Departamento de Sistemas Informáticos y Computación

Un Método de Desarrollo de Hipermedia Dirigido por Modelos

Tesis Doctoral

Carlos Solís Pineda

bajo la supervisión del Dr. José Hilario Canós Cerdá
y la Dra. María del Carmen Penadés Gramaje

Diciembre 2008

Tesis doctoral
©Carlos Solís Pineda
Impreso en España
Todos los derechos reservados
Pintura de la portada: *Untitled (Breck Girl)*, 1953, Martín Ramírez.
Esta tesis ha tenido una duración de 5 años, en ese periodo ha sido financiada por las siguientes organizaciones: el Consejo Nacional de Ciencia y Tecnología de México (CONACYT) durante 4 años con la beca 178867/193415. Fundación Carolina (España) ha sufragado la matrícula y pasajes aéreos México-España cada 6 meses durante los primeros 3 años. El grupo de investigación de Ingeniería del Software y Sistemas de Información (ISSI) del Departamento de Sistemas Informáticos y Computación de la Universidad Politécnica de Valencia (España) ha financiado la asistencia a congresos y talleres internacionales y nacionales a través de los subproyectos: PRISMA (OASIS Platform for Architectural Models) del proyecto DYNAMICA (Dynamic and Aspect-Oriented Modeling for Integrated Component-Based Architectures, proyecto TIC2003-7804-C05-01), y MOMENT (Model Management) del proyecto META (Models, Enviroments, Transformations and Applications, proyecto TIN2006-15175-C05-01).

Resumen

El desarrollo de sistemas hipermedia presenta dificultades inexistentes en los sistemas tradicionales. Se requiere modelar dominios de aplicaciones complejas y organizar su estructura de manera clara para proveer un fácil acceso a la información. La respuesta de la *Ingeniería del Software* a tal problema son los métodos de análisis y diseño de sistemas de hipermedia que proponen al modelo navegacional como solución.

La mayor parte de los métodos de la también llamada *Ingeniería Web* han seguido una aproximación basada en la estructura que muchas veces no captura toda la semántica de un problema determinado, ya que se limitan a construir caminos navegacionales derivados de las relaciones captadas en modelos como el Entidad-Relación o los Diagramas de Clases de UML. Tal estrategia puede ser útil en algunos casos, pero partir sólo del modelo estructural lleva a un modelo navegacional muy limitado, que no responde a la complejidad de los modelos navegacionales de aplicaciones reales. En general, existe un proceso de negocio subyacente, cuyo conjunto de tareas y su orden dan lugar a una parte de la navegación, y que no es abordado por los mencionados métodos.

En esta tesis se presenta el Método de Desarrollo de Hipermedia Dirigido por Modelos (MDHDM), que se fundamenta en el modelo de proceso como columna vertebral de las aplicaciones. El modelado conceptual está orientado al modelado del proceso, aunque también se define el modelo estructural. Ambos son usados en la obtención de modelos navegacionales más realistas, en tanto en cuanto cubren los aspectos estructural y dinámico.

Para definir de forma precisa el ciclo de de vida del método, los modelos, y sus correspondencias, se recurre a la *Ingeniería Dirigida por Modelos*. En particular, se define un metamodelo navegacional que permite reflejar las características navegacionales tanto del modelo de proceso como del estructural. El paso entre las sucesivas fases del método se efectúa mediante transformaciones de modelos, y plantillas de generación de código.

Resum

El desenvolupament de sistemes hipermèdia presenta dificultats inexistents en els sistemes tradicionals. Es requerix modelar dominis dáplicacions complexes i organitzar la seua estructura de manera clara per a proveir un fàcil accés a la informació. La resposta de l'*Enginyeria del Software* a tal problema són els mètodes dánàlisi i disseny de sistemes de hipermèdia, els quals proposen al model navegacional com a solució.

La major part dels mètodes de la també anomenada *Enginyeria Web* han seguit una aproximació basada en léstructura que moltes vegades no captura tota la semàntica dún problema determinat, ja que es limiten a construir camins navegacionals derivats de les relacions captades en models com ara lÉntitat-Relació o els Diagrames de Classes dÚML. Lésmentada estratègia pot ser útil en alguns casos, però partir només del model estructural porta a un model navegacional molt limitat, que no respon a la complexitat dels models navegacionals dáplicacions reals. En general, hi ha un procés de negoci subjacent, el conjunt de tasques del qual i el seu orde déxecució donen lloc a una part de la navegació que no és abordada pels esmentats mètodes.

En esta tesi es presenta el Mètode de Desenvolupament de Hipermèdia Dirigit per Models (MDHDM), que es fonamenta en el model de procés com a columna vertebral de les aplicacions. El modelatge conceptual està orientat al modelatge del procés, tot i què també es definix el model estructural. Ambdós són usats en lóbtenció de models navegacionals més realistes, en tant que cobrixen els aspectos estructural i dinàmic.

Per a definir de forma precisa el cicle de de vida del mètode, els models, i les seues correspondències, es recorre a la *Enginyeria Dirigida per Models*. En particular, es definix un metamodel navegacional, el qual permet reflectir les característiques navegacionals tant del model de procés com de léstructural. A partir dáquestos, el pas entre les successives fases del mètode séfectua per mitjà de transformacions de models i plantilles de generació de codi.

Abstract

The development of hypermedia systems raises challenges not found in traditional Information systems. Specifically, modelling complex domains, including a clear organization of their structure, is needed to allow easy access to information. As a result, *Software Engineering* researchers have proposed hypermedia analysis and design methods which propose the navigational model as a solution to cope with the complxity of hypermedia.

Most of the so-called *Web Engineering* methods have followed a structure based approach, which creates navigational paths derived from the relations of models such as the Entity-Relationship or the UML class diagrams. Such strategy is only useful in some cases because it leads to limited navigational models that do not take into account the complexity of the navigation in real applications. In general, there is an underlying business process, whose tasks and order give rise to part of the navigation. Such processes are not considered in most methods.

In this thesis we introduce the Model Driven Hypermedia Development Method (MDHDM), which is based on the process model as the reference for building applications. The conceptual modelling is aimed at modelling the process, as well as defining the structural model. Both are used for obtaining more realistic navigational models, which take into account structural and dynamic aspects.

Model Driven Engineering is used in order to define in a precise way the life cycle of the method, the models and their mappings. Specifically, a navigational metamodel is defined that reflects the navigational characteristics of both the process model and the structural model. The transition between the phases of the method is carried out through model transformations, and code generation templates.

Palabras clave

Método de Diseño de Hipermedia, Modelo de Proceso, Proyección del Proceso, Diseño Navegacional, Ingeniería Dirigida por Modelos.

Paraules clau

Mètode de Disseny de Hipermedia, Model de Procés, Projecció del Procés, Disseny Navegacional, Enginyeria Dirigida per Models.

Keywords

Hypermedia Development Method, Process Model, Process Projection, Navigational Design, Model Driven Engineeering.

¡Oh! cuando yo fui joven ávido he frecuentado
los santos y doctores, y oí cosas sublimes
sobre esto y sobre aquello: mas siempre me ha pasado
volverme por la puerta por donde había entrado.
Omar Khayyam.

Dedicatoria

Dedico esta tesis particularmente a mi madre María, a pesar de lo fugaz que he sido estos años, su cariño es perenne.

A mi padre Carlos y mis hermanos Cristina y Jorge que me han apoyado en todo lo que he emprendido, y nunca dejaron de animarme.

A Nour por enseñarme a aflorar mis sentimientos, e impulsarme a terminar la tesis en tiempos de desaliento.

Agradecimientos

Agradezco especialmente a mis supervisores, Dr. José Hilario Canós y Dra. María del Carmen Penadés, por sus valiosos comentarios y paciencia durante la revisión de esta tesis, y por su guía, enseñanzas, e incesante ayuda.

Quiero mostrar mi gratitud al Dr. Isidro Ramos por haberme recibido en el grupo de investigación de Ingeniería de Software y Sistemas de Información (ISSI), con lo que este trabajo dejo de ser un proyecto.

Doy las gracias al Dr. Marcos Borges y al Dr. Patricio Letelier, por sus consejos sobre cómo y para qué escribir una tesis, lo cual me dio motivación para continuar.

Esta tesis se enriqueció con las aportaciones recibidas durante las reuniones del grupo de investigación de Sistemas de Información Avanzados. Estoy agradecido con sus integrantes: Dr. José Hilario Canós, Dra. María del Carmen Penadés, Profesor visitante Dr. Marcos Borges de la UFRJ de Brasil, Dr. Patricio Letelier, y al doctorando Manuel Llavador.

Quiero agradecer a: Dr. Isidro Ramos, Dr. Emilio Insfrán, Dra. Silvia Abrahao, y el Dr. José Ángel Carsí, por haberme dado importantes mejoras y sugerencias en las reuniones de ISSI. Además, agradezco a la Dra. Nour Ali por la revisión que hizo de la tesis.

Mis compañeros doctorandos del grupo ISSI hicieron más agradable el ambiente de estudio, y además me proporcionaron útiles comentarios: Dra. Jennifer, María Eugenia, Manuel, Rogelio, Abel, Antonio, Alejandro y Cristobal, a todos ellos también gracias.

Agradezco al Dr. Felipe López y al Dr. Alfonso San Miguel por haberme impulsado a realizar este reto. A Caissa por darme amigos, a Fragiskos y Faisal por su amistad fraterna. A Carlos y Paloma por su afectuosa acogida al llegar a España. A mis amigos Marga, Nelly, Gonzalo, mi primo Javier, y mis sobrinos: Christian, Areli y Sarahí.

Me gustaría agradecer al Consejo Nacional de Ciencia y Tecnología (México) y a Fundación Carolina (España), por su apoyo financiero, que permitió realizar esta tesis.

Índice general

Índice de figuras XVIII

Índice de tablas XXIII

Acrónimos XXIV

I Introducción 1

1. Introducción 3

1.1. La ingeniería de software y los sistemas hipermedia 3
1.2. Motivación del trabajo 5
1.3. Problema 7
1.4. Hipótesis 8
1.5. Objetivos 8
1.6. Estructura de la tesis 9

II Estado del arte 13

2. Métodos de desarrollo de hipermedia 15

2.1. Introducción 15
2.2. HDM 16
2.3. RMM 20
2.4. OOHDM 24
2.4.1. Procesos con OOHDM 27

2.5. WebML 29
2.5.1. Procesos con WebML 32
2.6. UWE 34
2.6.1. Procesos con UWE 36
2.7. Otros métodos 39
2.7.1. OO-H 39
2.7.2. OOWS 41
2.7.3. WSDM 43
2.7.4. Ariadne 44
2.7.5. EORM 47
2.7.6. PML 47
2.7.7. Magrathea 48
2.7.8. Hera 49
2.7.9. SOHDM 49
2.8. Comparativa de los métodos 50
2.9. Conclusiones 55

3. Ingeniería dirigida por modelos 57
3.1. Introducción 57
3.2. Ingeniería Dirigida por Modelos 58
3.2.1. Modelos, metamodelos y sus relaciones 58
3.2.2. Transformación de modelos 60
3.3. Arquitectura Dirigida por Modelos 62
3.3.1. Conceptos básicos de MDA 62
3.3.2. Meta-Object Facility 65
3.4. Espacios Tecnológicos en MDA 67
3.4.1. XML 68
3.4.2. ATL 69
3.4.3. MOMENT 72
3.5. Marco comparativo 75
3.5.1. Características generales 75
3.5.2. Características operacionales 77
3.5.3. Características de la plataforma de soporte 78

3.5.4. Evaluación. 79
3.6. MDE e hipermedia 82
3.6.1. OOHDMDA 82
3.6.2. UWE 83
3.6.3. MIDAS 83
3.7. Conclusiones 84

III MDHDM 87

4. MDHDM 89
4.1. Introducción 89
4.2. Método de Desarrollo de Hipermedia 90
4.2.1. Definición del ciclo de vida 91
4.2.2. Definición de los metamodelos y sus restricciones . . . 95
4.2.3. Identificación de modelos PIM y PSM 96
4.2.4. Especificación de marcas de modelado 97
4.2.5. Especificación de las transformaciones 98
4.3. Conclusiones 99

5. Modelado conceptual 101
5.1. Introducción 101
5.2. Casos de uso 102
5.2.1. Características 102
5.2.2. Notación 103
5.2.3. Metamodelo 104
5.3. Flujos de trabajo 107
5.3.1. Características 108
5.3.2. Notación 109
5.3.3. Metamodelo 111
5.4. Modelo estructural 119
5.4.1. Características 120
5.4.2. Metamodelo 120
5.5. Correspondencias entre metamodelos 124

5.5.1. De casos de uso a flujo de trabajo 124
5.5.2. Del flujo de trabajo al modelo estructural 127
5.5.3. Ejemplo 127
5.6. Conclusiones 130

6. Diseño Navegacional 131
6.1. Introducción 131
6.2. Modelo navegacional 132
6.2.1. Nodos 132
6.2.2. Enlaces 134
6.3. Metamodelo Navegacional 139
6.4. Proceso para el diseño navegacional 146
6.5. Proyección del modelo de proceso 147
6.5.1. Ejemplo de proyección 153
6.6. Obtención del modelo navegacional fuerte 154
6.6.1. Marcas del modelo de proceso 154
6.6.2. Transformación del modelo de proceso 155
6.7. Obtención del modelo navegacional débil 164
6.7.1. Transformación basada en las relaciones 165
6.7.2. Transformaciones complejas 167
6.7.3. Atajos 168
6.7.4. Marcas aplicables al modelo estructural 169
6.7.5. Vista del modelo estructural por actor 173
6.8. Conclusiones 174

7. Diseño de la interfaz abstracta 175
7.1. Introducción 175
7.2. Metamodelo de la interfaz abstracta 176
7.3. Esquemas de las instancias de datos 181
7.4. Interfaces Abstractas de Usuario 182
7.4.1. Vista de tareas 183
7.4.2. IAU de las clases navegacionales 184
7.4.3. IAU de las visitas guiadas 185

7.4.4. IAU de índices 186
7.5. Ejemplo de interfaz abstracta 187
7.6. Conclusiones 192

8. Implementación 195
8.1. Introducción 195
8.2. Eclipse Modeling Framework 196
8.3. Verificación de los modelos 198
8.4. Proyección del proceso 200
8.5. Atlas Transformation Language 203
8.6. Generación de código 205
8.7. Modelos específicos de la plataforma 206
8.7.1. Modelo de persistencia 206
8.7.2. Modelo navegacional 208
8.7.3. Modelo de presentación 208
8.8. Conclusiones 211

9. Casos de estudio 213
9.1. Introducción 213
9.2. Librería electrónica 213
9.2.1. Modelo de proceso 214
9.2.2. Modelo estructural 216
9.2.3. Modelo navegacional fuerte 216
9.2.4. Modelo navegacional débil 219
9.3. Sistema de revisión de artículos 220
9.3.1. Modelo de proceso 222
9.3.2. Modelo estructural 222
9.3.3. Proyecciones 224
9.3.4. Modelo navegacional fuerte 225
9.3.5. Modelo navegacional débil 226
9.4. Implementación 227
9.5. Conclusiones 228

IV Conclusiones 229

10. Conclusiones y trabajo futuro 231

10.1. Contribuciones 232

10.2. Trabajo futuro 235

10.3. Publicaciones 237

Bibliografía 241

A. Transformaciones 253

A.1. Del modelo de proceso al navegacional 253

A.2. Del modelo conceptual al navegacional 263

B. Verificaciones 275

B.1. Verificación del modelo de proceso 275

B.2. Verificación del modelo conceptual 281

C. Plantillas de las Interfaces Abstractas 289

C.1. Plantillas de generación de los esquemas XML 289

C.2. Interfaces abstractas por omisión 290

Índice de figuras

1.1. Librería electrónica 6

2.1. Notación de HDM 17
2.2. Ejemplo en HDM 18
2.3. Notación de RMM 20
2.4. Ejemplo en RMM 21
2.5. Diagrama de clases navegacionales en OOHDM 25
2.6. Diagrama de contextos en OOHDM 26
2.7. Modelo de proceso OOHDM 28
2.8. Notación de WebML 30
2.9. Ejemplo en WebML 32
2.10. Primitivas de proceso de WebML 32
2.11. Notación y ejemplo en UWE 37
2.12. Modelo UWE con enlaces de proceso 38
2.13. Clases de proceso 38
2.14. Árbol CTT 42
2.15. Descripción de una tarea 43
2.16. Evolución de los métodos 54

3.1. Relaciones InstanceOf y RealizationOf 60
3.2. Elementos de una transformación 61
3.3. Niveles de abstracción en MDA 63
3.4. Patrón de transformación en MDA [MM03] 64
3.5. Torre reflexiva de MOF 66

3.6. Metamodelos orientado a objetos y relacional 70
3.7. Proyectores en MOMENT 72

4.1. Notación de SPEM 91
4.2. Ciclo de desarrollo genérico de los MDH 92
4.3. Modelado del proceso en el diseño navegacional 92
4.4. Ciclo de desarrollo en MDHDM 92
4.5. Modelado del proceso 93
4.6. Diseño navegacional 94
4.7. Diseño de la presentación 95
4.8. Modelos y transformaciones en MDHDM 96

5.1. Notación para representar casos de uso 103
5.2. Metamodelo de casos de uso 105
5.3. Componentes principales de un flujo de trabajo 109
5.4. Notación BPMN 110
5.5. Metamodelo de flujo de trabajo 112
5.6. Flujo de control 113
5.7. Taxonomía de tareas de un flujo de trabajo 113
5.8. Ramificación y unión parcial 114
5.9. Ramificación y unión total 115
5.10. Proceso con eventos intermedios 118
5.11. Flujo de datos 119
5.12. Metamodelo estructural 121
5.13. Casos de uso de la librería electrónica 128
5.14. Proceso derivado del caso de uso compra 128
5.15. Proceso derivado del caso de uso modificar carro 129
5.16. Proceso derivado del caso de uso cerrar compra 129

6.1. Tipos de nodo 133
6.2. Interpretación de los enlaces débiles 135
6.3. Composición de nodos 135
6.4. Enlaces con memoria 136
6.5. Notación de vista de tarea y enlace fuerte 137

6.6. Ejemplo de vistas de tarea 138
6.7. Metamodelo navegacional 140
6.8. Vistas de tarea y enlaces 141
6.9. Pasos para construir el modelo navegacional de un actor. . . . 146
6.10. Regla de transformación 149
6.11. Regla de proyección 1 149
6.12. Regla de proyección 2 150
6.13. Reglas de proyección 3 y 6 *(Or/And)-gateway* 150
6.14. Reglas de proyección 4 y 7 *(Or/And)-gateway* 150
6.15. Reglas de proyección 5 y 8 *(Or/And)-gateway* 151
6.16. Reglas de proyección 9 y 12 *(Or/And)-gateway* 151
6.17. Reglas de proyección 10 y 13 *(Or/And)-gateway* 152
6.18. Reglas de proyección 11 y 14 *(Or/And)-gateway* 152
6.19. Ejemplo de la proyección para el actor α 153
6.20. Sucesor de un nodo 157
6.21. Transformación del contenedor del modelo de proceso 158
6.22. Transformación de un nodo sincronizado 158
6.23. Transformación de las tareas de captura de datos 159
6.24. Transformación de las tareas de selección 160
6.25. Transformación de las tareas de edición/visualización 160
6.26. Transformación XOR gateway diferido 160
6.27. Transformación XOR gateway 161
6.28. Transformación del And-gateway (1) 162
6.29. Transformación del And-gateway (2) 163
6.30. Serialización de tareas 163
6.31. Transformación de un subproceso 164
6.32. Transformación de asociación 1-1 166
6.33. Transformación de asociación 1-N 166
6.34. Transformación de asociación M-N 166
6.35. Transformación de una relación en estrella 167
6.36. Transformación de una cadena de relaciones 1-N 168
6.37. Transformación inversa de una cadena de relaciones 1-N . . . 168
6.38. Marcas de omitido y fusión 170

6.39. Marca Tour 171
6.40. Marca NoNavegable 172
6.41. Creación del nodo Home 173

7.1. Metamodelo de la IAU 177
7.2. Interfaz instanciada de un libro 187
7.3. Modelo navegacional del ejemplo de IAU 188

8.1. Modelo Ecore 196
8.2. Editor tabular de modelos Ecore 197
8.3. Editor de instancias 198
8.4. Construcción de una instancia 199
8.5. Consola de ejecución de ATL 203
8.6. Arquitectura del código generado 206
8.7. Metamodelo Java y MVC 207
8.8. Proveedor de datos 209
8.9. Clases del PSM de presentación 209
8.10. Interacción del PSM de presentación 210

9.1. Librería electrónica 214
9.2. Subproceso de selección de libros 215
9.3. Subproceso de modificación del carro de compras 215
9.4. Subproceso del cierre de compra 216
9.5. Modelo estructural de la librería electrónica 216
9.6. Modelo navegacional fuerte de la librería 217
9.7. Modelo navegacional fuerte de la librería-II 220
9.8. Modelo navegacional débil 221
9.9. Modelo de proceso del sistema de revisión de artículos 221
9.10. Modelo conceptual del sistema de revisión de artículos 222
9.11. Proyecciones del sistema de revisión de artículos 223
9.12. Modelos navegacionales fuertes 225
9.13. Modelo débil del sistema de revisión de artículos 227
9.14. Modelo navegacional fuerte para los autores 227
9.15. Modelo navegacional fuerte para los revisores 228

Índice de tablas

2.1. Fases, actividades y productos de Ariadne 46
2.2. Comparativa de los métodos(1) 52
2.3. Comparativa de los métodos(2) 53
2.4. Métodos que modelan procesos de negocios 54

3.1. Espacios Tecnológicos en MDA 67
3.2. Características generales 79
3.3. Características operacionales 80
3.4. Características de la plataforma de transformación 80
3.5. Características de OOHDMDA 83
3.6. Características de MIDAS 84

4.1. Transformaciones automáticas 98

5.1. Plantilla de caso de uso 104
5.2. Equivalencia de caso de uso a actividad 125
5.3. Correspondencia de inclusión a flujo de trabajo 125
5.4. Correspondencias de extensión a flujo de trabajo 126

6.1. Marcas aplicables al modelo de proceso 155
6.2. Marcas del modelo navegacional derivadas del proceso 155
6.3. Transformaciones del modelo de proceso a navegacional . . . 156
6.4. Transformaciones del modelo de estructural al navegacional . 165
6.5. Marcas aplicables al modelo estructural 169

7.1. Representación XML de los elementos de la interfaz 180

Acrónimos

ADV Abstract data views

ADM Ariadne Development Method

API Application Programming Interface

ATL Atlas Transformation Language

AMMA ATLAS MegaModel Management Architecture

AMW ATLAS Model Weaver

AM3 ATLAS MegaModel Management Tool

ATP ATLAS Technical Projectors

BPEL Business Process Execution Language

BPMN Business Process Modelling Notation

CASE ingeniería de software asistida por computador

CRC Class-Responsability-Collaborator

CTT Concur Task Trees

DDL Data Definition Language

ECA Event Condition Action

E-R Entity-Relationship Model

EMF Eclipse Modelling Framework

EORM Enhanced Object Relationship Methodology

HDM Hypertext Design Model

HTML HyperText Markup Language

IAU Interfaz Abstracta de Usuario

MDA Model Driven Architecture

MDE Model Driven Engineering

MDHDM Model Driven Hypermedia Development Method

MOF Meta-Object Facility

MOMENT Model Management Framework

MVC Model-View-Controller

OCL Object Constraint Language

OOHDM Object Oriented Hypermedia Design Method

OOHDMDA OOHDM-MDA

OOSE Object Oriented Software Engineering

OMG Object Management Group

OMT Object-modeling Technique

OO-H Object Oriented Hypermedia Method

OOWS Object Oriented Web Solution

ORM Object Role Model

OWL Ontology Web Language

PIM Platform Independent Model

PSM Platform Specific Model

PML Process Modeling Language

RDF Resource Description Framework

QVT Query View Transformation

RMM Relationship Management Methodology

RMDM Relationship Management Data Model

RUP Rational Unified Process

SOHDM Scenario-Based Object-Oriented Hypermedia Design Methodology

SPEM Software Process Engineering Metamodel

WebML Web Modeling Language

WfMS Workflow Management System

WSDM Web Site Design Method

WWW World Wide Web

UML Unified Modeling Language

XMI XML Metadata Interchange

XML Extensible Markup Language

XSL Extensible Stylesheet Language

XSLT XSL Transformations

XUL XML-based User-interface Language

Parte I

Introducción

Capítulo 1

Introducción

1.1. La ingeniería de software y los sistemas hipermedia

El concepto de *hipertexto* fue definido por Nelson en 1965 [Nel65] como: "texto o imágenes interconectadas de forma compleja que no puede representarse en papel". Es decir, texto no secuencial con ramificaciones seleccionables por el lector, de modo que éste mejora su experiencia de lectura al profundizar en los temas que le interesan o donde requiere de explicaciones más amplias. El hipertexto rompe con la linealidad del discurso de lectura, le quita rigidez y le da libertad al lector [Lan92]. Nelson en el mismo artículo menciona la palabra *hipermedia* (sin definirla de forma amplia), y presenta como ejemplo de ésta al *hiperfilm*, un tipo de película con múltiples secuencias.

Un concepto similar al hipertexto fue propuesto por Bush en 1945 en su artículo "As we may think"[Bus45]. Bush imaginó un gran registro de conocimiento llamado *Memex*, que puede ser continuamente extendido, almacenado, consultado. En Memex los conceptos están asociados entre ellos como en redes semánticas. El objetivo de Memex era la extensión de la mente humana, debido a que se considera que ésta trabaja mediante asociaciones.

Hipertexto e hipermedia han estado ligados desde que aparecieron en el vocabulario. La definición de hipermedia en [Low99] es: "una aplicación

que utiliza relaciones asociativas entre información contenida en diversos medios de datos, con el propósito de facilitar el acceso y la manipulación de la información encapsulada por los datos". Las definiciones de hipertexto e hipermedia del *World Wide Web Consortium* son: "texto que no está restringido por la linealidad", e "hipertexto multimedia"[W3C95].

Un nodo es una unidad de información que contiene texto o elementos multimedia como vídeo, sonido o imágenes. Un enlace conecta dos nodos de forma unidireccional y está asociado a un elemento específico dentro del nodo donde se muestra. La característica esencial de la hipermedia son las redes de nodos de información conectados mediante enlaces. La acción de activar un enlace, localizado en un nodo, y llegar al contenido del nodo apuntado por el enlace se conoce como *navegar*. La *World Wide Web* (WWW o Web) [BL99] es el sistema de hipermedia más usado y tal vez el sistema con mayor impacto social en el mundo actual.

El desarrollo de sistemas hipermedia presenta dificultades inexistentes en los sistemas tradicionales. Se requiere modelar dominios de aplicaciones complejas y organizar su estructura de manera clara para proveer un fácil acceso a la información. Además, las relaciones entre nodos y la posición en la red de nodos es desconocida por los usuarios, lo cual hace difícil decidir cuál es la siguiente navegación dentro del sistema, tal problema se conoce como *perderse en el hiperespacio* [OJ00]. La solución de ambos es el objetivo del *diseño de navegacional*. También, entre otros problemas que plantea su desarrollo, se encuentra que requieren la participación de personas con habilidades diferentes: autores, bibliotecarios, diseñadores gráficos y programadores, y en consecuencia dificultades de administración y planeación. A pesar de que las aplicaciones hipermedia comenzaban a verse comúnmente en los 90s, la necesidad de métodos de diseño de hipermedia recibió empuje con el gran crecimiento de sitios *web* en *Internet*.

En las primeras etapas de desarrollo de la Web parecía que las aplicaciones eran caóticas, carecían de estructura, y hacían falta métodos de desarrollo adecuados. Asimismo, la tolerancia a errores por parte de los usuarios es menor, porque la distribución a través de Internet dañaría en poco tiempo la imagen de calidad del proveedor de la aplicación. Debido a

los problemas mencionados se necesitan métodos que permitan el desarrollo rápido de sistemas y prototipos. En resumen, el desarrollo de hipermedia también requiere de un proceso estructurado y sistemático.

La *ingeniería de software* proporciona las herramientas de análisis y diseño necesarias para que los desarrolladores puedan representar de forma abstracta un problema, y a partir de ésta generar una solución informática. Los métodos de análisis y diseño de sistemas de hipermedia contienen esos objetivos más los añadidos por la dimensión de hipermedia que son básicamente la navegación a través de la información y su presentación.

1.2. Motivación del trabajo

La conjunción de hipertexto y multimedia, por un lado, y la popularización de la Web, por otro, generaron desafíos que los métodos tradicionales no eran capaces, aparentemente, de resolver. Entre ellos destaca el diseño y control de la navegación por espacios complejos de información, que ha llevado a los métodos de diseño de hipermedia y de la llamada *Ingeniería Web* a presentar el modelo navegacional como la gran diferencia frente a los métodos tradicionales.

El modelo navegacional es su principal aportación. Éste se deriva del modelo de datos o estructural e incluye como principal elemento las llamadas *estructuras de acceso* que sirven para indicar los caminos navegacionales disponibles en la aplicación. Aunque es discutible que la navegación sea exclusiva de la hipermedia o la Web (en cualquier aplicación existe navegación, tal vez no basada en enlaces, pero sí apoyada en otro tipo de elementos de interfaz de usuario), un mérito indiscutible de los métodos de Ingeniería Web ha sido explicitar la necesidad de un diseño cuidadoso de la misma. Surgieron métodos como *Hypertext Design Model* (HDM) [GPS93], *Object Oriented Hypertext Design Method* (OOHDM) [SRJ96] y más recientemente *Web Modeling Language* (WebML) [CFB00]. Estos ampliaron los métodos de análisis y diseño incluyendo la dimensión de navegación.

Los métodos de desarrollo de hipermedia proponen estructurar el proceso de desarrollo en una serie de fases, usualmente, modelado conceptual, diseño

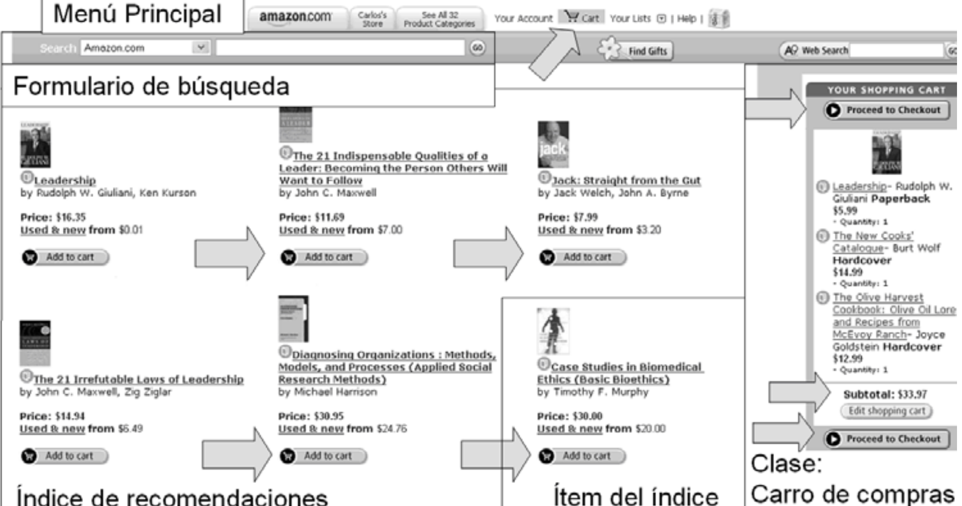


Figura 1.1: Librería electrónica

navegacional, diseño de la presentación e implementación. La mayor parte de los métodos han seguido una aproximación basada en la estructura que muchas veces no captura toda la semántica de un problema determinado, ya que se limitan a construir caminos navegacionales derivados de las relaciones captadas en modelos como el Entidad-Relación, o el Diagrama de Clases de UML. Dicha estrategia puede ser útil en algunos casos, pero partir sólo del modelo estructural lleva a un modelo navegacional muy limitado, que no responde a la complejidad de los modelos navegacionales de aplicaciones reales. Considérese, por ejemplo, el portal de una librería electrónica tal como Amazon (http://www.amazon.com).

La figura 1.1 muestra una captura de pantalla correspondiente a una fase intermedia del proceso de compra de un libro. Si bien algunos enlaces corresponden, claramente, a navegación derivada del modelo estructural (por ejemplo, los enlaces que conducen al detalle de un libro, el menú principal y el botón de envío del formulario de búsqueda), hay otros que de ningún modo tienen ese origen, sino que aparecen en esa pantalla por el hecho de que el usuario, en ese momento del proceso de compra, debe realizar una serie de acciones (indicados por las flechas). En el ejemplo, el usuario

está en la actividad de selección: puede añadir un nuevo producto o, como ya ha seleccionado algunos productos, puede editar el carro de compras, o cerrar la compra. En otras palabras: en general existe un proceso de negocio subyacente, cuyo conjunto de tareas y el orden entre las mismas dan lugar a una parte muy importante de la navegación de la aplicación Web.

Los enfoques derivados del modelo de datos son adecuados para muchas aplicaciones; sin embargo, existen otras como los *Sistemas de Gestión de Emergencias* [CMLP03] [CAJ04], el comercio electrónico y diversos procesos de negocio realizados en las Intranets de las organizaciones en las cuales se tiene un proceso bien definido, y donde los caminos navegacionales no son determinados únicamente por el modelo estructural, sino por el modelo de proceso, también denominado flujo de trabajo (*workflow*) [Mem99], y éste es lo que realmente le interesa seguir al usuario. Actualmente las aplicaciones hipermedia son ampliamente usadas, y existe la necesidad de obtenerlas a partir de especificaciones de modelos de proceso [SvdAA99].

Pese a ello, sólo recientemente algunos de los métodos mencionados han tenido en consideración la perspectiva de proceso en la obtención del modelo navegacional. La naturaleza dinámica de la realidad no queda plasmada en modelos estructurales, sino que son los modelos de proceso los que realmente la capturan. Dicha dimensión ha sido ignorada repetidamente por los métodos de Ingeniería Web y tratada de una forma poco natural, siempre como añadido a la generación basada en la estructura del modelo navegacional. En las metodologías existentes el modelo de proceso es introducido durante el diseño navegacional, por lo cual, el modelo resultante no refleja el proceso que ejecuta el usuario. Es necesario partir del modelo de proceso para explicitar las tareas que ejecutará el usuario mientras usa el sistema, y distinguir, claramente, la existencia de dos tipos de enlaces: los estructurales y los de proceso.

1.3. Problema

Los modelos navegacionales diseñados a partir de las relaciones entre los datos del modelo estructural no permiten modelar, de manera realista, la

navegación. En los sistemas hipermedia y *web* se observan enlaces y comportamientos que no son factibles de explicar a partir del modelo estructural. Las cuestiones a resolver son: ¿de dónde surgen los modelos navegacionales? y ¿cómo se deben diseñar? Es decir, se requiere averiguar cuál es el origen de los enlaces y nodos mostrados en las aplicaciones hipermedia. Además, se deben proporcionar las técnicas de modelado que permitan analizar y desarrollar tales sistemas. Las preguntas anteriores llevan, además, a la siguiente: ¿cómo presentar la información del sistema de modo que se permita el acceso a la misma durante la ejecución de un proceso en donde participan un actor o varios?

1.4. Hipótesis

De acuerdo al problema presentado en la sección anterior, se plantea la siguiente hipótesis:

> Basándose en el flujo de control y de datos de los modelos de proceso se podrían modelar las navegaciones complejas de las aplicaciones hipermedia.

1.5. Objetivos

El objetivo principal de la tesis es definir un método para el desarrollo de hipermedia que se fundamente en el modelo de proceso como columna vertebral de las aplicaciones, y que utilice las técnicas actuales de ingeniería de software. El método debe permitir modelar navegaciones más complejas que las obtenidas a través de los métodos basados en la estructura de los datos.

Para alcanzar el objetivo se propone utilizar la *Ingeniería Dirigida por Modelos* (MDE). En MDE los modelos son entidades de primer orden que permiten especificar la funcionalidad de un sistema sin detalles de implementación, y que son usados para la generación del código del sistema. MDE

es adecuado para tratar las aplicaciones hipermedia y *web* porque las diversas partes de interés (*concerns*) de un sistema se definen con diferentes modelos (proceso, conceptual, navegación, presentación). A continuación se presentan los subobjetivos en los que se descompone el objetivo principal.

Antes de abordar el desarrollo del método se deben conocer los trabajos previos en el área, y también se debe profundizar en MDE:

- Analizar el estado del arte de los métodos de desarrollo de hipermedia.
- Estudiar los conceptos y técnicas MDE, para seleccionar un marco de trabajo de MDE adecuado al desarrollo del método.

Utilizando el marco de trabajo de MDE seleccionado se deben efectuar las tareas de definición del método:

- Definir el ciclo de vida del método.
- Definir una estrategia de modelado conceptual centrada en el modelo de proceso.
- Definir un metamodelo navegacional que tenga en cuenta la estructura del proceso.
- Especificar cómo obtener la navegación a partir de los modelos conceptuales.
- Definir la interfaz de presentación del modelo navegacional.
- Definir cómo implementar a partir de los modelos elaborados.
- Aplicar el método a casos de estudio.

1.6. Estructura de la tesis

En esta tesis se presenta al Método de Desarrollo de Hipermedia Dirigido por Modelos (MDHDM). El trabajo está organizado en cuatro partes. La primera parte es la introducción, donde se explican los motivos de esta tesis,

y se define el problema a resolver y la estrategia a seguir. La segunda parte trata de los trabajos relacionados y de los conceptos de MDE necesarios para definir el método. La tercera parte define en su primer capítulo en qué consiste MDHDM. Posteriormente, en cada capítulo se explican sus diferentes etapas, y se incluye uno sobre los casos de estudio. La cuarta parte contiene las conclusiones y bibliografía. Adicionalmente, se incluyen los apéndices con las transformaciones de modelos, verificaciones de modelos, y plantillas de generación de las interfaces abstractas de usuario (IAU). A continuación se dan los detalles de cada capítulo del resto de la tesis:

- Parte II: Estado del arte.
    - Capítulo 2: Métodos de desarrollo de hipermedia.
      Muestra un análisis de los diversos métodos de desarrollo de hipermedia y *web*. Se describen sus principales características y notación, además, en los casos más influyentes se incluye un ejemplo de sus modelos. Al final del capítulo se efectúa una comparativa de los métodos.
    - Capítulo 3: Ingeniería dirigida por modelos (MDE).
      Presenta los conceptos de MDE y de la Arquitectura Dirigida por Modelos (MDA). Después se explican algunos espacios tecnológicos sobre los que podría desarrollarse el método, y se hace una evaluación de éstos. Por último presenta cómo se ha usado MDE en los métodos de desarrollo de hipermedia y *web*.
- Parte III: Método de Desarrollo de Hipermedia Dirigido por Modelos.
    - Capítulo 4: MDHDM.
      Este capítulo presenta las características generales de MDHDM. Define el ciclo de vida del método y los pasos para transformarlo en un método MDE: identificación los metamodelos involucrados, su clasificación como PIM o PSM y la especificación de marcas y transformaciones entre metamodelos.
    - Capítulo 5: Modelado conceptual.
      Define los modelos utilizados durante el modelado conceptual: los

modelos de comportamiento, casos de uso y proceso; y el modelo de clases o estructural. Se presentan sus características, notación y metamodelo. También se incluyen las correspondencias entre los modelos de: casos de uso y flujos de trabajo, flujos de trabajo y estructural, y un ejemplo de su aplicación.

- Capítulo 6: Diseño navegacional.
  Define el modelo navegacional de MDHDM, su notación, restricciones y metamodelo. Además muestra cómo pasar del modelo de proceso y estructural al modelo navegacional, a través de proyecciones y transformaciones, lo cual da lugar a la obtención de los modelos navegacional fuerte y débil. Se explican cada una de las funciones de transformación y las marcas aplicables a los modelos origen.

- Capítulo 7: Diseño de la Interfaz Abstracta.
  Se presenta un modelo para especificar interfaces abstractas de usuario (IAU) de forma independiente de la tecnología y que permite vincular a los elementos de la interfaz con los datos a presentar. Se especifica el metamodelo de las IAU, y también se muestra cómo derivar las IAU a partir del modelo navegacional.

- Capítulo 8: Implementación.
  Se explica cómo se ha implementado el método y se presenta su prototipo. Se detalla cómo se construyen los metamodelos y modelos, y las restricciones y transformaciones. También se explica cómo se generó el código y la estructura de los PSM.

- Capítulo 9: Casos de estudio.
  Se desarrollan dos casos de estudio: el de una librería electrónica y el de un sistema de revisión de artículos. De cada uno se especifican sus modelos conceptuales y de navegación, mediante los modelos y transformaciones definidas en los capítulos 5 y 6.

■ Parte IV: Conclusiones.

- Capítulo 10: Conclusiones y trabajo futuro.

Presenta las conclusiones, contribuciones, y trabajo futuro de esta tesis.

  - Bibliografía.

- Apéndice A: Transformaciones ATL.
  Contiene a las transformaciones de modelos para obtener el modelo navegacional a partir de los modelos de proceso y estructural.

- Apéndice B: Verificaciones.
  Muestra las restricciones OCL que verifican los modelos en forma de transformaciones ATL.

- Apéndice C: Plantillas de generación de IAU.
  Presenta las plantillas para generar las IAU por omisión para los elementos del modelo navegacional.

Parte II

Estado del arte

Capítulo 2

Métodos de desarrollo de hipermedia

2.1. Introducción

Los métodos de desarrollo hipermedia surgieron para, principalmente, resolver el problema del diseño de la navegación sobre espacios complejos de información, y de presentación de la información en la interfaz de usuario. Las principales contribuciones de los métodos de desarrollo de hipermedia, (en adelante, sólo se les llamará métodos) son ciclos de vida, técnicas, notaciones y heurísticas para efectuar el desarrollo de aplicaciones hipermedia.

El desarrollo de sitios *Web* también es cubierto por los métodos debido a que son un tipo de sistema hipermedia. La popularidad de la *Web* contribuye notablemente a remarcar la necesidad de estos métodos. Realmente, cualquier aplicación también cuenta con navegación, la cual puede ser efectuada con medios diferentes a los hiperenlaces (por ejemplo, ventanas y botones), por lo que las técnicas introducidas por los métodos pueden ser, igualmente, útiles en otros contextos.

Los métodos definen las etapas y actividades necesarias para efectuar la construcción completa de un sistema hipermedia. La mayoría de los métodos definen, con nombres particulares cada uno, las siguientes etapas:

Análisis conceptual. Trata de la especificación del dominio del problema, a través de la definición de datos y sus relaciones.

Diseño navegacional. Establece los caminos de acceso a la información y sus permisos de visibilidad.

Diseño de la presentación. Define cómo se muestra la información en la interfaz de usuario.

Implementación. Es la construcción del software a partir de las artefactos generados en las etapas previas.

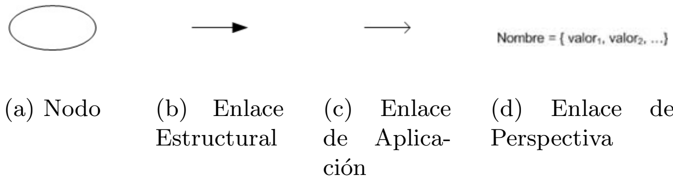
En este capítulo se presentan, por orden cronológico, algunos métodos que se han usado como referencia básica y que son los más conocidos y utilizados. Se describen sus características y aportaciones más destacadas, su ciclo de desarrollo, su notación y cómo utilizan el modelado de procesos, si es que lo consideran. Se hace hincapié en su manera de modelar aplicaciones hipermedia que implementan procesos. Se incluye además una sección de otros métodos donde se presentan aquellos que han sido menos relevantes para elaborar este trabajo y de los cuales existe menos información en la literatura. Posteriormente se presenta una comparativa de los métodos, y finalmente en la sección 2.9 las conclusiones del capítulo.

2.2. Hypertext Design Model (HDM)

HDM [GPS93], [GMP95] es considerado uno de los primeros métodos, y ha sido la base de otros como RMM [ISB95] y OOHDM [SR95b], presentados en las siguientes secciones. En HDM se distinguen dos etapas en el diseño de aplicaciones: diseño a gran escala (*design in the large*), y el diseño a pequeña escala (*design in the small*). El diseño a gran escala se refiere al diseño global y a los aspectos estructurales de la aplicación hipermedia, en otras palabras, trata de la definición de las relaciones conceptuales entre los nodos de la aplicación hipermedia. El diseño a pequeña escala se refiere al desarrollo del contenido de los nodos de hipermedia, y está relacionado con la implementación de éstos, pues trata, de solucionar problemas como la

obtención de la información desde una base de datos, y con qué herramientas de desarrollo se programará, etc.

La notación de HDM se muestra en la figura 2.1.



(a) Nodo (b) Enlace Estructural (c) Enlace de Aplicación (d) Enlace de Perspectiva

Figura 2.1: Notación de HDM

La información está agrupada en nodos, representados mediante óvalos (figura 2.1a) e identificados por un nombre. HDM distingue los siguientes tipos de nodos:

- Una unidad es un nodo que no se compone por ningún otro y es una hoja en la jerarquía.

- Un componente está integrado por un conjunto de unidades o componentes, éstos tienen una relación de pertenencia con el componente superior, es decir, los componentes forman jerarquías.

- Una entidad es una estructura de información que describe un objeto abstracto o real, que es parte del dominio de aplicación. Una entidad define un tipo, y cada entidad del mismo tipo comparte la misma estructura y conexiones con las demás, y es un *componente raíz*.

Los nodos se relacionan mediante enlaces, que pueden ser, también, de tres tipos:

- Los enlaces estructurales (figura 2.1b) sirven para conectar componentes que pertenecen a la misma entidad, es decir, conectan las entidades, componentes y unidades de una jerarquía. Este tipo de enlaces se identifican por una flecha triangular.

- Un enlace de aplicación (figura 2.1c) sirve para conectar nodos de distintas jerarquías, este tipo de enlace se representa por una flecha normal.

- Un enlace de perspectiva (figura 2.1d) corresponde a una vista de la aplicación. Cada perspectiva define un conjunto de propiedades visibles en la jerarquía de componentes. Las perspectivas se definen a nivel de entidad, y cada uno de sus componentes debe tenerla definida también.

Los otros elementos de navegación son las *estructuras de acceso o colecciones*. Una *colección* es un conjunto de enlaces que apuntan a nodos hipermedia. Las colecciones pueden ser de dos tipos: índices y visitas guiadas. Un *índice* permite navegar directamente a los nodos que pertenecen a la colección, y una *visita guiada* muestra la colección de objetos a través de un recorrido secuencial hacia adelante y atrás. Las estructuras de acceso son componentes y no tienen un símbolo para especificarlas, sólo se indican, adicionalmente, en el nombre del nodo como etiqueta.

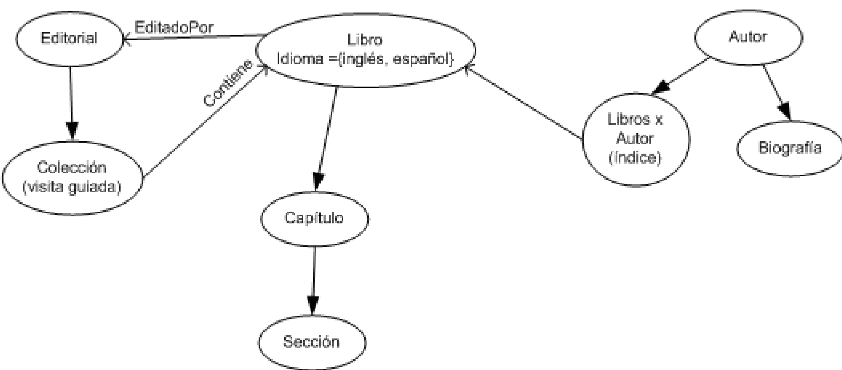


Figura 2.2: Ejemplo en HDM

La figura 2.2 muestra un modelo HDM de una librería electrónica, la cuál está conformada por libros clasificados en colecciones de alguna editorial. *Editorial, libro y autor* son las entidades o componentes del nivel más alto. El nodo *capítulo* es un componente, pero no una entidad. La jerarquía correspondiente a *editorial* se descompone en *colecciones* (de libros), la de

libro en *capítulos*, y éstos a la vez en *secciones*. La entidad *autor* forma una jerarquía con los nodos índice de *libros* y *biografía*, los cuales son unidades. Los enlaces que conforman la jerarquía son estructurales, y los que interconectan las jerarquías son enlaces de aplicación. Los *libros* tienen la perspectiva idioma que puede ser inglés o español, de modo que los nodos *capítulo* y *sección*, se pueden agrupar mediante esa perspectiva.

HDM se centra en el diseño a gran escala. El diseño a pequeña escala prácticamente no es abordado en el método y se limita sólo a la asignación de contenido a los nodos. El proceso de desarrollo en HDM está integrado por los siguientes pasos:

1. Identificar el grupo de entidades del mundo real y sus componentes. Las relaciones entre una entidad y sus componentes determinan la navegación estructural.

2. Identificar dentro de los objetos del mundo aquellos que tienen estructura y conexiones similares, es decir, los que son del mismo tipo.

3. Determinar el conjunto de colecciones de objetos y sus miembros. En este paso se diseñan los enlaces de aplicación derivados de las colecciones.

4. Se determina si el tipo de navegación de cada colección es mediante un índice o una visita guiada.

HDM propuso las estructuras de acceso, que fueron usados por los métodos posteriores. También distinguió claramente la composición de los nodos hipermedia, y la necesidad de diferentes vistas para una misma entidad. Las desventajas de HDM son que carece de una notación clara para representar las estructuras de acceso, debido a que todo son óvalos; además, su modelo de datos no es estándar, pues no es relacional ni orientado a objetos.

2.3. Relationship Management Methodology (RMM)

Según sus autores, la hipermedia es un medio para administrar relaciones entre objetos de información, y de ahí el nombre de la metodología. RMM [ISB95] es una metodología para el desarrollo de aplicaciones de hipermedia que tienen una estructura regular definida mediante entidades y relaciones entre éstas. Ejemplos típicos de información estructurada a los que se puede aplicar RMM son catálogos y bases de datos. Los modelos construidos con las primitivas de RMM son llamados Modelo de Datos de Administración de Relaciones (*Relationship Management Data Model*, RMDM).

La navegación es modelada a través de enlaces unidireccionales, enlaces bidireccionales, y estructuras de acceso; éstas pueden ser índices, visitas guiadas, índice-visita-guiada y grupos (ver figura 2.3). A continuación se describen brevemente cada una de ellas.

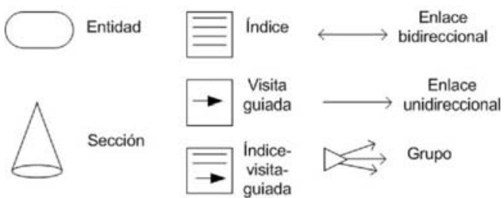


Figura 2.3: Notación de RMM

- Enlaces: conectan una entidad con sus secciones, o las secciones de una misma entidad entre sí. Las conexiones pueden ser bidireccionales o unidireccionales, y se denominan enlaces estructurales. Por ejemplo, el enlace entre la información general del *autor* y su *biografía* en la figura 2.4 es estructural y bidireccional. Cuando conectan entidades o estructuras de acceso entre sí, los enlaces son unidireccionales, y se denominan, sólo, enlaces.

- Grupo: Es una construcción similar a un menú; está formado por una lista de enlaces.

- Índice: es un grupo que contiene una lista de enlaces hacia instancias de entidades a las que provee un acceso directo.

- Visita guiada: es un camino lineal a través de una colección de instancias de entidades, permitiendo el movimiento a la instancia anterior y a la siguiente. Hay algunas variantes de visitas guiadas: en primer lugar, la *visita guiada circular*, en donde el último elemento tiene como siguiente al primero; en segundo lugar la *visita guiada con retorno* al nodo principal, en el que hay una descripción de la visita guiada, y es el nodo inicial y final de la misma; y por último, una variante con diversos nodos de entrada, es decir, una combinación de índice y visita guiada llamada *índice-visita guiada*.

Además, es posible cualificar índices y visitas guiadas con predicados lógicos para determinar el conjunto de instancias que son accesibles desde ellos.

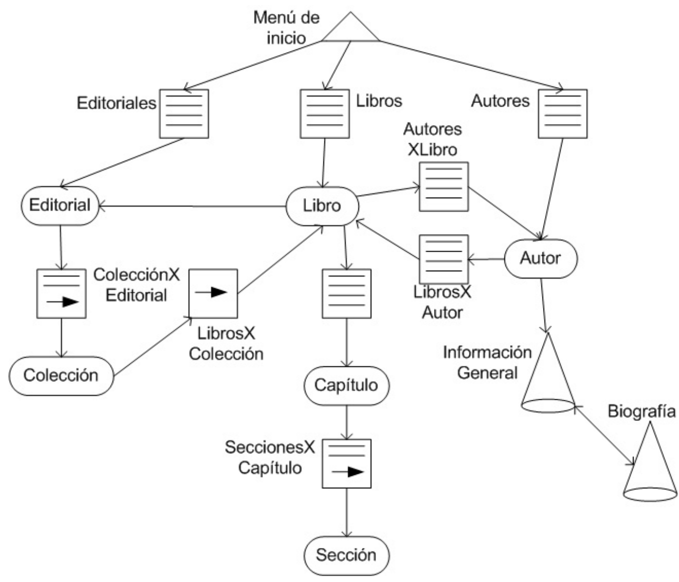


Figura 2.4: Ejemplo en RMM

El modelo RMDM de la figura 2.4 muestra que una *editorial* tiene acceso a sus *colecciones de libros* a través de un índice-visita-guiada, y cada

colección se puede recorrer a través de otra visita guiada. Un *libro* está relacionado con un índice de *capítulos*, y éste con sus *secciones*. El nodo de *autores* se alcanza desde un *libro* a través de un índice, y desde los *libros* también se puede navegar a los *autores* a través de un índice. Los *autores* son un nodo con dos secciones, una con los datos generales del *autor* y otra con la *biografía* extendida del autor. Los enlaces que conectan a la entidad *autor* y sus secciones son de tipo estructural. Cuando se visita la entidad *autor*, se muestra la sección de información general, porque es la sección de cabecera.

El proceso de desarrollo de RMM consiste en siete etapas:

1. Diseño Entidad-Relación (E-R).
   Consiste en representar el dominio de la aplicación a través de un diagrama E-R [Che85]. Los elementos del modelo son las primitivas del dominio (*domain primitives*): entidades, atributos y relaciones. Las entidades y sus atributos representan objetos abstractos o físicos y sus propiedades. Las relaciones describen las asociaciones entre los diferentes tipos de entidades. Las asociaciones pueden ser uno a uno o uno a muchos. Las relaciones muchos a muchos se factorizan en dos relaciones uno a muchos, para facilitar su conversión a estructuras de acceso en el paso 3.

2. Diseño de secciones (*Slice Design*).
   Este paso consiste en dividir una entidad en secciones significativas (*slices*) y organizarlas dentro de una red de hipertexto.

3. Diseño navegacional.
   En esta etapa se diseñan los caminos que existirán en el grafo del documento hipermedia. Cada relación en el diagrama E-R se analiza para decidir qué tipo de estructura de acceso debe reemplazarla. Las relaciones 1-1 son hiperenlaces. Las estructuras de acceso que pueden reemplazar a una relación 1-N son índices o visitas guiadas. Se aconseja utilizar una visita guiada cuando el número de instancias es menor a diez, y cuando no hay una clave única para diseñar un índice. Cuando

el conjunto de instancias es grande y existe una clave discriminadora se recomienda utilizar un índice en lugar de una visita guiada. Después de que se han reemplazado todas las relaciones, se agrupan las entidades que forman un conjunto de interés a través de un grupo (menú). Este último paso sirve para construir los caminos de acceso desde un nodo origen.

4. Diseño del protocolo de conversión.
   Se establecen un conjunto de reglas de conversión de los elementos de RMM hacia la plataforma tecnológica de implementación. Esta etapa es realizada manualmente por los programadores.

5. Diseño de la interfaz de usuario.
   Consiste en el diseño de la distribución de los elementos en las interfaces, es decir, dónde se localizaran las etiquetas, atributos y enlaces, y qué apariencia deben tener cada uno de ellos.

6. Comportamiento en tiempo de ejecución.
   Trata sobre el comportamiento de los nodos hipermedia, y si presentarán un historial de la navegación, un enlace hacia la página anterior, si se deben memorizar las páginas visitadas (caché), o si los nodos se deben computar cada vez, etc.

7. Construcción y pruebas.
   Se siguen las técnicas establecidas en la ingeniería de software tradicional. En el caso de aplicaciones hipermedia se debe tener especial cuidado en las pruebas de los caminos navegacionales.

RMM fue una mejora respecto a HDM, porque presenta una notación más completa y utiliza el modelo de datos relacional, y no uno específico para hipermedia.

2.4. Object Oriented Hypermedia Design Method (OOHDM)

El método de desarrollo de hipermedia *Object Oriented Hypermedia Design Method* (OOHDM) [SR95b], [SRJ96], [SR95a], [SR98] introduce el modelado orientado a objetos en el desarrollo de hipermedia. En OOHDM se modela la navegación a través del diagrama de clases navegacionales y del diagrama de contextos.

El diagrama de clases navegacionales es una vista del modelo estructural. *Cada conjunto de usuarios tiene una en particular*, y sus elementos son clases y asociaciones. Las clases reflejan la estructura de un documento de hipermedia, y son definidas como una vista de las clases conceptuales y se construyen a partir de lenguajes de consulta como los especificados en [Kim95] y [Be03]. Las asociaciones del diagrama de clases navegacionales pueden corresponder a enlaces (*anchor*) o estructuras de acceso que son incluidas como atributos de las clases.

En la figura 2.5 se muestra el diagrama de clases navegacionales correspondiente al ejemplo de la librería. La clase *editorial* contiene un índice de sus colecciones (procede de la asociación 1-N entre *editorial* y *colección*). El nodo de *colecciones* contiene una visita guiada con los *libros de la colección.* Cuando se muestra un *libro*, éste contiene un índice con sus *autores*, tal índice es un atributo de la clase navegacional (en términos de OOHDM); y la lista de *capítulos* se muestra en otro nodo, lo que se indica como *anchor* (hiperenlace). Desde la lista de *capítulos* se puede recorrer las *secciones* del *libro* a través de una visita guiada.

Complementario al diagrama de clases navegacionales, se tiene el diagrama de contextos. Un contexto navegacional es una colección de *objetos navegacionales* que satisfacen una condición, es decir, se puede definir como una consulta. Un contexto permite recorrer el conjunto de nodos hacia adelante y hacia atrás del mismo modo que una visita guiada. Los contextos se definen por clase navegacional, y en algunos casos se puede cambiar de contexto dentro de una misma clase. Por ejemplo, al recorrer un contexto de *libros* en orden alfabético, cada nodo muestra un *libro*, que pertenece a algu-

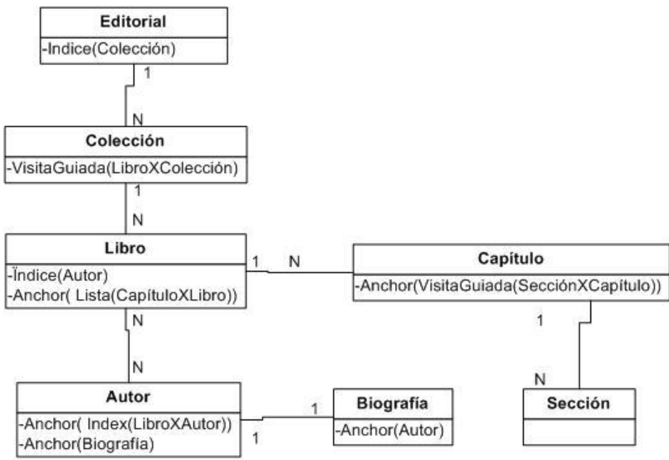


Figura 2.5: Diagrama de clases navegacionales en OOHDM

na colección bibliográfica definida como contexto, entonces se puede visitar el siguiente *libro* alfabéticamente o el siguiente en la *colección bibliográfica*, lo cual sería un cambio de contexto. Algunos contextos se derivan de las asociaciones entre clases navegacionales (por ejemplo, *libros por autor*); en cambio, otros son sólo colecciones de objetos navegacionales (por ejemplo, los libros con cierta palabra en el título). Además, en el diagrama de contextos se especifica el nodo raíz de la aplicación, que es el punto de acceso a los contextos disponibles.

En la figura 2.6 se muestra el diagrama de contextos y las estructuras de acceso. Los contextos son los rectángulos con línea continua, las estructuras de acceso las de línea discontinua, y las flechas los enlaces entre ellos. Las estructuras de acceso son:

- Índice, una colección dinámica de enlaces, en la figura 2.6 *Editorial* es un índice.
- Visitas guiada, un camino lineal a través de una colección de objetos, todos los contextos de la figura 2.6 son visitas guiadas.
- Menú, una colección estática de enlaces, en la figura 2.6 hay un menú principal.

Desde el menú principal se puede navegar a dos formularios para búsqueda de libros y autores o a un índice de editoriales. Ellos llevan la navegación a tres contextos, a partir de los cuales se puede navegar a los demás contextos que son derivados de las relaciones de las clases navegacionales.

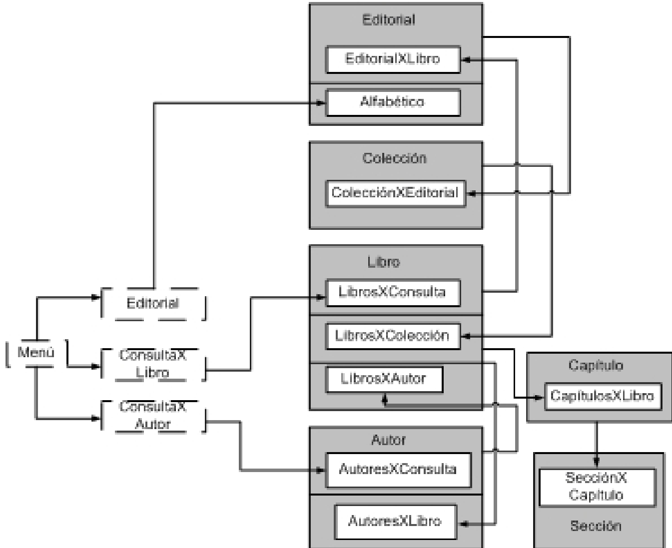


Figura 2.6: Diagrama de contextos en OOHDM

El proceso de desarrollo de OOHDM consta de las siguientes actividades:

1. Análisis conceptual.
   Se construye un modelo del dominio de la aplicación, a través de las técnicas del modelado orientado a objetos. Se identifican las clases y sus relaciones, que pueden ser de asociación, agregación, composición, y generalización-especialización. El resultado de esta etapa es un modelo estructural compuesto por clases, asociaciones y atributos, y es similar al diagrama de clases del *Unified Modelling Language* (UML) [Gro04].

2. Diseño navegacional.
   Sirve para reorganizar la información del modelo estructural y determinar la forma en la que será mostrada a los usuarios. El modelo

navegacional está integrado por el *diagrama de clases navegacionales* y *el diagrama de contextos*.

3. Diseño de la interfaz abstracta.
   En esta etapa se define la forma en la que serán percibidos los objetos a través de la interfaz de usuario y también la apariencia que tendrán. La separación de diseño navegacional y de la interfaz de usuario permite dividir las tareas del desarrollo, así como tener diferentes interfaces para un mismo modelo navegacional. En OOHDM se utilizan vistas abstractas de datos (*abstract data views*, ADV) [CIdLS93]. Mediante un ADV se representa la estructura estática de la interfaz, la composición de objetos y los eventos a los que responden.

4. Implementación.
   Es la última etapa, en la que, a partir de los modelos diseñados, se deben escoger las correspondencias con los objetos concretos de la plataforma de implementación. Es por lo tanto, una etapa totalmente dependiente de la plataforma de implementación escogida.

2.4.1. Modelado de procesos con OOHDM

En [RSL03] se presenta una extensión de OOHDM que incluye el modelado de procesos. Esta consiste en utilizar estereotipos para diferenciar en el diagrama de clases navegacionales entre clases de datos y actividades del proceso. El proceso se modela con un diagrama de actividad de UML, en el que cada actividad corresponde a una clase navegacional con el estereotipo *act node*. Las clases navegacionales de datos tienen el estereotipo *entity node*. Desde un nodo *act node* se puede navegar hacia otros nodos con el mismo estereotipo, lo cual tiene como efecto el avance del proceso. Si desde una actividad se navega hacia un nodo *entity*, entonces se tiene que definir el efecto de esta navegación respecto al proceso; hay tres posibles: suspender (*Suspend*), abortar (*Abort*), y terminar (*Terminate*). En cambio, la navegación desde un nodo *entity* a un nodo *activity* se puede etiquetar como iniciar (*Start*) o continuar (*Resume*).

Un enlace *Start* se muestra en un nodo del modelo navegacional que puede iniciar un proceso. Un enlace *Suspend* o *Abort* se muestra en un nodo actividad que tiene un enlace a un nodo del modelo navegacional. Un enlace *Abort* reinicia el proceso, de modo que los enlaces *Start* deben mostrarse otra vez en los nodos del modelo estructural. Un enlace *Suspend* permite navegar a los nodos del modelo navegacional, y regresar al estado alcanzado en el proceso mediante enlaces *Resume*. El enlace de tipo *Terminate* finaliza la ejecución del proceso.

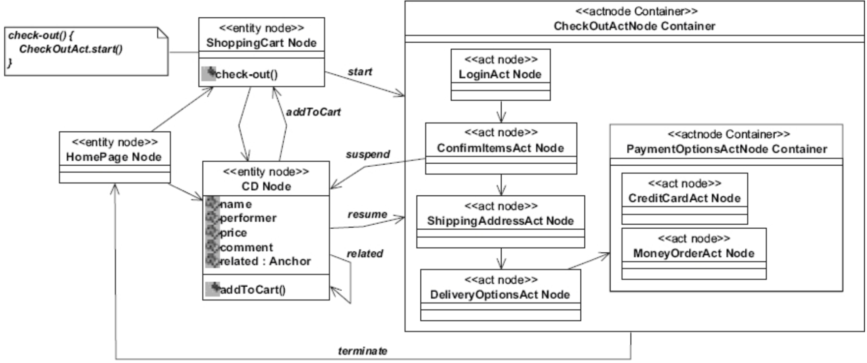


Figura 2.7: Modelo de proceso OOHDM (extraído de [RSL03])

El control de qué enlaces deben mostrarse en los nodos del modelo navegacional y en los nodos de actividades se realiza mediante un contenedor llamado *contexto de proceso*. Un proceso define un contexto navegacional y los nodos del modelo estructural usados en el proceso pertenecen al mismo contexto que los nodos actividad. Los nodos actividad se pueden agrupar en contextos que representan un subproceso. En la figura 2.7 se muestra un ejemplo de un proceso definido con las primitivas de OOHDM. El ejemplo trata de un comercio electrónico, los nodos *entity* son los productos y el carro de compras; y los nodos *activity* los pasos necesarios para efectuar el cierre de la compra registrarse, confirmar e indicar el tipo de pago.

Las principales contribuciones de OOHDM son el uso de la orientación a objetos para modelar del diagrama navegacional, y el diagrama de contextos, que permite cambiar de colecciones de objetos y enriquece la navegación. Encontramos deficiencias en su notación como el solapamiento entre estruc-

turas de acceso y asociaciones en el diagrama de clases, y una representación poco clara de los tipos disponibles de estructuras de acceso. Respecto al modelado de procesos, sólo considera procesos monousuario, sin actividades automáticas, y que sólo puede tener ramificaciones condicionales. Además, su punto de partida es el modelo de datos, y por tanto el modelado de proceso viene como un añadido. Las entidades de proceso son incorporadas al nivel de modelado navegacional; nótese sin embargo, que en el ejemplo, la navegación previa (para seleccionar *cds o libros*) no es considerada como parte del proceso.

2.5. Web Modeling Language (WebML)

A diferencia de los otros métodos, WebML [CFB00] [BCFM00], [BCC+03] no hace énfasis en ser un método de desarrollo de hipermedia, sino un método para diseñar sitios *web*. WebML utiliza XML [SMMP+06] para representar los modelos generados en cada etapa del desarrollo, lo que permite que sean fácilmente consumidos por generadores de código fuente que usan transformaciones XSLT [Cla99]. WebML cuenta con una herramienta de desarrollo llamada WebRatio que permite construir sus modelos y generar el código de la aplicación hipermedia mediante transformaciones XSLT.

El proceso de desarrollo que propone se compone de la especificación de requisitos, el diseño del modelo de datos E-R, diseño del hipertexto, diseño de la arquitectura e implementación, cerrando el ciclo con una etapa de evaluación y pruebas. Como el proceso es iterativo, cada una de las etapas del proceso puede retroalimentar a las anteriores. La última etapa es la instalación del sistema, que pasa de estar en un estado de desarrollo a uno de producción.

Las etapas de desarrollo de WebML se centran en la construcción de 4 modelos: estructural, de hipertexto, de presentación y de personalización.

1. El *modelo estructural* o de datos representa las entidades y atributos relevantes en el dominio del problema. La notación que sigue WebML es E-R.

2. El *modelo de hipertexto* es llamado *site view*, que describe las vistas del modelo estructural que estarán publicadas en el sitio *web*. El *site view* se divide en dos modelos, el de navegación y el de composición. El *modelo de composición* define qué páginas estarán contenidas en el sitio *web*, y qué contenido tiene cada página. En el modelo de composición se construyen nodos de hipermedia con enlaces estructurales o internos, que no son convertidos en hiperenlaces. Los elementos del modelo de composición son llamados unidades. El modelo de navegación sirve para especificar los enlaces entre páginas.

3. El *modelo de presentación* en WebML se efectúa a través de la aplicación de hojas de estilo XSL a documentos XML que representan una instancia del modelo navegacional, y el resultado de la transformación es un documento HTML [HJR99].

4. El *modelo de personalización* está conformado por las entidades predefinidas usuario y grupo. Las características de estas entidades se usan para mostrar contenido individualizado.

Las unidades para construir el modelo de composición (ver figura 2.8) son:

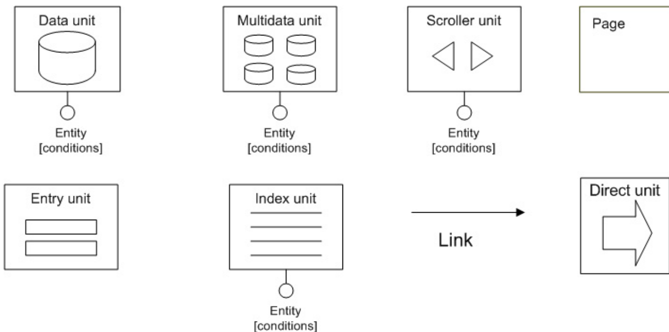


Figura 2.8: Notación de WebML

- La unidad de datos (*data unit)* representa una entidad del modelo estructural y sus atributos visibles. Es decir, una unidad de datos

está definida por una entidad del modelo relacional y los atributos que debe mostrar. La unidad multidatos (*multidata unit)* presenta varias instancias de una entidad simultáneamente, de acuerdo a algún criterio de selección.

- Un índice (*index unit)* representa a un conjunto de referencias a entidades, donde cada referencia presenta una descripción de la entidad a la que apunta.

- Una barra de desplazamiento (*scroller unit)* representa la visualización secuencial de un conjunto de entidades y está asociado a una unidad de datos; es más conocida como visita guiada.

- Una unidad de filtrado (*entry unit)* está asociada a una entidad y contiene una lista de atributos de filtrado, y un predicado condicional. Es un formulario que permite establecer condiciones de selección a índices, barras de desplazamiento, o unidades multidatos.

- Una unidad directa (*direct unit)* representa una relación uno a uno entre dos entidades.

El ejemplo de la *librería* se muestra con la notación de WebML en la figura 2.9. El modelo que se muestra es un *site view* con los modelos de composición y navegación. El modelo de composición está formado por cada nodo individual y por cada página. La página principal muestra un enlace a una página que es un índice de *editoriales*, y contiene también dos formularios de búsqueda. Aunque no se incluye el concepto de secciones como en RMM, en WebML una unidad de datos es una selección de atributos de una entidad de datos del modelo E-R; por ejemplo, la unidad de datos *biografía* se deriva de la entidad *autor* y sólo selecciona el campo *biografía*. El modelo de navegación está integrado por los enlaces que cruzan la frontera de las páginas y que salen de las unidades que se presentan de forma individual.

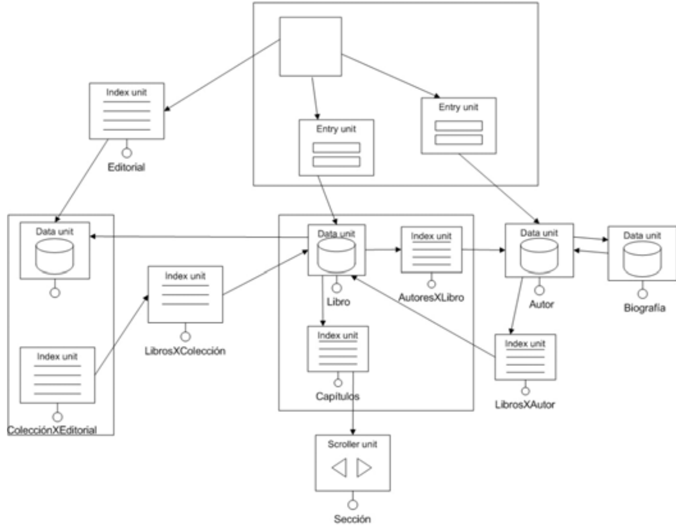


Figura 2.9: Ejemplo en WebML

2.5.1. Modelado de procesos con WebML

WebML ha sido extendido con algunas primitivas de flujo de trabajo [BCC+03]. Las primitivas que incluye son: *start activity, end activity, assign, switch e if*. La representación gráfica de las primitivas se muestra en la figura 2.10.

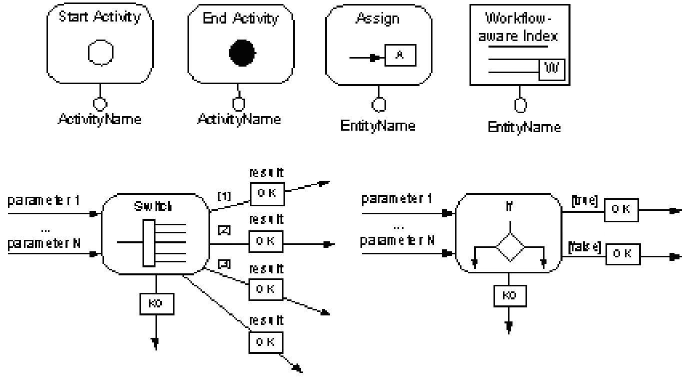


Figura 2.10: Primitivas de proceso de WebML

Start Activity y End Activity delimitan una porción de hipertexto de-

dicada a la ejecución de una actividad. La primitiva *Start Activity* implica la creación de una instancia de la actividad y el registro del momento de creación. En cambio, la primitiva *End Activity* asigna a la actividad el estado de completada y registra el momento en que ocurre ello. La primitiva *Assign* permite asignar un dato generado por una actividad a otra que lo consume. Las primitivas *Start Activity, End Activity y Assign* son de control y no tienen un efecto en la interfaz hipermedia.

Las primitivas *Switch e If* efectúan una navegación condicional basada en el estado del proceso; ambas repercuten en la interfaz hipermedia porque afectan en la decisión del nodo destino de una navegación. Al incorporar las primitivas de proceso, las estructuras navegacionales de WebML presentadas en la sección 2.5 tienen una variante que conoce el estado del proceso, y son llamadas *Workflow Aware Units*; un ejemplo es un índice que utiliza como parámetro de selección una variable del proceso.

Para obtener un modelo de hipermedia que implementa un proceso primero se diseña el flujo de trabajo, después se obtienen el *swimlane* de cada actor, y cada *swimlane* se diseña con las primitivas de WebML para procesos e hipermedia. El procedimiento de diseño del proceso y de obtención de *swimlanes* está fuera del alcance de WebML. Las construcciones de flujo de trabajo como secuencia de actividades y decisiones se pueden implementar fácilmente como secuencias de páginas y con navegaciones condicionales *Switch e If*. Las actividades en paralelo de un flujo de trabajo se tienen que rediseñar como secuencias de actividades. La navegación desde un nodo que representa una actividad a otro del espacio de hipermedia que no es parte del proceso no está definida.

WebML ha sido extendido para usar procesos con la notación *Business Process Modelling Notation* [BCFM06]. El diseñador a partir de las actividades en el modelo BPMN tiene que diseñar una red de hipermedia equivalente con las primitivas que hemos mostrado. El estado del proceso es controlado en las aplicaciones generadas con WebML mediante un metamodelo de flujo de trabajo que es generado como parte de la aplicación.

La principal aportación de WebML fue el uso de XML para representar los modelos y transformarlos. Su modelo conceptual es E-R y esto le impone

algunas limitaciones, que son superadas por modelos más ricos como el OO. En WebML no hay una separación del modelo de proceso y del modelo navegacional. La navegación entre los nodos de actividades y los nodos del modelo navegacional de datos no está permitida. Es decir, se navega a través del proceso o a través del modelo navegacional de datos, pero no puede haber una mezcla de ambos tipos de navegación.

2.6. UML based Web Engineering (UWE)

El objetivo de UWE [HK01] es proveer un método con una notación basada en UML. Para conseguirlo, sus autores desarrollaron un *profile* de UML. Además de la notación utilizan el proceso unificado de desarrollo de software (*Rational Unified Process* o RUP) [BRJ99] como metodología para realizar aplicaciones hipermedia, por lo cual el proceso es iterativo e incremental. El método es muy similar a OOHDM, y la principal diferencia radica en la notación. Sus etapas del ciclo de vida son:

1. Análisis de requisitos.
   El análisis de requisitos se expresa a través de la especificación de los casos de uso del sistema.

2. Diseño conceptual.
   En esta etapa se representa el dominio del problema con un diagrama de clases de UML. Los casos de uso sirven como entrada para elaborar tarjetas *Clase-Responsabilidad-Colaborador* (CRC) [BC89], o para la identificación de verbos y sustantivos, entre otras técnicas, que permiten determinar las clases, métodos y atributos.

3. Diseño navegacional.
   El diseño navegacional tiene dos etapas: la definición del espacio de navegación y el diseño de las estructuras de navegación. El primero es una vista del diagrama conceptual, y la segunda define las estructuras de acceso que permiten visitar los objetos del espacio navegacional.

4. Diseño de la presentación.

   El modelo de presentación en UWE está muy relacionado con los elementos de las interfaces definidas en HTML. Estos elementos también están definidos como estereotipos de UML. Los elementos del modelo de presentación son: anclas, entradas de texto, imágenes, audio y botones. Cada clase del modelo navegacional tiene asignada una clase del modelo de presentación; las clases del modelo de presentación son equivalentes a las ADV de OOHDM.

El espacio de navegación se define mediante el diagrama de clases de UML. Sin embargo, las clases navegacionales son diferenciadas de las clases comunes de UML a través del estereotipo *NC*. Las clases conceptuales que son importantes para el usuario permanecen en el modelo navegacional. Aquellas clases que no se visitan, pero que contienen atributos importantes, no aparecen más en el modelo navegacional, y sus atributos se muestran como parte de otras clases. En el caso de vistas complejas se emplea *Object Query Language* [Be03] para construirlas. Además, para evitar caminos navegacionales largos, algunas asociaciones adicionales son incorporadas al modelo de navegación. Las asociaciones en este diagrama también están etiquetadas con un estereotipo y representan la navegación directa entre clases. Las composiciones en el diagrama de clases navegacionales son interpretadas como la creación de un nodo de hipermedia compuesto, en la que varios nodos se muestran juntos.

La estructura navegacional define las estructuras de acceso que permiten visitar los objetos del espacio navegacional. Las estructuras de acceso son menús, índices, visitas guiadas, y formularios. Todos ellos son clases con estereotipos. La notación de las estructuras de acceso es similar a la de RMM y se muestra en la figura 2.11a.

- Los índices tienen referencias a una colección de objetos, y permiten la navegación directa a ellos.

- Las visitas guiadas contienen una colección de referencias, y permiten la navegación secuencial a través de la misma. Los índices y visitas

guiadas pueden definir la colección de objetos a la que están asociados de forma dinámica mediante el uso de formularios de entrada y condiciones de selección. Por supuesto, los índices y visitas guiadas pueden referirse a colecciones fijas de objetos.

- Un menú es un objeto navegacional que tiene un número fijo de asociaciones a estructuras de acceso u objetos.
- Un formulario permite al usuario ingresar información para completar las condiciones de selección de objetos pertenecientes a las colecciones de índices y visitas guiadas.

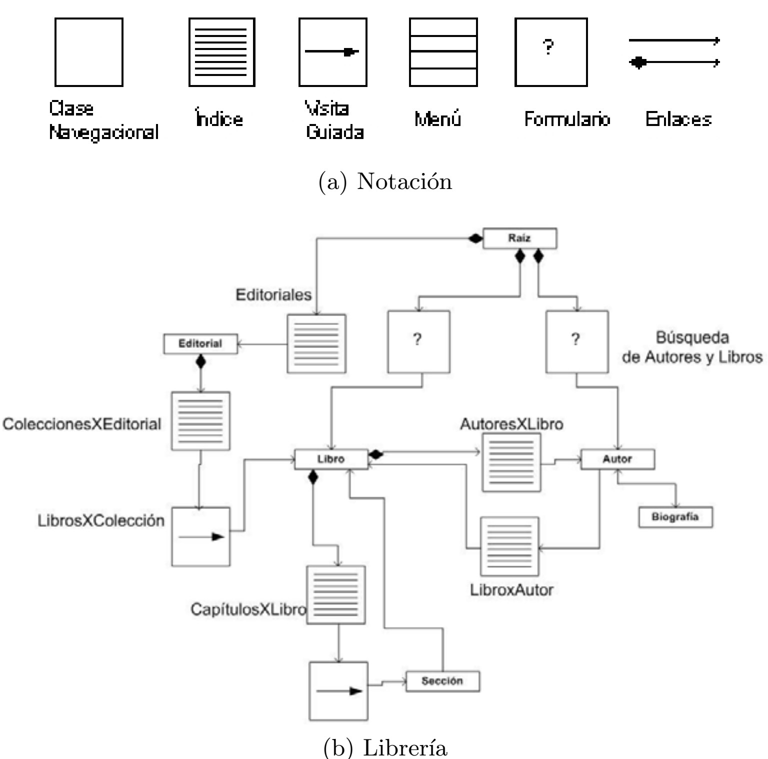
Los elementos de otros modelos, o elementos que aún no existen se pueden incluir a través de estereotipos en UML.

La figura 2.11b muestra el ejemplo de la librería que se ha estado siguiendo en la notación de UWE. Solamente, se destaca que la construcción de nodos complejos se efectúa a través de asociaciones por composición (ver figura). Por ejemplo, la editorial contiene un índice de sus colecciones.

2.6.1. Modelado de procesos con UWE

Cuando se modela un proceso con UWE [KKCM04], los casos de uso son etiquetados como de navegación o de proceso; de los casos de uso de navegación se derivan clases navegacionales, y de los casos de uso de proceso las clases que representan una actividad o subproceso. El principal inconveniente de efectuarlo así es que los casos de uso no son la mejor opción para modelar un flujo de control, puesto que el modelo de casos de uso no ordena los casos.

UWE define dos tipos de enlace, enlaces de proceso y enlaces navegacionales. Un enlace de proceso relaciona una clase navegacional y una clase de proceso. Los enlaces de proceso pueden ser unidireccionales o bidireccionales. Un enlace unidireccional de una clase navegacional a una clase de proceso indica que la navegación pasa a la clase de proceso (en la figura 2.12 el enlace entre *ShoppingCart* y *Checkout*). Los enlaces de proceso bidireccionales conectan una clase de proceso y una clase navegacional e indican que a



(a) Notación

(b) Librería

Figura 2.11: Notación y ejemplo en UWE

partir de la clases navegacional se puede iniciar un proceso y que al término de éste, la navegación continua en la clase navegacional (en la figura 2.12 el enlace entre *product* y *AddToCart*). Un enlace de proceso bidireccional reemplaza a dos enlaces de proceso unidireccionales que tienen a la misma clase navegacional como origen y destino.

A partir de cada actividad y de cada subproceso se deriva una clase de proceso. Un contenedor de actividades o subproceso contiene su estado en un atributo, y el contenedor está relacionado con cada actividad contenida mediante una asociación con cardinalidad 0..1. Conforme el estado del proceso avanza, se crean las instancias de las clases de proceso asociadas. En la figura 2.13 se muestran las clases de proceso del subproceso *CheckOut*.

El estado del proceso es controlado a través del estado de cada clase de proceso. En UWE no hay una separación de actividades manuales y au-

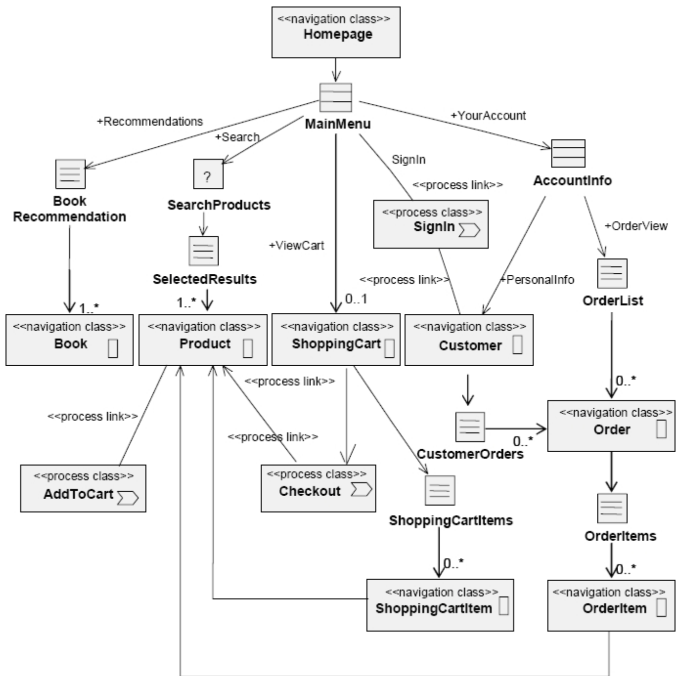


Figura 2.12: Modelo UWE con enlaces de proceso (extraído de [KKCM04])

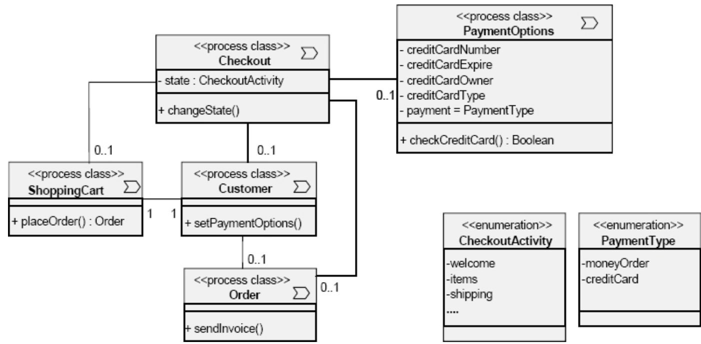


Figura 2.13: Clases de proceso

tomáticas, ni considera procesos multiusuario. Por ejemplo, en la figura 2.12 el enlace de proceso *AddToCart* tiene como destino el nodo *Product*, pero debería ser una actividad automática, lo cual no es señalado en ninguna parte. Además el proceso *AddToCart* y el proceso *CheckOut* son modelados de

forma independiente. Tampoco se explica la resolución del conflicto entre la clase *ShoppingCart* del proceso y la clase *ShoppingCart* del modelo de navegación, ni la gestión de los enlaces de proceso. La mayor deficiencia de UWE es que el modelado de proceso se hace después de diseñar la navegación; por eso, el proceso *AddToCart* y *CheckOut* se tratan de forma independiente, y la navegación para seleccionar productos no es vista como parte del proceso.

2.7. Otros métodos

En esta sección se presenta brevemente otros métodos, cuya información disponible es menor, o bien, han tenido menos influencia en el campo de hipermedia.

2.7.1. Object Oriented Hypermedia Method (OO-H)

OO-H [KKCM04], [GGC03] inicia el análisis del sistema mediante casos de uso; en este sentido, es parecido a SOHDM y a todos los métodos que son posteriores a la aceptación *de facto* de los casos de uso como método inicial de análisis. OO-H reconoce que además de la navegación que procede de la estructura de datos, hay otra derivada de la funcionalidad del sistema, y por ello, clasifica la navegación en estructural y semántica.

El proceso de desarrollo de OO-H se compone de las siguientes actividades:

1. Análisis del sistema. Los requisitos del usuario son plasmados en casos de uso; a partir de ellos, a través de técnicas conocidas de análisis orientado a objetos se deriva el modelo estructural, que se representa como un diagrama de clases UML.

2. Definición de las unidades semánticas. Un grupo de casos de uso que tiene como fin realizar un determinado requisito es llamado unidad semántica de navegación. Para cada grupo de usuarios, se determina en qué unidades semánticas participa; las unidades semánticas a las que puede navegar un usuario se agrupan a través de un menú.

3. Diseño navegacional. La navegación semántica se deriva de los casos de uso en los que participa el usuario, mientras que la navegación estructural se deriva de las relaciones de los datos. La navegación estructural no se refleja en un cambio en la intención de efectuar un cierto caso de uso. El orden de navegación entre las unidades semánticas se define mediante los enlaces semánticos. Este modelo es llamado modelo de análisis de la navegación.

   Después se diseñan las clases navegacionales, que son una vista de las clases del modelo estructural. Un grupo de clases navegacionales usadas para efectuar un requisito es llamado destino navegacional. La relación entre las clases navegacionales dentro de un destino navegacional es a través de enlaces internos (I-link). En cambio, las relaciones entre clases de diferentes destinos navegacionales se relacionan a través de enlaces de travesía (T-link). El nodo inicial de un destino navegacional es indicado mediante un enlace de requisito (R-link). Los servicios que proporcionan las clases navegacionales son efectuados a través de los enlaces de servicio (S-link).

4. Diseño de la presentación. La presentación es diseñada con vistas abstractas de datos, las cuales definen una jerarquía de elementos gráficos que permiten diseñar la presentación de las clases navegacionales.

OO-H puede verse como una extensión a OOHDM con casos de uso y enlaces de servicio. OO-H, a través de casos de uso, y su técnica de encadenar la ejecución de éstos, llega a establecer un modelo parecido al de proceso. Una mejor opción es utilizar directamente un modelo de proceso, que permite modelar la interacción conjunta de los diferentes usuarios y, tiene una notación más expresiva para el flujo de control. Además, las construcciones básicas de los procesos son difíciles de analizar usando sólo casos de uso. En [KKCM04] se presenta cómo se modelan procesos con OO-H, se puede observar que los casos de uso son reemplazados por este tipo de modelo.

2.7.2. Object Oriented Web Solution (OOWS)

OOWS [FVR+03] es una extensión al método OO-Method [Pas92], el cual permite capturar los requisitos funcionales de un sistema orientado a objetos para generar una especificación formal en el lenguaje OASIS [LRSP98]. A partir de la especificación formal, y mediante un compilador de modelos, se genera un prototipo del sistema. OOWS le proporciona a OO-Method los modelos de navegación y presentación.

El proceso de desarrollo de OOWS se divide en dos grandes fases: modelado conceptual y desarrollo de la solución. El modelado conceptual se subdivide en tres etapas:

1. Captura de requisitos funcionales. Éstos se capturan a través de casos de uso y escenarios, que luego son usados para construir el modelo conceptual.

2. Modelado conceptual clásico. A través de modelos estructurales, funcionales y dinámicos, se captura la estructura y comportamiento del sistema.

3. Modelado de la navegación y de la presentación. El modelo navegacional es construido a partir del diagrama de clases, y se plasma en forma de vistas de navegación.

Para empezar la definición de los mapas de navegación, primero se identifican los tipos de usuario y se categorizan. Para cada tipo de usuario se debe especificar su vista de navegación. Una clase navegacional contiene los atributos y operaciones visibles a un usuario. Una vista de navegación contiene las clases navegacionales y los caminos de navegación disponibles para un usuario. Los caminos de navegación se agrupan en subsistemas (menús), y éstos se pueden descomponer en otros subsistemas o en contextos. Un contexto (similar al de OOHDM) es un conjunto de nodos hipermedia. Un contexto está formado por objetos navegacionales. Un contexto puede ser visitado a través de estructuras de acceso (índices y formularios). Un índice permite un acceso directo a un nodo del contexto, mientras que, un formulario permite restringir el espacio de objetos visibles en el contexto. El modelo

de presentación en OOWS trata con algunas cuestiones como la paginación y ordenamiento de la información que pueden ser tratadas en las estructuras de acceso.

Modelado de procesos en OOWS

En [VFP05] se presenta una técnica para la especificación de requisitos basado en la descripción de las tareas de los usuarios. Las tareas de mayor nivel se descomponen en tareas más específicas, mediante una estructura de árbol, y su orden de ejecución se define con restricciones temporales (ver figura 2.14); este tipo de especificación es llamado *Concur Task Trees* [PMM97].

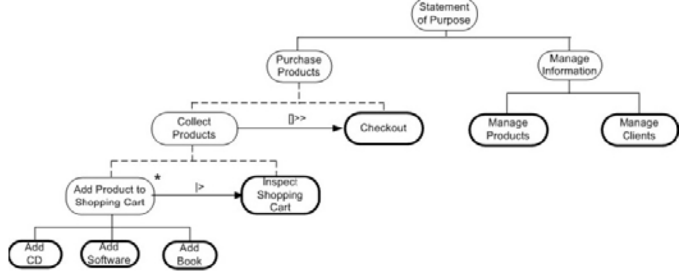


Figura 2.14: Árbol CTT (extraído de [VFP05])

Las restricciones usadas son:

- $T_1| = |T_2$. Las tareas T_1 y T_2 se pueden efectuar en cualquier orden.
- $T_1[]T_2$. Una de las tareas es ejecutada exclusivamente.
- $T_1||T_2$. Las tareas T_1 y T_2 se ejecutan de forma paralela.
- $T_1[> T_2$. La ejecución de T_1 deshabilita la ejecución de T_2.
- $T_1[] >> T_2$. T_2 es habilitado cuando la T_1 es terminada.
- $T_1| > T_2$. T_1 puede ser interrumpido por T_2 y reiniciado hasta que T_2 termina.
- T_1*. T_1 puede ejecutarse n veces.

Cada tarea elemental es detallada con diagramas de actividad de UML (figura 2.15); cada tarea puede ser marcada como tarea de entrada, de salida o función del sistema. De cada actividad de salida se deriva un contexto navegacional, y de cada tarea de función un enlace, localizado en el contexto de la actividad de salida anterior.

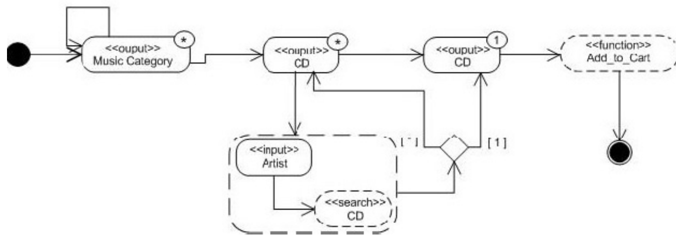


Figura 2.15: Descripción de una tarea

Las ventajas de OOWS radican en el uso de un modelo formal para la verificación de los modelos y la generación automatizada de un prototipo de la aplicación. En cuanto al aspecto de navegación es muy similar a OOHDM y hace uso de los contextos navegacionales de éste. Respecto al modelado de procesos, sería más sencillo describir todo el proceso mediante una sola notación, las restricciones temporales presentadas se pueden definir con notaciones para especificación de procesos como BPMN [Gro06].

2.7.3. Web Site Design Method (WSDM)

WSDM [TL98] tiene como principal característica que es centrado en los usuarios. Las etapas en las que consiste el método son: modelado del usuario, diseño conceptual, diseño de la implementación e implementación. El modelado del usuario lo dividen en dos fases: clasificación de usuarios y descripción de usuarios. El diseño conceptual, también consta de dos fases: modelado orientado a objetos y diseño navegacional.

Los diferentes tipos de usuario encontrados tienen diferentes perspectivas de la información y de usabilidad. La vista que tienen los usuarios del modelo conceptual es llamada Modelo de Objetos del Usuario; este modelo no incluye la totalidad de las clases que modelan una organización, sino sólo

las que son importantes para un tipo determinado de usuario. Las perspectivas, por tipo de usuario, que existen de una clase del modelo conceptual son modeladas mediante subclases; por ejemplo, si existe una clase factura que es vista por compradores y vendedores, entonces habría dos subclases.

El diseño navegacional de la aplicación se efectúa por cada perspectiva encontrada. El nodo raíz de un modelo navegacional está determinado por los tipos de usuarios encontrados, los cuales se agrupan en la capa de contexto. La información que interesa al tipo de usuario se agrupa en componentes que son accesibles desde el nodo raíz. Finalmente, se crean enlaces a los objetos de perspectiva que integran tal componente. Los modelos navegacionales que se diseñan con WSDM son de tipo jerárquico.

Modelado de procesos en WSDM

Debido a su estructura jerárquica de datos, el modelado de procesos en WSDM [TC03] se efectúa a través de *Concur Task Trees*(CTT) [PMM97] como en OOWS (figura 2.14). El árbol tiene una tarea inicial que puede descomponerse en otras de menor granularidad, y mediante restricciones temporales se indica la precedencia de las tareas y el número de veces que se puede ejecutar. Las restricciones que se pueden modelar son las mismas a las mostradas en la sección de OOWS.

En [TC03] se puede observar cómo el proceso de compra electrónica se modela fácilmente con CTT; sin embargo, las correspondencias entre los árboles CTT y los modelos de navegación, no se proporcionan. La principal limitación del método es que sólo es apropiado para modelar estructuras hipermediales jerárquicas.

2.7.4. Ariadne Development Method (ADM)

ADM [DMA05] tiene como principales características el modelado de contenido multimedia, de requisitos funcionales, de usuarios y de políticas de seguridad. Estos requisitos son remarcados en Ariadne porque considera que los métodos anteriores no les tienen en cuenta suficientemente.

El proceso de desarrollo en Ariadne consta de tres fases: diseño conceptual, diseño detallado y evaluación. Las fases no se efectúan de forma secuencial, sino que depende del tipo de proceso software que se siga.

1. Diseño conceptual. Está enfocado en la identificación de tipos abstractos de datos de componentes, funciones y relaciones.

2. Diseño detallado. Trata de la especificación de características y comportamiento del sistema, a un nivel de detalle tal que se pueda generar de forma semi-automática.

3. Evaluación. Se refiere a la evaluación de la usabilidad del sistema y de los prototipos, para mejorar el diseño conceptual o detallado.

En este método se considera que las aplicaciones hipermedia tienen seis vistas fundamentales:

- Navegación. La vista de navegación se refiere a la estructura de nodos y enlaces hipermedia, que definen los caminos navegacionales y estructuras de acceso (un tipo de nodo).

- Presentación. Se refiere al diseño de las plantillas que se aplican a un tipo de nodo para obtener su presentación final; las plantillas deben considerar criterios de usabilidad.

- Estructura. Los nodos hipermedia se construyen a partir de entidades de datos y relaciones, la estructura se refiere al modelo de datos.

- Comportamiento. Describe cómo debe reaccionar el sistema ante la ocurrencia de un evento en la interfaz de usuario.

- Proceso. La vista de proceso está enfocada a la especificación de cómo funciona el sistema.

- Seguridad. Permite definir el contenido visible para cada usuario y cuál es modificable o personalizable.

Fase	Actividad	Producto
Diseño conceptual	Definición de la estructura lógica	Diagrama estructural
	Estudio de la funcionalidad	Diagrama navegacional
		Especificación funcional
	Especificación de entidades	Diagramas internos
		Catálogo de atributos
		Catálogo de eventos
	Modelado del usuario	Diagrama de usuarios
	Definición de las políticas	Tabla de categorías
	de seguridad	Tabla de acceso
Diseño detallado	Identificación de instancias	Diagrama de nodos instanciados
		Diagrama de usuarios instanciados
	Especificación de funciones	Especificación de estructuras
	de acceso	
		Especificación de las funciones
	Especificación de instancias	Diseño detallado de los
		diagramas internos
		Reglas de autorización
		Asignación de usuarios
	Definición de las características	Especificación de la presentación
	de presentación	
Evaluación	Desarrollo del prototipo	Prototipo
	Evaluación	Documento de evaluación
		Reporte de conclusiones

Tabla 2.1: Fases, actividades y productos de Ariadne

Cada fase se descompone en actividades que producen algún artefacto (ver tabla 2.1). El método proporciona al menos un artefacto que permite el análisis de las seis vistas de un sistema hipermedia.

Ariadne permite la definición de contenido multimedia en los nodos a través de restricciones temporales y espaciales, es decir, una sección del contenido de un nodo ocupa un espacio durante un tiempo.

Un nodo puede tener enlaces derivados de los eventos que inician la ejecución de operaciones definidas en los requisitos funcionales, los eventos deben especificar la lista de elementos en los que puede ocurrir y sus condiciones de activación, y se debe mantener un catálogo de eventos.

Otra de las actividades relevantes del diseño conceptual es el modelado del usuario. Se deben identificar los tipos de usuario y sus necesidades de interacción específicas. Las responsabilidades o funciones de los usuarios se agrupan en roles; que a su vez se agrupan en equipos que definen conjuntos de usuarios que colaboran.

La seguridad de los nodos hipermedia se establece mediante una tabla

de acceso. Los permisos de acceso (ver, personalizar y editar) son asignados a roles y nodos.

Las características a resaltar del método son el modelado de contenido multimedia, la seguridad y el modelado del usuario. Es decir, sus aportaciones están dirigidas a mejorar el contenido. Los requisitos funcionales del método se expresan a través de especificaciones y listas de eventos; en cambio, los demás métodos lo hacen a través de casos de uso. La mayor deficiencia del método radica en la especificación de los requisitos funcionales, porque a partir de una especificación de proceso se pueden identificar los grupos de usuario, entidades de datos, y requisitos funcionales (actividades). Además, se observa qué actividades están asignadas a cada grupo.

2.7.5. EORM. Enhanced Object Relationship Methodology

El método de modelado de sistemas hipermedia *Enhanced Object Relationship Methodology* (EORM)[Lan94], al igual que OOHDM, es orientado a objetos, y tiene como objetivo llevar los conceptos de orientación a objetos a los sistemas hipermedia. El primer paso es realizar un modelo conceptual usando OMT [Rum91]. Sin embargo, el paso fundamental de este método es el enriquecimiento de las relaciones; para ello, las relaciones entre los objetos del modelo original son modeladas como objetos. Las relaciones representan los enlaces entre los objetos hipermedia. Los tipos de enlaces son modelados a través de una jerarquía de clases, de modo que, cada clase le da una semántica diferente al enlace.

2.7.6. PML. Process Modeling Language

PML [NS01] un lenguaje de especificación interfaces basados en procesos [NS01], y sus construcciones son: secuencia, selección, ejecución en paralelo, y acciones.

Secuencia. Indican que una tarea se ejecuta después de la finalización de otra.

Selección. Sirven para efectuar decisiones basadas en la salida de la actividad previa y elegir exclusivamente un camino.

Ejecución en paralelo . Indican que las tareas de cada rama se ejecutan de forma paralela. Sin embargo, en PML se ejecuta secuencialmente cada una de las ramas. La siguiente tarea se efectúa hasta que se terminan todas las ramas.

Tarea Una tarea tiene asociado un *script* o un formulario, si la tarea es manual entonces tiene asociado un formulario y si es automática entonces tiene asociado un *script*.

La navegación que se diseña con este lenguaje sólo es aplicable a un usuario; si el proceso tuviera varios actores, entonces se tendría que controlar el estado del proceso a través de las acciones y en la lógica de los *scripts*. La principal carencia de este trabajo es que no es parte de un método de desarrollo de sistemas, no tiene una etapa de modelado conceptual ni de análisis, tampoco explica como integrarlo con otras primitivas de hipermedia. El concepto de proceso que propone es una secuencia de formularios que van ejecutando alguna acción. La máquina virtual de PML controla el despliegue del siguiente formulario y la ejecución de *scripts*. Este trabajo es poco citado en los artículos de hipermedia e ingeniería *web*, aunque aparece un año antes que el artículo de WebML sobre diseño de hipertexto dirigido por flujos de trabajo [BCC+03].

2.7.7. Magrathea

Magrathea [SHG06] parte de modelos BPMN para la ejecución de procesos. Los modelos BPMN que utiliza se componen sólo de actividades y *Xor-gateways* (decisiones y uniones). Un proceso es visto del mismo modo que en PML, como una secuencia de actividades manuales (formularios) y actividades automáticas (*scripts*). Es decir, reemplaza el lenguaje PML, por modelos BPMN y mediante un enfoque MDA (sección 3.3) genera código Ruby que ejecuta el proceso. Magrathea no está tampoco orientado a cual-

quier tipo de proceso si no que propone algunos tipos de procesos que son líneas de producto.

2.7.8. Hera

En Hera [BFH06] se definen procesos a través de diagramas de actividad de UML. El diagrama de actividad es transformado a un sistema de transiciones de estado con etiquetas, las etiquetas son conceptualizadas como eventos. Un flujo de trabajo puede representarse como diagramas de estado o como reglas donde dado un evento entonces se ejecuta una acción (*Event Condition Action*, ECA) [Ell79]. La representación de flujos de trabajo mediante algún tipo de diagrama con estados explícitos o mediante reglas es equivalente [LLP+06]. En Hera los procesos son multi-actor y se define formalmente los grupos de actividades de cada actor (*clusters*). Las actividades en Hera son de captura de información, de mostrado de información, o automáticas de actualización de datos. Hera extiende el concepto de proceso definido en PML a varios actores e incluye un nuevo tipo de actividad que es para el despliegue de información. En su modelo de proceso cada *cluster* no depende de los *clusters* de otros usuarios, lo cual no es frecuente en los flujos de trabajo donde es común encontrar dependencias entre actividades.

2.7.9. Scenario-Based Object-Oriented Hypermedia Design Methodology (SOHDM)

SOHDM [OHK98] propuesto en 1998, tiene un proceso de desarrollo que consta de seis etapas, que se detallan a continuación:

1. Análisis del dominio.
   El análisis inicial del sistema es a través de un modelo de escenarios, para el que SOHDM proporciona una notación basada en diagramas de flujo y eventos. Los escenarios son una combinación de casos de uso [Jac92] y diagramas de flujo de datos [You88].

2. Modelado orientado a objetos.
   La identificación de las clases y sus relaciones se efectúa mediante tar-

jetas CRC, y se representan en un diagrama de clases con la notación de *Object-modeling Technique* (OMT) [Rum91].

3. Diseño de las vistas.
   Una vez que se tiene el modelo orientado a objetos, se proyecta la vista de clases y relaciones que son visibles en el modelo hipermedia, es decir, se eliminan las clases, atributos, y relaciones que no son visibles. Después se crean las vistas que agrupan información de otras clases del modelo orientado a objetos, y son llamadas unidades navegacionales.

4. Diseño navegacional.
   Las unidades navegacionales son clasificadas en simples, si muestran una sola instancia, e índices y visitas guiadas cuando muestran más de una. Se construye un menú hacia los escenarios alternativos que se presentan al usuario. Este paso puede aplicarse por cada grupo de usuario del sistema.

5. Implementación y construcción.
   Se diseñan las interfaces de usuario en HTML, y también se modela el esquema de la base de datos. El método también define una notación para el diseño de la interfaz de usuario que presenta a las unidades de navegación.

SOHDM es una extensión a los métodos que usan el modelado orientado a objetos, a los que proporciona escenarios, para modelar la funcionalidad del sistema. Sin embargo, el concepto de escenario, no corresponde con los escenarios de UML.

2.8. Comparativa de los métodos

El árbol de la figura 2.16 muestra los cambios que han ocurrido en los métodos de desarrollo de hipermedia. En la raíz del árbol se encuentra HDM, que estableció los pasos básicos: análisis conceptual, diseño navegacional y de la presentación, e implementación.

El modelado conceptual se realiza mediante modelos relacionales (RMM, WebML) u orientados a objetos (OOHDM, UWE, OO-H). Cuando el modelado conceptual es orientado a objetos se utiliza la notación de UML, aunque los métodos que surgieron antes que él, utilizaron OMT, ORM u otras notaciones.

El modelo navegacional se empieza a construir con la proyección de las clases o entidades que son visibles. Después reemplazan las relaciones 1-n y m-n por estructuras de acceso. Finalmente, se genera una página inicial con enlaces a estructuras de acceso que permiten la navegación a las entidades o clases del modelo navegacional.

Cada método define su propia notación para representar el modelo navegacional. Respecto a las notaciones se puede encontrar que la de OOHDM está muy consolidada. Sin embargo, OOHDM carece de un metamodelo bien definido que indique las construcciones válidas. De hecho podemos encontrar variantes de su notación gráfica en cada publicación relacionada con ese método. Las notaciones de WebML y UWE son más claras y están mejor documentadas. UWE tiene como ventaja adicional que es un *profile* de UML, es decir, un modelo navegacional es un diagrama de clases de UML con estereotipos.

En el diseño de la presentación casi todos los métodos siguen la técnica propuesta en OOHDM, que es el diseño abstracto de interfaces. La fase de implementación no es abordada por los métodos, y en el mejor de los casos cuentan con alguna herramienta que permite generar código ejecutable en cierto lenguaje.

OOHDM y EORM proporcionaron un modelo conceptual OO y RMM una notación más clara. El método que ha tenido más descendientes es OOHDM: SOHDM le han añadido análisis por escenarios, OO-H casos de uso, UWE casos de uso y notación UML, OOHDMDA una implementación mediante MDA. WSDM surge desde el punto de vista del diseño centrado en el usuario.

Las tablas 2.2 y 2.3 muestran el ciclo de vida y notación de modelado usadas en cada etapa por los métodos vistos; además, se incluye una columna con el nombre de su herramienta, en caso de existir.

Tabla 2.2: Comparativa de los métodos(1)

Método	**Proceso**	**Notación**	**Herramientas**
Ariadne	Diseño conceptual Diseño detallado Evaluación	Diagrama de clases Diagrama navegacional Diagrama de usuarios Especificaciones funcionales Diagramas internos Diagrama de nodos instancia Diagrama de usuarios instanciados Prototipo Reporte de conclusiones	
EORM	Diagrama de clases Enriquecimiento de las relaciones Diseño navegacional	Diagrama de clases Diagrama de clases Diagrama navegacional EORM	Ontos-Studio
MIDAS	Captura de requisitos Modelado conceptual Modelado navegacional Generación de la presentación Generación del modelos de datos	Casos de uso Diagrama de clases Diagrama navegacional UWE ADV-XML SQL-99	MIDAS
OO-H	Análisis con casos de uso Modelado conceptual Análisis y diseño navegacional Diseño de la presentación Implementación	Casos de uso Diagrama de clases Diagrama navegacional OO-H ADV	VisualWade
OOHDM	Modelado conceptual Diseño navegacional Diseño de la presentación Implementación	Diagrama de clases Diagrama de contextos ADV	OOHDM-Web HyperSD
OOHDMDA	Modelado conceptual Diseño navegacional Diseño de la presentación Implementación	Diagrama de clases Diagrama de contextos ADV	OOHDMDA
OOWS	Captura de requisitos Modelado conceptual Diseño navegacional Diseño de la presentación Implementación	Casos de uso Diagrama de clases Diagrama de contextos ADV	Olivanova modeler

Tabla 2.3: Comparativa de los métodos(2)

Método	**Proceso**	**Notación**	**Herramientas**
RMM	Diseño ER Diseño de rebanadas Diseño navegacional Diseño del protocolo de conversión Diseño de la interfaz Diseño del comportamiento en tiempo de ejecución Construcción y pruebas	Diagrama E-R Diagrama RMDM Diagrama RMDM	RMM-Case
SOHDM	Análisis de requisitos Modelado conceptual Diseño de vistas Diseño navegacional Implementación	Diagrama de escenarios Diagrama de clases Diagrama de clases Diagrama navegacional SOHDM	
WSDM	Modelado del usuario Modelado Conceptual Modelado de objetos Diseño navegacional Diseño de la implementación Implementación	Diagrama de clases Diagrama de capas WSDM	
WebML	Análisis de requerimientos Diseño de datos Diseño de hipertexto Diseño de la arquitectura Implementación Pruebas Puesta en producción Mantenimiento	Diagrama E-R *Site view*	WebRatio
UWE	Análisis de requerimientos Diseño conceptual Diseño navegacional Diseño de la presentación Implementación y pruebas	Casos de uso Diagrama de clases Diagrama de clases con esterotipos Clases de presentación (ADV)	Argo-UWE

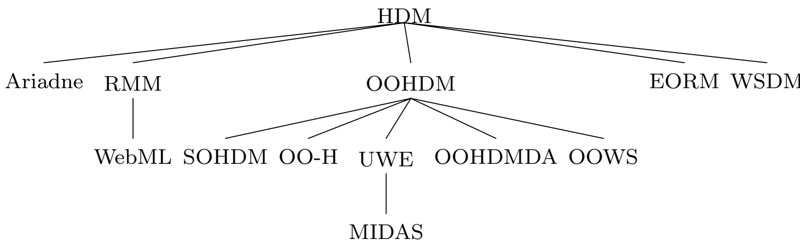


Figura 2.16: Evolución de los métodos

Tabla 2.4: Métodos que modelan procesos de negocios

Método	**Notación**
Hera	Reglas Event-Condition-Action
Magrathea	BPMN
OO-H	Diagramas de actividad
OOHDM	Diagramas de actividad
OOWS	Concur Task Trees
PML	PML
UWE	Diagramas de actividad
WebML	BPMN
WSDM	Concur Task Trees

Los métodos que modelan procesos de negocios y la notación que utilizan para ello se muestra en la tabla 2.4. Algunos de los métodos utilizan para el modelado de procesos el estándar más aceptado, que es BPMN, y otros prefieren los diagramas de actividad de UML, dado que utilizan UML en otras etapas del modelado del sistema. El tipo de procesos que se puede modelar con BPMN y UML es el mismo, porque los métodos no utilizan las primitivas de BPMN que no están incluidas en UML. Sólo Hera modela los flujos de trabajo con reglas ECA, lo que le permitiría modelar procesos muy dinámicos; sin embargo, habría que estudiar su impacto en la navegación.

El lenguaje PML permite la construcción de procesos estructurados. En cambio, los que utilizan la notación gráfica de BPMN podrían construir procesos no-estructurados. Otra diferencia importante es el modelado de la sincronización entre las actividades de diferentes actores; sólo WebML y Hera, permiten el diseño de procesos multi-actor.

2.9. Conclusiones

En este capítulo se han presentado los principales métodos de desarrollo de hipermedia y una comparativa de ellos. Los métodos basados en casos de uso construyen un *modelo de proceso* a partir de los mismos, el cual hace falta validar. El modelo navegacional derivado del proceso es un añadido al modelo de navegación derivado de los datos. Los métodos están enfocados a generar un espacio de navegación obtenido del modelo de conceptual de datos, y otro del proceso, que es visto como una secuencia de formularios, sin embargo, un proceso es más complejo que ello. La navegación sobre los datos debe ser vista como parte integral del proceso, si dicha navegación tiene un fin además de la búsqueda en sí misma. Es decir, la navegación puede conceptualizarse en dos capas y ambas suceden en paralelo, una en el espacio del proceso y otra en el espacio de datos. De modo que las propiedades de hipermedia pueden conservarse y la navegación en el proceso no sólo es una sucesión de formularios.

A partir de la comparativa se puede observar que es necesario un método que:

- Tenga un ciclo de vida que considere como modelo principal al modelo de proceso.
- Permita obtener el modelo navegacional, para cada usuario, desde el modelo de proceso. Tal modelo se complementa con la navegación derivada del modelo estructural.
- Que la navegación en el espacio del proceso y de datos sean complementarias, pero no excluyentes.
- Defina formalmente los modelos y sus transformaciones, apoyándose en estándares.
- Permita generar prototipos desde los modelos.

Capítulo 3

Ingeniería dirigida por modelos

3.1. Introducción

En el pasado se han creado abstracciones que facilitan el desarrollo y especificación de sistemas, entre las que se incluyen los primeros lenguajes de programación y posteriormente las herramientas para la ingeniería de software asistida por computador (CASE). Las herramientas CASE están orientadas a la solución de un problema dentro de un entorno tecnológico específico, en lugar de especificar la solución del problema en términos de un dominio de aplicación. La Ingeniería Dirigida por Modelos (*Model Driven Engineering*, MDE) [Ken02] es un esfuerzo por elevar más el nivel de abstracción durante el desarrollo del software. MDE tiene como pilares el uso de lenguajes específicos de dominio o metamodelos y la transformación de modelos.

Esta tesis utiliza los conceptos propuestos en MDE como base procedimental. En este capítulo se introducen MDE (sección 3.2) y la Arquitectura Dirigida por Modelos (*Model Driven Architecture*, MDA) [MM01] (sección 3.3), una realización de MDE. En la sección 3.2.1 se definen los conceptos de modelo, metamodelo, transformación de modelos, y de plataforma de transformación de acuerdo a la ontología usada en MDE. Los conceptos de

MDA se explican en la sección 3.3.1 y la torre reflexiva Meta-Object Facility (MOF) en la sección 3.3.2. Después en la sección 3.4 se explican algunos espacios tecnológicos dentro de MDA y en 3.5 se evalúan, lo cual sirve para decidir cuál usar durante el desarrollo de este trabajo.

3.2. Ingeniería Dirigida por Modelos (MDE)

En MDE el desarrollo de software se centra en la especificación de modelos y transformaciones y en la reutilización de éstos. La premisa principal de MDE es que los programas se generarán de forma automatizada a partir de sus correspondientes modelos [Sel03]. Su objetivo es superar lo que hacen las herramientas CASE, que sólo pueden generar esqueletos de programas, y que tienen poca flexibilidad debido a que no puede ser cambiadas por el usuario. En MDE incluso la semántica de ejecución de los sistemas tendría que especificarse como modelo.

El desarrollo de software en MDE está basado en la *refinación de modelos* a través de transformaciones, es decir, transformar modelos con un mayor nivel de abstracción en otros con menor nivel de abstracción, que permiten efectuar de forma más sencilla la generación de código. A continuación se presenta un conjunto de definiciones, necesarias para entender MDE/MDA.

3.2.1. Modelos, metamodelos y sus relaciones

La representación mediante modelos de los artefactos software tiene por objetivo reducir la complejidad y los problemas asociados al desarrollo de software. La guía de MDA [MM01] define modelo como “la descripción o especificación de un sistema y su ambiente para cierto propósito, y frecuentemente representado como una combinación de dibujos y texto”. El texto puede ser en un lenguaje de modelado o en lenguaje natural. En [KWB03] se define un modelo como “una descripción de un sistema, escrita en un lenguaje bien definido” y un lenguaje de modelado como “el lenguaje usado para expresar un modelo”; también se indica que el modelado es “la actividad de definir un modelo a través de un lenguaje de modelado”.

Respecto al término metamodelo, también se pueden encontrar varias definiciones. En [Sei03] "el modelo de un lenguaje de modelado es un metamodelo". Para [MM01], un metamodelo es "un modelo de modelos". Un metamodelo está compuesto por conceptos y las relaciones entre éstos; en tal sentido, un metamodelo es una ontología para la construcción de modelos. Cuando los conceptos de un metamodelo pueden describirse con el mismo lenguaje de modelado que define ese metamodelo se dice que el metamodelo es *reflexivo*.

Cuando un modelo es construido según un metamodelo bien definido, se dice que el modelo es *conforme* al metamodelo. También, es común decir que tal *modelo es una instancia del metamodelo*, sin embargo, este caso indica que el metamodelo se representa con metaclases, y que un modelo está compuesto por objetos. Por ejemplo, si tenemos un documento XML, se puede validar respecto a un esquema XML y saber si satisface la relación *conforme a*. En este caso hay una clara relación entre el documento XML y el lenguaje de modelado determinado por el esquema XML. En cambio, si tenemos un modelo y un sistema que lleva a la realidad tal modelo, se dice que el sistema es una *realización* del modelo. En este caso el sistema no es una instancia del modelo UML [Sei03], [BJT05]. La relación inversa a la realización es la *representación* (un modelo representa un sistema).

El metamodelado puede aplicarse de forma sucesiva; el resultado de esto es una jerarquía de modelos que se conoce como *arquitectura de metamodelado* o pila de metamodelos. Cada nivel de la pila es un nivel de modelado que permite construir los modelos del nivel inferior.

En la figura 3.1 se muestran dos arquitecturas de metamodelado, una basada en UML y otra en esquemas XML. En la primera en L3 se encuentra el metamodelo de UML que se define así mismo en L2, y en L1 se encuentran los modelos de UML que son instancias del metamodelo UML de L2. En la arquitectura de metamodelado de XML, en L3 se encuentra el meta-esquema XML, y en L2 los esquemas XML que son instancias del meta-esquema XML; en L1 están los documentos XML que son instancias de los esquemas XML de L2. El nivel inferior L0 representa la realidad, en este caso se encuentran los sistemas que realizan a los modelos UML.

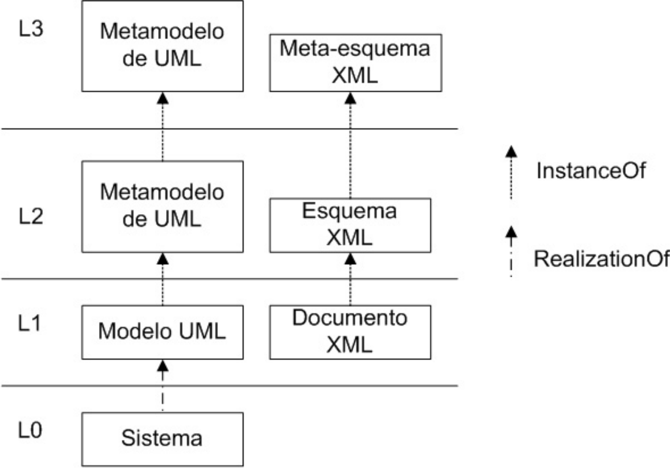


Figura 3.1: Relaciones InstanceOf y RealizationOf

Cualquier arquitectura de metamodelado presenta el problema de cerrar el nivel superior de la pila, de tal forma que no se necesite un metamodelo de nivel superior siempre. Para cerrar el nivel superior de una arquitectura de metamodelado se requiere de un metamodelo reflexivo en el nivel superior de la torre [Sei03].

3.2.2. Transformación de modelos

El proceso automatizado de utilizar varios modelos como entrada y producir uno o más modelos como salida, siguiendo un conjunto de reglas de transformación, se conoce como *transformación de modelos* [SK03]. En [KWB03] se define una transformación como la generación automática de un modelo destino a partir de un modelo origen, de acuerdo a una especificación. En [MM01] se define la transformación de modelos como el proceso de convertir un modelo en otro del mismo sistema. La transformación de modelos según [SK03] puede efectuarse de tres formas: mediante la manipulación directa del modelo a través de una *Application Programming Interface* (API), a través de una representación intermedia que permita a otras herramientas manipular el modelo, o mediante un lenguaje de transformación de modelos, que tiene construcciones específicas para ello. En un proceso MDE

se prefiere la tercera opción.

Las transformaciones de modelos pueden especificarse usando muchos lenguajes, inclusive los de propósito general. Sin embargo, la tendencia en MDE es usar aquellos dedicados a la transformación de modelos. La *interoperación de transformaciones de modelos* es la posibilidad de usar una transformación en distintas plataformas, lo cual es factible, si se usan estándares como QVT, o estándares *de facto* como *Atlas Transformation Language* (ATL) [JAB+06], [JK06].

Los elementos de una transformación de modelos se muestran en la figura 3.2.

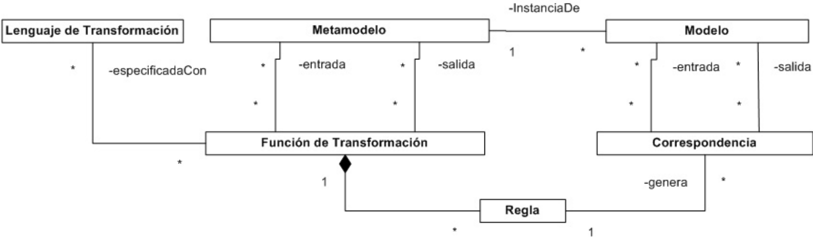


Figura 3.2: Elementos de una transformación

El proceso de transformación de modelos se efectúa a través de una *función de correspondencia o transformación*, compuesta por un conjunto de *reglas de transformación* [KWB03]. La entrada de la función de transformación es un conjunto de modelos, cuyos metamodelos corresponden al dominio de la función. La salida es un conjunto de modelos que pertenecen a un conjunto de metamodelos de salida o codominio.

La función de transformación define las *correspondencias abstractas* entre elementos de los metamodelos de entrada y los de salida. En [KWB03] se define una *correspondencia* entre modelos como la ejecución de una función de transformación, es decir, la relación entre los elementos concretos de un modelo de entrada y otro de salida no se conoce hasta que se ejecuta la transformación. Por ejemplo, una función de transformación (compuesta por una sola regla) podría ser *todo X se transforma en un Y*, (correspondencia abstracta); en cambio, si la entrada es un modelo $M_1 = \{X_1, X_2\}$, al aplicar

la transformación se obtiene $M_2 = \{Y_1, Y_2\}$, y se establecen las siguientes correspondencias (X_1, Y_1) y (X_2, Y_2).

Las *transformaciones horizontales* en MDE ocurren si el modelo origen y el modelo destino pertenecen a la misma categoría de acuerdo a cierta clasificación de modelos, en caso contrario, es *vertical* [Ken02].

Las transformaciones de modelos se ejecutan en una *plataforma de transformación*, que se compone de uno o más metamodelos para definir los metamodelos de entrada y salida de las funciones de transformación, un lenguaje de transformación y un entorno tecnológico de ejecución. Un término aplicado a esta combinación de elementos es el de *espacio tecnológico* [KBJV06].

3.3. Arquitectura Dirigida por Modelos

MDA es un marco de trabajo cuyo objetivo central es resolver el problema del cambio de tecnología e integración en el desarrollo de sistemas software. La idea principal de MDA es usar modelos, de modo que las propiedades y características de los sistemas queden plasmadas de forma abstracta, y por tanto, los modelos no serían afectados por tales cambios.

3.3.1. Conceptos básicos de MDA

MDA define dos clases de modelos: independientes de la plataforma (*Platform Independent Model*, PIM) y específicos de la plataforma (*Platform Specific Model*, PSM). Las definiciones de la guía de MDA [MM01] son las siguientes:

Plataforma. *Una plataforma es un conjunto de subsistemas y tecnologías que provee un conjunto coherente de funcionalidades a través de interfaces y unos patrones específicos de uso, los cuales pueden ser empleados por cualquier aplicación sin que ésta tenga conocimiento de los detalles de cómo la funcionalidad es implementada.*

PIM. *Un modelo independiente de la plataforma es una vista de un sistema desde un punto de vista independiente de la tecnología.*

PSM. *Un modelo específico de la plataforma es una vista de un sistema que depende de la tecnología donde se ejecutará el sistema.*

La figura 3.3 muestra los niveles de abstracción de MDA. Los PIM son más abstractos que los PSM y éstos más que el código. Por tanto, una transformación horizontal es de PIM a PIM o de PSM a PSM, y una transformación es vertical es de PIM a PSM o de PSM a código.

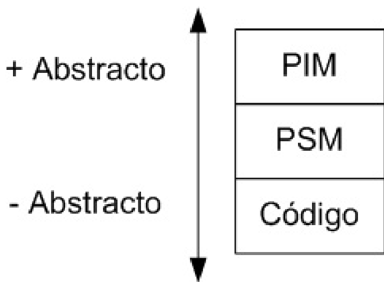


Figura 3.3: Niveles de abstracción en MDA

El desarrollo de un sistema de acuerdo al marco de trabajo de MDA se inicia con la construcción de un PIM. Después el PIM es transformado en uno o más PSM. Por último, el código es generado a partir de los PSM. La operación fundamental en MDA es la transformación de PIM a PSM.

El patrón de transformación de MDA [MM03] se muestra en la figura 3.4, que representa una función de transformación de modelos la cual requiere como entrada un PIM, y tiene como salida un PSM. La función, además, recibe información adicional del usuario, a través del *modelo de marcado*, que sirve para indicar qué reglas aplicar, cómo transformar cierto elemento del modelo origen, e indicar decisiones de diseño.

Los PIM y PSM se marcan antes de la transformación, idealmente éstos no deberían contaminarse con las marcas, por eso el patrón recibe el modelo de marcado como entrada. Sin embargo, lo más común es utilizar *tagged values*, por lo que el patrón de transformación real es de PIM-marcado a PSM. Por ejemplo, en un diagrama de clases se pueden marcar las clases persistentes con la etiqueta *persistente*; lo cual indicaría que esas clases se quieren transformar en tablas de un PSM que representa una base de datos.

La guía de MDA dice que los modelos PIM y PSM se expresan como modelos UML y que las transformaciones deben automatizarse al máximo posible. La guía menciona como alternativas de transformación la directa (vía programación), el uso de patrones y el metamodelado.

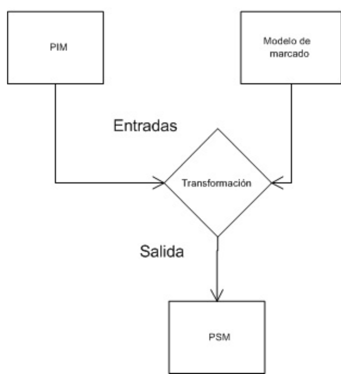


Figura 3.4: Patrón de transformación en MDA [MM03]

Desde el punto de vista de MDE también interesan las transformaciones de PIM a PIM y de PSM a PSM (de acuerdo a la clasificación de modelos de MDA).

Las versiones iniciales de MDA sirvieron como base de MDE, que generalizó MDA, y que define las transformaciones en el contexto del metamodelado. En MDE la forma más aceptada de definir los modelos es a través del metamodelado, y las transformaciones a través de lenguajes de transformación de modelos. En cambio, en la propuesta original de MDA el metamodelado no es una condición necesaria. Las versiones posteriores de MDA incorporaron después las ideas de MDE, que dieron lugar a *Meta-Object Facility* (MOF) y *Query View Transformation* (QVT).

Los conceptos encontrados en el campo de MDE se añadieron a la Guía de MDA en su actualización [MM03]. MDA es una realización de MDE basada en las tecnologías y estándares de OMG. El estándar para el metamodelado es MOF [Gro03a], y el lenguaje de transformación QVT [Gro05].

3.3.2. Meta-Object Facility. (MOF)

MOF tiene como función permitir la interoperación de herramientas de metamodelado, y de las que dan soporte al proceso de desarrollo software basado en MDA. MOF es una arquitectura de metamodelado que consta de cuatro capas.

El nivel M0 contiene los datos de la aplicación o los objetos de un sistema informático en ejecución. Las instancias del nivel M0 son modeladas en el nivel M1. La relación que existe entre las entidades en M0 y los modelos en M1 es que un modelo en M1 es realizado por un sistema en M0. Los modelos localizados en M1 son expresados con los lenguajes definidos a través de metamodelos ubicados en M2. Finalmente, el nivel superior M3 contiene un modelo de los lenguajes de M2 o meta-metamodelo. Este modelo es precisamente MOF.

Un ejemplo de la torre reflexiva de MOF se muestra en la figura 3.5. En la parte superior de la torre está el metamodelo MOF. En el nivel M2 están definidos algunos metamodelos como UML, E-R y Java; todos ellos son instancias de MOF. En el nivel M2 se definen transformaciones de modelos con los elementos definidos por los lenguajes de modelado (en la figura una transformación de UML a E-R y otra de UML a Java). En el nivel M1 están definidos un modelo UML, uno E-R y un programa Java; los dos últimos se obtienen mediante la ejecución de transformaciones. En este ejemplo, también, se puede distinguir entre modelos PIM y PSM. Los modelos UML y E-R son PIM, en cambio, el programa java es un PSM.

MOF específica una *interfaz de programación de aplicaciones* (API) la cual permite definir modelos y metamodelos, o consultar sus elementos. La API es reflexiva [Gro03b], es decir, la interfaz es independiente del metamodelo en M2. Esto sólo es posible entre los niveles M1, M2 y M3, los relativos al modelado, porque como se mencionó, la relación entre M1 y M0 es sólo conceptual.

MOF es un metamodelo muy similar a UML. Los modelos que se pueden definir mediante MOF son diagramas de clases. La principal diferencia entre MOF y UML es que MOF utiliza sólo cinco constructores básicos para definir

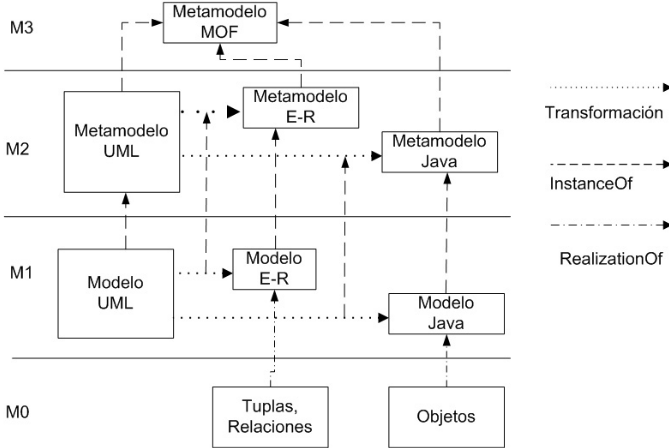


Figura 3.5: Torre reflexiva de MOF

un lenguaje de modelado: clases, generalizaciones, asociaciones, atributos y operaciones.

Las clases son usadas para definir los tipos de elementos del lenguaje de modelado. La generalización permite establecer una relación de generalización/especialización entre clases. Las asociaciones sirven para definir relaciones entre clases, y las hay simples y de composición. Las asociaciones son binarias y unidireccionales, es decir, sólo una de las entidades conoce la asociación. La composición es un tipo especial de asociación. Un elemento que pertenece a una composición no puede existir sin estar contenido por algún elemento compuesto. Los atributos se usan para definir las propiedades de los elementos del modelo. Las operaciones son especificadas sólo para las clases, y se puede indicar su lista de parámetros, pero no su semántica.

La arquitectura de metamodelado MOF está acompañada de otros dos estándares, XMI y QVT. El primero es un esquema XML para la representación de modelos MOF, y el segundo es el lenguaje estándar de transformación de modelos MOF.

Tabla 3.1: Espacios Tecnológicos en MDA

Espacio Tecnológico	**Metamodelo**	**Lenguaje**	**Entorno**
ATLAS	Ecore, KM3	ATL	Eclipse
Borland Together	Ecore	QVT Operational (imperativo)	Eclipse
SmartQVT	Ecore	QVT Operational (imperativo)	Eclipse
MOMENT	Ecore	Maude	Eclipse
XML	Esquema XSD de XMI	XSLT	Lenguajes de propósito general

3.4. Espacios Tecnológicos en MDA

MDA y MOF son estándares que sólo establecen guías para la definición de las tecnologías que los implanten, es decir, carece de un entorno de ejecución. En esta tesis se define un proceso MDE alineado en lo posible a MDA. Para conseguirlo, es necesario elegir un espacio tecnológico, cuya principal característica es su lenguaje de transformación de modelos. Se ha impuesto, además, la restricción de que el lenguaje sea declarativo en lugar de imperativo, porque así se pueden definir las correspondencias sin mezclarlas con el código de construcción y de búsqueda de elementos de un modelo.

Los espacios tecnológicos que satisfacen los requerimientos planteados son ATL [JAB+06], MOMENT [BCR05] y XML [SMMP+06]. Hay algunas implementaciones de QVT imperativo como *SmartQVT y Borland Together*, las cuales quedan fuera de la evaluación; se prefiere QVT declarativo. La tabla 3.1 muestra los espacios tecnológicos en MDA con su metamodelo de M3 (variantes de MOF) y entorno.

ATLAS y MOMENT están basadas en MOF y tienen como lenguaje de modelado a Ecore (un subconjunto de MOF) [BBM03]. Sin embargo, sus lenguajes de transformación son diferentes; ATLAS usa ATL [JAB+06] y MOMENT emplea QVT sobre Maude [CDE+02]. En cuanto a sus entornos de ejecución, las dos son *plug-ins* de Eclipse [Hol04]. Otro espacio tecnológico aplicable a MDA es XML [SMMP+06], porque los modelos MOF tienen una representación como documentos XML, por lo cual, se pueden transformar a través de plantillas XSLT [Cla99]. A continuación se describen los espacios tecnológicos mencionados.

3.4.1. XML-XSLT

Cuando se usa XML como espacio tecnológico de MDA, los modelos y metamodelos se definen como documentos XMI y el lenguaje de transformación empleado es XSLT [Cla99]. XSLT, o Transformaciones del lenguaje extensible de hojas de estilo es un lenguaje para transformar documentos XML a otros documentos XML o a formatos tales como HTML, PDF, SVG, o texto, etc. XSLT es además un estándar del W3C.

Una transformación XSLT consiste en un conjunto de reglas, cada una de las cuales tiene un patrón de entrada, que sirve para identificar los nodos del documento de entrada a los que tiene que aplicarse la regla. Durante una transformación XSLT, el árbol del documento XML de entrada es recorrido. Al visitar un nodo, se evalúa si satisface el patrón de entrada de alguna regla; si lo cumple, entonces se debe generar la salida de acuerdo a la regla. El orden de aplicación de las reglas no es importante, y suele haber un conflicto de resolución si varias reglas tienen el mismo patrón de entrada. La transformación continúa con el recorrido del árbol, y aplicando las reglas subsecuentes.

XSLT utiliza el lenguaje XPath [DC99] para identificar los nodos del documento de entrada. XPath puede seleccionar nodos navegando el árbol en cualquier dirección, y aplicando predicados basados en la posición, contenido y atributos de los nodos. Las expresiones XPath son usadas para seleccionar el conjunto de nodos de entrada, para evaluar condiciones y para calcular valores que serán usados en los nodos del árbol de salida.

XSLT se puede aplicar a la transformación de modelos mediante la serialización de modelos UML o MOF en XMI. Sin embargo, la mayor desventaja de este enfoque radica en que la transformación se define a un nivel sintáctico. Es decir, un modelo es una instancia del esquema XML que es equivalente a un metamodelo MOF. La transformación entre los modelos no se establece mediante la transformación de los elementos del metamodelo, sino como la de elementos de un documento XML a otro. O sea, un nodo del documento que se encuentra en cierta posición, se transforma en un conjunto de nodos XML del documento destino, y se colocan en cierta posición del árbol del

documento destino. Si se quiere transformar un concepto inexistente en el metamodelo origen o en el destino XSLT no lo informa. Además, en XSLT las transformaciones dependen de la profundidad de los modelos, lo cual, a su vez depende de la definición de los metamodelos. El formato XMI es muy difícil de leer y entender. Las referencias en un modelo XMI indican la posición de los elementos del modelo de forma diferente al estándar XML, esto hace muy complicado escribir las transformaciones. Como en XSLT todo es relativo a la posición entonces no se tienen en cuenta las relaciones semánticas, que se indican en el metamodelo, de los elementos.

3.4.2. ATL. Atlas Transformation Language

ATL fue una propuesta enviada a OMG para ser adoptada como estándar para QVT. ATL es uno de los componentes de ATLAS MegaModel Management Architecture (AMMA) [FJ05], que se completa con: AMW (ATLAS Model Weaver), AM3 (ATLAS MegaModel Management Tool) y ATP (ATLAS Technical Projectors). AMW sirve pare definir correspondencias entre modelos gráficamente. AM3 es un repositorio de modelos, que permite almacenar modelos, compartirlos de forma distribuida, y controlar versiones. Por último, ATP está integrado por programas que permiten leer modelos definidos con metamodelos diferentes a MOF.

ATL es un lenguaje de transformación híbrido (declarativo e imperativo) para transformar modelos MOF. El lenguaje de consulta de modelos es el *Object Constraint Language* (OCL) [WK98]. Una función de transformación de modelos en ATL está integrada por reglas de transformación. La función se aplica a un conjunto de modelos de entrada y tiene como salida uno o más modelos. Los metamodelos de cada uno de los modelos participantes son parte de la entrada de la función de transformación.

En la figura 3.6 se muestran dos metamodelos muy simples, que servirán para ejemplificar la transformación del listado 3.1. El metamodelo orientado a objetos tiene los siguientes elementos: una *Clase* se compone de *Atributos*, los cuales tienen un tipo de dato (*TipoDeDato*). Además, una clase puede tener una superclase. Y el metamodelo relacional contiene *Tablas* compuestas

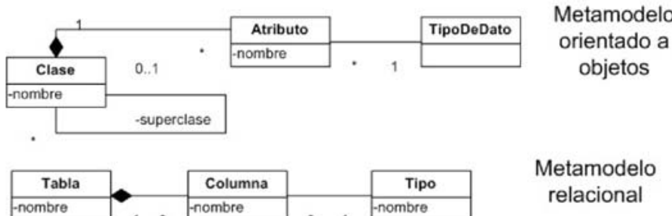


Figura 3.6: Metamodelos orientado a objetos y relacional

por *Columnas* de cierto *Tipo*.

Al inicio del programa hay una línea de encabezado (listado 3.1) que indica cuales son los modelos y metamodelos de entrada y salida, y también, los nombres para identificarlos. Los metamodelos de entrada y salida pueden ser una lista separada por comas para indicar N modelos de entrada o de salida. Un programa ATL se compone de un conjunto de reglas. Un elemento del metamodelo origen no puede satisfacer más de una regla de transformación, en caso contrario, habría un error en tiempo de ejecución. El único caso que lo admite es que sean excluyentes mediante la precondiciones de la reglas.

Una regla de transformación en ATL consta de las siguientes secciones: *name*, que indica el nombre de la regla (*rule claseAtabla*). Para identificar un elemento de un metamodelo se utiliza el nombre del metamodelo seguido de *!* y el nombre del elemento del metamodelo. Después se indica la clase de los elementos que se transformarán mediante la palabra *from*. Después se define la condición para efectuar la transformación, y finalmente la sección de construcción de los elementos de salida indicada por la palabra *to*.

En el listado 3.1 se transforman las *clases* sin superclase en una *tabla*. Los *atributos* y *tipos de dato* se transforman en *columnas* y *tipos* respectivamente. Los *atributos* que son parte de subclases se incorporan como *columnas* a la *tabla* que se obtiene de la clase raíz de una jerarquía de clases, para lo cual se utiliza una consulta OCL definida en el *helper superClase*.

Listado 3.1: Transformación ATL

```
module ClasesARelacional;
```

```
create OUT : Relacional from IN : Clases;

helper context Clases!Clase def: superClase(): Clases!Clase =
if self.super->size() = 0 then
   self
else
   self.super.superClase()
endif;

rule claseAtabla{
from Clases!Clase(
 super->size() = 0
)
to relacional!Tabla(
    nombre <- nombre
 )
}

rule atributoAColumna{
from Clases!Atributo
to relacional!Columna(
    nombre <- nombre,
        tabla  <- atributo.tipo.superClase(),
        tipo   <- TipoDeDato
 )
}

rule TipoDeDato{
from Clases!TipoDeDato
to relacional!Tipo(
    nombre <- nombre
 )
}
```

La semántica de un programa ATL es la siguiente: para cada elemento del modelo de entrada, se verifica si satisface la precondición de una regla; si es así, entonces se revisa que no fue transformado antes (en caso contrario ocurre un error en tiempo de ejecución). Después, se construyen los elementos de salida y se resuelve la asignación de valores. Si se asigna un tipo primitivo con cardinalidad 1, entonces la expresión del lado izquierdo

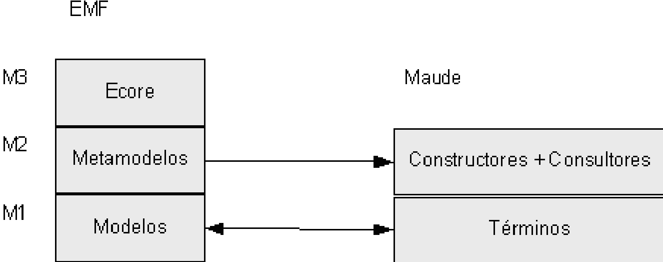


Figura 3.7: Proyectores en MOMENT

de la regla debe devolver exactamente un elemento de ese tipo. En el caso de que sea una colección, la expresión puede devolver uno o más elementos a partir de una consulta OCL y se verifica que el tipo primitivo de cada elemento de la colección sea del mismo que se define en el metamodelo. Si lo que se resuelve es la asignación de una referencia con cardinalidad uno, se verifica que exista un enlace dinámico correspondiente al elemento del modelo origen. En el caso de colecciones de clases se efectúa el paso anterior para cada elemento de la colección y se verifica que la cardinalidad máxima no exceda la definida en el metamodelo.

3.4.3. Model Management Framework. MOMENT

MOMENT es un marco de trabajo para la gestión de modelos en el contexto de MDA [BCR05]. MOMENT utiliza QVT para especificar las transformaciones, que son convertidas a especificaciones en Maude, donde realmente la transformación es ejecutada. MOMENT proyecta los metamodelos y modelos definidos en MOF a álgebras y términos algebraicos definidos en Maude [CDE+02] (figura 3.7). Las especificaciones algebraicas de Maude pertenecen a la familia de OBJ y su mecanismo de deducción ecuacional proporciona la semántica operacional de los operadores de modelos.

Los operadores de gestión de modelos que proporciona son:

1. Intersección y unión. La unión devuelve un modelo con los elementos que participan en A o B. La intersección devuelve los elementos que

únicamente participan en A y B.

2. Diferencia. Efectúa la diferencia o resta entre, es decir, el conjunto de elementos de un modelo que no se encuentran en el otro (A-B).

3. ModelGen. Realiza la transformación de un modelo A que es instancia del metamodelo MM_A en el modelo destino B que es instancia del metamodelo MM_B.

Los operadores de intersección, unión y diferencia tienen como entrada dos modelos que pertenecen al mismo metamodelo y el resultado es otro modelo, también del mismo metamodelo. El operador *ModelGen* es el que se usa en la transformación de modelos. La especificación QVT de una transformación es proyectada a un programa en Maude que ejecuta el *ModelGen*.

La transformación QVT de los metamodelos de la figura 3.6 se muestra en el listado 3.2. Al inicio de la transformación se indican los metamodelos y sus identificadores (*transformation ClasesARelacional*). Después se indica que el identificador de una tabla es su nombre (*Key Tabla*), lo cual sirve para asignar las columnas procedentes de *atributos* de subclases a la *tabla* correcta. Las reglas de transformación se definen con la palabra *relation*. Simplificando QVT, se puede decir que los elementos de entrada son indicados mediante *checkonly*, y los elementos de salida son construidos en las secciones *enforce*. Así que hay tres reglas, una *clase* y sus subclases se transforman en una sola *tabla*, un *atributo* en una *columna* y un *tipodeDato* en un *Tipo*.

Listado 3.2: Transformación QVT

```
transformation ClasesARelacional ( clases: Clases; relacional
    : Relacional ) {

        key Tabla {nombre};

        top relation ClaseAtabla {
       cln: String;

           checkonly clases c: Clase {
```

```
     nombre = cln
    }

    enforce relacional t: Tabla {
            nombre = if (c.superclase->size() = 0)
                then
                                                cln
                                        else
                                                c.superclase.
                                                   nombre
                                        endif
    } where {
     atributoAColumna(c, t);
}
 }

 relation atributoAColumna {
    an: String;

checkonly domain clases  c:Clase {
    atributo = a: Atributo {
           nombre = an,
                          tipo = t : TipoDeDato { }
    }
}

enforce domain relacional c: Columna {
   nombre = an,
          tipo = tc : Tipo { nombre = t.nombre }
    }
 }

 top relation tipoDeDatoATipo {

checkonly domain clases  td: TipoDeDato {}

enforce domain relacional t: Tipo {
        td.nombre = t.nombre
    }
 }
```

```
}
```

Posteriormente, la transformación QVT es proyectada a un programa Maude. Maude es un lenguaje de tipos basado en lógica ecuacional y reescritura de términos. Cada metamodelo define una signatura, que se compone de un conjunto de *sorts*, uno por cada clase del metamodelo, y cada regla de transformación es convertida en un conjunto de ecuaciones. La ejecución de las reescrituras dan por resultado un modelo transformado. Para más detalles al respecto de este proceso se puede consultar [Bor08].

3.5. Marco comparativo para lenguajes de transformación en el espacio tecnológico de MDA

En esta sección se define un marco comparativo para lenguajes de transformación en el espacio de MDA. Primero se abordan las características generales como tipo de lenguaje y de reglas. Después se trata de los tipos de operación como de validación, incremental y su completitud. Finalmente, se definen algunas características de la plataforma de ejecución.

3.5.1. Características generales

Las características principales de un lenguaje de transformación de modelos son que posee una sintaxis dedicada a la consulta y otra para la construcción de modelos. Como se mencionó, se prefieren los lenguajes declarativos, porque no hace falta indicar el orden de ejecución del programa. Para poder evaluar este tipo de lenguajes es importante definir las propiedades de las reglas de transformación.

Lenguaje de consulta y construcción dedicados. Los espacios tecnológicos en MDA deben permitir la consulta y filtrado de información de los elementos de los modelos de entrada.

Tipo de lenguaje: declarativo, imperativo e híbrido. Las transformaciones son declarativas, si se especifica qué hace en lugar de cómo lo hace. En los lenguajes declarativos se definen reglas de transformación que

son aplicadas si la condición especificada en el lenguaje de consulta se satisface. Por otra parte, las transformaciones son imperativas si se especifica qué operaciones deben aplicarse al modelo de entrada para obtener el modelo de salida. Este es el caso de los lenguajes imperativos de propósito general. El lenguaje de transformación es híbrido si el lenguaje permite construcciones imperativas y declarativas.

Cardinalidad de la reglas. Indica cuántos elementos de entrada se permiten como entrada y cuántos elementos de salida pueden construirse. Las cardinalidades permitidas son 1-1, 1-N, M-1 y M-N.

Reglas de transformación condicionales. Una transformación se aplica si el modelo de entrada satisface una condición. Los lenguajes de consulta de modelos permiten establecer condiciones en la selección de elementos, lo cual permite definir reglas de transformación excluyentes mediante alguna condición.

Reglas de transformación genéricas. Una transformación genérica es aquella que se define entre superclases y se aplica, también, a las clases que la especializan. Esto evita que el código común de la transformación tenga que escribirse en cada una de las clases específicas, es decir, una regla puede extender a otras.

Reglas de transformación abstractas. Una transformación abstracta se puede definir sólo si hay transformaciones genéricas. Los metamodelos en MOF pueden contener jerarquías de clases y algunas clases ser abstractas. La transformación de un elemento abstracto, define una transformación que debe aplicarse al momento de transformar elementos que descienden de un nodo abstracto.

Reglas de transformación bidireccionales. Una regla de transformación es bidireccional si su especificación permite obtener los elementos del modelo origen a partir de los elementos del modelo destino.

Reglas de transformación transaccionales. Una transformación puede tener puntos de aceptación (*commit*) o de regreso al punto anterior (*rollback*)

dependiendo de información del modelo de entrada. Por ejemplo, si una entidad del modelo de entrada se debe transformar en n entidades del modelo de salida, pero en un punto determinado de la transformación no hay información suficiente entonces se realizaría un *rollback* para dejar al modelo de salida consistente.

Reglas de transformación con correspondencia condicional. Una correspondencia condicional permite indicar en la sección de construcción o salida de una regla una condición para construir un elemento u otro del modelo de salida.

Reglas de transformación por elemento de entrada. El lenguaje puede permitir una o varias reglas de transformación por elemento de entrada, y debe definir la semántica de transformación cuando es válido tener más de una regla por elemento.

3.5.2. Características operacionales de los lenguajes de transformación

Además de la ejecución directa, la transformación de modelos tiene otros tipos de operación que resultan de interés para el desarrollo de las mismas, las cuales se describen en esta sección.

Validación del modelo de entrada/salida. Se refiere a si se verifica que los modelos de entrada/salida satisfacen las restricciones impuestas por sus metamodelos antes y después de ejecutar una función de transformación.

Validación de la completitud de las transformaciones. Se verifica que una función de transformación permita obtener un modelo destino válido de acuerdo a las restricciones de cardinalidad y composición.

Operación en modo de validación. Dada una función de transformación y los modelos de entrada y salida, se verifica que la función de transformación permita obtener el modelo de salida a partir de los modelos de entrada.

Operación en modo incremental. Supongamos que un modelo de salida se obtuvo a partir de una función de transformación y de un modelo de origen. El modelo origen ha sido cambiado por el usuario y al ejecutar otra vez la transformación se actualiza el modelo de salida con los elementos procedentes de los cambios en el modelo de entrada. Es decir, la nueva transformación no ejecuta el total de las reglas de transformación, sino sólo las afectadas por los cambios.

Trazabilidad de la ejecución de transformaciones. El lenguaje de transformación genera modelos adicionales con trazas a partir de las reglas de transformación. Esta propiedad sirve para identificar el origen de los elementos generados.

Define el comportamiento de ejecución de las transformaciones cuando el modelo de entrada no está bien definido. Cada motor de transformación debe definir qué sucede si el modelo de entrada no satisface las restricciones de su metamodelo.

Control automático de los elementos transformados. Los mecanismos de transformación, es decir, los algoritmos subyacentes que efectúan las transformaciones, deben controlar qué elementos del modelo origen ya han sido transformados. La principal implicación de esto es que el usuario del lenguaje de transformación no debe indicarlo como parte de las reglas de transformación.

La estrategia de aplicación de las reglas es determinista o no determinista. Si para un modelo de entrada dado, la estrategia de evaluación y aplicación de las reglas de transformación sigue el mismo orden cada vez que se aplica a dicho modelo, entonces la estrategia es determinista, en caso contrario es no-determinista .

3.5.3. Características de la plataforma de soporte

El desarrollo de transformaciones de modelos no es una tarea sencilla, por lo cual las plataformas de transformación deben proveer a los desarrolladores un conjunto de herramientas que lo faciliten.

Tabla 3.2: Características generales

	Atlas	**MOMENT**	**XML**
Lenguaje de consulta dedicado	OCL	OCL	XPath
Lenguaje de construcción dedicado	ATL	QVT	XSLT
Tipo de lenguaje	Híbrido	Declarativo	Declarativo
Cardinalidad de la reglas	1-1 1-N M-1 M-N	1-1 1-N M-1 M-N	1-1 1-N M-1 M-N
Reglas de transformación condicionales	Sí	Sí	Sí
Reglas de transformación genéricas	No	No	No
Reglas de transformación abstractas	No	No	No
Reglas de transformación bidireccionales	No	No	No
Reglas de transformación transaccionales	No	No	No
Reglas de transformación con correspondencia condicional	Sí	Sí	Sí

Compilador y depurador. Se evalúa si cuenta con un compilador y/o depurador del lenguaje de transformación de modelos.

Generación de texto/código. Se refiere a las facilidades que proporciona la herramienta para generar texto o código a partir del lenguaje de consulta de modelos.

Lenguaje Gráfico. La definición gráfica de las transformaciones es una importante ayuda a los usuarios poco expertos en los lenguajes de transformación (por ejemplo, QVT tiene un lenguaje gráfico [Gro05]).

Mecanismos de modularidad. Se evalúa si existe alguna forma de integrar transformaciones anteriores a través de librerías o módulos.

3.5.4. Evaluación.

La tabla 3.2 muestra que las consultas en ATL y MOMENT se hacen a través de OCL y que en el espacio tecnológico XML se usa XPath. OCL es un lenguaje especializado para consultar modelos MOF y UML. En cambio, utilizar XPath para consultar modelos XMI es complicado debido a que las expresiones XPath dependen de la representación y posición de los elementos XML. ATL y MOMENT resultan ventajosos en este aspecto respecto al espacio tecnológico XML. El lenguaje de transformación también debe tener

Tabla 3.3: Características operacionales

	ATL	**MOMENT**	**XSLT**
Validación del modelo de entrada/salida	No	No	No
Validación de la completitud de las transformaciones	No	No	No
Operación en modo de validación	No	No	No
Operación en modo incremental	No	No	No
Trazabilidad de la ejecución de transformaciones	No	Sí	No
Comportamiento especificado cuando el modelo de entrada no está bien definido	No	No	No
Control automático de los elementos transformados	Sí	Sí	Sí
Estrategia determinista/no-determinista	Sí	Sí	Sí

Tabla 3.4: Características de la plataforma de transformación

Características de la plataforma de soporte	**ATL**	**MOMENT**	**XSLT**
Compilador y depurador	Sí	Sí	Sí
Generación de texto/código	ATL queries Plantillas JET	Plantillas JET	XSLT
Lenguaje Gráfico	Sí	Sí	Sí
Mecanismos de modularidad	Sí	Sí	Sí

una sintaxis especializada para la construcción de los elementos del modelo de salida, los tres la tienen. En los casos de ATL y MOMENT se puede definir más de una regla de transformación si son excluyentes mediante una condición. En el caso de XSLT se aplican todas las reglas.

La tabla 3.3 muestra que MOMENT es el único que soporta de forma integral la trazabilidad. Ninguno de los espacios tecnológicos comparados tiene algún modo de operación especial como operación incremental o validación. La tabla 3.4 indica que los tres espacios tecnológicos cuentan con herramientas de compilación, depuración, etc.

El lenguaje de transformación XSLT satisface la gran mayoría de los requerimientos necesarios en un lenguaje de transformación usado para MDE. Es un lenguaje estándar y de muy amplio uso. Sin embargo, es complicado escribir transformaciones XSLT y aún es más difícil escribirlas con formatos como XMI. Una transformación XSLT se hace por medios sintácticos y no usando la semántica de los metamodelos. Es decir, se requiere de soporte a un nivel semántico, como el proporcionado por el editor XSMapper [LC06], para poder describir las transformaciones mediante las relaciones semánticas entre los elementos de dos esquemas XML.

El lenguaje de transformación XSLT no se puede aplicar fácilmente a la transformación de los elementos de un metamodelo, porque las transformaciones se deben escribir con respecto a la representación XML del modelo MOF. En otras palabras, existe un problema de acoplamiento entre los lenguajes de consulta de modelos MOF y los lenguajes de consulta de XML.

MOMENT ofrece la ventaja de los operadores genéricos de transformación de modelos; no obstante, éstos pueden ser implementados en XSLT o ATL. En MOMENT se realiza de forma natural la transformación de modelos mediante la lógica ecuacional. Resulta muy claro que al principio se tiene un término en un álgebra que se transforma en un término de otra y que los metamodelos representan la signatura del álgebra. Sin embargo, los mecanismos de navegación sobre modelos de tipo UML, que resultan naturales en OCL, eran limitados (2005), porque la versión de MOMENT del 2007 fue la primera en disponer de OCL y QVT. Los experimentos de transformaciones con Maude efectuados en este trabajo mostraron la necesidad de usar QVT y algunos defectos de transformar directamente con Maude. Contar con un marco formal tiene poco impacto en la necesidad de realizar transformaciones en un método de desarrollo de hipermedia.

ATL permite efectuar la transformación de modelos mediante un lenguaje muy parecido a OCL y a QVT, por lo cual es fácil de usar con modelos MOF. ATL fue uno de los lenguajes candidatos al estándar QVT y es quizá el más usado de los lenguajes de transformación similares a QVT, por lo cual, se ha convertido en un estándar *de facto*. ATL proporciona herramientas de desarrollo integradas en Eclipse, entre ellas un compilador y un depurador. ATL también es fácilmente incorporable a aplicaciones en Java, tiene un amplio grupo de usuarios y soporte. Es un proyecto oficial de Eclipse, por lo cual, alcanza una difusión muy grande. Está bien documentado y hay muchos ejemplos de transformaciones. ATL pertenece a una plataforma de gestión de modelos que también es parte de los proyectos de Eclipse. La principal desventaja de ATL es que no tiene una semántica formal, aunque su algoritmo de ejecución está claramente especificado.

A partir de la comparativa, se puede observar que las transformaciones MDE se pueden especificar con cualquiera de los 3 lenguajes, y que ninguno

tiene una ventaja abrumadora sobre los demás. Desde el punto de vista de un usuario de transformaciones, no se valora como importante el soporte formal con el que está implementada una herramienta de transformación, sino las facilidades que dispone para desarrollar sus transformaciones (por ejemplo, que cuente con un depurador, que el tiempo de aprendizaje de los lenguajes de transformación no sea demasiado largo, y que funcione bien). Este trabajo se desarrolla en ATL debido a que es un lenguaje de transformación ampliamente probado y con herramientas con la mejor usabilidad.

3.6. Métodos MDE y MDA para desarrollo de hipermedia

Como se ha visto en este capítulo, las características principales de una implementación de MDE son definir los modelos y metamodelos a través de un lenguaje de metamodelado, facilitar la construcción de modelos a través de servicios de las metaclases, y transformar los modelos en otros con lenguajes de transformación especializados.

La mayoría de los métodos de desarrollo de hipermedia son dirigidos por modelos, sólo EORM y WSDM no usan de forma extensiva modelos o carecen de modelos precisos. Métodos como MIDAS, OO-H, OOHDM, OOWS y WebML generan código a través de sus herramientas CASE. Sin embargo, la representación de sus modelos y las transformaciones son parte del código embebido en sus herramientas. Es decir, son dirigidos por modelos, pero no se pueden catalogar como MDE.

3.6.1. OOHDMDA

El método llamado OOHDMDA [SD05] proporciona como PIM el modelo conceptual y el modelo navegacional de OOHDM. Sin embargo, el modelo navegacional puede ser derivado parcialmente del primero con una transformación de PIM a PIM. En OOHDMDA desde el PIM navegacional se deriva un PSM que permite generar *java servlets*.

Tabla 3.5: Características de OOHDMDA

Método	Proceso	Notación	Herramientas
OOHDMDA	Modelado conceptual Diseño navegacional Diseño de la presentación Implementación	Diagrama de clases Diagrama de contextos ADV	OOHDMDA

3.6.2. UWE

En [Koc06] se mencionan sus experimentos para usar transformaciones de modelos dentro del método UWE, sin embargo, el autor llega a la conclusión de que son difíciles de efectuar debido al estado de las herramientas de transformación. Aunque usó ATL que es una de las que tiene mayor desarrollo y soporte. También probó transformaciones en QVT-P (http://qvtp.org) de la organización QVT-Partners, una herramienta que es sólo una prueba de concepto.

3.6.3. MIDAS

MIDAS [CMV03], [MCdC04] es un método de desarrollo de hipermedia basado en MDA. Su proceso de desarrollo es el siguiente:

1. Modelado de requisitos. Se capturan los requisitos a través de casos de uso.

2. Modelado Conceptual. Se construye un modelo conceptual OO, independiente de la plataforma (PIM).

3. Modelado Navegacional. El modelo conceptual se representa mediante un esquema XML y se transforma a un modelo navegacional. (Las heurísticas de transformación son las de RMM [ISB95] (sección 2.3), y su notación la de UWE (figura 2.11a).

4. Generación del modelo de datos. El modelo conceptual es transformado al modelo objeto relacional, que es un modelo específico de la plataforma (PSM).

Tabla 3.6: Características de MIDAS

Método	**Proceso**	**Notación**	**Herramientas**
MIDAS	Captura de requisitos	Casos de uso	MIDAS
	Modelado conceptual	Diagrama de clases	
	Modelado navegacional	Diagrama navegacional UWE	
	Generación de la presentación	ADV-XML	
	Generación del modelos de datos	SQL-99	

5. Generación del modelo de presentación. El modelo navegacional es transformado a un documento XML, que es la plantilla del modelo de presentación.

El modelado de requisitos funcionales en MIDAS es a través de casos de uso; éstos son etiquetados como servicio funcional básico, servicio funcional. Después, se desarrolla un diagrama de actividad para conocer el orden de ejecución de los casos de uso. De cada caso de uso funcional se obtiene un *slice* funcional en el diagrama navegacional, y de cada caso de uso estructural se obtiene un *slice* estructural.

La principal aportación del método es su técnica para derivar, desde los caso de uso, estructuras navegacionales de tipo funcional o estructural. El método, prácticamente, construye un modelo de proceso, al establecer un orden de ejecución de los casos de uso y una separación de éstos por usuario, tal como se propuso en [Pen02].

En cuanto, al empleo de MDA, no se puede justificar que MIDAS utilice ese espacio tecnológico, pues no se menciona que los metamodelos estén definidos con MOF, ni que las transformaciones se hagan con un lenguaje de transformación de modelos.

3.7. Conclusiones

En este capítulo se ha presentado el paradigma MDE, sus conceptos fundamentales y las mejoras que aporta al desarrollo de software. En esta tesis se ha optado por utilizar el paradigma de MDE porque eleva el nivel

de abstracción en el desarrollo de software, lo que facilita el diseño de aplicaciones por expertos del dominio del problema. En un proceso MDE los modelos son entidades de primer orden que permiten especificar la funcionalidad de un sistema sin detalles de implementación, es decir, el diseñador no se concentraría en los aspectos tecnológicos sino en el proceso y consulta de la información. Además, las transformaciones permiten la generación semiautomática del sistema.

Las ventajas mencionadas previamente pueden ser utilizadas en el desarrollo de aplicaciones hipermedia y web porque dividen a las partes de interés de un sistema en modelos: de proceso, conceptual, de navegación, de presentación, entre otros. Además, el paso de un modelo de a otro se define usualmente mediante heurísticas que podrían tratarse como transformaciones de modelos.

También, se ha elegido la implementación de MDE basada en MDA, porque cuenta con estándares y soporte tecnológico. Se han presentado algunos espacios tecnológicos de MDA: ATLAS, MOMENT y XML. De acuerdo a los conceptos definidos de MDE es posible descartar otras propuestas que pretenden ser implementaciones de MDE en el espacio tecnológico de MDA. De los espacios tecnológicos en MDA se ha efectuado una comparativa y el resultado de ello es el uso de la combinación MOF ATL XMI como plataforma de transformación de modelos.

Parte III

Método de desarrollo de hipermedia dirigido por modelos (MDHDM)

Capítulo 4

Método de Desarrollo de Hipermedia Dirigido por Modelos (MDHDM)

4.1. Introducción

El objetivo principal de esta tesis es proponer un método para el desarrollo de hipermedia que se fundamenta en, por un lado, considerar el proceso como el componente central de la aplicación a desarrollar, y por otro lado, utilizar una aproximación dirigida por modelos, basada en MDA. Tal método ha sido llamado *Model Driven Hypermedia Development Method* (MDHDM).

Una de las características más importantes de los sistemas hipermedia es la navegación. MDHDM aborda cómo obtener el modelo navegacional a partir de un modelo de proceso. En MDHDM el modelo navegacional tiene dos dimensiones, una es la navegación guiada por el proceso y otra la navegación dirigida por la estructura. La primera está determinada por los enlaces de proceso o fuertes, y la segunda por los enlaces estructurales o débiles. Los enlaces fuertes hacen avanzar el estado del proceso, y tienen como efecto, algunas veces, la ejecución de tareas automáticas, mientras que los débiles sólo permiten dirigirse a otro nodo de información, siguiendo los

caminos derivados de las asociaciones entre datos. Tal distinción permite una guía metodológica clara para el desarrollo de aplicaciones hipermedia en las que tanto el proceso como la estructura de los datos aportan información para generar el modelo navegacional.

En el capítulo 2 se ha presentado una revisión de los principales métodos de desarrollo de hipermedia. La idea subyacente en todos ellos es que el ciclo de vida de desarrollo de una aplicación hipermedia tiene una serie de etapas similares a las del desarrollo convencional más una dedicada al diseño navegacional y de la presentación. En cada etapa se genera un nuevo artefacto, y la transformación de éstos en las diferentes etapas se basa en la aplicación de un conjunto de heurísticas. Éstas no están definidas formalmente y pocas de ellas se aplican de forma automatizada, lo que hace que el equipo de desarrollo se enfrente al problema de interpretarlas adecuadamente. Además, desde el punto de vista del desarrollador no está claro cómo traducir los elementos de los modelos a una plataforma tecnológica específica.

En MDHDM se definen de forma precisa los modelos, a través de metamodelos MOF y restricciones OCL; ambos permiten definir modelos semánticamente correctos. Las heurísticas proporcionadas por el método se expresan de forma precisa mediante reglas de transformación de modelos en ATL, lo que permite el desarrollo semi-automático del sistema siguiendo una aproximación MDE. Es decir, el modelo navegacional se obtiene basándose en transformaciones de modelos a partir de los modelos de proceso y estructural.

En este capítulo se presenta una introducción a MDHDM, sus principales características y los pasos aplicados a la definición del método. En los siguientes capítulos se explica cada uno con mayor detalle.

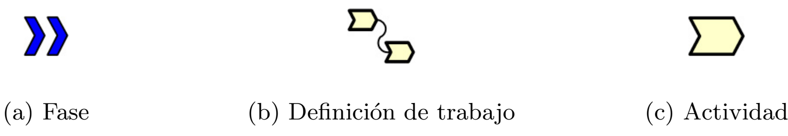
4.2. Método de Desarrollo de Hipermedia Dirigido por Modelos (MDHDM)

En las siguientes secciones se describen las características de MDHDM. Se empieza por su ciclo de vida y se compara con el de otros métodos.

Después se explican sus cualidades desde el punto de vista de MDE, se abordan sus metamodelos, los tipos de modelos PIM y PSM, las marcas de modelado, y las transformaciones.

4.2.1. Definición del ciclo de vida

El ciclo de vida de MDHDM es representado con la notación del *Software Process Engineering Metamodel* (SPEM) [Gro07]. La figura 4.1 muestra los elementos de SPEM utilizados y que se explican, muy brevemente, a continuación: una *actividad* es una unidad de trabajo, una *definición de trabajo* es un grupo de actividades, y una *fase* incluye actividades o definiciones de trabajo. Las entradas y resultados de una actividad son *productos del trabajo* que se representan como un documento o folio. Los elementos incluidos por una fase se indican mediante una flecha discontinua. Cuando se necesita definir el orden de ejecución de las tareas o definiciones de trabajo se utiliza un diagrama de actividad de UML.



(a) Fase (b) Definición de trabajo (c) Actividad

Figura 4.1: Notación de SPEM

El ciclo de desarrollo de software en MDHDM es muy similar al seguido en otros métodos: análisis, diseño e implementación. Las fases del ciclo de vida son independientes del tipo de proceso de desarrollo seguido (incremental, iterativo, espiral, etc.). La figura 4.2 muestra el ciclo de vida de los métodos de desarrollo de hipermedia.

El modelo de proceso, por lo general, no es tomado en cuenta, lo cual lleva a la obtención de modelos navegacionales, únicamente, centrados en los datos. Es decir, las tareas que efectúan los usuarios durante la navegación no son modeladas. Otros métodos incluyen el modelado del proceso después de haber hecho el diseño navegacional, es decir, incluyen primitivas de hipermedia que permiten definir parcialmente un proceso (ver figura 4.3).

Figura 4.2: Ciclo de desarrollo genérico de los métodos de desarrollo de hipermedia

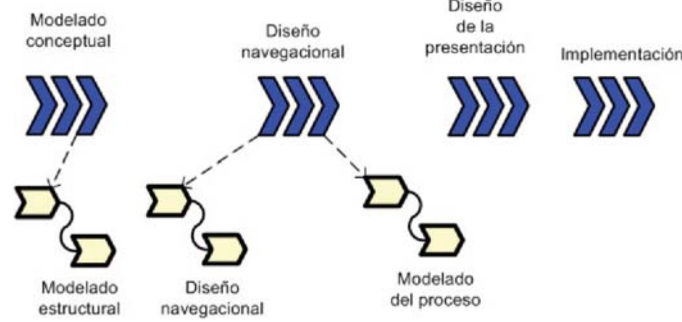


Figura 4.3: Ciclo con el modelado del proceso en el diseño navegacional

En cambio en MDHDM el modelo de proceso es el inicial. La figura 4.4 muestra el ciclo de vida de MDHDM.

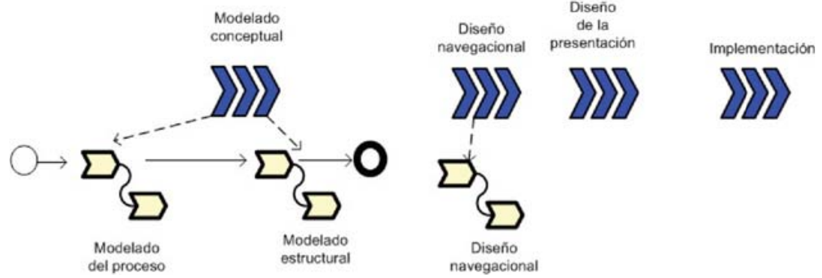


Figura 4.4: Ciclo de desarrollo en MDHDM

A partir del modelo de proceso se puede obtener parcialmente parte del modelo estructural y, posteriormente, desde los modelos de proceso y estructural se genera el modelo navegacional. Si no se cuenta con un modelo

de proceso, entonces en MDHDM se proporciona una técnica que permite obtener uno después de aplicar el análisis mediante casos de uso [Jac92] y de conocer la estructura organizacional. La figura 4.5 muestra los pasos para la obtención del modelo de proceso. Esta técnica se presentó inicialmente en [Pen02]. Básicamente consiste en clasificar los casos de uso según correspondan a actividades indivisibles o actividades compuestas o subprocesos. Después a partir de las relaciones entre casos de uso y del diseño organizacional se obtiene el flujo de control.

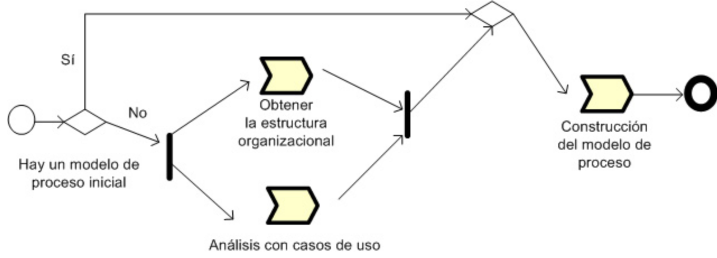


Figura 4.5: Modelado del proceso

Los casos de uso de mayor nivel en la jerarquía organizacional son casos de uso compuestos, y los casos de uso componentes son efectuados en las entidades organizacionales de menor nivel. Este modelo de proceso preliminar debe ser refinado y validado por los analistas. Para una mayor explicación el lector debe dirigirse a la sección 5.5.1, donde se presentan las reglas de derivación del modelo de proceso.

Después de haber obtenido el modelo de proceso, se procede a elaborar el modelo estructural. Se debe tener en cuenta que los objetos dato del modelo de proceso se corresponderán con clases del modelo estructural. Las relaciones que mantienen entre sí los objetos dato también deben transferirse al modelo estructural. El diseño del modelo estructural se puede efectuar mediante las técnicas de UML.

El siguiente paso es el diseño navegacional. Los pasos de los que se compone el diseño navegacional se muestran en la figura 4.6. El primer paso es efectuar la proyección del proceso por actor, es decir, obtener la vista que un usuario tiene del proceso. Además, las proyecciones aún se definen con

las primitivas del modelo de proceso. La diferencia entre las proyecciones y el modelo de proceso completo es la inclusión de tareas sincronizadas. Estas se pueden representar mediante eventos intermedios o bien con etiquetas a las actividades que permanecen como parte de la vista. En el caso más común de procesos estructurados [KtHB00] (en los que cada ramificación tiene su correspondiente unión), se reemplazan las secciones efectuadas por otros usuarios hasta encontrar la unión que cierra el bloque. Sin embargo, en otros casos no es posible hacerlo así, y se tiene que efectuar con las reglas de proyección presentadas en el capítulo 6. Las proyecciones sirven para obtener el espacio navegacional llamado *modelo navegacional fuerte*, debe su nombre debido a que los enlaces de este modelo son llamados enlaces fuertes.

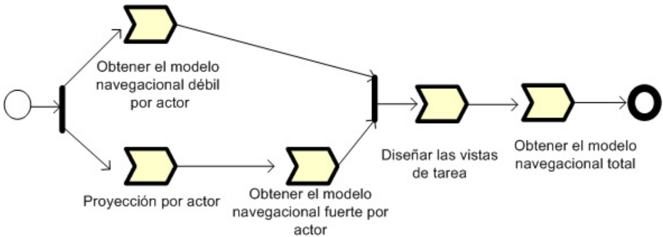


Figura 4.6: Diseño navegacional

A continuación se marca el modelo estructural para definir el espacio donde puede navegar un determinado actor. Dicho espacio navegacional es llamado *modelo navegacional débil*, los enlaces de este modelo son llamados enlaces débiles. También, se debe diseñar el modelo navegacional por tarea efectuada, es decir, construir la vista de las tareas o el espacio de hipertexto donde se puede navegar en cada una.

Después de efectuar la proyección del modelo de proceso, y el marcado del modelo estructural, entonces se pueden aplicar las heurísticas para la obtención del modelo navegacional, que nos conducen a la obtención del modelo navegacional con enlaces fuertes y al de enlaces débiles. Las heurísticas se explican en el capítulo6.

Una vez efectuado el diseño navegacional, entonces se puede realizar el diseño de la presentación (ver capítulo 7). Es decir, el diseño de las interfaces

con las que interactúa el usuario. En MDHDM las interfaces son abstractas, y por tanto, también independientes de la tecnología. Se proporciona un metamodelo con los elementos XML para el diseño de la interfaz (ver sección 7.2), las plantillas para la derivación de los esquemas XML de las instancias de datos que se presentarán en la interfaz (ver sección 7.3) y las interfaces abstractas de usuario (ver sección 7.4). Los pasos para efectuar el diseño de la presentación (figura 4.7) son: primero, la generación de los esquemas de las instancias de datos y después, el diseño de las interfaces abstractas, ambos de acuerdo al modelo navegacional, y a través de las plantillas mostradas en el capítulo 7.

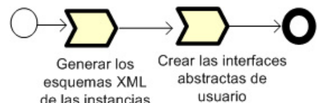


Figura 4.7: Diseño de la presentación

El último paso es la implementación. El equipo de desarrollo usaría los diseños efectuados con el método para hacer esta tarea. Para seguir con MDA se debe contar con un conjunto de modelos PSM que proporcionan la ejecutabilidad a los modelos PIM. A partir de los PSM se procede a la generación del código. En MDHDM se han escogido algunos PSM que permiten crear un prototipo (ver sección 8.7). Además, los usuarios del método pueden crear sus propios PSM.

4.2.2. Definición de los metamodelos y sus restricciones

La propuesta de MDHDM está basada en MDA, es decir, en la especificación de modelos a partir de metamodelos y en la transformación de modelos con reglas. La figura 4.8 muestra MDHDM desde la perspectiva MDA.

Se deben definir los metamodelos de los modelos incluidos en el ciclo de vida propuesto por el método. Se han definido los metamodelos MOF y restricciones OCL de los siguientes modelos: Casos de uso, Modelo de Proceso, Modelo Estructural y Modelo Navegacional.

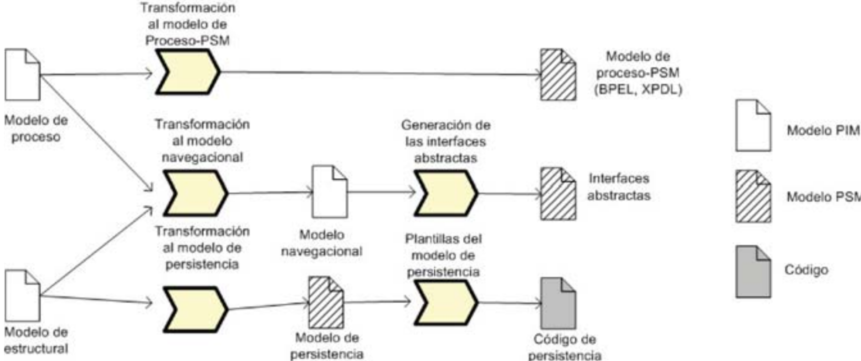


Figura 4.8: Modelos y transformaciones en MDHDM

Las interfaces abstractas de usuario o modelo de presentación no se representan con metamodelos y modelos MOF, sino que se definen mediante *tags* XML que pertenecen a un metamodelo de elementos gráficos.

4.2.3. Identificación de modelos PIM y PSM

De los modelos integrantes del proceso MDHDM los modelos estructural y de proceso son independientes de la plataforma tecnológica porque representan el dominio del problema. El modelo navegacional también se considera independiente de la plataforma porque, a pesar de que surge como elemento fundamental de los diseños de hipermedia, bien puede ser usado en el diseño de aplicaciones basadas en ventanas, además de que hay varias tecnologías de implementación de hipermedia (la gran mayoría en desuso debido a que últimamente sólo se considera la Web).

Los modelos que son dependientes de la tecnología son el modelo de persistencia del modelo estructural, el modelo de proceso-PSM (ejecutable), y el modelo de presentación-PSM. El modelo de persistencia está básicamente representado por tecnología relacional u objeto-relacional, aunque también sería factible considerar bases de datos XML o algún otro medio de persistencia.

El modelo de proceso-PSM representa al proceso ejecutable. La selección de este PSM depende de la complejidad del proceso. En el caso de los

procesos más simples el estado puede controlarse mediante la sesión de navegación; en otros casos, a través de una base de datos con una tabla con el estado del proceso relacionada con los datos asociados a la instancia del proceso; por último, para los más complejos, sí es recomendable usar un sistema de gestión de flujos de trabajo. Es este último caso se generarían especificaciones en BPEL [Ope06] o XPDL [WfM02].

El modelo de presentación-PSM está muy ligado a la tecnología hipermedia que se utilizará. En muchos casos la presentación es en HTML [HJR99], pero es preferible utilizar otras tecnologías como XUL [BKO+02] o XForms [Dub03] que separan los datos de la interfaz.

A partir de los tres modelos específicos de la tecnología se generará código fuente. En el caso del modelo de persistencia se generarían las especificaciones del lenguaje de definición de datos (DDL) de la base de datos. La presentación se llevaría a plataformas como Java, C# o PHP que generarían el documento HTML o XForms, entre otros.

4.2.4. Especificación de marcas de modelado

En general, para convertir de un modelo origen a uno destino es necesario algún tipo de marcado debido a que el modelo origen no tiene la información suficiente para ser transformado automáticamente. El modelo de marcado es una de las principales formas de intervención del diseñador del sistema.

El marcado de modelos utilizado en MDHDM es muy simple: un conjunto de pares (nombre, valor) que se asocia a algún elemento de un metamodelo. Mediante esta información se expresan las decisiones de diseño que deben aplicarse al ejecutar una transformación.

En el método el marcado de modelos es imprescindible para generar la vista que tiene un actor del proceso del modelo estructural; también es necesario para definir la visibilidad de la información, crear nuevos nodos que agrupan la información o determinar la estructura de acceso preferida, entre otras cosas. También, las proyecciones del modelo de proceso deben de marcarse para indicar los tipos de actividades. Las marcas definidas se presentan en las secciones 6.6 y 6.7.

Tabla 4.1: Transformaciones automáticas

Modelo origen	Modelo destino	Transformación
Proceso	Navegacional	ATL
Estructural	Navegacional	ATL
Proceso y estructural	Navegacional	ATL
Navegacional	Interfaz Abstracta Esquemas XML de los datos	Plantillas Plantillas
Interfaz abstracta	Interfaz PSM	XSLT
Estructural	Persistencia	ATL y plantillas
Proceso	Proceso-PSM	ATL y plantillas

4.2.5. Especificación de las transformaciones

A partir de las recomendaciones y patrones de diseño se puede identificar a las transformaciones automáticas entre modelos. Las transformaciones más importantes del método son las del modelo de proceso y estructural al modelo navegacional, y la generación de la presentación abstracta a partir del modelo navegacional. Los PSM que permiten generar el prototipo están sujetos al vaivén de los cambios en la tecnología, y además hay una multitud de ellos. Los PSM utilizados como ejemplo se explican en el capítulo sobre la implementación. La tabla 4.1 resume las transformaciones automáticas en MDHDM.

La principal técnica de transformación de modelos aplicada en el método son las transformaciones ATL. Las funciones de transformación se han definido en ATL, porque en este momento es el lenguaje de transformación con más herramientas de desarrollo, es parecido a QVT, y, se ha convertido en el estándar *de facto*. Las transformaciones se han planteado a partir de las heurísticas que se mencionan en los diversos métodos de diseño de hipermedia y otras nuevas introducidas por MDHDM. Además las transformaciones ATL tienen en cuenta el conjunto de marcas aplicables a los modelos de proceso y estructural.

Otra técnica de transformación utilizada se aplica a los documentos XML de la interfaz, y el camino natural es a través de transformaciones XSLT; el paso del modelo de presentación abstracto a uno específico de la tecnología se efectúa mediante éstas. La última técnica de transformación que se utiliza es la generación de código mediante plantillas. Si el origen es un modelo

MOF, entonces se puede hacer una consulta ATL que permite efectuar la generación del código.

Una parte importante del desarrollo se centra en las transformaciones entre modelos. Por lo tanto, las pruebas y la localización de errores de programación también deben plantearse. El desarrollo de las transformaciones para un proceso MDE se realiza a través de iteraciones y de pruebas. La prueba de que las transformaciones funcionan se hace generando los modelos destino manualmente y comparando el resultado de las transformaciones, es decir, se pueden comprobar mediante pruebas unitarias. Se define un conjunto de modelos de entrada, salida, se aplica una transformación y se comparan la salida de la transformación con los modelos de salida esperados.

4.3. Conclusiones

En este capítulo se ha presentado brevemente el método de desarrollo de hipermedia MDHDM. Se puede observar que el punto de partida del método es el modelo de proceso. Éste permite definir la dinámica del sistema, los diferentes tipos de usuarios, e identificar las actividades automáticas y manuales. Del modelo de proceso se pueden obtener algunas clases del modelo estructural a partir de los *objetos dato* o información de entrada y salida de las actividades del proceso. El modelo estructural se elabora con las técnicas de UML.

Posteriormente, se debe proyectar el modelo de proceso, por cada actor, para obtener la visión que tiene cada uno de dicho proceso. De las proyecciones se obtendrá el modelo navegacional fuerte que contiene los enlaces de proceso, es decir, aquellos enlaces que tiene efectos en el cómputo y estado del proceso. El modelo estructural debe marcarse para indicar algunas decisiones de diseño de hipermedia respecto a la visibilidad, agrupación y navegabilidad. Una vez marcado el modelo entonces se puede obtener el modelo navegacional débil. El modelo navegacional final es la unión de los dos modelos navegacionales. Es importante antes de unirlos analizar la información que es visible en cada tarea, es decir, diseñar las vistas de tarea. Aunque algunas de las heurísticas permiten obtener enlaces de las vista de tareas, es

recomendable hacerlo manualmente.

Aplicando una visión desde el paradigma MDE se deben especificar formalmente los metamodelos y transformaciones. La implementación de MDE utilizada en MDHDM es MDA, y por ello los metamodelos se definen con MOF y restricciones OCL. El proceso de desarrollo se ve como un encadenamiento de transformaciones, donde los pasos automáticos son transformaciones en el lenguaje ATL y la generación de código se efectúa mediante plantillas.

Capítulo 5

Modelado conceptual

5.1. Introducción

En este capítulo se definen los metamodelos y restricciones OCL de los modelos construidos durante el modelado conceptual del sistema hipermedia según el ciclo de vida propuesto en MDHDM. El modelado conceptual en MDHDM incluye el modelado del flujo de trabajo y modelado estructural. Además, opcionalmente, se pueden especificar los casos de uso.

Como ya se ha dicho, el eje central de MDHDM es el modelo de proceso que se especifica en la sección 5.3. El modelo de proceso describe las actividades que efectúan todos los actores, de forma coordinada, para llegar al objetivo final de la organización o de los usuarios del sistema.

Sin embargo, cuando se carece de una especificación del proceso se puede empezar el análisis a través de casos de uso, como se explica en la sección 5.2. A partir de ellos, se pueden obtener las versiones iniciales del modelo de flujo de trabajo y del modelo estructural mediante la técnica descrita en la sección 5.5.1.

El modelo estructural, descrito en la sección 5.4, representa las entidades que componen el dominio del problema y son la información necesaria para ejecutar el proceso. Una parte del modelo estructural puede derivarse a partir del análisis del flujo de datos del proceso, como se muestra en la sección 5.5.2. Finalmente, en la sección 5.6 se dan las conclusiones del capítulo.

5.2. Casos de uso

Los casos de uso sirven para especificar los requisitos funcionales de un sistema y están enfocados a los usuarios que no son expertos en el desarrollo de sistemas, con la intención de que plasmen de forma simple las tareas de valor que se efectúan en la organización. Los casos de uso han sido descritos en diversas propuestas como [Jac92], [BRJ99] y [CL99]; sin embargo, la definición y el metamodelo de casos de uso utilizados en este trabajo son una extensión del presentado en [Pen02]. En las siguientes secciones se explican las principales características, notación y metamodelo de los casos de uso.

5.2.1. Características

Los casos de uso son un medio de análisis que debe ser construido, colaborativamente, entre los diversos participantes del proceso de desarrollo de software: usuarios, desarrolladores, y expertos en el dominio, entre otros. Un caso de uso representa una secuencia de eventos automáticos o manuales que producen un resultado de valor para algún actor del proceso [Kru98]. Los casos de uso permiten identificar cuándo se ejecutan funciones realizadas por el sistema y las interacciones entre el sistema y los actores involucrados. Un actor es cualquiera que puede interactuar con el sistema: personas, dispositivos u otros sistemas. La ejecución del caso de uso se produce si el sistema satisface un conjunto de pre-condiciones. Al término de su realización el sistema alcanza otro estado y produce un resultado, que se caracteriza mediante las post-condiciones.

Existen dos tipos de relaciones entre casos de uso: inclusión y extensión. La relación de inclusión se presenta cuando un caso de uso inicia la ejecución de otro, como parte de los eventos efectuados en su especificación. La relación de extensión indica que el caso de uso es ampliado por la especificación nueva si se satisface una condición. Un caso de uso puede extender únicamente a otro, y podría ser extendido por varios.

Los caso de uso definidos en [Pen02] tienen, además, las propiedades: tipo de proceso, y datos de entrada/salida. El tipo de proceso sirve para indicar si una tarea se efectúa de forma manual o automática, si es un caso de uso

complejo, entonces el tipo de proceso será complejo. Los datos de entrada son la información necesaria para ejecutar el caso de uso, y los datos de salida la generada por él.

5.2.2. Notación

El diagrama de casos de uso muestra a los casos de uso, sus relaciones y los actores que intervienen. La notación para representar los casos de uso se muestra en la figura 5.1.

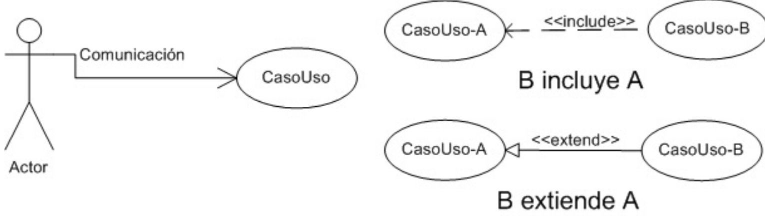


Figura 5.1: Notación para representar casos de uso

Un caso de uso se representa mediante un óvalo, los actores con monigotes. Una flecha entre un actor y un caso de uso indica una comunicación. La inclusión de un caso de uso y la extensión se representan con flechas discontinuas; la flecha usada para la inclusión contiene el estereotipo *include* y la de extensión el de *extend*.

Los casos de uso generalmente se recogen mediante una plantilla (ver tabla 5.1) que es rellenada por los analistas del sistema, en la que se identifica el caso de uso mediante un número o clave y un nombre. Se debe incluir una descripción textual del caso de uso y cuál es su objetivo. En la plantilla se indican las precondiciones necesarias para iniciar el caso de uso y las post-condiciones impuestas después de su ejecución. La parte más importante detalla los eventos que ocurren durante la ejecución del caso de uso. Los eventos se pueden describir de forma textual o de forma gráfica como diagrama de secuencia. También, se lista a los actores participantes y se les vincula con los eventos que ejecutan. La última parte de la plantilla corresponde a la especificación de las relaciones de inclusión y extensión. Los

campos de esta sección dependen del metamodelo, en este caso corresponden a la especificación de los puntos inclusión y extensión.

Tabla 5.1: Plantilla de caso de uso

Caso de uso	Nombre
Id	Identificador
Diagrama	
Propósito	Explicación textual del caso de uso
Tipo de proceso	[Automático—Manual—Compuesto]
Actores	Lista de actores
Datos de entrada	
Datos de salida	
Precondiciones	Condición de ejecución
Postcondiciones	Estado de salida
Flujo de eventos	
Extendido por	
Evento de referencia	
Condición	
Cuando	
Incluye a	
Evento de referencia	
Cuando	

5.2.3. Metamodelo

La inclusión de un metamodelo preciso de casos de uso en este trabajo, tiene como finalidad facilitar la definición de flujos de trabajo, a partir de los procesos de negocio que se efectúan en las organizaciones. El objetivo es proporcionar una etapa metodológica que nos permita analizar el sistema que se quiere desarrollar. Así, se puede seguir el análisis clásico de sistemas mediante casos de uso, tal como se define en [Jac92].

En esta sección se describen las propiedades y relaciones de los elementos de los modelos de casos de uso, a través de un metamodelo MOF mostrado

en la figura 5.2. El metamodelo presentado es una modificación del definido en [Pen02], en este caso los puntos de inclusión y extensión son iguales que los del metamodelo de casos de uso de UML [Gro04]. El metamodelo de [Pen02] incluye los eventos que forman parte del caso de uso, permite indicar quienes son los actores participantes, o cuándo se ejecuta el caso a través de condiciones, y también se puede especificar cuáles son los datos de entrada y salida del caso de uso, en cambio el de UML no contiene dichos elementos. Tal información es relevante si se quiere analizar un proceso a partir de casos de uso.

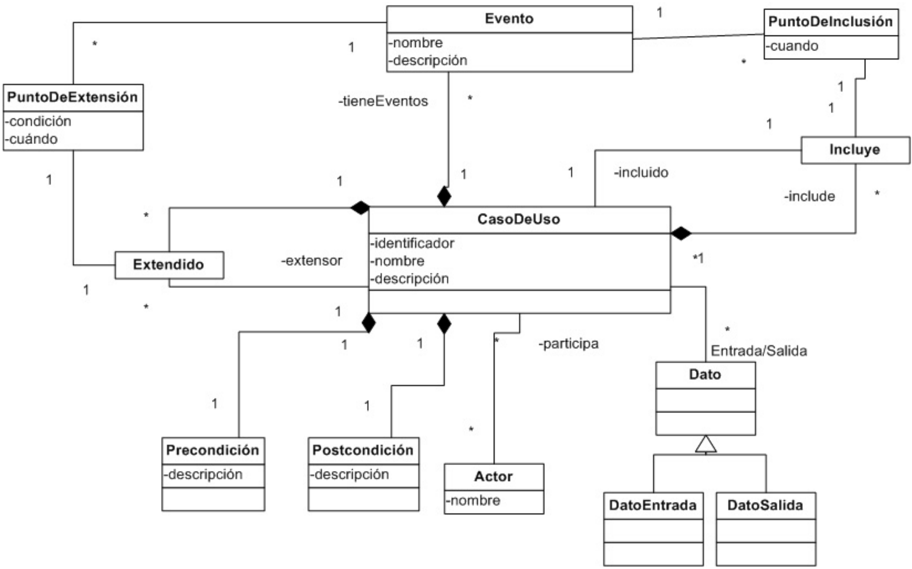


Figura 5.2: Metamodelo de casos de uso

Un caso de uso contiene las propiedades: nombre, descripción e identificador, que sirven para identificarlo y explicar su propósito. En un caso de uso pueden participar n actores. Un caso de uso tiene un flujo de eventos. Cuando un caso de uso incluye a otro entonces existe una relación de inclusión entre ambos. La relación inclusión se asocia con un punto de inclusión que indica un evento del caso de uso que incluirá al otro. Esto quiere decir que ese evento se usa como punto de referencia para efectuar la inclusión. El punto de inclusión tiene un atributo llamado *cuando* que indica si el caso de

uso a incluir se coloca antes o después del evento de referencia. Los puntos de extensión funcionan del mismo modo que los puntos de inclusión, excepto que tienen una condición que los habilita, y pueden indicar el reemplazo del evento de referencia si en su atributo *cuando* indican un reemplazo. Finalmente, un caso de uso tiene una precondición y postcondición, y además, está relacionado con n datos de entrada/salida. En [Pen02] se define como caso de uso elemental aquél que no tiene relaciones de uso o extensión, mientras que complejos son aquéllos que sí usan o son extendidos por otros.

Las diferencias de este metamodelo con el presentado en [Pen02] se mencionan a continuación. Las relaciones de extensión e inclusión son metaclases. Además, a través del atributo *cuando* del punto de extensión se define si el caso de uso incluido debe ejecutarse antes o después de cierto evento del flujo principal. Otra diferencia de menor importancia, es que los datos de entrada y salida, así como las postcondiciones y precondiciones tienen una superclase común, las clases *Dato* y *Condición*.

Restricciones

Las restricciones que debe satisfacer un caso de uso se presentan a continuación en OCL.

Consulta 5.1: Devuelve el conjunto de casos de uso que extienden a otro

```
context CasoDeUso::Extensores() : Set(CasoDeUso):
   if self.extendido->size() > 0 then
     self.extendido.union(
          self.extendido
      ->collect(x | x.CasoDeUso.Extensores())
         )->flatten()
   else
           Set{}
   endif
```

Restricción 5.2: Un caso de uso extendido no puede ser usado por ningún caso de uso que lo extiende.

```
context CasoDeUso inv extiendePeroNoUsa:
  self.Extensores()
```

```
    ->forAll( x | x.incluye.CasoDeUso <> self )
```

Restricción 5.3: Un caso de uso no puede extenderse así mismo, ni tampoco contener ciclos en la relación

```
context CasoDeUso inv sinAutoreferencia1:
        self.extendido->collect(x | x.CasoDeUso )
        ->forAll( x | x <> self)
and
    not self.Extensores()->includes(self)
```

Restricción 5.4: La referencia de un punto de extensión pertenece a un evento del caso de uso extendido

```
context PuntoExtension::EventoReferencia inv :
  self.extendido.CasoDeUso.evento
  ->includes(self.evento)
```

Las mismas restricciones que se mostraron para la relación de extensión deben satisfacerse para la relación de inclusión.

5.3. Flujos de trabajo

Un flujo de trabajo o proceso de negocio es un conjunto de actores, tareas, recursos e información que permiten resolver algún problema en una organización. Un sistema de administración de flujo de trabajo sirve para coordinar los elementos mencionados y dar pauta a la ejecución de las tareas, en el momento oportuno, de forma automatizada. Las primeras ideas sobre los flujos de trabajo surgieron en los años 70, sin embargo, con la tecnología disponible en ese momento no fue posible la automatización de los trabajos de oficina, entre los precursores se encuentran [Zis77] y [Ell79]. Los primeros procesos de negocio automatizados fueron efectuados por motores de flujo de trabajos centralizados, sin embargo, con la llegada de Internet los procesos no sólo abarcan el interior de una organización (Intraorganizacionales), sino que pueden efectuarse entre actores situados en distintas organizaciones y lugares, es decir, los motores de flujo de trabajo realizan procesos distribuidos y que colaboran (Interorganizacionales).

A pesar de los cambios tecnológicos, los conceptos usados desde el surgimiento de estos sistemas permanece vigente y con pocas adiciones. En lo que resta de la sección se describirán los componentes de los sistemas de flujo de trabajo y también, se proporcionará un metamodelo, tomado de [Pen02].

5.3.1. Características

El objetivo básico de un flujo de trabajo es ejecutar las tareas, que lo conforman, según una especificación. Cada ejecución del proceso es conocida como *instancia* del proceso, caso, ciclo del proceso o asignación. La ejecución de una instancia de proceso tiene una identidad única y unos atributos llamados variables del proceso, atributos del caso o instancia, o parámetros de operación.

Según [AH04], un sistema de gestión de flujo de trabajo *Workflow Management System* (WfMS) tiene como principal objetivo administrar, de forma eficiente, el trabajo que debe aplicarse a una instancia del proceso. Las instancias del proceso pueden representar objetos reales o virtuales a los que se les aplica un trabajo, con lo cual obtienen un valor añadido. Las actividades aplicadas a una instancia del proceso significan algún tipo de trabajo, ya sea efectuado de forma manual o bien automática. Una tarea es una unidad de trabajo que es efectuada por un recurso. Un recurso es el nombre genérico de quién puede efectuar un trabajo de valor: personas, dispositivos, máquinas o sistemas, etc.

Las actividades o el trabajo aplicados a una instancia del proceso son determinados por el camino de ejecución que sigue el caso desde su creación hasta su fin. Los caminos que puede seguir son condicionalmente escogidos haciendo uso de las variables del proceso. Las condiciones y caminos que determinan el orden de ejecución de las actividades es conocido como flujo de control.

Los flujos de trabajo tienen otras dimensiones importantes, además del flujo de control: la infraestructura informática y la estructura organizacional [LR99]. La dimensión de infraestructura informática indica qué programas o procesos informáticos son ejecutados por una tarea. La estructura organi-

zacional representa a los actores y unidades organizacionales que ejecutan las tareas.

Un flujo de trabajo es un proceso constituido por tareas que requiere de un número determinado de recursos para la ejecución de las mismas (figura 5.3).

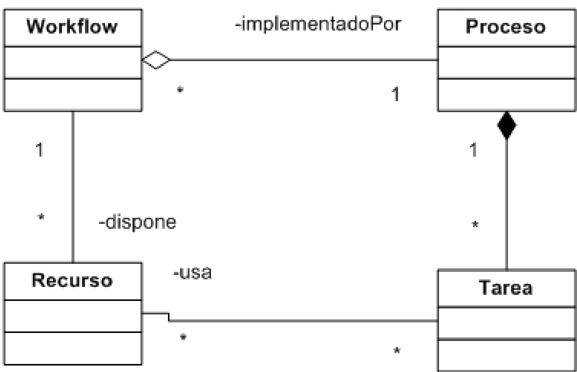


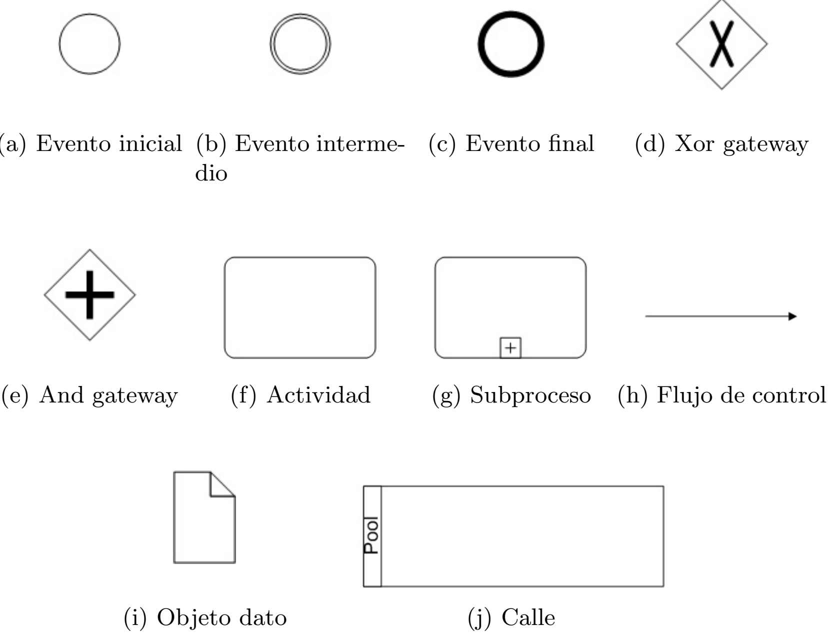
Figura 5.3: Componentes principales de un flujo de trabajo

5.3.2. Notación

La notación que utilizaremos para representar el flujo de control y de datos de un proceso es la propuesta por el *Object Management Group* (OMG) y denominada BPMN, la cual se ha convertido en el estándar *de facto* para especificar procesos de negocio. Antes, el estándar era el modelo de referencia de la *Workflow Management Coalition* [Mem99].

BPMN es muy flexible, y puede ser adaptada a las necesidades de modelado de muchos tipos de proceso. La semántica de la notación se explica en la siguiente sección. A continuación se describen los elementos notacionales utilizados.

- Eventos. Los eventos representan algo que ocurre durante la ejecución del proceso. Los eventos pueden alterar la ejecución del proceso disparando alguna actividad. Hay tres tipos de eventos: inicial, final e intermedio. Precisamente su nombre indica en qué momento ocurren. Los eventos se representan mediante círculos (figuras 5.4a, 5.4b y 5.4c).



(a) Evento inicial (b) Evento intermedio (c) Evento final (d) Xor gateway

(e) And gateway (f) Actividad (g) Subproceso (h) Flujo de control

(i) Objeto dato (j) Calle

Figura 5.4: Notación BPMN

- Gateways. Los *gateways* representan los puntos de convergencia y divergencia. Es decir, los puntos donde un camino se ramifica, o se une. Éstos se representan mediante rombos.

  - Xor-gateway. Representa un punto donde el flujo continúa por alguno de los caminos de salida, o un punto donde varios caminos se unen, y la ejecución continúa cuando el flujo ha sido completado en alguno de los caminos de entrada. Se representa con un rombo con una X en el interior (figura 5.4d).
  - And-gateway. Representa un punto donde el flujo continúa por todos los caminos de salida, o un punto donde todos los caminos de entrada se unen, y la ejecución continúa cuando el flujo ha sido completado por todos los caminos de entrada. Se representa por un rombo con un + en su interior (figura 5.4e).

- Actividad. Es la unidad de trabajo atómico, es decir, no puede ser dividida en partes. Las actividades pueden ser automáticas o manuales. Una actividad se representa por un rectángulo con esquinas redondeadas (figura 5.4f).

- Subproceso. Es un proceso que se ejecuta como parte de otro. Se representa con un rectángulo con las esquinas redondeadas y un símbolo de + (figura 5.4g).

- Flujo de control. Sirve para indicar el orden de ejecución de las tareas. Se representa mediante una flecha (figura 5.4h).

- Objeto dato. Representa la información transmitida entre tareas. Un objeto dato se asocia a una flecha del flujo de control y su símbolo es un folio con la esquina doblada (figura 5.4i).

- Calle. Es el conjunto de actividades del proceso asignado a un actor. Se representa mediante un rectángulo que contiene las actividades (figura 5.4j).

Con los elementos presentados de BPMN se pueden construir los 5 patrones básicos de flujo de control: secuencias, ejecución en paralelo (ramificación total), sincronización (unión total), ramificación exclusiva (ramificación parcial), y unión parcial. Además, la combinación entre ellos permite especificar la mayoría de los patrones de control [AH02]. En [WAD+06] se describe la expresividad y limitaciones de BPMN.

5.3.3. Metamodelo

El metamodelo de flujo de trabajo presentado en esta sección está basado en el desarrollado en [Pen02], sin embargo, ha sido completado con restricciones en OCL, que garantizan que los modelos construidos representen flujos de trabajo válidos. El metamodelo se representa como diagrama de clases MOF con restricciones OCL.

BPMN sólo se usa como notación, porque su metamodelo aún no ha sido aprobado por la OMG. Además, BPMN contiene muchos elementos y

permite construcciones que son difíciles de explicar e inclusive podrían ser incorrectas debido a su flexibilidad. Por lo anterior, en este trabajo se han restringido las construcciones permitidas, y se les ha dado una semántica que se explica en esta sección. También, se ha puesto atención a que los patrones básicos del flujo de control [AH02] se puedan realizar, pues son la base para construir patrones más complejos. La figura 5.5 muestra el metamodelo del flujo de trabajo. En las siguientes secciones se explicarán sus elementos, en primer lugar el flujo de control, después el flujo de datos, y finalmente, la semántica de los eventos.

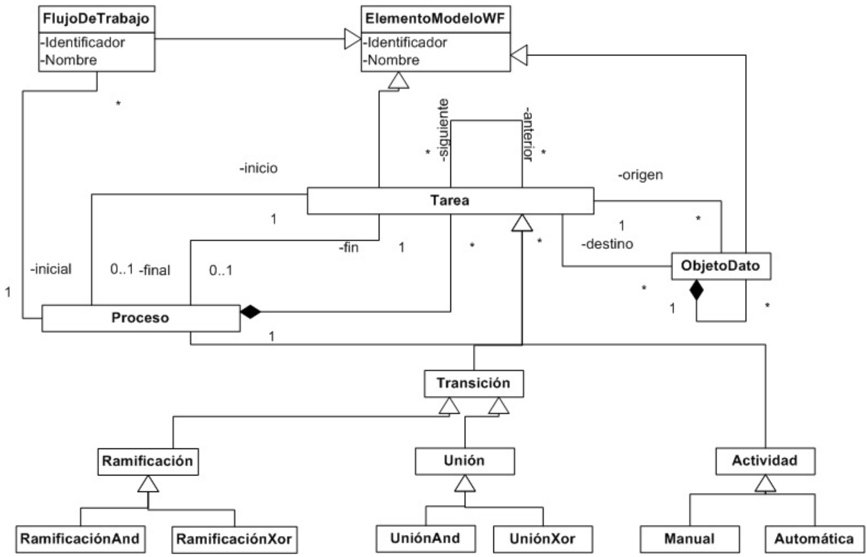


Figura 5.5: Metamodelo de flujo de trabajo

Flujo de control

El flujo de control de un proceso es el conjunto de tareas y caminos que las conectan. Un proceso tiene una tarea inicial, una tarea final y un conjunto de tareas intermedias. Todas las tareas son alcanzables desde la tarea inicial, y desde cualquier tarea se puede llegar a la tarea final. Las tareas son de tres tipos: actividades, subprocesos, y transiciones. Una actividad es una unidad de trabajo indivisible, la cual puede ser automática o manual. En

caso de ser manual debe asignarse a un actor, y también se deben indicar los recursos que requiere, los cuales son generalmente datos. Un subproceso tiene la misma estructura que un proceso, y es una agrupación de otras tareas, incluyendo otros subprocesos. Las tareas que integran un proceso deben mantener la relación que representa el flujo de control, es decir, contienen una lista con sus antecesores y sus sucesores. La figura 5.6 muestra un modelo MOF con las relaciones necesarias para representar el flujo de control de un proceso.

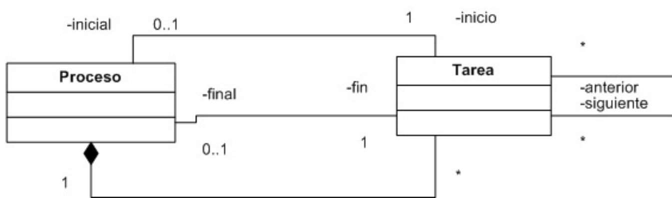


Figura 5.6: Flujo de control

La figura 5.7 muestra la taxonomía de tareas. Las tareas pueden ser actividades automáticas o manuales, procesos, o transiciones. Las transiciones son: ramificación parcial, unión parcial, ramificación total y unión total.

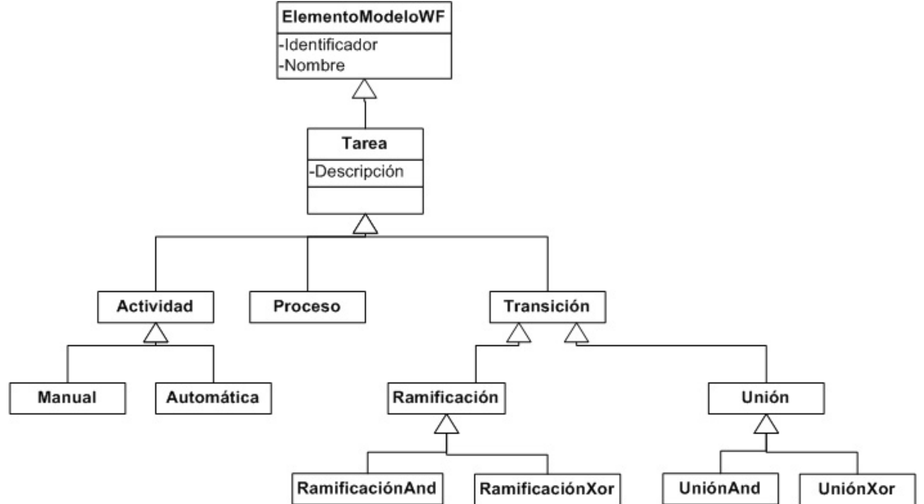


Figura 5.7: Taxonomía de tareas de un flujo de trabajo

Las transiciones sirven para ramificar o unir caminos del flujo de control. Un nodo de ramificación representa una decisión que se basa en el estado de las variables de proceso o en los datos de salida de una actividad. La ramificación parcial, representada con un *Xor-gateway* de BPMN, tienen un flujo de entrada y n flujos de salida (figura 5.8). Cada camino de salida tiene asociada una condición, esto quiere decir que el camino donde la condición es verdadera lleva a la siguiente actividad por ejecutar. En cambio, un nodo de unión asíncrona o unión parcial representa la función contraria a la ramificación parcial. La unión parcial, representada con un *Xor-gateway*, es un nodo con n caminos de entrada y un sólo camino de salida conectado a la actividad que se ejecutará en cuanto alguno de los flujos de control de entrada esté activo, es decir, alguna de las tareas precedentes fue terminada (figura 5.8).

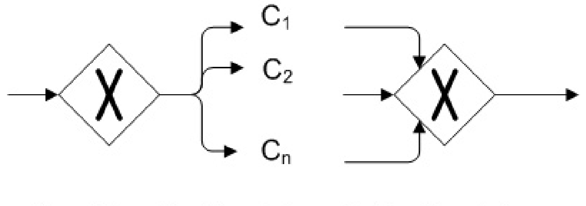


Figura 5.8: Ramificación y unión parcial

La ejecución en paralelo de tareas, representada con un *And-gateway*, tiene un flujo de control de entrada y n flujos de control de salida, y todas las tareas conectadas a los flujos de control de salida se efectuarán en un hilo de ejecución (figura 5.9). La reunión de caminos y la ejecución de la siguiente tarea, hasta que todos los flujos de control de entrada han terminado, se efectúa mediante una unión total, representada por un *And-gateway*, que tiene n flujos de control de entrada y uno de salida (figura 5.9).

A lo largo de esta sección se han mencionado las restricciones que debe de satisfacer el flujo de control. A continuación se presenta su definición en OCL.

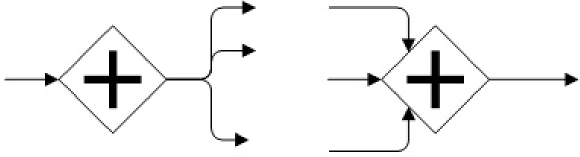


Figura 5.9: Ramificación y unión total

Restricción 5.5: Las tareas que no son de ramificación sólo pueden tener un nodo sucesor.

```
Context Tarea inv unSucesor:
        (not self.oclIsTypeOf(Ramificación))
                implies self.siguiente->size() <= 1
```

Restricción 5.6: Los nodos que no son de unión sólo pueden tener un nodo antecesor.

```
Context Tarea inv unAntecesor:
        not self.oclIsTypeOf(Unión) implies
        self.anterior->size() <= 1
```

Restricción 5.7: Si una tarea es la inicial de un proceso, entonces no tiene antecesores.

```
Context Tarea inv inicialSinAntecesor :
    self.inicial->size() = 1
          implies
        self.anterior->size() = 0
```

Restricción 5.8: Si una tarea es la final de un proceso, entonces no tiene sucesores.

```
Context Tarea inv finalSinSucesor:
        self.final->size() = 1
        implies
        self.siguiente->size() = 0
```

Restricción 5.9: Los nodos que pertenecen a un proceso no pueden estar desconectados.

```
Context Tarea inv conectado:
          (self.anterior->size() = 0
            implies
                self.inicial->size()  = 1
           )
        and
          (self.siguiente->size()= 0
           implies
           self.final->size() = 1 )
```

La siguiente consulta devuelve el conjunto de tareas que son sucesoras de las tareas del conjunto origen, y que además no pertenecen a él. Para encontrar las tareas sucesoras de un nodo esta operación se invoca con los parámetros origen = {nodo}, y delta = {}.

Restricción 5.10: Consulta el conjunto de sucesores de un tarea.

```
Context Tarea def
  sucesor(origen: Set(Tarea), delta: Set(Tarea))
: Set(Tarea) =
        let
           sucesores :
                Set(Tarea) =
                        origen->collect(n|n.siguiente)
                        ->flatten()
                        ->select( s | origen->excludes(s))
                        ->asSet()
        in
      if sucesores = Set{} then
             origen
         else
             origen.union(
                  Tarea::sucesor( origen.union(delta),
                                  sucesores)
                                               )
         endif
```

Restricción 5.11: Todas las tareas de un proceso son sucesoras de la tarea inicial.

```
Context Proceso inv sucesorDelInicio:
let
   sucesores : Set(Tarea) =
        Tarea::sucesor( Set{inicial}, Set{})
in
   self.tarea
   ->forAll(x | sucesores->includes(x))
```

Restricción 5.12: Todos los nodos deben tener como sucesor al nodo final del proceso al que pertenecen.

```
Context Tarea inv antecesorDelFinal:
 self <> self.Proceso.final
        implies
 Tarea.sucesor(Set{self},{})
                ->including(self.Proceso.final )
```

Eventos

En BPMN existen tres clases de eventos: de inicio, fin e intermedios. La semántica de los eventos de inicio y fin está bien definida: cada proceso o subproceso tiene exactamente una tarea inicial indicada por el evento de inicio, y también una final que se conecta al evento de finalización.

En cambio, el significado de los eventos intermedios de BPMN no es claro, por lo cual, se propone que se interpreten como un or-exclusivo pospuesto. En MDHDM los eventos intermedios se usan para modelar los eventos originados por el usuario. Un evento originado por el usuario ocurre durante la ejecución de una actividad, por tanto, éstos son parte de la información de salida de la actividad. Al final de la actividad se evalúa, si ocurre el evento, en tal caso se sigue con la actividad indicada por el evento. Del mismo modo se pueden incorporar los eventos de excepción y temporización.

En la figura 5.10 se muestra un ejemplo. Un proceso que consta de tres actividades secuenciales, la primera (Actividad-A) se indica por el evento inicial, y la última (Actividad-C) se conecta al evento final. En la Actividad-

B y la Actividad-C puede ocurrir un evento que cancela su ejecución y que dirige el flujo de control a la Actividad-A.

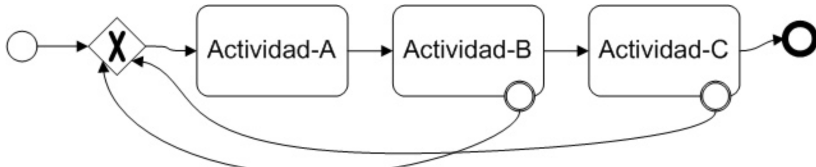


Figura 5.10: Proceso con eventos intermedios

En BPMN hay otros tipos de eventos como los de enlace, reglas, compensación y cancelación. Los dos primeros no son incluidos en MDHDM, porque su definición en el estándar es ambigua, y los dos últimos resultan redundantes con el evento de excepción, del que serían una especialización.

También, en MDHDM se utilizan los eventos intermedios como puntos de sincronización entre las proyecciones de los usuarios. Es decir, un evento intermedio sirve para activar la ejecución de las tareas correspondientes a otro actor. Sin embargo, este uso de los eventos intermedios no es parte del modelado conceptual, sino una de las tareas del diseño navegacional.

Flujo de datos

El flujo de datos representa la información que fluye a través de las actividades a lo largo de la ejecución de un proceso. La figura 5.11 muestra el fragmento del metamodelo correspondiente al flujo de datos. La información que pasa a través de una transición y conecta dos actividades es llamada objeto dato, y está compuesto por atributos, referencias a otros objetos dato y objetos dato internos.

No es obligatorio que una tarea necesite o genere información. Sin embargo, si produce información y no es la tarea inicial o final de un subproceso entonces, puede entregarla a otra tarea que pertenece a sus sucesores. La tarea inicial puede recibir información, pero ésta es gestionada por el proceso o por el subproceso, y lo mismo ocurre para las tareas finales. Además de los objetos dato cada proceso puede tener variables locales, las cuales son leídas o escritas por sus actividades. Las restricciones OCL necesarias son:

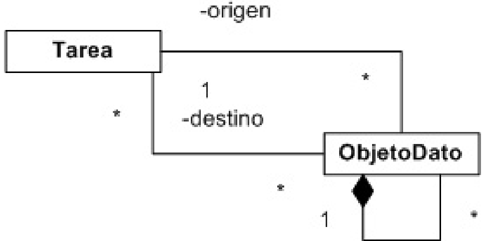


Figura 5.11: Flujo de datos

Restricción 5.13: Si el destinatario de los datos es la tarea inicial de un subproceso entonces el origen es el subproceso.

```
context DataFlow inv :
    self.target.inicial->size() = 1
  implies
    self.source.oclAsType(Proceso).inicio = self.target
```

Restricción 5.14: Si el origen de los datos es la tarea final de un subproceso entonces el destinatario es el subproceso.

```
context Transition inv :
    self.source.final->size() = 1
  implies
        self.target = self.source.oclAsType(Proceso).final
```

Restricción 5.15: Si la tarea tiene sucesores entonces éstos deben recibir los datos.

```
context DataFlow inv :
    self.source.siguiente->size() > 0
  implies
    self.source.siguiente->exists(x |self.target = x)
```

5.4. Modelo estructural

El modelo estructural representa la vista estática del sistema software, es orientado a objetos, y está compuesto por clases y las relaciones entre ellas.

5.4.1. Características

Las clases representan a entidades reales del dominio del problema. Una clase es una abstracción de un conjunto de instancias que presentan las mismas características de comportamiento y estado. Una clase tiene un nombre, atributos y servicios. El nombre de la clase debe ser una palabra representativa de las características de los objetos de la clase. Los atributos son las propiedades o características de los objetos de una clase; mediante ellos se puede describir a cualquier objeto. Los valores que tienen los atributos en un instante dado describen el estado del objeto en ese momento. El comportamiento de los objetos de una clase está definido por los servicios o métodos que ofrece, y su ciclo de vida es especificado, por lo general, mediante un diagrama de estados.

Las relaciones entre las clases pueden ser de diferentes tipos: asociación, agregación, composición, y especialización/generalización. La asociación señala que dos elementos tienen una relación. La relación de composición indica que un elemento componente es parte de otro, llamado compuesto, y sus existencias están vinculadas, es decir, el objeto componente no tiene una existencia independiente del compuesto. La relación de agregación indica que un elemento componente es parte de otro (compuesto), sin embargo, en este caso el componente puede tener una existencia independiente del compuesto. Una relación de especialización/generalización indica que una clase desciende o especializa el comportamiento de sus clases antecesoras. Para representar el modelo estructural usamos la notación de UML.

5.4.2. Metamodelo

El metamodelo del modelo estructural propuesto (figura 5.12) coincide con el de MOF [Gro03b], con la única diferencia que las asociaciones del propuesto son bidireccionales. De modo que las asociaciones son navegables en ambos sentidos al expresar restricciones OCL, y tampoco se requiere definir una referencia inversa para especificar completamente una asociación entre dos clases. MOF y el metamodelo presentado tienen en común que no cuentan con clases-asociación ni asociaciones n-arias, a diferencia de otros

modelos estructurales como el de UML, así se evita definir su semántica.

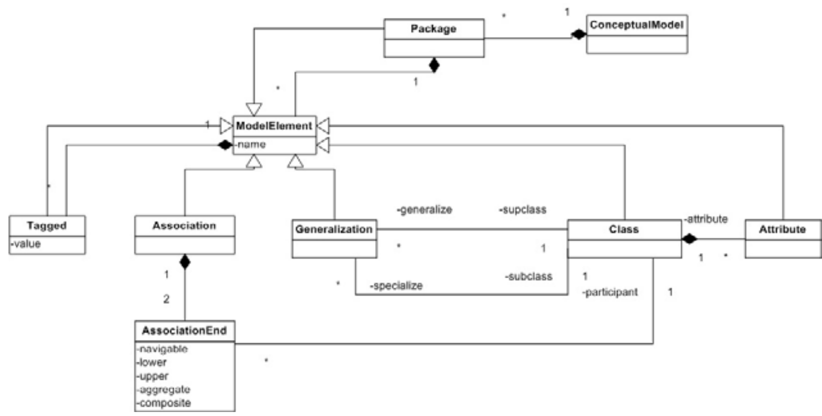


Figura 5.12: Metamodelo estructural

Los paquetes agrupan a elementos del modelo y sirven para organizarlos. Una clase se compone de n atributos. Las clases y atributos se identifican por un nombre. Los nombres de las clases deben ser irrepetibles en el contexto del paquete al que pertenece la clase, mientras que los atributos son únicos en el contexto de su clase compuesta.

Las asociaciones pueden tener un identificador, pero no es obligatorio. Cuando hay varias asociaciones que relacionan las mismas clases, cada extremo de las asociaciones debe tener un nombre diferente. Las asociaciones en este metamodelo sólo son binarias como en MOF. Las asociaciones de composición y agregación se indican en uno de los extremos de la asociación, es decir, la clase *Association-end* tiene como atributos booleanos: *aggregate y composite*. Sólo uno de los extremos de una asociación puede ser compuesto o agregado, pero no ambos. Los atributos *lower y upper* de un *Association-end* sirven para indicar la cardinalidad mínima y máxima en ese extremo. Si un extremo es compuesto entonces tiene como cardinalidad mínima 1. El atributo navegable cuando es verdadero indica que los objetos de la clase asociada conocen a los elementos con los que se relacionan. En una asociación debe haber al menos un extremo navegable.

Una generalización/especialización relaciona a dos clases, una corres-

ponde a la superclase o supertipo, y la otra a la subclase o subtipo. Una generalización/especialización debe ser única en el modelo, además no deben formarse ciclos, es decir, no se puede encontrar una clase que sea su propia superclase.

El metamodelo de UML [Gro04], a diferencia del propuesto, contiene el elemento *AssociationClass* que es una especialización de *Association* y de *Class*. Una clase asociación es una abstracción de una asociación que puede tener sus propios atributos, operaciones, y comportamiento, tal como una clase. Otra diferencia es que en UML las asociaciones pueden tener más de dos *Association-end*.

En este metamodelo no se incluyen los métodos porque sólo aparecerían de forma nominal, y el modelado de su función tampoco es abordado, debido a que no son relevantes para la navegación. A continuación se presentan las restricciones que debe satisfacer el modelo estructural.

Restricción 5.16: Una clase no puede tener atributos con el mismo nombre.

```
Context Class inv  nombreÚnico:
        self.attribute->isUnique(x | x.name)
```

Restricción 5.17: En un paquete no pueden existir dos clases con el mismo nombre.

```
Context Package inv nombreÚnico  :
        self.class->isUnique(x | x.name)
```

Restricción 5.18: En un paquete no puede haber dos asociaciones con el mismo nombre.

```
Context Association inv nombreÚnico:
        self.association->isUnique(x | x.name)
```

Restricción 5.19: Los extremos de una asociación tienen un nombre único dentro de la asociación.

```
Context Association inv AEndÚnico:
        self.associationEnd->isUnique(x | x.name)
```

Restricción 5.20: Sólo un extremo de la asociación puede ser compuesto.

```
Context Association inv unaComposición:
        self.associationEnd->select(x|x.composite)
                           ->size() <= 1
```

Restricción 5.21: Sólo un extremo de la asociación puede ser agregado.

```
Context Association inv unaAgregación :
        self.associationEnd->select(x|x.aggregation)
                           ->size() <= 1
```

Restricción 5.22: Un extremo de asociación no puede ser compuesto y agregado al mismo tiempo.

```
Context AssociationEnd inv :
        (self.composite implies not self.aggregate)
  and
    (self.aggregate implies not self.composite)
```

Restricción 5.23: Una generalización/especialización no puede ocurrir más de una vez.

```
Context Generalization inv Única :
not Generalization.allInstances()->
  exists(x | x.supClass = self.supClass and
             x.subClass = self.subClass and
             x <> self)
```

Consulta 5.24: Devuelve el conjunto de clases que son superclases de otra.

```
Context Class def: superClasses(origen: Class)
: Set(Class) =
if self.supertype->isEmpty() then
  Set{}
else
  let supercs : Set(Class) =
          self.supertype->collect(x|x.supClass)
                        ->asSet()
  in
        supercs.union(
                supercs->select( y | y <> origen)
```

```
                  ->collect(z|z.superClasses(origen))
                  ->flatten()->asSet() )
endif
```

Restricción 5.25: Las generalizaciones/especializaciones no pueden construir ciclos.

```
Context Class inv sinCiclos:
   not self.superClasses(self)->includes(self)
```

5.5. Correspondencias entre metamodelos

Una vez presentado el metamodelo de casos de uso, el de proceso y el estructural se pueden definir las correspondencias entre ellos. En la sección 5.5.1 se explica cómo a partir de un modelo de casos de uso se puede obtener la versión inicial del proceso efectuado en una organización. Finalmente, en la sección 5.5.2 se describe cómo obtener parte del modelo estructural desde el modelo de proceso.

5.5.1. Correspondencias de casos de uso a flujo de trabajo

La correspondencia entre el metamodelo de casos de uso extendidos y el metamodelo de flujo de trabajo se puede establecer debido a que comparten muchas similitudes, y también, porque los caso de uso extendidos tienen propiedades adicionales. Las correspondencias presentadas se basan en las de [Pen02].

Cada caso de uso elemental es transformado en una actividad, y sus propiedades son transformadas en las de la actividad, de acuerdo a la tabla 5.2.

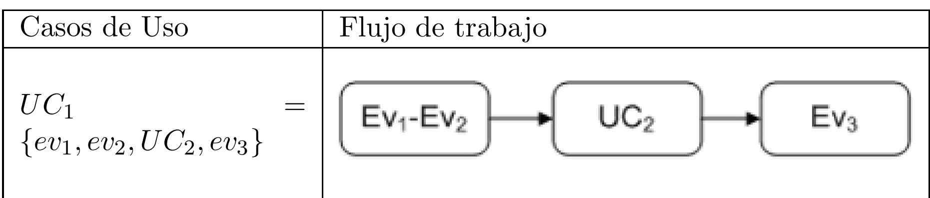
Los casos de uso complejos se transforman en subprocesos. Cuando un caso de uso presenta una relación de inclusión corresponde a una secuencia de actividades y subprocesos en el flujo de trabajo. Cada secuencia de eventos se transforma en una actividad, y el caso de uso empleado si es elemental en otra actividad, si no será otro subproceso. Si un caso de uso UC_1 define la siguiente relación de eventos: ev_1, ev_2, UC_2, ev_3, el caso de uso efectúa los

Tabla 5.2: Equivalencia de caso de uso a actividad

Caso de uso elemental	Actividad
Nombre del Caso de Uso	Nombre de la Actividad
Tipo de Proceso	Tipo de Actividad
Datos de entrada	Datos de entrada
Datos de salida	Datos de salida
Actores	Actores
Flujo de Eventos	Flujo de Acciones

eventos ev_1 y ev_2, después efectúa el caso de uso UC_2, y enseguida el evento ev_3. Los eventos ev_1, ev_2 se transforman en una actividad conectada por un flujo de control a la actividad o subproceso derivados del caso de uso UC_2, y éste se conecta por un flujo de control a una actividad que corresponde al evento ev_3. La tabla 5.3 muestra el ejemplo de relación de uso que se ha explicado.

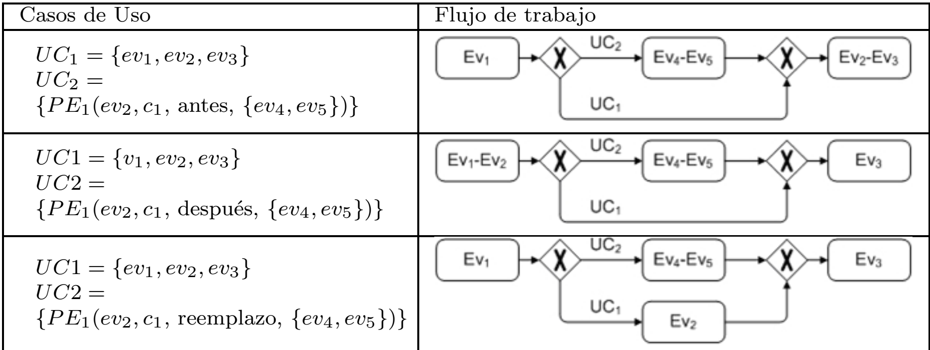
Tabla 5.3: Correspondencia entre una relación de inclusión entre caso de uso y un flujo de trabajo

Casos de Uso	Flujo de trabajo
$UC_1 = \{ev_1, ev_2, UC_2, ev_3\}$	

En el caso de relaciones de extensión, se genera un subproceso a partir del caso de uso que es extendido. Los puntos de extensión se transforman en ramificaciones XOR antes o después del evento indicado dependiendo del tipo de punto de extensión. La condición del XOR corresponde a la condición del punto de extensión. Por ejemplo, se tiene un caso de uso con la relación de eventos: $UC_1 = \{ev_1, ev_2, ev_3\}$, y otro caso de uso UC_2, que define un punto de extensión $PE1(ev_2, antes) = \{ev_4, ev_5\}$, en este caso un *Xor-gateway* de ramificación es insertado antes del evento ev_2, una de la ramas de salida del *Xor-gateway* es hacia la actividad que efectúa los eventos $\{ev_4, ev_5\}$, y otra de las ramas se conecta a un *Xor-gateway* de unión antes del evento ev_2. La

tabla 5.4 muestra gráficamente la correspondencia entre puntos de extensión y flujos de trabajo.

Tabla 5.4: Equivalencias entre una relación de extensión entre caso de usos y un de flujo de trabajo



Casos de Uso	Flujo de trabajo
$UC_1 = \{ev_1, ev_2, ev_3\}$ $UC_2 =$ $\{PE_1(ev_2, c_1, \text{antes}, \{ev_4, ev_5\})\}$	Ev_1 UC_2 Ev_4-Ev_5 Ev_2-Ev_3 UC_1
$UC1 = \{v_1, ev_2, ev_3\}$ $UC2 =$ $\{PE_1(ev_2, c_1, \text{después}, \{ev_4, ev_5\})\}$	Ev_1-Ev_2 UC_2 Ev_4-Ev_5 Ev_3 UC_1
$UC1 = \{ev_1, ev_2, ev_3\}$ $UC2 =$ $\{PE_1(ev_2, c_1, \text{reemplazo}, \{ev_4, ev_5\})\}$	Ev_1 UC_2 Ev_4-Ev_5 Ev_3 UC_1 Ev_2

Hay que tener en cuenta, durante la transformación, que la descripción de los eventos, de un caso de uso, puede incluir condiciones y ciclos, y éstos deben plasmarse también en el modelo de proceso.

Uno de los pasos para derivar el flujo de trabajo, es aplicar las equivalencias entre un caso de uso y un flujo de trabajo que se acaban de explicar. Los demás pasos necesarios son presentados en [Pen02] y se listan a continuación.

1. **Modelar la estructura organizacional.** Se construye el organigrama con las unidades organizacionales.

2. **Crear los casos de uso.** Por cada tarea que se efectúa en la organización se construye un caso de uso.

3. **Relacionar los casos de uso.** Se identifican las relaciones de extensión e inclusión entre casos de uso. Esta tarea la efectúa el responsable de cada unidad organizacional.

4. **Generar el flujo de trabajo.** Se aplican las equivalencias descritas previamente.

5. **Refinar el flujo de trabajo.** Si existen tareas desconectadas, se completa el flujo de control y se comprueba el modelo obtenido.

5.5.2. Correspondencias del flujo de trabajo al modelo estructural

Desde el modelo del flujo de trabajo se pueden obtener algunas entidades del modelo estructural puesto que están presentes en el flujo de datos.

1. Se identifica la clase de cada objeto dato.

2. Se determina si un objeto dato es compuesto. Es decir, si agrupa a dos objetos dato a través de una relación de agregación o composición. La composición se puede identificar si el objeto dato contiene a otro objeto dato en su interior y no es factible dividirlos, es decir, su existencia no puede separarse.

3. Las agregaciones y asociaciones se identifican a través de los identificadores de otros objetos que son parte del objeto dato analizado.

4. Se determinan las propiedades de los objetos dato y de ellos se derivan los atributos de la clase.

5. Se examina si los objetos dato tienen atributos y asociaciones en común, en tal caso, se determina que pertenecen a una jerarquía de clases.

5.5.3. Ejemplo

La figura 5.13 muestra los casos de uso con los que interactúa un usuario durante una compra en una librería en línea. El caso de uso del cliente consiste en efectuar una *Compra*. Los casos de uso *SeleccionarLibro* y *AgregarLibro* son incluidos en el caso de uso *Compra*. *SeleccionarLibro* se ejecuta para localizar algún libro de interés para el usuario. *AgregarLibro* consiste en añadir el libro seleccionado al carro de compras. En tanto que los casos de uso *ModificarCarro* y *CerrarCompra* se pueden ejecutar si la condición *el carro de compras tiene productos* es verdadera, por lo que se añaden al caso de uso *Compra* a través de puntos de extensión. *ModificarCarro* permite modificar el contenido del carro de compras, y *CerrarCompra* sirve para ingresar los datos de pago y entrega del pedido.

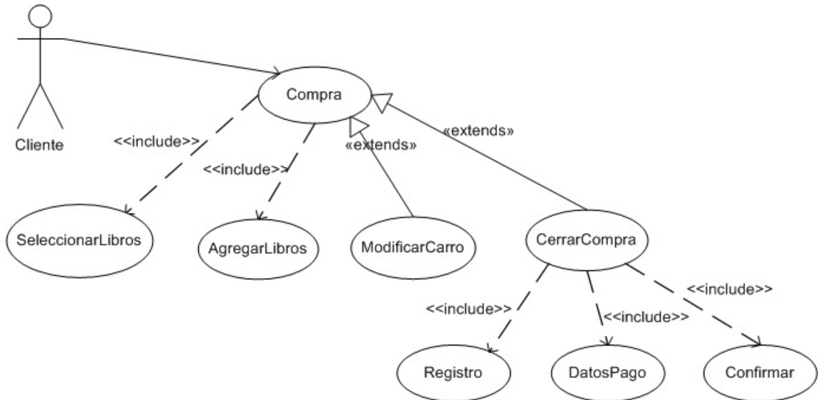


Figura 5.13: Casos de uso de la librería electrónica

De acuerdo con las correspondencias explicadas anteriormente, el proceso obtenido es el siguiente. Del caso de uso *Comprar* se deriva un subprocesos porque es extendido e incluye a otros. El caso de uso *SeleccionarLibro* consiste navegar a través de las colecciones de libros o utilizar el formulario de búsqueda hasta seleccionar un libro y corresponde a una actividad manual. El caso de uso *AgregarLibro* contiene el evento de añadir el libro al carro de compras, lo cual es realizado de forma automática, por lo que corresponde a una actividad automática. Los casos de uso *ModificarCarro* o *CerrarComprar* se pueden ejecutar si el carro de compras no está vacío.

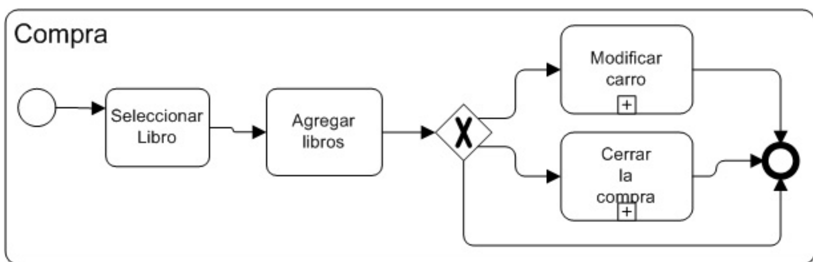


Figura 5.14: Proceso derivado del caso de uso compra

El caso de uso *ModificarCarro* está conformado por los siguientes eventos *presentar el carro de compras*, y después el cliente decide entre *quitar un libro del carro*, *cambiar la cantidad de libros* o no cambiar nada. Por tanto,

el caso de uso *ModificarCarro* es un subproceso donde la actividad iniciales presentar el carro de compras y después se ejecuta alguna de las otras dos actividades automáticas.

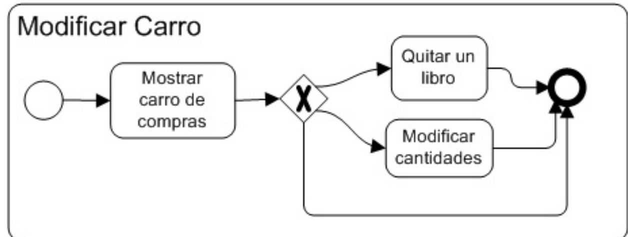


Figura 5.15: Proceso derivado del caso de uso modificar carro

Del casos de uso *CerrarComprar* se deriva un subproceso porque incluyen a otros. El caso de uso *Registro* se compone de los eventos: *presentar formulario de registro*, *validar el registro del usuario*, si el registro es correcto entonces, permite la ejecución del caso de uso *DatosPago*. El caso de uso *DatosPago* presenta un formulario para la captura de la información del pago y envío, que se almacenan. Una vez terminado el caso de uso *DatosPago* se puede confirmar la compra o cancelarla, si el usuario decide cancelarla entonces, se eliminan los datos de pago y envío, si el usuario acepta entonces, se crea una orden de envío.

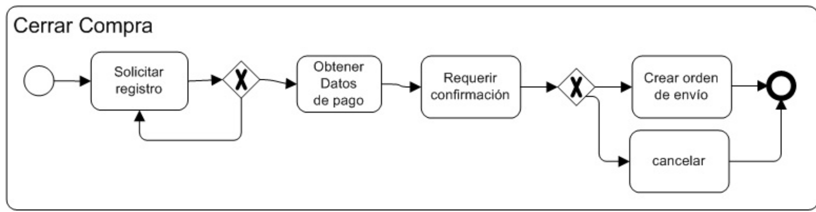


Figura 5.16: Proceso derivado del caso de uso cerrar compra

El ejemplo también muestra que la ocurrencia de eventos en un caso de uso está regida por condiciones, las cuales se transformaron en *Xor-gateways*. El proceso obtenido a través de las correspondencias es incompleto y debe ser refinado por los analistas, en este caso hacen faltan algunos caminos que permitan repetir las tareas de seleccionar libros y de edición del carro de

compras.

5.6. Conclusiones

En este capítulo se han presentado los modelos y técnicas necesarias para analizar el dominio del problema. Para poder realizar especificaciones correctas de estos sistemas se ha presentado el metamodelo de: casos de uso, el de flujos de trabajo, y el estructural. Se han definido como metamodelos MOF más restricciones OCL.

El modelo de casos de uso permite analizar la funcionalidad del sistema, tiene como ventajas que es muy conocido por los analistas de sistemas, y que en su especificación se emplea el lenguaje del usuario.

El flujo de trabajo proporciona una visión global del proceso efectuado en la organización y brinda una especificación de alto nivel. Además, puede ser usada por los sistemas de gestión de flujos de trabajo como entrada para su posterior ejecución. El modelo de flujo de trabajo es representado en BPMN.

El modelo estructural presenta las entidades y relaciones del dominio en cuestión. Las entidades y relaciones participan en el flujo de trabajo como información requerida para efectuar una actividad o como producto de ellas, es decir, conforman el flujo de datos. El modelo estructural contiene clases y sus relaciones, y es similar al modelo de clases de MOF.

El ciclo de análisis de un sistema se inicia con la especificación de los casos de uso que realiza cada unidad organizacional. Las unidades organizacionales de mayor nivel efectúan casos de uso complejos, con relaciones de inclusión o extensión hacia los casos de uso realizados por las unidades organizacionales de menor nivel. A partir de la especificación completa de casos de uso se obtiene una versión inicial del flujo de trabajo efectuado en la organización, aplicando las relaciones de transformación que se han presentado. En cambio, el modelo estructural se puede empezar a diseñar a partir de los objetos dato del modelo de proceso, porque éstos permiten identificar algunas clases.

Capítulo 6

Diseño Navegacional

6.1. Introducción

En MDHDM el diseño navegacional sirve para definir los caminos navegacionales, por cada uno de los actores del proceso, de acuerdo con la vista que tiene un actor del proceso y sus datos. Como se verá a lo largo de este capítulo, la navegación la dividiremos en dos tipos: la guiada por el proceso, que constituye la columna vertebral del sistema, y la navegación a través de las relaciones entre datos. Los primeros tienen como efecto la ejecución de las tareas y el avance del estado del proceso, mientras que los segundos permiten navegar a través de las asociaciones entre datos.

El capítulo se estructura como sigue: en la sección 6.2 se define el modelo navegacional; en la sección 6.3 su metamodelo y sus restricciones. La sección 6.4 indica los pasos del diseño navegacional para obtener el modelo navegacional de un actor. En la sección 6.5 se explica cómo efectuar la proyección del proceso para un actor determinado. Después en la sección 6.6 se aborda cómo obtener el modelo navegacional a partir de la proyección. Finalmente, se presenta en la sección 6.7 cómo obtener el modelo navegacional derivado de la estructura, y en la sección 6.8 las conclusiones.

6.2. Modelo navegacional

Un sistema de hipermedia se compone de nodos conectados a través de enlaces. Un nodo se compone de información (texto, imágenes, vídeos, etc.) y enlaces vinculados a la información. Seguir un enlace, o *navegar*, genera un cambio de contenido en la interfaz que mostraría el contenido del nodo apuntado por el enlace. El modelo navegacional sirve para indicar los caminos que existen de un nodo a otro, y para definir qué información de un nodo se muestra a cada actor.

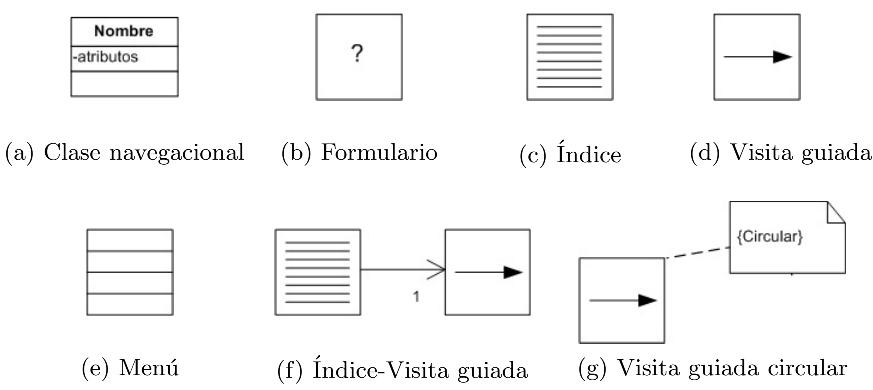
En el capítulo 2 se presentaron varios métodos de diseño de hipermedia y sus notaciones para representar el modelo navegacional. En este trabajo se ha optado por usar la notación de UWE [HK01] porque se basa en UML. No existe una notación estándar para definir modelos navegacionales y UWE al menos utiliza UML que es un estándar muy conocido. La notación en MDHDM es similar a UWE porque utiliza los mismos símbolos para los siguientes conceptos: clases navegacionales, estructuras de acceso y las asociaciones navegables. Sin embargo, los metamodelos subyacentes en UWE y MDHDM son diferentes. La distinción de enlaces débiles y fuertes, los enlaces con memoria y las vistas de tarea no se encuentran en UWE. A continuación se presenta el modelo navegacional de MDHDM.

6.2.1. Nodos

Los nodos representan la información que puede ser indicada por un enlace o servir como origen de los enlaces. Los tipos de nodos disponibles en MDHDM se muestran en la figura 6.1 y se describen a continuación.

Clases navegacionales

Una *clase navegacional* (figura 6.1a) representa una vista de un conjunto de clases conceptuales o de objetos dato del proceso. Los atributos de las clases navegacionales se derivan, generalmente, de los atributos correspondientes de las clases conceptuales, aunque es posible tener algunos que sean resultado de una fórmula expresada en OCL. Las clases navegacionales, al



(a) Clase navegacional (b) Formulario (c) Índice (d) Visita guiada

(e) Menú (f) Índice-Visita guiada (g) Visita guiada circular

Figura 6.1: Tipos de nodo

igual que las conceptuales, pueden especializar/generalizar a otras a través de relaciones de herencia. Los *objetos navegacionales* son instancias de las clases navegacionales.

Estructuras de acceso

Las *estructuras de acceso* (figura 6.1b a 6.1g) sirven para facilitar la navegación de una clase a otra. Básicamente, reemplazan a las asociaciones del modelo estructural, cuando son de cardinalidad 1-N o M-N. Los tipos de estructuras disponibles son:

Formulario. Permite la captura de datos, por ejemplo, para completar el criterio de selección de otra estructura de acceso, o bien, como entrada para efectuar un actividad del proceso (figura 6.1b).

Índice. Los índices proveen acceso directo a un conjunto de objetos navegacionales. Además, se pueden construir dinámicamente a partir del criterio de selección indicado (figura 6.1c).

Visita guiada. Las visitas guiadas proveen acceso secuencial a un conjunto de objetos navegacionales, es decir, son una lista ordenada de objetos (figura 6.1d). Esta lista también puede construirse mediante un criterio

de selección. Desde el primer elemento de la lista se puede pasar a otro usando enlaces de avance y retroceso.

Menú. Es un conjunto de enlaces navegacionales. A diferencia de índices y vistas guiadas, su conjunto de enlaces es fijo (figura 6.1e).

Índice-Visita guiada. Un índice sirve como elemento de entrada a la lista de objetos de la visita guiada. En ese caso ambos deben contener el mismo conjunto de elementos. Además, la visita guiada presenta un enlace de vuelta al índice (figura 6.1f).

Visita guiada circular. Es un tipo de visita guiada en donde el último elemento de la lista presenta un enlace al primero. Se marca la visita guiada con la etiqueta *Circular* (figura 6.1g).

6.2.2. Enlaces

Como se ha mencionado, un enlace es una conexión entre nodos. En UWE los enlaces son las *asociaciones navegables*, las cuales tienen dos extremos que pueden tener asociados un rol, una cardinalidad y una dirección de navegación (como una asociación de UML). La dirección de la navegación está indicada mediante una flecha en los extremos de la asociación. En MDHDM las asociaciones navegables son llamadas *enlaces débiles*.

Enlaces débiles o de datos

Los enlaces débiles provienen de las asociaciones entre datos del modelo estructural y no tienen ningún efecto sobre el proceso que efectúa un actor. Un modelo navegacional sólo conformado por nodos y enlaces débiles se denomina modelo navegacional débil. La figura 6.2 muestra ejemplos de enlaces débiles con diferentes cardinalidades y dirección de navegación. Un enlace (figura 6.2a) que permite navegar de un objeto navegacional a otro se representa como un enlace débil con cardinalidad 1-1 entre dos clases, además, la navegabilidad podría ser en ambos sentidos (figura 6.2b). Si de un objeto navegacional se puede navegar a n objetos de otra clase (figura

6.2c), entonces esa colección de enlaces debe ser parte de una estructura de acceso. Por tanto, el objeto origen estará forzosamente relacionado con una estructura de acceso, es decir, habría un enlace débil con cardinalidad 1-1 entre la clase origen y la estructura de acceso, y un enlace débil entre la estructura de acceso y la clase destino.

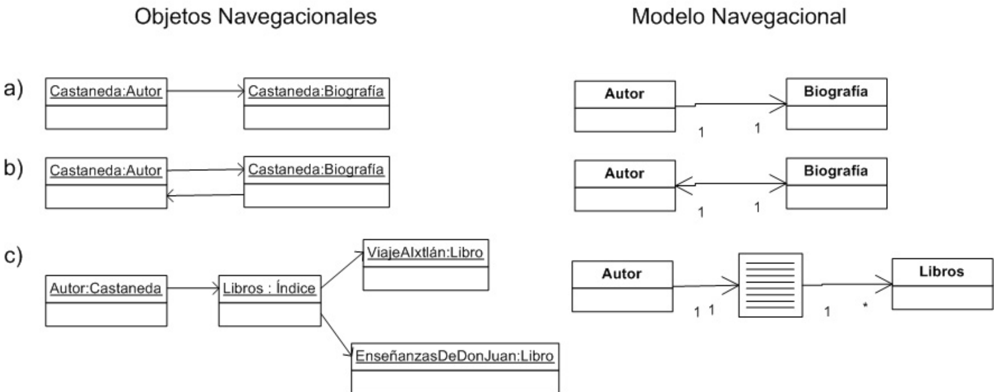


Figura 6.2: Interpretación de los enlaces débiles

Nodos compuestos Un nodo que reúne a otros es un nodo compuesto. Para indicar que un nodo se muestra junto con otro se añade al enlace débil el símbolo de composición de UML (rombo negro). La figura 6.3 indica que cuando se muestra un objeto navegacional *autor*, éste incluye un índice, con enlaces a objetos de la clase *libro*, mientras que esto no ocurre en el caso 6.2c.

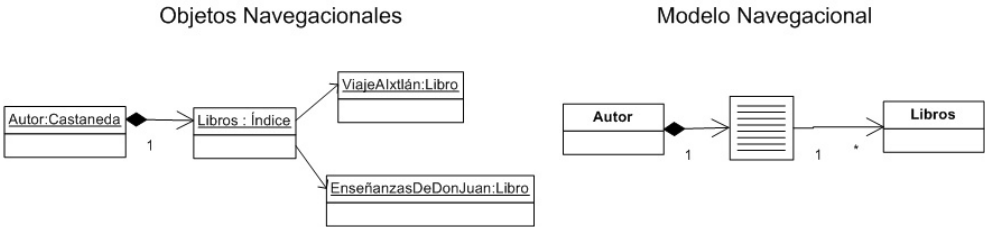


Figura 6.3: Composición de nodos

Información contextual Los enlaces débiles transmiten información contextual que viaja de un nodo a otro, lo cual sirve para determinar qué objeto navegacional debe mostrarse en el nodo destino. Cuando un enlace tiene como origen una clase navegacional entonces se transmite su identificador de objeto. Si el receptor es otra clase navegacional entonces podrá mostrar a la instancia asociada al identificador de objeto recibido, y si el receptor es un índice entonces mostraría el conjunto de nodos relacionados con el identificador. En el caso de índices, los enlaces transmiten el identificador de objeto del elemento seleccionado por el usuario. Los enlaces de las visitas guiadas transmiten el identificador de objeto del elemento que están mostrando.

Enlaces con memoria En MDHDM se puede memorizar la información contextual enviada por un nodo. Un enlace débil puede etiquetarse con la marca *(Memoriza, Nombre)*, que indica que la información contextual enviada por el enlace es almacenada e identificada por *Nombre*. La recuperación de dicha información puede ocurrir en cualquier momento y añadirse a algún enlace etiquetado con la marca de *(Recuerda, Nombre)*. Por ejemplo, en la figura 6.4 se memoriza la información contextual enviada por el enlace entre la clase navegacional *NClass-A* y el índice. En los nodos de la clase navegacional *NClass-B* se muestra un enlace al último nodo *NClass-A* que se visitó para llegar al nodo *NClass-B*.

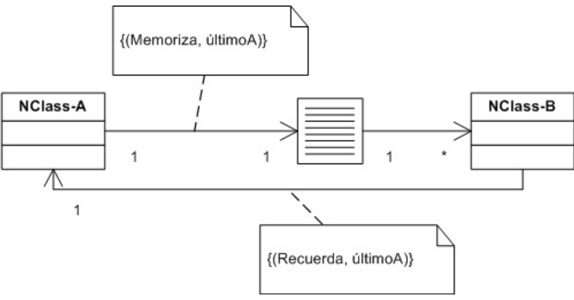


Figura 6.4: Enlaces con memoria

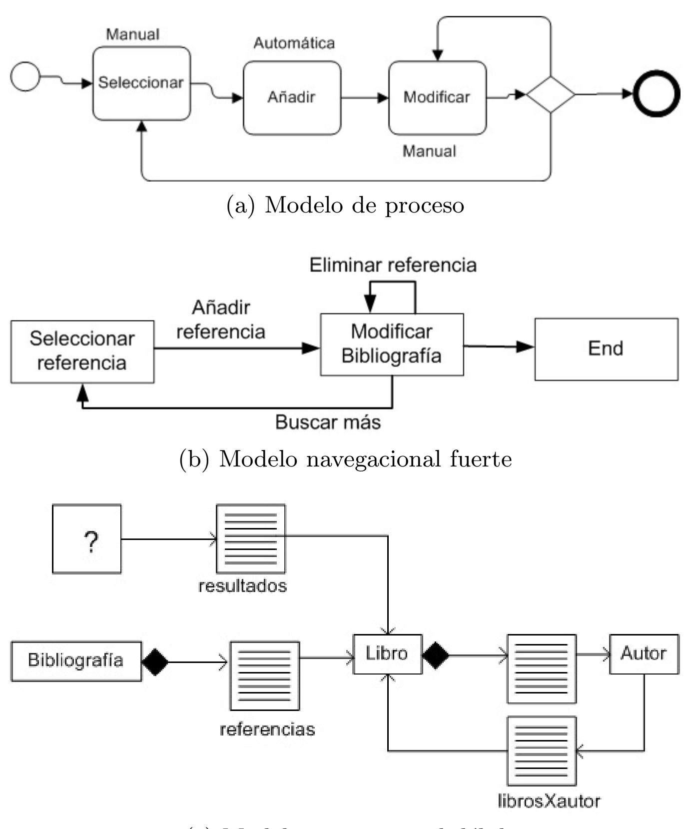
Enlaces fuertes o de proceso

Los enlaces fuertes se derivan a partir del flujo de control del modelo de proceso, pero únicamente desde la proyección del proceso por cada actor. Se puede navegar a través de un enlace de proceso sólo cuando está habilitado, es decir, únicamente cuando el proceso ha alcanzado cierto estado. Si un enlace de proceso no está habilitado, entonces no se debe mostrar en el documento hipermedia o si se muestra no es navegable. Los enlaces de proceso tienen como propiedades el nombre de la tarea del modelo de proceso que disparan, y una lista con los nodos que los activan. La representación gráfica de un enlace de proceso es una flecha (como los enlaces normales), pero con una línea más gruesa y punta triangular (ver figura 6.5).

Figura 6.5: Notación de vista de tarea y enlace fuerte

Los enlaces de proceso unen vistas de tarea. Las vistas de tarea son la representación del estado del proceso en el modelo navegacional. Una vista de tarea tiene un nodo inicial, que recibe la navegación cuando la vista de tarea es activada. Los enlaces de proceso que parten de una vista de tarea se muestran sólo en el conjunto de nodos navegables que los activan. Si la lista de nodos de activación es vacía entonces los enlaces de proceso se muestran en cualquier nodo. La representación gráfica de una vista de tarea es mediante un rectángulo (ver figura 6.5), las vistas de tareas incluidas en otra, se muestran dentro del rectángulo de la vista de tarea contenedora. Un enlace de proceso es navegable sí y sólo sí está habilitado por una vista de tarea y el nodo visitado pertenece al conjunto de nodos de activación del enlace de proceso.

Al modelo navegacional conformado por las vistas de tarea y los enlaces de proceso se le denomina modelo navegacional fuerte (ver figura 6.6). Es



(a) Modelo de proceso

(b) Modelo navegacional fuerte

(c) Modelo navegacional débil

Figura 6.6: Ejemplo de vistas de tarea

posible navegar de una vista de tarea a otra si se tiene como contexto un nodo que activa a algún enlace de proceso, lo cual produce el cambio de la vista de tarea activa. Además, la navegación a través de un enlace de proceso puede desencadenar la ejecución de tareas automáticas, de ahí que en sus propiedades se puede indicar la activación o disparo de una tarea (aunque no repercute en la navegación). De modo que hay una navegación a través de los enlaces de proceso y vistas de tarea, y otra estructural a través de los enlaces débiles. Sin embargo, toda navegación estructural ocurre dentro del contexto de una vista de tarea.

En el ejemplo de la *librería electrónica*, que se ha presentado en el capítulo 2, se podrían tener las siguientes vistas de tarea: una para la *selección*

de *referencias bibliográficas* y otra para *modificar la bibliografía*; esta última permite *eliminar referencias bibliográficas* seleccionadas previamente (ver figura 6.6). De la vista de tarea de *selección* se puede pasar a la vista de tarea de *modificar bibliografía* a través del enlace de *añadir referencia*, el cual activaría la tarea automática *añadir*.

En la vista de tarea de *modificar bibliografía* se puede navegar a través del enlace de proceso *quitar* y de *buscar más*; el primero lleva a la misma vista de tarea y el segundo a la vista de tarea de *seleccionar*. El enlace de proceso *añadir referencia* activa una tarea que recibe como entrada un *libro* para añadirlo a la *bibliografía*, y después se navega a la vista de tarea *modificar*. Es decir, en la vista de tarea *seleccionar referencia* los nodos que activan el enlace de proceso *añadir referencia* son los nodos: *índice de resultados*, *libro* y el *índice librosXAutor*. En la vista de tarea *modificar* el nodo que activa el enlace de proceso *eliminar referencia* es cada elemento del índice *referencias*, y el nodo que activa el enlace de proceso *buscar más* es cualquiera que se visite.

Excepciones Otro elemento importante en la navegación es el manejo de excepciones durante la ejecución de la aplicación. Una excepción redirige la navegación a otro nodo si ocurre un error inesperado. Al usuario se le informa sobre el error ocurrido, y se le presentan enlaces con las opciones de compensación. En MDHDM los eventos generados por el usuario y las excepciones se transforman en enlaces de proceso. Sin embargo, un enlace de proceso derivado de un evento de excepción en la aplicación es activado de forma automática (sin intervención humana), y etiquetado con la palabra *excepción*.

6.3. Metamodelo Navegacional

En esta sección se explica el metamodelo navegacional que se ha definido (ver figura 6.7). Todas las clases descienden de la clase abstracta elemento (*ModelElement*) que representa cualquier elemento del modelo. Los elementos pueden ser marcados con etiquetas que pueden tener un valor (*Tagged-*

Value). Las clases superiores del metamodelo son las clases abstractas nodo (*Node*) y enlace (*Link*). La primera representa cualquier cosa a la cual puede apuntar un enlace (nodos navegables y vistas de tarea), y la segunda representa a los enlaces.

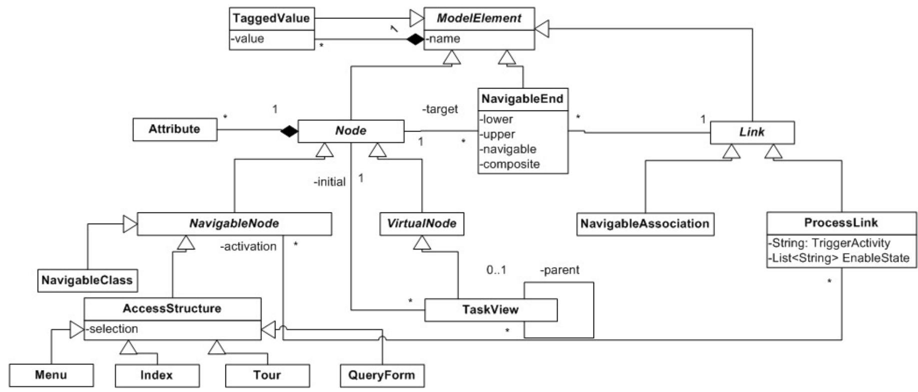


Figura 6.7: Metamodelo navegacional

Un nodo está compuesto por atributos, que representan información hipermedia. La clase nodo se especializa en las clases nodo navegable (*NavigableNode*) y nodo virtual (*VirtualNode*). Un nodo navegable es aquel que tiene contenido hipermedia (información y enlaces). En cambio, un nodo virtual redirige la navegación a otros nodos, y no posee contenido hipermedia. Los nodos navegables se clasifican en clases navegacionales (*NavigableClass*) y estructuras de acceso (*AccessStructure*). Un nodo del tipo clase navegacional representa un conjunto de nodos (objetos navegacionales) que comparten el mismo tipo de atributos y enlaces hacia otros nodos.

Una estructura de acceso representa una colección de enlaces hacia nodos navegables. Las estructuras de acceso son: índice (*Index*), visita guiada (*Tour*), menú (*Menu*) y formulario (*QueryForm*). Los índices y menús muestran enlaces hacia los nodos de la colección, mientras que las visitas guiadas muestran un elemento cada vez y enlaces a otros elementos de la colección. Las estructuras índice y visita guiada tienen una relación hacia la clase a cuyas instancias apuntan, y también pueden tener un criterio de selección de objetos.

Una vista de tarea (*TaskView*) es un nodo virtual y permite definir una red de hipermedia que depende del estado del proceso. Las vistas de tarea tienen un nodo navegable inicial y un conjunto de enlaces de proceso que permiten navegar a otras vistas de tarea (ver figura 6.8), porque los enlaces de proceso tienen como extremos a las vistas de tarea. Las vistas de tarea redirigen la navegación a su elemento de inicio. Las vistas de tarea contenidas dentro de otra indican a su contenedora a través de la asociación *parent*. Los enlaces de proceso de la vista de tarea contenedora también deben activarse en los nodos que les habilitan.

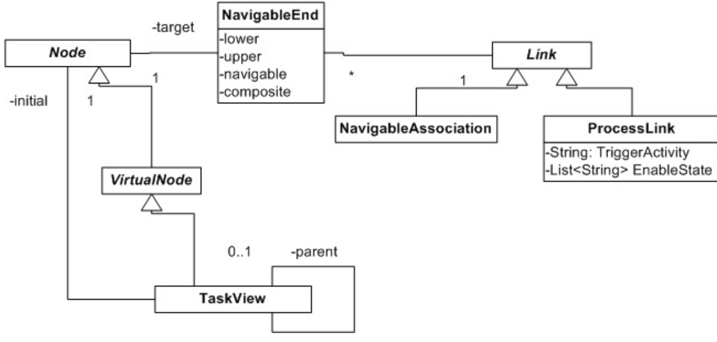


Figura 6.8: Vistas de tarea y enlaces

Los enlaces (*Link*) se especializan en dos clases: asociación navegable (*NavigableAssociation*) o enlace débil y enlace de proceso o fuerte (*ProcessLink*). Una asociación navegable representa un único enlace o un conjunto de enlaces que parte de un nodo hacia otro. Un enlace tiene un conjunto de extremos de asociación (*NavigableEnd*) conectados a los nodos que une.

Un enlace de un nodo a otro se representa con una asociación navegable en cuyos extremos se encuentran los nodos que conecta; además, la cardinalidad en cada extremo es 1. Un enlace tiene al menos un extremo navegable. Un enlace puede ser bidireccional, es decir, navegable desde ambos extremos. Un enlace que une dos nodos y que es navegable desde el nodo origen al destino modela a los hiperenlaces de la *Web*. Un enlace bidireccional entre dos nodos corresponde a dos hiperenlaces de la *Web*. La cardinalidad de una asociación navegable es n en el extremo del nodo destino cuando el origen

es una estructura de acceso índice o visita guiada, lo cual indica que en la instancia de la estructura de acceso hay n enlaces hacia los nodos destino. Los enlaces pueden tener n extremos, es decir, un enlace puede permitir la navegación hacia múltiples nodos. Los enlaces débiles se derivan de las asociaciones binarias del modelo estructural y tienen sólo dos extremos.

Los enlaces de proceso unen vistas de tarea, es decir, en este caso los *NavigableEnd* están asociados a *Task Views* (se debe garantizar a través de una restricción). Los enlaces de proceso se muestran en un nodo navegable si la vista de tarea los habilita y el nodo pertenece a los nodos de activación del enlace. La navegación a través de un enlace de proceso tiene como efecto alcanzar una nueva vista de tarea, el cambio del estado del proceso y la consiguiente ejecución de actividades. Las vistas de tarea que habilitan a un enlace de proceso son aquellas que están conectadas al extremo no navegable del enlace de proceso.

Las asociaciones navegables transmiten información contextual de un nodo a otro, identificadores de objetos en el caso de clases, índices y visitas guiadas. La información que envía un formulario son los datos capturados en él.

Las restricciones que deben satisfacerse en cuanto a la construcción de clases, atributos y la jerarquía de herencia son idénticas a las del modelo estructural (sección 5.4); por ello, no se muestran otra vez. A continuación se muestran las restricciones que debe satisfacer el modelo navegacional. Para escribirlas de forma más simple se han definido dos funciones para seleccionar el primer y el segundo extremo de una asociación navegable.

Restricción 6.1: Selección de los extremos de una asociación.

```
Context NavigableAssociation def: first =
  self.NavigableEnd->first();

Context NavigableAssociation def: last =
   self.NavigableEnd->last();
```

Restricción 6.2: Los enlaces tienen al menos dos extremos de asociación.

```
Context Link inv:
```

```
      self.NavigableEnd->size() >= 2
```

Restricción 6.3: Los enlaces débiles tienen dos extremos de asociación.

```
Context NavigableAssociation inv:
      self.NavigableEnd->size() = 2
```

Restricción 6.4: Las asociaciones 1-1 pueden presentarse entre un menú y cualquier nodo navegable, entre clases navegacionales, entre clases o formularios e índices y visitas guiadas.

```
Context NavigableAssociation inv:
(
  self.oclIsTypeOf(NavigableAssociation)
  and
  self.first.max = 1
  and
  self.last.max = 1
)
implies
(
    self.first.target.oclIsTypeOf(Menu)
  or
        ( self.first.target.oclIsTypeOf(NavigableClass)
         and
      self.last.target.oclIsTypeOf(NavigableClass)
         )
  or
        ( (self.first.target.oclIsTypeOf(NavigableClass)
           or
           self.first.target.oclIsTypeOf(Queryform)
           )
           and
       (self.last.target.oclIsTypeOf(Index)
            or
            self.last.target.oclIsTypeOf(Tour)
           )
         )
)
```

La cardinalidad n se representa con -1.

Restricción 6.5: Las asociaciones 1-N pueden presentarse entre índices y visitas guiadas, hacia otros iguales o clases navegacionales.

```
Context NavigableAssociation inv:
(
  self.oclIsTypeOf(NavigableAssociation)
  and
  self.first.max = 1
  and
  self.last.max = -1
)
implies
(
        ( self.first.target.oclIsTypeOf(Index)
          or
      self.first.target.oclIsTypeOf(Tour)
         )
  and
        ( (self.last.target.oclIsTypeOf(Index)
           or
       self.last.target.oclIsTypeOf(Tour)
           or
           self.last.target.oclIsTypeOf(NavigableClass)
           )
         )
)
```

Restricción 6.6: Las asociaciones 1-1 deben tener al menos un lado navegable.

```
Context NavigableAssociation inv:
  self.first.navigable
or
  self.last.navigable
```

Restricción 6.7: La dirección de la navegación en una asociación 1-N es hacia el lado N.

```
Context NavigableAssociation inv:
  (self.first.max = 1 and self.last.max = -1)
implies
```

```
    (self.last.navigable and (not self.first.navigable))
```

Restricción 6.8: La composición puede presentarse en cualquiera de los extremos de una asociación 1-1, pero únicamente en un lado.

```
Context NavigableAssociation inv:
  (self.first.max = 1 and self.last.max = 1)
implies
  ( not
    ( self.first.composite
      and
          self.last.composite)
  )
```

Restricción 6.9: La composición no puede presentarse en el extremo 1 de una asociación 1-N.

```
Context NavigableAssociation inv:
  (self.first.max = -1 and self.last.max = 1)
implies
  ( not self.last.composite )
```

Restricción 6.10: No puede presentarse un ciclo de asociaciones compuestas.

```
Context NavigableNode def: getCompositePartners() :
Set(NavigableAssociation) =
     self.NavigableEnd
         ->select( x | x.composite = true )
         ->collect(x | x.NavigableAssociation)
         ->collect(x | x.NavigableEnd.target)
         ->select(x|x <> self)

Context NavigableNode def:
CompositePartner(partner:NavigableNode) : Boolean =
    self.getCompositePartners()->includes(partner) )
  or
    self.CompositePartner(partner)

Context NavigableNode inv:
   not self.CompositePartner(self)
```

Restricción 6.11: Los enlaces de proceso tiene al menos un extremo navegable.

```
Context ProcessLink inv:
  self.NavigableEnd->exists(x|x.navigable)
```

Restricción 6.12: Los enlaces de proceso tiene como origen y destino una vista de tarea.

```
Context ProcessLink inv:
  self.NavigableEnd->forAll(x|x.target.oclIsKindOf(TaskView))
```

6.4. Proceso para el diseño navegacional

En MDHDM son necesarios para efectuar el diseño navegacional el modelo de proceso y el modelo estructural. A estos modelos se les añade información adicional, que conoce el diseñador, a través de etiquetas. Los modelos marcados son la entrada del proceso de transformación automatizado, que da como salida los modelos navegacionales fuerte y débil de cada uno de los actores del proceso. Los pasos para efectuar el diseño navegacional se muestran en la figura 6.9, y son los siguientes.

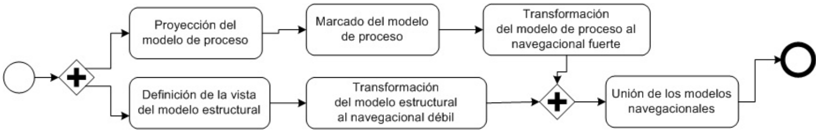


Figura 6.9: Pasos para construir el modelo navegacional de un actor.

Para un actor dado:

1. *Proyección del modelo de proceso.* Sirve para generar la vista que un actor tiene sobre el proceso (sección 6.5).

2. *Marcado del modelo de proceso.* En este paso se etiquetan las tareas manuales para indicar sus características, lo que permitirá transformarlas a elementos del modelo navegacional (sección 6.6.1).

3. *Transformación del modelo de proceso a navegacional fuerte.* A través de reglas de transformación se convierte un modelo de proceso en un modelo navegacional fuerte (sección 6.6.2).

4. *Definición de las vistas del modelo estructural.* Se identifican los elementos del modelo estructural que son visibles para un actor. Se etiqueta el modelo estructural para especificar los elementos visibles, seleccionar la navegabilidad de las asociaciones, y conformar clases navegacionales (sección 6.7.4).

5. *Transformación del modelo estructural a navegacional débil.* Mediante reglas de transformación se convierte el modelo estructural en un modelo navegacional débil (sección 6.7.1).

6. *Unir los modelos navegacionales.* Se efectúa la unión de conjuntos de los modelos navegacionales obtenidos en los pasos 3 y 5.

   En el resto del capítulo se explica cada uno de los pasos descritos.

6.5. Proyección del modelo de proceso

Este paso tiene como entrada el modelo de proceso del sistema, y su objetivo es obtener un grafo que contenga únicamente las tareas que dependen de un determinado actor. La proyección sirve para encontrar el flujo de control del cual provienen los enlaces de proceso del modelo navegacional, y también muestra cuándo la navegación en cierta sección del proceso debe terminar debido a que continúa con actividades de otros actores. La sección del proceso que efectúa un actor está delimitada por puntos de sincronización. Los puntos de sincronización indican que una tarea realizada por dicho actor necesita de la finalización de tareas efectuadas por otros actores. Un punto de sincronización puede corresponder al nodo inicial de un proyección. Esto indica que todas las actividades incluidas en la proyección dependen de la ejecución previa de una actividad efectuada por otro actor. En cambio, un punto de sincronización al final de un a proyección indica que da paso a la ejecución de actividades de otros actores. La representación de estos puntos

de sincronización son, respectivamente, un nodo de inicio o finalización con un sobre en el interior.

Los pasos para obtener la proyección son:

1. La sustitución de las tareas que realizan los otros actores por subprocesos que las engloban.

2. Indicar los puntos de sincronización donde estos nuevos subprocesos tienen como antecesor o sucesor una actividad o subproceso del actor de la proyección. Los puntos de sincronización son representados como eventos intermedios de BPMN, y se colocan entre las actividades del actor y los subprocesos efectuados por otros. Los puntos de sincronización se pueden marcar para indicar si el flujo continúa fuera de la proyección o si retorna, las marcas son *out* e *in* respectivamente.

La proyección se efectúa mediante iteraciones, cada iteración trata de ejecutar, en el orden definido, alguna regla; si se logra ejecutar alguna, entonces se efectúa otra iteración. Se llega al final de las iteraciones cuando ya no se puede ejecutar ninguna regla. El algoritmo sería:

```
Para todas las reglas de R1 a Rn
  Si Ri se puede ejecutar entonces {
      ejecuta la regla Ri
      reinicia ciclo (i=0)
  }
```

En resumen, la proyección se basa en la aplicación de tres reglas:

- Las secuencias de actividades efectuadas por otros actores son agrupadas en subprocesos.

- Si se forman secuencias de nuevos subprocesos de otros actores, entonces se vuelven a agrupar en otro subproceso.

- Si un subproceso está rodeado por *gateways*, entonces éstos se incorporan a un subproceso.

Antes de presentar las reglas para efectuar la proyección, se explica cómo se representarán las reglas de transformación. Del lado izquierdo se muestra el patrón del modelo origen, después una flecha, que representa la regla de transformación, y del lado derecho de ésta, el patrón resultante en el modelo destino (ver figura 6.10).

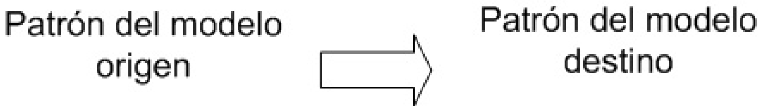


Figura 6.10: Regla de transformación

A continuación se muestran las reglas detalladas para realizar la proyección para un hipotético actor α, otros actores se indican mediante distintas letras griegas.

1. Una secuencia de actividades o subprocesos realizadas por otros actores ubicada entre *gateways* es reemplazado por un subproceso (figura 6.11).

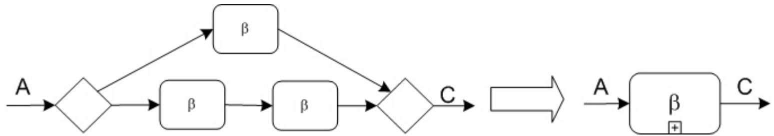


Figura 6.11: Regla de proyección 1

2. Una secuencia de actividades que no efectúa el actor α se reemplaza por un subproceso con dos puntos de sincronización. La secuencia puede tener longitud 1 (figura 6.12).

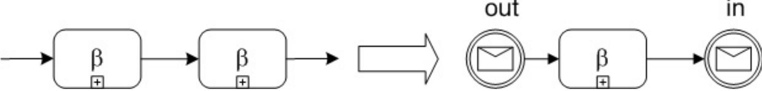


Figura 6.12: Regla de proyección 2

3. Un *Or-gateway* de ramificación precedido por un subproceso con puntos de sincronización es reemplazado por un *Or-gateway* con puntos de sincronización (figura 6.13).

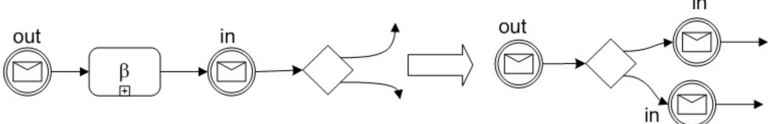


Figura 6.13: Reglas de proyección 3 y 6 *(Or/And)-gateway*

4. Un *Or-gateway* de unión con puntos de sincronización precedido por n subprocesos sincronizados es reemplazado por un *Or-gateway* con puntos de sincronización (figura 6.14).

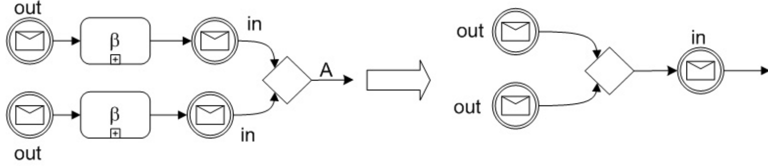


Figura 6.14: Reglas de proyección 4 y 7 *(Or/And)-gateway*

5. Un *Or-gateway* de unión seguido por algunos subprocesos sincronizados es reemplazado por un *Or-gateway* con puntos de sincronización (figura 6.15).

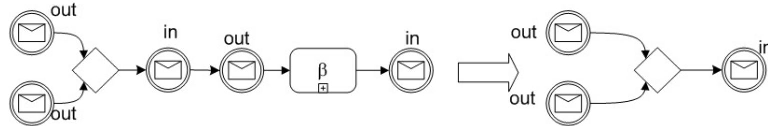


Figura 6.15: Reglas de proyección 5 y 8 *(Or/And)-gateway*

6. Un *And-gateway* de ramificación precedido por un subproceso sincronizado reemplazado por un *And-gateway* con puntos de sincronización (figura 6.13).

7. Un *And-gateway* de unión precedido por n subprocesos sincronizados es reemplazado por un *And-gateway* con puntos de sincronización(figura 6.14).

8. Un *And-gateway* de unión seguido por algunos subprocesos sincronizados es reemplazado por un *And-gateway* con puntos de sincronización (figura 6.15).

9. Un *Or-gateway* de ramificación seguido de algunos subprocesos de sincronizados es reemplazado por un *Or-gateway* con puntos de sincronización (figura 6.16). Si alguno de los caminos queda conectado a un *Or-gateway* de unión se elimina.

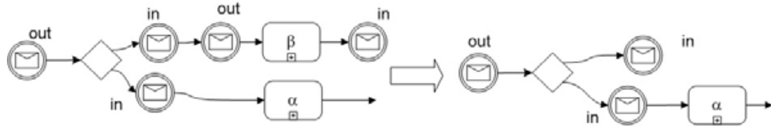


Figura 6.16: Reglas de proyección 9 y 12 *(Or/And)-gateway*

10. Un *Or-gateway* de ramificación sincronizado conectado al nodo de inicio y a una actividad de α, es reemplazado por un nodo de inicio sincronizado conectado a la actividad (figura 6.17).

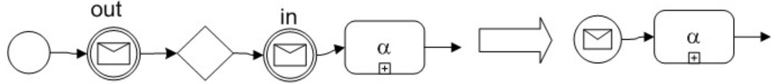


Figura 6.17: Reglas de proyección 10 y 13 *(Or/And)-gateway*

11. Un *Or-gateway* de unión, conectado al nodo de finalización y a una actividad del proyectado, es reemplazado por un nodo de finalización sincronizado conectado a la actividad (figura 6.18).

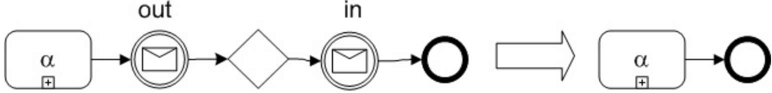


Figura 6.18: Reglas de proyección 11 y 14 *(Or/And)-gateway*

12. Un *And-gateway* de ramificación seguido de algunos subprocesos de sincronización es reemplazado por un *And-gateway* con puntos de sincronización (figura 6.16).

13. Un *And-gateway* de ramificación sincronizado, conectado al nodo de inicio y a una actividad de α es reemplazado por un nodo de inicio sincronizado conectado a la actividad (figura 6.17).

14. Un *And-gateway* de unión, conectado al nodo de finalización y a una actividad de α es reemplazado por un nodo de inicio sincronizado conectado a la actividad (figura 6.18).

15. Las tareas automáticas que efectúa el sistema son vistas como efectuadas por otro actor sólo si su duración en el tiempo es larga y rompe la navegación. Una tarea automática de corta duración no interfiere con la navegación, mientras que una de larga duración sí debe ser tratada como una tarea efectuada por otro actor. Una tarea de larga duración es aquella que tarda más de 10 segundos [Nie99], con un tiempo mayor

a este se pierde la atención del usuario, y además, le da una sensación de incertidumbre, debido a que no sabe si ocurrió un fallo o si es la respuesta normal.

6.5.1. Ejemplo de proyección

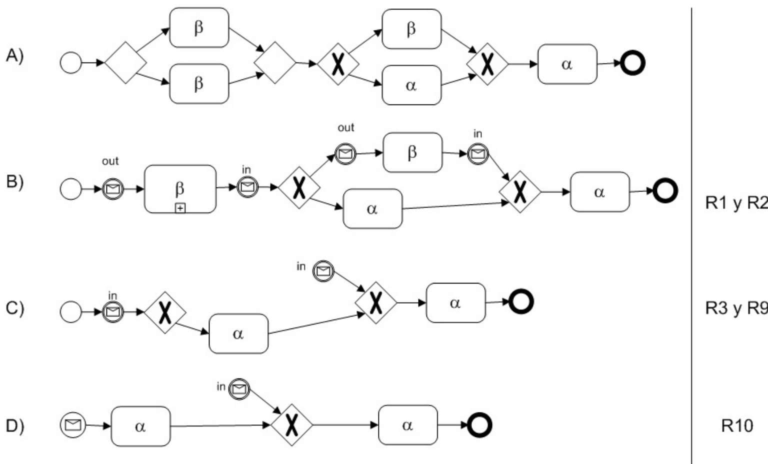


Figura 6.19: Ejemplo de la proyección para el actor α

El proceso de la figura 6.19A se proyecta para el actor α. Después de aplicar las reglas 1 y 2 se obtiene el proceso de la figura 6.19B. Estas reglas reemplazan a las secuencias de actividades del actor β y las colocan entre puntos de sincronización. Después se aplica la regla 3 y la regla 9, obteniendo el proceso de la figura 6.19C, que unen a los procesos sincronizados efectuados por el actor β a los *gateways*. También, mediante la regla 9 se elimina uno de los caminos por quedar unidos directamente los dos *gateways*. Finalmente, mediante la regla 10 se reemplaza al *gateway* conectado al nodo de inicio, se indica que el inicio de las tareas correspondientes a la proyección dependen de la finalización de tareas efectuadas por el actor β; finalmente, mediante la regla 11 se elimina el *gateway* conectado al nodo de finalización. La proyección resultante para el actor α es la mostrada en la figura 6.19D.

6.6. Obtención del modelo navegacional fuerte

El modelo navegacional fuerte se obtiene a partir de la proyección del proceso para un actor. En esta sección se definen las marcas aplicables al modelo de proceso y las reglas de transformación al modelo navegacional.

6.6.1. Marcas del modelo de proceso

Aunque el proceso de transformación es automático, no deja de ser guiado por el diseñador, que debe conocer las transformaciones alternativas (aplicación de unas reglas u otras según ciertos criterios) y qué datos adicionales se requieren para ejecutar una transformación. Las etiquetas o marcas sirven para indicar qué transformaciones quiere el diseñador que se apliquen a los elementos del modelo. Las marcas, en algunos casos, proporcionan información adicional, que no está disponible en el modelo a transformar. Esto suele ocurrir porque un modelo de alto nivel de abstracción carece de detalles.

Las marcas aplicables a los modelos de proceso y estructural (secciones 6.6.1 y 6.7.4 respectivamente) se han identificado, y deben de considerarse durante la ejecución de las funciones de transformación. En el caso del modelo de proceso sirven para indicar distintos tipos de actividades manuales y si un *xor gateway* es diferido. En el caso del modelo estructural sirven para agrupar elementos, definir la visibilidad de los elementos e indicar estructuras de acceso.

Antes de ejecutar una transformación, además de verificar que un modelo esté bien construido, también, se debe verificar que cuente con la información adicional requerida. Estos datos se proporcionan a la transformación a través de un conjunto de etiquetas y valores asociados a los elementos del modelo (*modelo de marcado*).

Las marcas se comprueban mediante una consulta en OCL (*hasTag*), también se ha definido la función opuesta *notHasTag*. Otra función (*testValue*) sirve para comprobar que una etiqueta tiene asignado un valor dado. Las consultas que se han mencionado se pueden ver en al apéndice A.

Las marcas aplicables al modelo de proceso se muestran en la tabla 6.1. Estas etiquetas sirven básicamente para identificar los tipos de actividades

manuales facilitando que puedan transformarse a elementos del modelo navegacional.

Tabla 6.1: Marcas aplicables al modelo de proceso

Marca	**Elemento**	**Efecto en el modelo navegacional**
Anonymous	Proceso	El inicio del proceso no es a través de formulario de identificación
Edition	Actividad Manual	Permite editar un objeto dato o variable del proceso
Selection	Actividad Manual	El usuario debe seleccionar un objeto dato
Visualization	Actividad Manual	Muestra a un objeto dato o variable del proceso
DataInput	Actividad Manual	Presenta un formulario para la captura de datos

El modelo navegacional, también, puede tener marcas relativas al modelo de proceso (ver tabla 6.2). Si un nodo navegacional inicia el proceso, se añade la marca *Start*. Esta etiqueta tiene efecto si no existe ninguna instancia del proceso iniciada por la navegación, es decir, el nodo inicial puede visitarse otras veces, pero no da lugar a nuevas instancias, a menos que ya se haya pasado por un nodo etiquetado con *End*. Si un nodo navegacional termina el proceso, se le añade la marca *End*.

Tabla 6.2: Marcas del modelo navegacional derivadas del proceso

Marca	**Elemento**	**Efecto en el modelo navegacional**
Start	NavigableNode	Inicia el proceso
End	NavigableNode	Finaliza el proceso

6.6.2. Transformación del modelo de proceso

En esta sección se presentan las reglas de transformación que permiten obtener a partir de los elementos de una proyección del modelo de proceso, los elementos del modelo navegacional fuerte. *Grosso modo*, una actividad manual corresponde a una vista de tarea, las flechas del flujo de control se convierten en enlaces de proceso, y cada sección de navegación delimitada por puntos de sincronización, se convierte en una alternativa en un menú.

Las reglas se pueden consultar en el apéndice A. La tabla 6.3 resume la transformaciones presentadas en esta sección.

Tabla 6.3: Reglas de transformación del modelo de proceso a navegacional

R1	Obtención del sucesor de un nodo
R2	Construcción de los enlaces de proceso
R3	Transformación del contenedor del proceso
R4	Transformación de nodos sincronizados
R5	Transformación de las tareas manuales
R6	Transformación de un xor gateway de ramificación diferido
R7	Transformación de un xor gateway de ramificación
R8	Transformación de las tareas automáticas
R9	Transformación de un and-gateway de ramificación
R10	Transformación de nodos de sincronización y subprocesos
R11	Transformación de los nodos de sincronización

R1. Obtención del sucesor de un nodo

El sucesor de una tarea es identificado recorriendo el modelo a través de una consulta OCL. Si la tarea no tiene una tarea manual como sucesor entonces los enlaces de proceso derivados llevarían al fin de la navegación; en algunos casos, el mismo nodo inicial hace este papel, así el usuario puede realizar otra ejecución del proceso.

Para buscar al sucesor de un nodo se invoca la consulta *BuscaSucesor*. Una actividad manual tiene como máximo una tarea sucesora y dependiendo de dicha tarea se determina el sucesor o se sigue buscando. Por ejemplo, si el sucesor es una actividad automática entonces se buscaría ahora el sucesor de tal tarea. Si el sucesor es un xor-diferido entonces devuelve el conjunto vacío porque se transforma en un menú contenido en la actividad manual anterior. Los *gateways* de sincronización son saltados por esta consulta y se devuelve al sucesor de éstos. La figura 6.20 muestra el resultado de la consulta *BuscaSucesor*.

R2. Construcción de los enlaces de proceso

Los enlaces de proceso se añaden a los nodos navegacionales a través de una regla que es invocada cada vez que se construye un nuevo elemento del modelo navegacional. Esta regla recibe la tarea origen y la tarea destino. Se

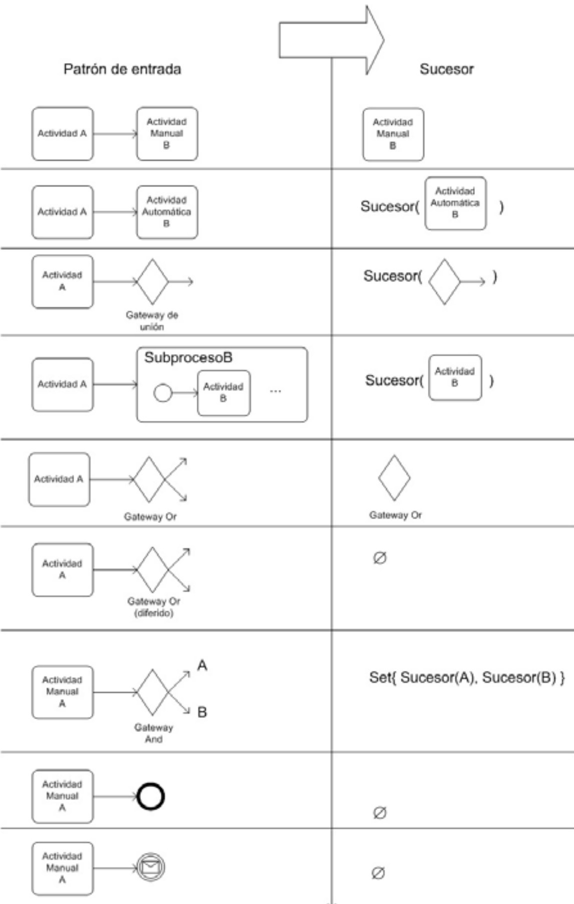


Figura 6.20: Sucesor de un nodo

cuenta con otra regla similar que recibe adicionalmente un conjunto con los elementos navegacionales donde el enlace de proceso se activa. Esta regla es invocada por aquellas que tienen que construir un enlace de proceso como parte de su patrón de transformación.

R3. Transformación del contenedor del proceso

El contenedor del proceso es la clase *Workflow*, a partir de este nodo se obtiene en el modelo navegacional de un actor el nodo inicial de la navegación. A partir de un modelo de proceso se construye un modelo navegacional

y se marca el primer nodo como inicial. Si el acceso no es anónimo entonces se construye un formulario que tendrá un enlace a la primera tarea. El nodo que funciona como inicio de la aplicación hipermedia es el derivado de la primera tarea manual, es decir, el nodo inicial de la primera vista de tarea. Dicho nodo debe ubicarse después de un formulario de identificación del actor, excepto en los casos donde los actores anónimos participan en el proceso. Esto ocurre, por ejemplo, en el comercio electrónico, donde el proceso lo inician los visitantes de la página.

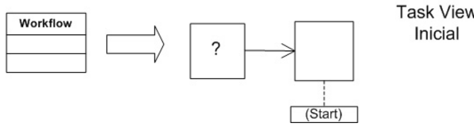


Figura 6.21: Transformación del contenedor del modelo de proceso

R4. Transformación de nodos sincronizados

Se denomina nodo sincronizado aquel precedido por un punto de sincronización. Un nodo sincronizado es transformado en un índice de objetos (aquellos que pueden ser usados como entrada de la siguiente actividad). En caso de que no exista un objeto dato de salida, entonces el actor tendría que seleccionar una instancia del proceso que está en el estado previo al que se va a ejecutar (figura 6.22). Este índice debe ser apuntado por un enlace débil desde el nodo inicial de la vista de tarea inicial. Además, este índice es el nodo de activación del enlace de proceso. El destino del enlace de proceso es la vista de tarea derivada de la actividad.

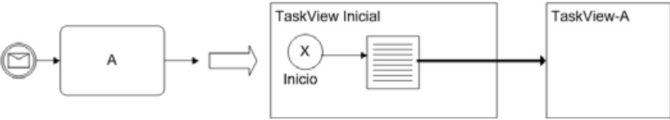


Figura 6.22: Transformación de un nodo sincronizado

R5. Transformación de las tareas manuales

Las tareas manuales deben etiquetarse e indicar el tipo de interacción que el actor efectúa durante la ejecución de la tarea. Las actividades de captura de datos son transformadas en un formulario de entrada. Estas tareas generan un objeto dato que debe pertenecer a alguna clase del modelo estructural, o ser un objeto transitorio que sólo sirve como entrada de la siguiente actividad. El patrón de entrada de la transformación está conformado por una actividad manual etiquetada como de captura de datos y que tiene un objeto dato de salida (figura 6.23).

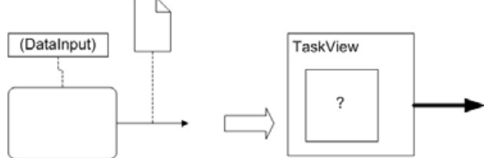


Figura 6.23: Transformación de las tareas de captura de datos

Una actividad de selección sirve para elegir un objeto dato que sirve como entrada de la siguiente actividad, es decir, el patrón de entrada de la transformación está conformado por una actividad manual etiquetada como de selección, una condición que deben cumplir los objetos seleccionados, y un tipo objeto dato que identifica la clase del objeto seleccionado. El patrón de salida generado es un índice con la condición especificada y un enlace de proceso. Si no se indica ninguna condición de selección, entonces en cualquier índice o visita guiada apuntando a la clase navegacional correspondiente al objeto dato, o cualquier instancia de dicha clase se consideran nodos de activación del enlace de proceso (figura 6.24). Además, se debe tener en cuenta que los índices permiten la selección múltiple, en cambio, las visitas guiadas y los objetos navegacionales no.

Una actividad de edición/visualización, permite editar/ver una variable de proceso o un objeto dato. Este tipo de actividad es transformada en una clase navegacional (ver figura 6.25).

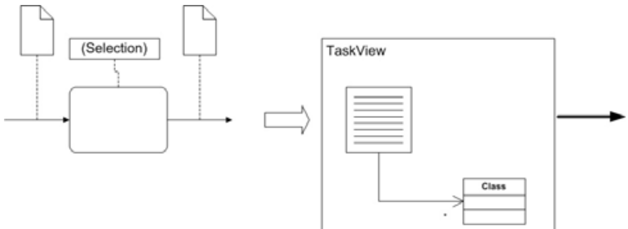


Figura 6.24: Transformación de las tareas de selección

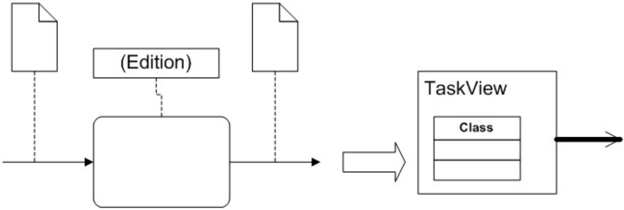


Figura 6.25: Transformación de las tareas de edición/visualización

R6. Transformación de un xor-gateway diferido

Las tareas manuales pueden ser etiquetadas como *deferred-or*, es decir la actividad manual sirve como entrada de un *gateway-Or* que efectúa una ramificación. En tal caso, el *gateway-Or* es convertido en un menú asociado a la vista de tarea con enlaces de proceso, uno por cada rama de salida. El nodo destino de cada rama es determinado a través de la función *getSuccesor*.

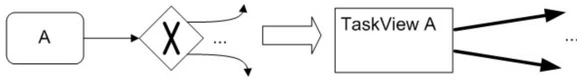


Figura 6.26: Transformación XOR gateway diferido

R7. Transformación de un xor-gateway

Un *xor-gateway* no diferido es transformado en una vista de tarea marcada con la etiqueta *Xor* y que tiene un conjunto de enlaces de proceso a las alternativas disponibles. Es decir, desde el nodo navegacional que le an-

tecede no se puede saber cuál es el siguiente nodo hasta que se efectúe una decisión de acuerdo al estado e información de entrada de la tarea.

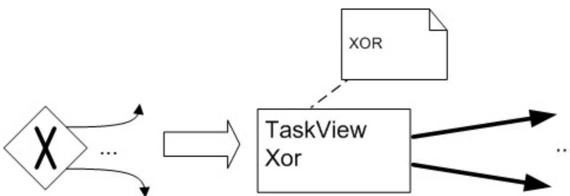


Figura 6.27: Transformación XOR gateway

R8. Transformación de las tareas automáticas

Las tareas automáticas no tienen una correspondencia con ningún elemento del modelo navegacional. Sin embargo, deben ser ejecutadas cuando, a través de un enlace fuerte el actor hace avanzar el estado del proceso. Los enlaces de proceso tienen como meta-información el estado que debe tener el proceso para estar activos, qué tarea disparan (esto último es usado en la generación de código), y cuál es el siguiente nodo navegable. Por ejemplo, en una secuencia de tres actividades donde la intermedia es una actividad automática, el enlace de proceso que parte del nodo derivado de la primera tarea manual tiene como destinatario la siguiente tarea manual, pero debe disparar la ejecución de la tarea automática.

Es importante marcar las tareas automáticas como de corta o larga duración, de acuerdo, al criterio mencionado en la sección 6.5. En los diagramas mostrados, si no se indica lo contrario, se asume que una tarea es de corta duración. El paso de proyección se pueden marcar las tareas para tener estos criterios en cuenta.

R9. Transformación de un And-gateway de ramificación

Un *And-gateway de ramificación* es transformado en un enlace con múltiples nodos destino. Los sucesores de este nodo son las tareas manuales que deben mostrarse. Por ejemplo, en la figura 6.28 los sucesores del enlace de

proceso son la transformación de las tareas manuales t_2 y t_4, las tareas automáticas t_1 y t_3 son ejecutadas antes de activar las actividades manuales. Además, debe considerarse que un enlace de proceso con múltiples destinos es el resultado de una transformación sólo si hay alguna actividad manual en alguno de los caminos paralelos (figura 6.28). En el caso de que haya un camino donde sólo hay actividades en paralelo, éstas deben ser activadas por el enlace de proceso anterior, y ejecutarse en su propio hilo. Es decir, los elementos navegacionales derivados de este caso sólo tienen un enlace de proceso (figura 6.29).

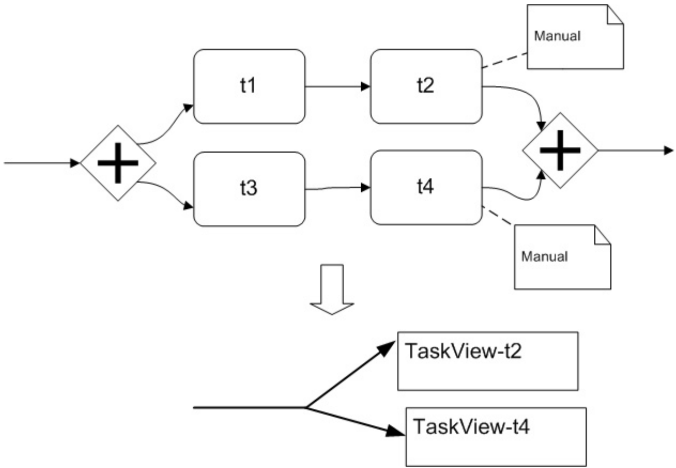


Figura 6.28: Transformación del And-gateway (1)

La regla de transformación debe considerar los dos casos anteriores. La regla de transformación verifica que el *gateway-And* de ramificación tenga como sucesor una tarea manual antes de alcanzar un *gateway-And* de sincronización. La función *hasManualSucesor()* efectúa esa búsqueda, y es similar a la función busca sucesor, excepto que cuando alcanza un *gateway-And* de sincronización concluye la búsqueda.

Si se desea evitar el paralelismo, para que el actor sólo se concentre en una tarea, entonces se pueden serializar las actividades en paralelo como dos secuencias independientes que se ejecutan una después de la otra (figura 6.30).

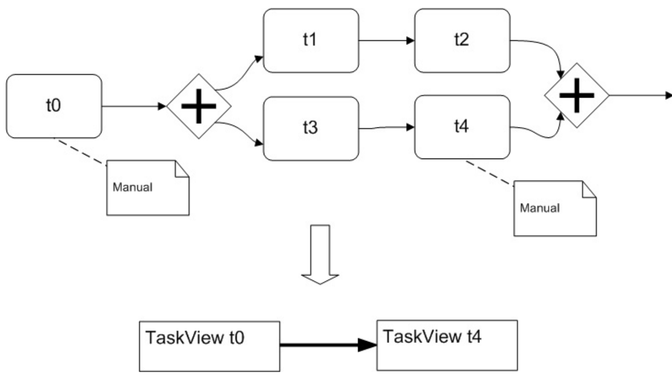


Figura 6.29: Transformación del And-gateway (2)

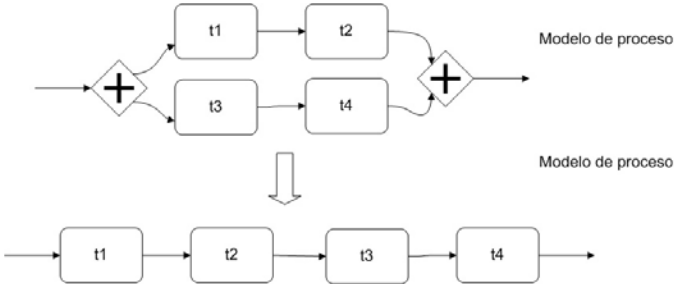


Figura 6.30: Serialización de tareas

R10. Transformación de subprocesos

La transformación de los subprocesos depende de la interpretación de los eventos intermedios en el borde de un subproceso. En el capítulo anterior se ha mencionado que los eventos intermedios pueden verse como un *Xor-gateway* de ramificación después de cada tarea del subproceso. Si se efectúa esta sustitución de los eventos intermedios entonces sólo hace falta que la consulta que busca los sucesores de un nodo, devuelva el sucesor de la tarea inicial del subproceso.

Un evento intermedio corresponde sólo a una navegación alternativa que es generada por el usuario, entonces, también es factible transformar un subproceso en una vista de tarea y el evento intermedio en un enlace de proceso. La figura 6.31 muestra la transformación de un subproceso de acuerdo a este

punto de vista. Cada tarea del subproceso se transforma con las reglas explicadas, y tienen como tarea contenedora a la del subproceso. La consulta que busca el sucesor de un nodo queda como se dijo en el párrafo anterior. Las vistas de tarea contenidas también tiene que activar los enlaces de proceso de la tarea contenedora.

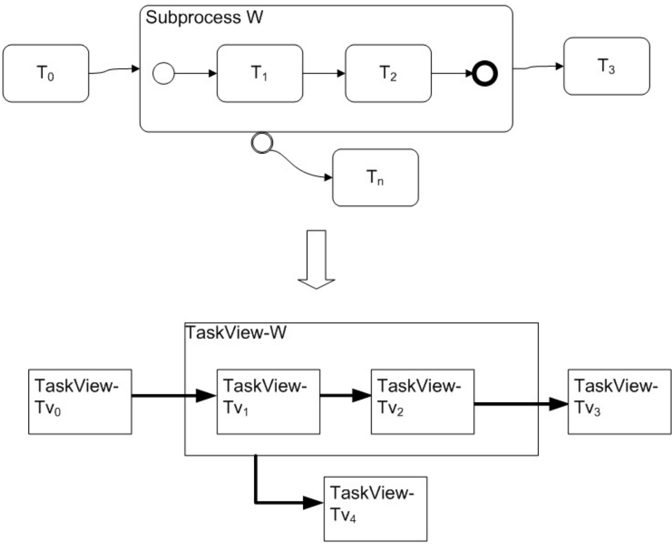


Figura 6.31: Transformación de un subproceso

R11. Transformación de los nodos de sincronización

Los *gateway-Or* y *gateway-And* de sincronización no participan en la transformación y son ignorados, porque no afectan a la navegación, lo cual está reflejado en la consulta que busca un sucesor.

6.7. Obtención del modelo navegacional débil

Algunas heurísticas para construir al modelo navegacional débil ya han sido explicadas en otros métodos de desarrollo que obtienen el espacio navegacional de las asociaciones existentes entre las clases del modelo estructural.

En este trabajo se han retomado dichas heurísticas, se han catalogado

y especificado a través de metamodelos MOF y transformaciones ATL. En la subsección 6.7.1 se explican las reglas de transformación que la mayoría de los métodos de desarrollo de hipermedia aplican; éstas aparecieron por primera vez en RMM y se basan en la cardinalidad de las asociaciones entre clases.

Se han añadido algunas transformaciones nuevas también basadas en las asociaciones entre las clases del modelo estructural. En concreto, en la sección 6.7.2 se abordan transformaciones que tienen en cuenta los caminos de longitud mayor que 1. En la sección 6.7.3 se definen los atajos que son enlaces derivados del encadenamiento de asociaciones. Las marcas aplicables al modelo estructural y las transformaciones con marcas se definen en la sección 6.7.4.

6.7.1. Transformación basada en las relaciones

Las transformaciones aplicadas al modelo estructural presentadas en esta sección son las mostradas en la tabla 6.4. Las transformaciones se representan igual que en la sección 6.6, pero ahora el modelo origen es el modelo estructural, y el modelo destino es el modelo navegacional.

Tabla 6.4: Reglas de transformación del modelo de estructural al navegacional

R12	Transformación de clases y atributos
R13	Transformación de asociaciones 1-1
R14	Transformación de asociaciones 1-N
R15	Transformación de asociaciones M-N

R12. Transformación de clases y atributos

Una clase conceptual corresponde a una clase navegacional. Los atributos de las clases se transforman en atributos de clases navegacionales.

R13. Transformación de asociaciones 1-1

Una asociación con cardinalidad 1-1 se convierte en una asociación navegacional, que conecta a las clases asociadas, y navegable en los dos sentidos

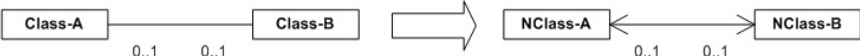


Figura 6.32: Transformación de asociación 1-1

(figura 6.32).

R14. Transformación de asociaciones 1-N

Una asociación con cardinalidad 1-N es transformada en una estructura de acceso y dos asociaciones navegacionales, una conectada a la clase con cardinalidad 1 y a la estructura de acceso, y otra desde ésta hacia la clase con cardinalidad n. Estas dos asociaciones son navegables en dirección al extremo n de la asociación. Además, se genera otra asociación navegable con cardinalidad 1 desde la clase con cardinalidad n hacia la de cardinalidad 1 (figura 6.33).

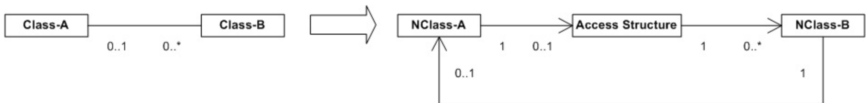


Figura 6.33: Transformación de asociación 1-N

R15. Transformación de asociaciones N-M

Las asociaciones con cardinalidad N-M son tratadas como dos asociaciones con cardinalidad 1-N, es decir, en este caso se crean dos estructuras de acceso (figura 6.34). La regla de transformación se base en la anterior.

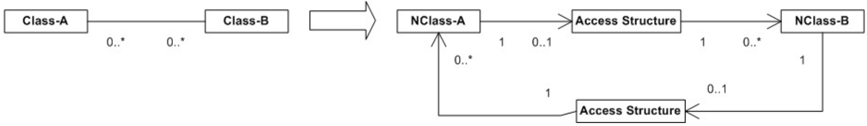


Figura 6.34: Transformación de asociación M-N

6.7.2. Transformaciones complejas del modelo navegacional débil

En esta sección se presentan otras transformaciones que se basan, también en las relaciones entre las entidades. Sin embargo, éstas tienen en cuenta la configuración de varias entidades asociadas.

Asociaciones en estrella

Una relación en estrella está conformada por una clase (base) con n relaciones 1-N hacia otras (hojas), este tipo de estructura se transforma a un encadenamiento de estructuras de acceso. Las clases hoja se transforman a una estructura de acceso, donde el usuario selecciona un elemento; entonces el identificador de un objeto de la clase base sirve como parámetro de la siguiente estructura de acceso, también procedente de una clase hoja. La figura 6.35 muestra a la clase B (base) relacionada con dos clases A y C (hojas), la cardinalidad de las asociaciones es 1-N en los dos casos. El primer índice seleccionaría las instancias de A que están relacionadas al menos con un B. El segundo índice recibe el identificador del elemento B elegido y su condición de selección obtiene las instancias de C relacionadas con B. Este tipo de transformación es útil en la realización de búsquedas.

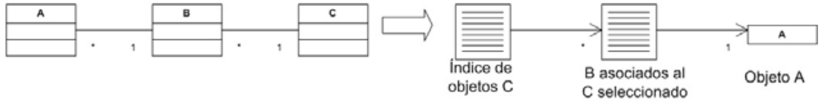


Figura 6.35: Transformación de una relación en estrella

Caminos 1-N

Otro tipo de acceso compuesto se obtiene al seguir los caminos de las relaciones 1-(1:N). En la figura 6.36 se muestra que la clase A tiene una relación 1-N con la clase B y ésta con la clase C. El acceso compuesto entre la clase A y C consistiría en una cadena de índices. El primer índice mostraría las instancias de B con las que se relaciona un A y el siguiente

índice mostraría las instancias de C que se relacionan con la B seleccionada en el primer índice. Al inicio de la cadena puede estar un índice de objetos A o la clase navegacional A. Este tipo de navegación es útil para representar colecciones de colecciones.

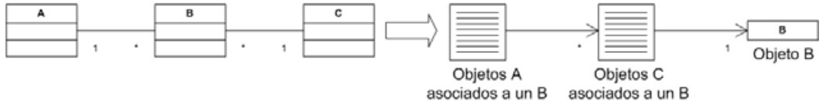


Figura 6.36: Transformación de una cadena de relaciones 1-N

Caminos 1-N inversos

El encadenamiento inverso de relaciones 1-N se puede efectuar cuando la cardinalidad mínima y máxima del primer extremo es 1, y no 0:1. En este caso se tiene como primer elemento un índice que selecciona las instancias de C, después se navega a un índice que muestra los objetos B relacionados con C, y finalmente un índice de los A relacionados con B (figura 6.37).

Figura 6.37: Transformación inversa de una cadena de relaciones 1-N

6.7.3. Atajos

Los atajos representan caminos que pueden establecerse directamente entre dos clases, debido a que existe una relación entre ellas, a través de otro elemento. Se tienen las clases A, B y C con relaciones con cardinalidad:

Cardinalidad 1-1-1. Es posible tener un atajo desde A a C, y es aplicable en forma transitiva a las relaciones 1-1 y 1-N de C.

Cardinalidad 1-1-N Es posible tener un atajo desde A hasta la estructura de acceso derivada de la relación 1-N.

Tabla 6.5: Marcas aplicables al modelo estructural

Marca	**Elemento**	**Efecto en el modelo navegacional**
Buscable	Atributo	El atributo puede mostrarse en formularios de búsqueda
Buscable	Clase	Un formulario de búsqueda es añadido a la página Home
Omitido	Clase	La clase no se muestra, ni sus subclases
Omitido	Atributo	El atributo no se muestra
Omitido	Asociación	La asociación no se muestra
Fusión	Asociación 1-1	Las clases asociadas se unen
Fusión	Asociación 1-N	La estructura de acceso derivada de la asociación se muestra en el nodo de la clase con cardinalidad 1
IndexTour	Asociación 1-N	La asociación se transforma en un índice y una visita guiada
Home	Modelo	Crea página Home, menú e índices
NoNavegable	Extremo de asociación	No se tiene en cuenta el extremo de la asociación
Compone	Extremo de asociación	Muestra al nodo asociado como composición
NoHome	Clase	No es incluida en el menú del Home
Tour	Asociación 1-N	La asociación se transforma en una visita guiada
CircularTour	Asociación 1-N	La asociación se transforma en en una visita guiada circular
UnoaUno	Asociación 1-N	La relación 1-N se transforma en una 1-1 en la que se visita el último nodo del extremo con cardinalidad 1

6.7.4. Marcas aplicables al modelo estructural

Para facilitar las transformaciones del modelo estructural al modelo navegacional, se añaden marcas o *tagged values* a los elementos, éstos representan la intromisión del diseñador en la transformación. La tabla 6.5 presenta las etiquetas identificadas, y su efecto en una transformación. Las reglas de transformación definidas previamente deben tener en cuenta el uso de estas marcas.

Marca omitido

La marca *Omitido* indica que una clase, asociación o atributo no debe ser tomado en cuenta al momento de efectuar la transformación de modelos. Cuando una clase es omitida, también lo son sus asociaciones y atributos (figura 6.38).

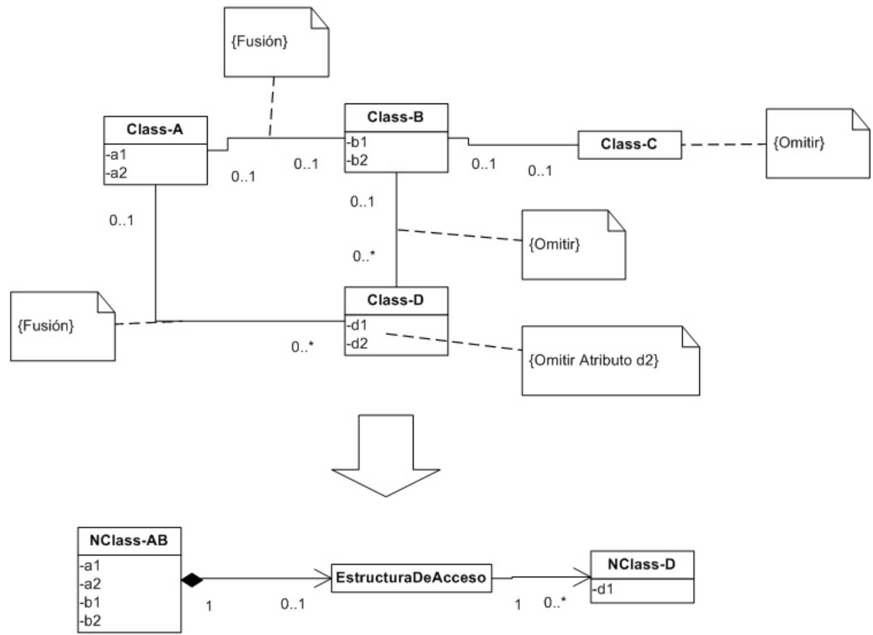


Figura 6.38: Marcas de omitido y fusión

La marca *Omitido* debe verificarse que no exista cuando se transforma un atributo, clase o asociación, si la tienen entonces no se aplica la regla. En el caso de las asociaciones debe verificarse que ninguno de sus extremos está relacionado con una clase que debe omitirse.

Marca fusión

La marca *Fusión* aplicada a una asociación con cardinalidad 1-1 indica que las dos clases asociadas forman una sola clase navegacional. Si esta marca se aplica a una asociación 1-N entonces la estructura de acceso derivada de la asociación se muestra como parte de una clase navegacional compuesta (figura 6.38).

Cuando la marca *Fusión* se aplica a una relación 1-1 entre clases, la clase asociada al primer extremo de la asociación se usa como pivote de la transformación. Es decir, un atributo que pertenece a la clase en el segundo extremo de la asociación se mueve a la primera. La fusión podría aglomerar

a varias clases que tienen asociaciones 1-1 entre ellas, sin embargo, se ha impuesto la restricción de que una clase sólo participe como máximo en una fusión. Para poder aplicar la transformación normal de clases, asociaciones y atributos, se verifica que no tengan la marca *Fusión*.

La transformación de asociaciones debe tener en cuenta, también, si las clases se fusionan. En el caso de que el destino de un extremo de asociación sea una clase fusionable entonces, se debe de cambiar el destinatario.

Marca Tour

En [ISB95] se recomienda usar una visita guiada cuando en una asociación 1-N el número de elementos incluidos es pequeña (unos 10 elementos). La marca *Tour* indica que se debe usar una visita guiada en la transformación de una asociación 1-N. También se pueden usar las marcas *IndexTour* o *CircularTour*. Éstas señalan que la asociación debe transformarse en un índice asociado a una visita guiada, una visita guiada circular o sólo una visita guiada (figura 6.39).

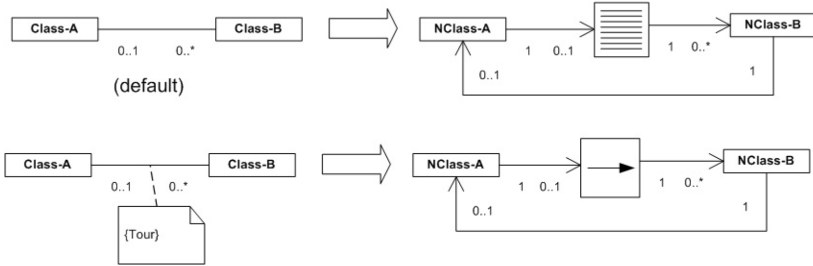


Figura 6.39: Marca Tour

Marca NoNavegable

En algunos casos cuando se tienen asociaciones con cardinalidad 1-1 o 1-N, no se quiere permitir la navegación en alguno de los extremos de la asociación, para evitarlo se usa la marca *NoNavegable* (figura 6.40). La marca *NoNavegable* en la transformación de una asociación 1-1 sirve para establecer el valor del atributo navegable de los extremos de la asociación.

En cambio, en la transformación de la relación 1-N evita la creación de la relación 1-1 o de la estructura de acceso, en este ejemplo, también se tiene en cuenta la fusión como fue explicado anteriormente.

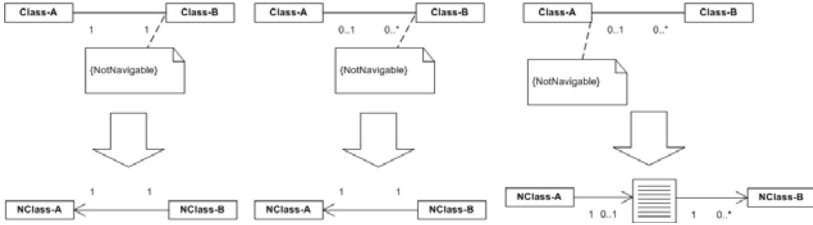


Figura 6.40: Marca NoNavegable

Marca UnoaUno

A las asociaciones 1-N se les puede cambiar el comportamiento de navegación para que parezcan una asociación 1-1 a través de la marca *UnoaUno*. De este modo la clase con el extremo de cardinalidad *0:N* puede tener una asociación navegable hacia la clase con cardinalidad 1, eso es posible si se memoriza el nodo a través del cual se llegó. Por tanto, a la asociación navegable que sale de la clase con cardinalidad 1 se le añade la marca *Memoriza* (figura 6.4).

Marca Home

La creación de la página *Home* puede ser vista como una transformación de modelos. Si la marca *Home* se añade al modelo estructural entonces, se crea una clase navegacional *Home* con un menú, y para cada clase que no tenga la marca *Omitir* se genera un índice y un enlace en el menú. Además, si la clase tiene la marca *Buscable* entonces se genera un formulario de búsqueda, que se añade al menú (figura 6.41). Los atributos de una clase conceptual que se señalan con la marca *Buscable*, deben mostrarse en los formularios de búsqueda relacionados con esa clase. Si la clase no tiene atributos con tal marca, entonces todos sus atributos se muestran en el formulario de búsqueda.

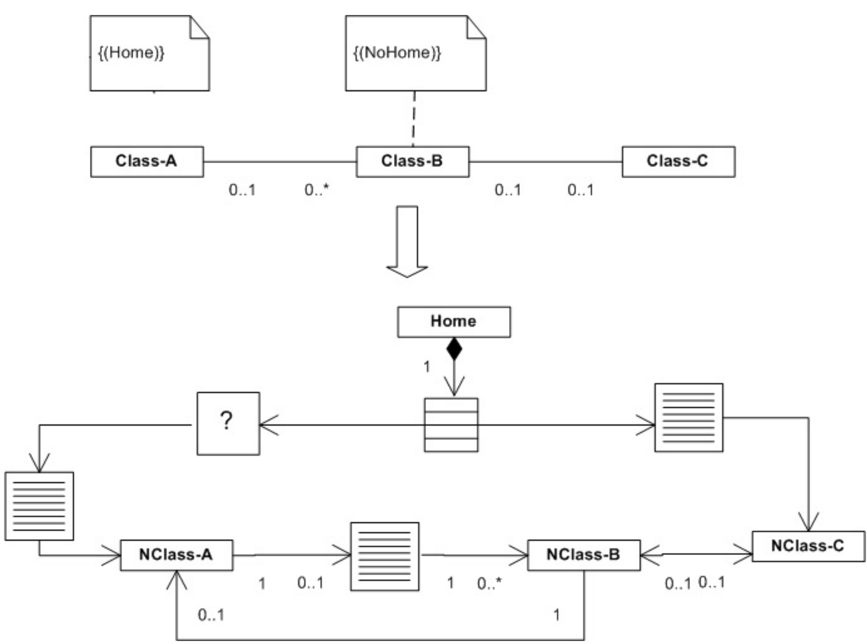


Figura 6.41: Creación del nodo Home

La generación del nodo *Home* se efectúa al momento de transformar el modelo completo, es decir, es una marca que no está relacionada con un elemento particular, sino con todo el modelo. La creación del nodo *Home* se hace a través de dos reglas, una que tiene en cuenta dicha marca y otra para el caso contrario.

6.7.5. Vista del modelo estructural por actor

El etiquetado del modelo estructural define la vista del modelo estructural que tiene cada actor. Un actor tiene asociado un modelo estructural etiquetado, lo que permite obtener un modelo navegacional débil con una particular visibilidad y estructuras de acceso.

La marca omitido permite eliminar clases, atributos y asociaciones, es decir, determina qué elementos nunca son visibles. Si para un actor es importante mostrar ciertos atributos o clases juntas entonces, se utiliza la marca de fusión. Otra marca que evita la visibilidad es la marca de *NoNavegable*,

es especialmente útil con las relaciones 1-N, para evitar la navegación desde un nodo relacionado a N elementos.

Cuando el modelo navegacional está dirigido por el proceso (ejemplos en el capítulo 9), entonces, las marcas relativas a la creación de un nodo inicial (*Home*), no pueden ser usadas. Las etiquetas que no se usarían son: *Home, NoHome y Buscable* (la última cuando se aplica a clases).

6.8. Conclusiones

En este capítulo se ha mostrado la especificación de un modelo navegacional como instancia del metamodelo MOF. Además, se han definido restricciones OCL que sirven para verificar la correcta construcción de los modelos navegacionales. Los modelos navegacionales de los diferentes actores se obtienen a partir del modelo de proceso definido en BPMN. El primer paso es la proyección del proceso por actor de acuerdo al algoritmo definido. El segundo paso es la aplicación de las reglas de transformación del modelo de proceso al modelo navegacional, y finalmente se aplican las reglas para transformar el modelo estructural a modelo navegacional. El modelo navegacional final es el resultado de la unión de los modelos navegacionales fuerte y débil. Para definir las reglas, se han identificado los patrones de transformación entre modelos. También, se han identificado los modelos de marcado para el modelo de proceso y el modelo estructural. Las reglas de transformación se han especificado en ATL y tienen en cuenta a los modelos de marcado.

Hay que destacar que el modelo navegacional obtenido permite la ejecución del proceso que se construyó durante la fase de análisis. Este modelo establece la navegación en dos niveles uno dirigido por los enlaces de proceso y otro por los obtenidos del modelo estructural. En el primer caso los enlaces conectan vistas de tarea, que representan el estado de la interfaz respecto al proceso en ejecución. La presencia de enlaces de proceso depende del estado del proceso y del nodo que se visita durante la navegación. En el segundo caso, los enlaces conectan datos del modelo estructural.

Capítulo 7

Diseño de la interfaz abstracta de usuario

7.1. Introducción

La interacción entre los usuarios y las aplicaciones hipermedia se efectúa a través de una interfaz de usuario. Existen diversas tecnologías de hipermedia tales como: Notecards [HMT87] o StorySpace [Ber02], aunque en la actualidad la prevalente es la Web. Para poder modelar las interfaces de usuario de forma independiente de la tecnología, se definen de forma abstracta o genérica.

Una Interfaz Abstracta de Usuario (IAU) consta de dos partes: una sección con los elementos propiamente de la interfaz (etiquetas, enlaces, botones, etc.) y otra que incluye las relaciones entre los elementos de la interfaz y la instancia de datos que se mostrará. En MDHDM las IAU y las instancias de datos se definen como documentos XML. Una instancia de datos satisface un esquema XML derivado del modelo navegacional, y su relación con los elementos de la interfaz se especifica a través de expresiones XPath [DC99]. Se ha elegido XML porque así se tiene cierta independencia de la tecnología que proporciona los datos de las instancias de clases navegacionales y estructuras de acceso.

La técnica presentada es parecida a las interfaces abstractas de OOHDM

[SRJ96], donde del modelo navegacional se genera una ontología expresada en *Ontology Web Language* (OWL) [vHM04], y se vinculan los elementos de la IAU con los elementos del modelo navegacional mediante expresiones en *Resource Description Framework* (RDF) [CK04]. Su principal desventaja es que esas definiciones sirven sólo como especificaciones, y, además, definir una interfaz gráfica en OWL es un tanto complejo (debido a que se expresan sentencias de lógica descriptiva codificadas en XML). En lugar de definir las interfaces abstractas de ese modo, se ha optado por vincular las IAU y los datos con XPath. Las ventajas de hacerlo así son que las interfaces son más sencillas de definir, y además obtener una implementación ejecutable de la interfaz abstracta es más directo. La desventaja de definirlo con XPath es que la información del modelo navegacional no está disponible; sin embargo, el modelo navegacional se usa como base para generar las IAU. De cada clase o estructura de acceso del modelo navegacional se genera un esquema XML que deben satisfacer sus instancias, y también, una IAU por omisión.

El capítulo está estructurado del siguiente modo: en la sección 7.2 se presenta un metamodelo que incluye conceptos tales como elementos de navegación, textuales, multimedia, y formularios. Todos ellos tienen una representación en XML. La especificación de los esquemas XML de los datos que se presentarán en la interfaz se define en la sección 7.3. En la sección 7.4 se muestra cómo se definen las IAU y cómo se enlazan los datos y la interfaz. En la sección 7.5 se presenta un ejemplo de interfaz. Finalmente en la sección 7.6 se dan las conclusiones del capítulo.

7.2. Metamodelo de la IAU

En esta sección se describe el metamodelo para la definición de las IAU (ver figura 7.1). Las clases abstractas que se encuentran en la parte superior de la jerarquía y que definen las características genéricas de los demás elementos son las siguientes:

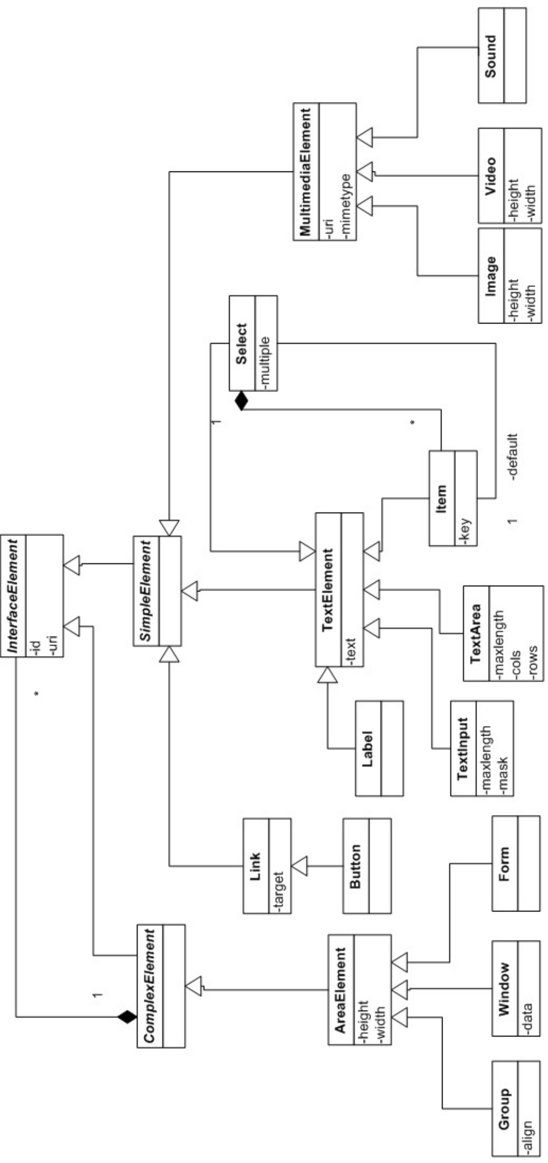


Figura 7.1: Metamodelo de la IAU

InterfaceElement. Es el elemento raíz y cualquier otro desciende de éste. Su atributos son: *id* que sirve como identificador del elemento, y *uri*, que permite localizar el elemento dentro de una red de hipertexto.

SimpleElement. Es un elemento indivisible de la interfaz, que no contiene a ningún otro como parte de él. Los elementos simples pueden ser elementos de texto, navegación o multimedia.

ComplexElement. Es un elemento compuesto, es decir, agrupa a un conjunto de elementos de la interfaz, los cuales pueden ser a su vez simples o compuestos.

Los elementos simples se especializan en dos subclases abstractas *TextElement* y *MultimediaElement*, y en una clase concreta *link* (definida más adelante).

TextElement. Representa a los elementos de tipo texto, como áreas de texto, campos de entrada o etiquetas. Contiene el atributo *text* que representa al contenido.

MultimediaElement. Representa a los elementos multimedia, como vídeos y sonidos, entre otros. Su atributo *mimetype* indica el tipo de fichero del elemento.

Los *ComplexElement* se especializan en la clase abstracta *AreaElement.*

AreaElement. Representa un área de la interfaz y sus propiedades son *width* y *height.*

A continuación se presentan las especializaciones definidas para cada clase abstracta. Los elementos que sirven para agrupar a varios elementos de la interfaz son llamados contenedores. El metamodelo contiene a los siguientes: *window*, *group* y *form*, todos descienden de *AreaElement.*

Window. Una ventana es un contenedor de un documento hipermedia. Está asociada a una instancia de datos en formato XML. El atributo *data* de la ventana es un *uri*, que permite localizarlos.

Group. Un grupo es un contenedor de elementos de la interfaz, que no puede contener ventanas. Tiene un atributo que define la alineación

de los elementos que contiene, siendo las más comunes horizontal y vertical.

Form. Es un grupo que envía como información contextual el contenido de sus campos de texto y de selección al nodo navegacional siguiente, que es determinado por los enlaces o botones del formulario.

Los elementos para captura o despliegue de texto descienden de *TextElement*.

Label. Es una etiqueta en la interfaz, que despliega el valor del atributo *text*.

TextInput. Es un campo de entrada, que permite la captura de su atributo *text*. Tiene una longitud máxima *maxlenght*, y un campo *mask* que sirve para efectuar validaciones acerca del texto capturado.

TextArea. Es un campo de entrada, que permite la captura de su atributo *text*, que se despliega en un área determinada por las variables: *cols*, para la longitud; y *rows*, para las filas. El texto capturado tiene una longitud máxima de *maxlength* caracteres.

Item. Es un par de valores *key* y *text*, que son usados en los elementos de selección (*select*).

Select. Contiene n elementos Item, y sirve para seleccionar a algunos de ellos. El elemento *select* contiene el atributo booleano *multiple*, que indica si permite seleccionar uno o más Items. También, contiene el atributo *default* que indica el *key* del ítem seleccionado por omisión.

Los elementos de navegación pueden ser hiperenlaces o botones. Los botones son una especialización de los enlaces.

Link. Es un hiperenlace que tiene como etiqueta el valor del atributo *name* y como destino de navegación su campo *target*.

Button. Un botón está asociado a un formulario y envía la información capturada en los campos del formulario. El botón tiene como etiqueta el valor del atributo *name* y como destino de navegación su campo *target*.

Los elementos de la interfaz que despliegan elementos multimedia son imágenes, vídeos y sonidos. La composición de elementos multimedia complejos se consigue a través de grupos de elementos multimedia dentro de un contenedor.

Image. Representa una imagen, y tiene como propiedades su ancho y alto.

Video. Representa un archivo de vídeo y tiene como propiedades su área de despliegue (ancho y alto).

Sound. Representa un archivo de sonido.

Sin embargo, se podrían incluir nuevos elementos multimedia como *Flash* heredando las propiedades del elemento *MultimediaElement*.

La representación XML de los elementos del metamodelo se muestra en la tabla 7.1. Como se verá, esta representación permite definir las interfaces mediante documentos XML similares a los mostrados en el apartado 7.5.

Tabla 7.1: Representación XML de los elementos de la interfaz

<table>
<tr><td>Group</td><td><group>...</group></td></tr>
<tr><td>Window</td><td><window data="uri">...</window></td></tr>
<tr><td>Label</td><td><label text="texto desplegado"/></td></tr>
<tr><td>TextInput</td><td><textinput maxlenght="x" type="y"/></td></tr>
<tr><td>TextArea</td><td><textarea cols="x" rows="y" maxlenght="z">texto</textarea></td></tr>
<tr><td>Item</td><td><item key="x">value</item></td></tr>
<tr><td>Select</td><td><select multiple="true|false"><item>*</select></td></tr>
<tr><td>Link, Button</td><td><link>
<name>etiqueta</name>
<target>uri</target>
</link></td></tr>
<tr><td>Image</td><td><image width="x" height="y" data="uri"></td></tr>
<tr><td>Video</td><td><video width="x" height="y" data="uri"></td></tr>
<tr><td>Sound</td><td><sound data="uri"></td></tr>
</table>

7.3. Esquemas de las instancias de datos

Del modelo navegacional se generan, mediante *helpers* en ATL, los esquemas XML que deben satisfacer las instancias de datos para clases y estructuras de acceso. Los *helpers* se presentan a continuación en pseudocódigo, pero se pueden consultar completos en el Apéndice C. La información de los modelos está en ficheros XMI, y este tipo de modelos es fácil de consultar utilizando OCL.

El pseudocódigo de la generación del esquema XML de las clases navegacionales se muestra en el listado 7.1. Inicialmente se genera un elemento complejo con el nombre de la clase; como cada instancia es identificada mediante un identificador de objeto, se añade un elemento simple *oid*. Por cada atributo de la clase, se construye un atributo simple con el mismo nombre y tipo.

Listado 7.1: Generación del esquema XML de una clase navegacional

```
EsquemaXMLClase
Entrada: Clase Navegacional C
Salida: Esquema XML de la clase navegacional C
Se genera un elemento complejo con nombre C.nombre
Por cada atributo A en C:
   Se genera un elemento simple con el tipo de dato de A
Por cada elemento navegacional E asociado
por composición a C:
   Se genera el esquema XML del Elemento E
```

El pseudocódigo para la generación del esquema XML de las visitas guiadas se muestra en el listado 7.2. Una visita guiada debe estar relacionada con una clase navegacional a la que pertenecen la colección de objetos que presenta.

Listado 7.2: Generación del esquema XML de una visita guiada

```
EsquemaXMLVisitaGuiada
Entrada: Visita guiada C
Salida: Esquema XML de la visita guiada C
Se genera un elemento complejo con nombre C.nombre
Se genera un elemento simple correspondiente
```

```
        al enlace de avance.
Se genera un elemento simple correspondiente
        al enlace de retroceso.
Se genera una referencia con multiplicidad 1
al elemento complejo de la clase navegacional E
que muestra la visita guiada C.
```

Los índices contienen n referencias a otros objetos navegacionales. Los atributos de una clase navegacional que deben mostrarse en un índice se indican en el modelo navegacional mediante la etiqueta *Index* o la pareja *(Index, Name)*; el primer caso indica que un atributo debe mostrarse en cualquier índice, y la otra etiqueta lo específica para un índice en particular. Es decir, en el modelo navegacional se puede definir la vista de una clase navegacional en un elemento determinado.

El esquema XML de un índice define un elemento complejo con su mismo nombre y contiene elementos de un cierto tipo, los cuales son indicados como contenido de la secuencia interna y con multiplicidad n. Si el índice muestra una vista de una clase navegacional, entonces se genera un elemento complejo correspondiente a esa vista, y se usa su nombre en el elemento referenciado por el índice. La generación de los esquemas XML de los índices se muestra en la figura 7.3.

Listado 7.3: Generación del esquema de un índice

```
EsquemaXMLÍndice
Entrada:Índice C
Salida: Esquema XML del índice C
Se genera un elemento complejo con nombre C.nombre
Se genera una referencia al elemento complejo
con multiplicidad n de la clase navegacional E
que muestra el índice C
```

7.4. Interfaces Abstractas de Usuario

Las IAU de los elementos del modelo pueden tener diversos diseños. En esta sección se describen las interfaces abstractas definidas por omisión para clases navegacionales, índices y visitas guiadas, los elementos más comunes

en el modelo navegacional, y cualquier otra IAU sería parecida a éstas. Las IAU, al igual que los esquemas XML de los datos, se generan a través de *helpers* en ATL.

Para construir las IAU se hace uso de los elementos del metamodelo de interfaz gráfica (sección 7.2), y de sentencias XPath que vinculan a éstos con los esquemas XML de datos (sección 7.3). Una sentencia de vinculación de datos tiene una sección de selección de los elementos XML de entrada, y una sección de salida con el elemento de la interfaz instanciado. Los elementos XML que permiten generar las interfaces, están precedidas por el espacio de nombres *aui*, y se explican a continuación:

interface. Este elemento indica el inicio de la definición de una IAU.

match. Permite seleccionar un conjunto de nodos en un documento XML, la expresión de selección se indica en su atributo *select*, y se define en XPath. La sección de salida es evaluada para cada nodo seleccionado.

attribute. Crea un atributo en el nodo XML que lo contiene; la expresión XPath de creación se define en su atributo *value*.

value. Asigna como contenido del nodo XML el resultado de la expresión XPath de su atributo *select*.

7.4.1. Vista de tareas

Las vistas de tarea carecen de interfaz debido a que son un nodo virtual. Sin embargo, modifican las IAU de los demás elementos dependiendo del estado del proceso. La vista de tarea añade enlaces de proceso a las IAU. Un enlace de proceso en la IAU no se distingue de los otros enlaces excepto porque lo identifica un parámetro adicional. Una vista de tarea puede estar contenida dentro otra, en cuyo caso, se tienen que aplicar todas las vistas de tarea derivadas de los subprocesos contenedores. El listado 7.4 muestra el pseudocódigo para generar las IAU para una vista de tarea específica. Las IAU generadas son la de todos los nodos alcanzables (navegando) desde el nodo inicial de la vista de tarea.

Listado 7.4: Generación de la IAU de una vista de tarea

```
GeneraIAU(T)
Entrada: Vista de tarea T
Inicio = Nodo inicial de la vista de tarea
Para cada nodo N alcanzable desde T.Inicio
         GeneraIAU(T,N)
```

La interfaz abstracta de un elemento de la interfaz es modificado por una vista de tarea del siguiente modo: primero se genera la IAU del elemento, y después se añaden los enlaces de proceso en los nodos de activación de los mismos, o en el nodo *Window* si son activos en cualquier nodo (ver listado 7.5).

Listado 7.5: Generación de la IAU de una vista de tarea(2)

```
GeneraIAU(T,N)
Entrada: Vista de tarea T
Salida: InterfazAbstracta I
Nodo navegacional: N
    InterfazAbstracta I = IAU(N, null)
    Para cada enlace de proceso P en T
       Si P está activado en cualquier nodo entonces
          Se añade P a elemento Window de I
       Sino
                Si N es un nodo de activación de P entonces
              Se añade P al elemento Window de I
          Sino
             Set{Nodo} I = Conjunto de nodos internos de I
             Para cada x en I
                Si x es un nodo de activación de P entonces
                     Se añade P al elemento x
```

A continuación se definen las funciones IAU de los elementos clase navegacional, índice y visita guiada.

7.4.2. IAU de las clases navegacionales

La interfaz abstracta por omisión de una clase navegacional se muestra en el listado 7.6.

- Una clase navegacional se presenta en una ventana.
- Cada atributo genera un grupo de etiquetas con su nombre, una etiqueta está destinada al nombre y otra al valor del atributo.
- En el caso de las asociaciones navegables, se construye un enlace que tiene una etiqueta y un destinatario. Generalmente, se incluye como parámetro el identificador del objeto origen.
- Cuando en el modelo navegacional hay asociaciones compuestas, entonces se tiene que anidar la definición del otro elemento navegacional.

Listado 7.6: Generación de la IAU de una clase navegacional

```
Entrada: Clase navegacional C
Entrada: Window raíz
Salida: InterfazAbstracta I
Si raíz es null entonces
    Se genera un elemento  I = interface
        Se genera un elemento Window W en I
Dentro de la ventana W:
   Se genera una etiqueta con el nombre de C en W
   Por cada atributo A en C:
        Se genera una etiqueta con el nombre de A.nombre
        Se vincula el valor de la etiqueta a la expresión
            XPath
    //A.nombre
    Por cada asociación A sin  composición en C:
      Se genera un enlace:
       con destino al elemento A.target
       con parámetro oid asociado
       a la expresión XPath /oid
   Por cada asociación A por composición en C:
       IAU(A.target,W)
```

7.4.3. IAU de las visitas guiadas

La interfaz abstracta de una visita guiada se muestra en el listado 7.7. Para el elemento de la visita guiada correspondiente a la clase navegacional

se aplican los pasos explicados en la sección anterior. Después se vinculan los enlaces del elemento siguiente y anterior con los *oids* indicados en la instancias de datos.

Listado 7.7: Generación de la IAU de una visita guiada

```
Entrada: Visita guiada V
Entrada: Window raíz
Salida: InterfazAbstracta I
Si raíz es null entonces
    Se genera un elemento  I = interface
        Se genera un elemento Window W
Dentro de la ventana W:
   Se genera una etiqueta con el nombre V.nombre
   Se genera el enlace de retroceso
       con destino a V
       con parámetro oid asociado
       a la expresión XPath //previous
   Se genera el enlace de avance
       con destino a V
       con parámetro oid asociado
       a la expresión XPath //next
   IAU(V.target, W)
```

7.4.4. IAU de índices

La interfaz abstracta de un índice se muestra en la figura 7.8. Se define de forma muy similar al despliegue de una clase navegacional, pero la expresión XPath en el *tag match* indica que se debe efectuar de forma repetitiva para un conjunto de nodos XML.

Listado 7.8: Generación de la IAU de un índice

```
Entrada: Índice C
Entrada: Window raíz
Salida: InterfazAbstracta I
Si raíz es null entonces
    Se genera un elemento  I = interface
        Se genera un elemento Window W
Dentro de la ventana W:
```

```
Se genera una etiqueta con el nombre C.nombre
Se genera un grupo de elementos
Por cada elemento en la instancia de datos:
    Se crea un enlace al elemento asociado
      Se le añade el parámetro oid asociado
      a la expresión XPath ./oid
    Por cada atributo A de C
       Se genera una etiqueta
       Se asocia a la expresión XPath
       ./Nombre del atributo
```

7.5. Ejemplo de interfaz abstracta

A continuación se explican los elementos de una IAU a través de un ejemplo, cuyo modelo navegacional se muestra en la figura 7.2.

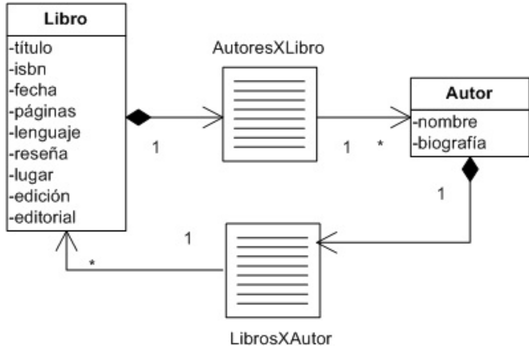


Figura 7.2: Interfaz instanciada de un libro

Una interfaz de usuario que muestra un *libro* podría ser como en la figura 7.3. En la figura se muestran los atributos del *libro* tales como *título, isbn, número de páginas,* entre otros. Nótese que el *autor* es un enlace que permite navegar a la información detallada del *autor*.

La interfaz del *libro* proviene del listado 7.9. El *libro* se presenta en un área de la interfaz, indicada por el elemento raíz *window*. Después cada uno de los atributos del objeto se muestra en una etiqueta, el elemento *label* despliega a su atributo *text*. La imagen del libro está especificada a través

Figura 7.3: Modelo navegacional del ejemplo de IAU

del elemento *image*. El *autor* es un enlace y se define con el elemento *link*. Existen además dos enlaces de proceso *Comprar* y *Buscar más*.

Listado 7.9: Interfaz instanciada de un libro

```
<window data="/book?oid=5865R">
   <group align="horizontal">
       <image src="123450.jpg"/>
       <group>
                   <label text="La nausea" />
                   <label text="de" />
                   <link>
                      <label>Jean Paul Sartre</label>
                      <target>autor</target>
                          <params>
                             <oid>34687</oid>
                          </params>
                   </link>
                   <group align="horizontal">
                         <label text="Edición:" />
                         <label text="1" />
                   </group>
                   <group align="horizontal">
                         <label text="Isbn:" />
                         <label text="9789500392624" />
                   </group>
```

```
                </group>
    </group>
    <link>
            <label>Comprar</label>
        <target>ModificarCarro</target>
            <params>
                <processLink>AgregaAlCarro</<processLink>
            </params>
    </link>
    <link>
            <label>Buscar más</label>
        <target>Busqueda</target>
            <params>
                <processLink>BuscarMas</<processLink>
            </params>
    </link>
</window>
```

Suponiendo que la vista de tarea actual es *seleccionar libros*, partir de ella se puede llegar a la vista de tarea *editar el carro* y a la de *buscar*. Cuando se empieza a generar la IAU entonces se crearía primero el elemento *Window* y se la añadirían los dos enlaces de proceso (ver 7.10).

Listado 7.10: IAU de un libro en la vista de tarea seleccionar

```
<aui:interface>
<window>
    <link>
            <label>Añadir al carro de compras</label>
        <target>ModificarCarro</target>
            <params>
                <processLink>AgregaAlCarro</<processLink>
            </params>
    </link>
    <link>
            <label>Buscar más</label>
        <target>Buscar</target>
            <params>
                <processLink>BuscarMas</<processLink>
            </params>
    </link>
```

```
</window>
</aui:interface>
```

La IAU del *libro* se completaría con la sección construida a partir de la clase navegacional libro. Algunos elementos son, por ejemplo:

- La etiqueta *título* que muestra el contenido del elemento *título* de la instancia de datos, porque está indicado mediante la expresión XPath *//título*.
- La lista de autores procede de un índice, por lo que la interfaz abstracta debe tener un elemento que permita la iteración sobre una colección de nodos.
- El elemento <*aui:match*> sirve para seleccionar un conjunto de nodos e iterar sobre éste, en el ejemplo, se desplegaría el nombre de todos los *autores*.

Listado 7.11: Interfaz abstracta de la clase libro

```
<aui:interface>
 <window>
  <group align="horizontal">
   <image ref="//imagen"/>
   <group>
        <label class="título" ref="//título"/>
        <label text="de" />
        <aui:match select="//autor">
         <label class="autor" ref="./nombre"/>
        </aui:match>
        <group align="horizontal">
         <label text="Edición:" />
         <label ref="//edición>
        </group>
        <group align="horizontal">
          <label text="Isbn:" />
          <label ref="//isbn/>
        </group>
  </group>
```

```
   <link>
           <label>Añadir al carro de compras</label>
       <target>ModificarCarro</target>
           <params>
              <processLink>AgregaAlCarro</<processLink>
           </params>
    </link>
    <link>
           <label>Buscar más</label>
       <target>Buscar</target>
           <params>
              <processLink>BuscarMas</<processLink>
           </params>
    </link>
  </window>
</aui:interface>
```

Los datos presentados en la interfaz provienen de un documento XML como el mostrado en el listado 7.12. Este documento XML inicia con el nombre de la clase del objeto, seguido de un elemento con el nombre de cada uno de los atributos y su valor. El *libro* incluye un índice de *autores*, que en este caso sólo contiene un elemento.

Listado 7.12: Instancia XML de un libro

```
<libro>
    <oid>5865R</oid>
    <titulo>La nausea</titulo>
        <isbn>9789500392624</isbn>
        <año>2003</año>
        <edición>1</edición>
        <imagen>123450.jpg</imagen>
        <autores>
           <autor>
              <oid>34687</oid>
                  <nombre>Jean Paul Sartre</nombre>
           </autor>
    <autores>
    <descripción>
           Es la novela que ...
    </descripción>
```

```
</libro>
```

Para poder generar la interfaz de las instancias de la clase navegacional *libro* es necesario vincular los datos a desplegar con los elementos de la interfaz (*label, link, etc.*). Para lograrlo es indispensable conocer el esquema XML de los datos. El esquema XML de los documentos que representan un *libro* se muestra en el listado 7.13, que se obtiene a través del algoritmo mostrado al inicio del capítulo.

Listado 7.13: Esquema de la clase libro

```
<xs:element name="libro" type="xs:string">
<xs:complexType>
  <xs:element name="oid" type="xs:string"/>
  <xs:element name="título" type="xs:string"/>
  <xs:element name="isbn" type="xs:string"/>
  <xs:element name="año" type="xs:integer"/>
  <xs:element name="edición" type="xs:string"/>
  <xs:element name="imagen" type="xs:string"/>
  <xs:element name="descripción" type="xs:string"/>
  <xs:element name="autores">
    <xs:complexType ref="autor" minoccurs="1" maxoccurs="*"/>
  </xs:element>
</xs:complexType>
</xs:element>
```

7.6. Conclusiones

Las IAU permiten especificar una interfaz de modo independiente de la tecnología. En este capítulo se ha mostrado cómo derivar las IAU a partir del modelo navegacional. Se ha definido un metamodelo con los elementos de las interfaces, los cuales incluyen texto, enlaces, multimedia, ventanas, entre otros. Se han especificado los esquemas XML de las instancias de datos que se mostrarán en la interfaz, así como las sentencias que vinculan la interfaz abstracta con los datos a través de expresiones en XPath. Esta técnica que vincula los datos con la interfaz es lo que marca la diferencia con las interfaces abstractas de otros métodos. También, se ha explicado que

las vistas de tareas añaden los enlaces de proceso a las interfaces abstractas dependiendo del estado del proceso debido a que siempre hay una vista de tarea activa.

Capítulo 8

Implementación

8.1. Introducción

En este capítulo se explica cómo se ha implementado el método y se presenta su prototipo. En la sección 8.2 se detalla cómo se construyen los metamodelos y modelos a través del *Eclipse Modelling Framework* (EMF). Una vez que se han definido los modelos es necesario verificarlos, es decir, saber si las restricciones OCL que se han definido se satisfacen. En la sección 8.3 se muestra cómo se verifican las restricciones a través de transformaciones en ATL. La sección 8.4 describe el programa que genera las proyecciones del modelo de proceso para cada actor, éste busca patrones en el flujo de trabajo y que efectúa la sustitución de actividades por subprocesos.

Cuando se cuenta con las proyecciones del modelo de proceso y con el modelo estructural marcado por actor, entonces se pueden aplicar las transformaciones ATL para obtener los modelos navegacionales. La sección 8.5 es acerca del entorno de ejecución de ATL. La técnica empleada para la generación de código se analiza en la sección 8.6. La implementación del prototipo requiere de la definición de algunos PSM, los cuales son descritos en la sección 8.7, incluyendo su función y arquitectura. Finalmente, en la sección 8.8 se dan las conclusiones del capítulo.

8.2. Eclipse Modeling Framework

La definición de los metamodelos y modelos requiere de un marco de trabajo que implemente los metaniveles de MOF. En este trabajo se utiliza el *Eclipse Modeling Framework* (EMF) [BBM03]. EMF es un marco de trabajo para el modelado estructural y la generación de código. Los conceptos disponibles para el modelado son proporcionados por el modelo Ecore que un subconjunto de MOF. EMF provee un editor de metamodelos y otro para sus instancias. También proporciona mecanismos de importación de ficheros XMI con la especificación de metamodelos o modelos. EMF cuenta con un API reflexivo para la manipulación de modelos y notificación de cambios en su estructura mediante eventos. El API de manipulación de modelos también verifica las restricciones de integridad (composición y cardinalidad) definidas por los metamodelos. En la figura 8.1 se muestra un diagrama simplificado de los principales elementos del modelo Ecore.

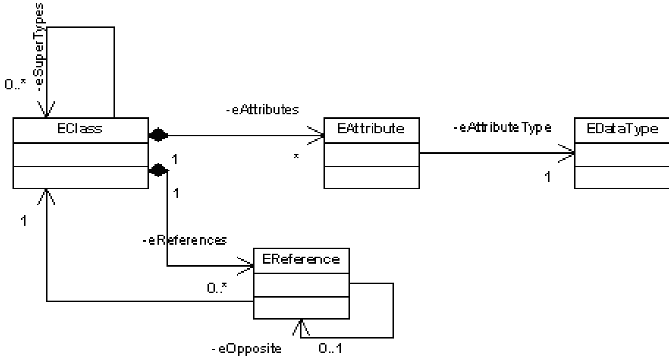


Figura 8.1: Modelo Ecore

La metaclase *EClass* sirve para representar a los elementos que contienen atributos y referencias. Una característica de este modelo es que la navegación es en un solo sentido, a través de referencias, las instancias de la clase *EReference* sirven para representarlas. Para construir una asociación, por tanto, son necesarias dos referencias opuestas. Este punto es relevante, porque las asociaciones en OCL son navegables en ambos sentidos.

La torre reflexiva de EMF sitúa al modelo Ecore en el nivel M3. Los

modelos construidos con los conceptos de Ecore son los metamodelos en M2. La construcción de un modelo Ecore se efectúa a través del editor tabular de modelos (figura 8.2). También es posible hacerlo a través de editores gráficos como Omondo [Omo]. Los metamodelos de proceso, estructural y navegacional se han definido como modelo Ecore. Los pasos para definir un modelo Ecore son:

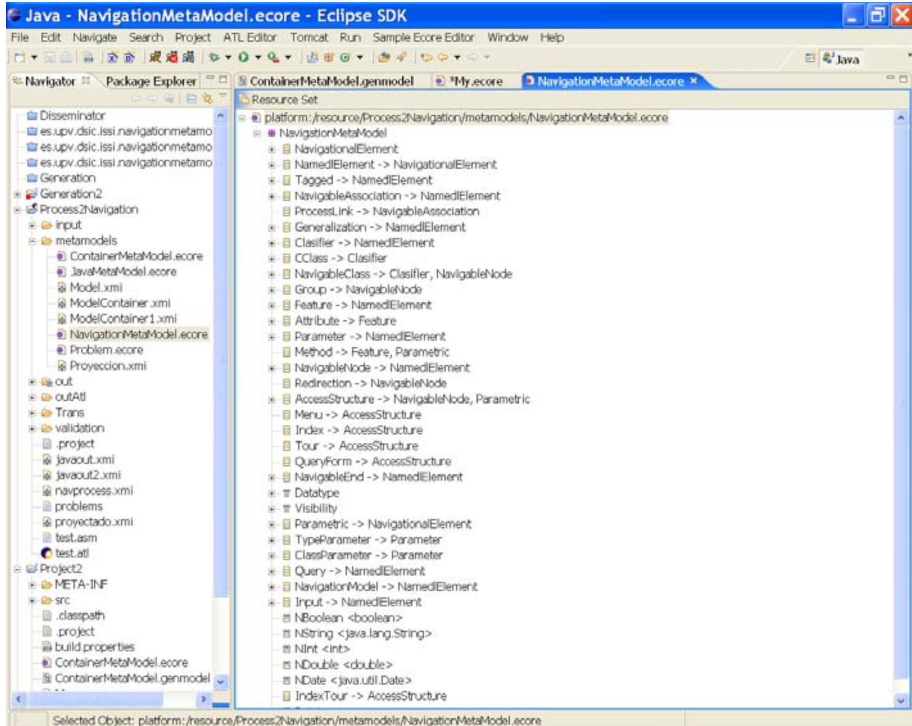


Figura 8.2: Editor tabular de modelos Ecore

1. Se crea un paquete *EPackage* donde residirán las clases del metamodelo.
2. Cada clase corresponde a un *EClass*.
3. Cada atributo a un *EAttribute*.
4. Cada asociación corresponde a dos *EReferences* opuestas.

Para construir un modelo se recurre al editor de instancias; para ejecutarlo, se selecciona la clase contenedora del modelo y se invoca al editor de instancias dinámicas (ver figura 8.3).

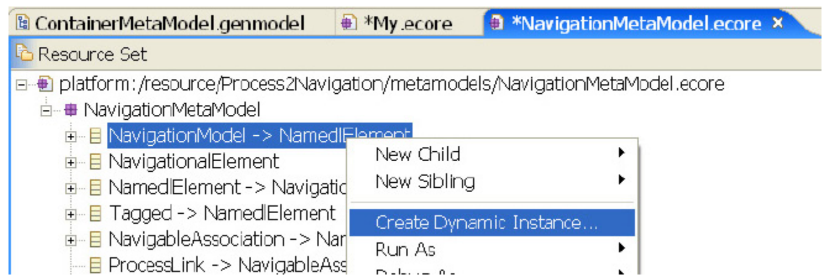


Figura 8.3: Editor de instancias

Una vez creado el objeto contenedor del modelo se pueden añadir otros objetos a la referencia que contiene a los demás elementos del modelo. Después se podrían especificar los elementos que están asociados entre sí, o asignar valores a los atributos (ver figura 8.4).

8.3. Verificación de los modelos

La verificación de modelos consiste en revisar si éstos satisfacen su especificación [Som04], es decir, si son válidos según el metamodelo y sus restricciones OCL. La verificación de un modelo la podemos dividir en dos etapas la primera es la construcción del metamodelo con el editor de instancias, y la segunda la verificación de las restricciones OCL. En el primer caso se comprueba que el modelo sólo contiene elementos definidos en el metamodelo y que cumple con las restricciones de cardinalidad y composición. Sin embargo, no es suficiente para verificar condiciones más complejas. La segunda parte o verificación de las restricciones OCL se efectúa con ATL. Debido a que ATL utiliza OCL como mecanismo de consulta, entonces es posible utilizar una transformación para verificar si se satisfacen las restricciones OCL.

Cada restricción OCL debe reescribirse de forma negada, y está se utiliza como condicionante de una transformación ATL. Por tanto, la transforma-

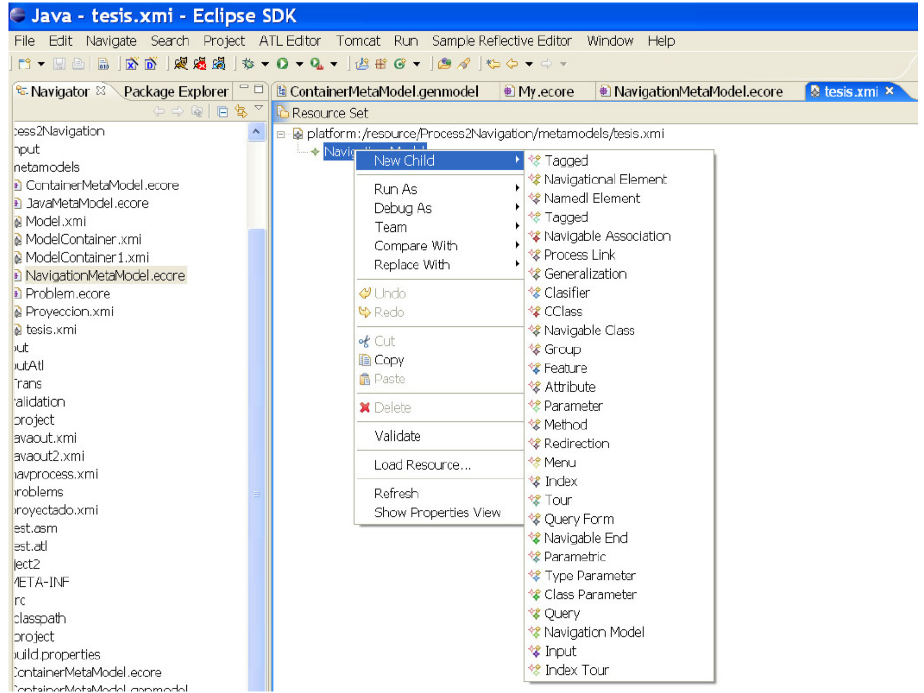


Figura 8.4: Construcción de una instancia

ción se ejecuta si hay elementos en el modelo que no satisfacen la restricción. La verificación de las restricciones OCL es una función de transformación del metamodelo origen al metamodelo de errores.

El metamodelo de errores contiene una clase *Error* que tiene como atributos una descripción del error y una ubicación del mismo. El primer atributo sirve para informar qué tipo de error se ha detectado, y el segundo es para indicar el identificador o nombre del elemento del modelo donde ocurrió. La salida de la transformación es un conjunto de mensajes que indican qué elementos violan alguna restricción.

Por ejemplo, la transformación ATL 8.1 genera un mensaje de error en el modelo de salida si una clase del modelo estructural tiene atributos con el mismo nombre. La sección que determina si una clase debe transformarse verifica si tiene atributos repetidos, mientras que la definición de la restricción OCL es al revés: una clase no tiene atributos repetidos. Las restricciones

OCL que se han presentado en este trabajo se han definido de este modo.

Transformación 8.1: *Transformación* 8.1: Restricción OCL como transformación ATL

```
--Una clase no puede tener atributos con el mismo nombre
rule ClassFeatureChecking {
        from
                aa : ConceptualMetaModel!CClass
                (
                    --Selecciona los atributos
                        let atts: Sequence(
                            ConceptualMetaModel!Attribute) =
                            aa.feature->select(x | x.
                            oclIsKindOf(ConceptualMetaModel!
                            Attribute))
            in
            --Para todos los atributos no hay ninguno con el
                nombre repetido
                        not atts->forAll( c1 | atts->forAll
                            (c2 | c1 <> c2 implies c1.name
                            <> c2.name))
                )
        to
          --Si atts es verdadero entonces tiene atributos
              repetidos y debe transformarse
          --a un mensaje de error
          aao : Problem!Problem
          (
                severity     <- #Error,
                location     <- 'Class ' + aa.name,
                description  <- 'Has attributes with the same
                    name'
          )
}
```

8.4. Proyección del proceso

Para realizar la proyección del proceso se ha desarrollado un módulo denominado proyector. Éste se encarga de aplicar las reglas explicadas en la

sección 6.5 para obtener la vista del proceso que tiene un actor determinado. El módulo es un programa en Java que usa el API reflexivo de EMF para consultar el modelo y buscar los patrones de las reglas. Inicialmente se quiso definir estas reglas con transformaciones ATL declarativas, pero las relaciones recursivas del metamodelo no las hace propicias. Una alternativa era usar ATL de forma imperativa; sin embargo, no se exploró.

Las entradas del programa son un fichero XMI con un modelo de proceso, el nombre del fichero de salida y el nombre del actor por el cual se proyectará. Las reglas se aplican en el orden definido y de forma iterativa. Cada vez que se ejecuta una regla, se vuelve a buscar la siguiente en la lista de reglas desde el principio. El programa termina cuando ya no se puede ejecutar ninguna regla más.

Listado 8.2: Algoritmo principal del proyector por actor

```
//Applies the rules
do{
  applied =  false;
  for(int i=0; i<rules.length;i++)
    if ( rules[i].perform(proc,act) ){
             applied = true;
                 break;
          }
} while(applied);
```

El listado 8.3 muestra la implementación de la regla que reemplaza los ciclos estructurados cuando son efectuados por otro actor. El resto de reglas incluidas en el proyector se implementa de modo similar, la regla del listado 8.3 se presenta de modo ilustrativo.

Listado 8.3: Regla de reemplazo de ciclos

```
public class Rule4 implements ProjectionRule {

...

public boolean perform(Process pr, Actor act)
 {
 boolean applied = false;
```

```
Object[] iter = pr.getOwns().toArray();
for(int i=0;i<iter.length;i++)
 if( iter[i] instanceof PartialJoin && ((Task)iter[i]).getNext
     ().size() == 1){
   Task ts = (Task) ((Task) iter[i]).getNext().get(0);
   if ((ts instanceof Process) && (tc.hasTag(ts, "coordinated
       ") || tc.hasTag(ts, "automatic")))
    if ( ts.getNext().get(0) instanceof PartialSplit ){
         Task orsplit = (Task) ts.getNext().get(0);
      if ((orsplit.getNext().size() == 2 || (orsplit.getInvend
          () != null))
                     && orsplit.getNext().contains(iter[i])
                        ){
       Process proc = cf.createProcessFromCycle("Rule4-" +
           count++,
                       new ArrayList(
                        Arrays.asList(new Object[]{ iter[i],
                              orsplit })), pr);
       ts.setOwner(proc);
           cf.createTag( tc.hasTag(ts, "coordinated") ? "
               coordinated" : "automatic", proc);
           applied = true;
          }
        }
 }
 return applied;
}
}
```

Hay que tener en cuenta que previamente cualquier conjunto de tareas ya fue reemplazado por subprocesos, en otras reglas que se aplicaron primero. Esta regla busca las uniones parciales en un subproceso, y por cada una que encuentre se verifica que uno de los sucesores del nodo sea una bifurcación parcial, que tiene como sucesor de su sucesor a la unión parcial en cuestión. Si la condición anterior se cumple entonces reemplaza el patrón por un subproceso.

8.5. Atlas Transformation Language

Las transformaciones en ATL se ejecutan a través de un *plug-in* de Eclipse que permite la edición de los programas, su depuración y ejecución. La figura 8.5 muestra la consola de ejecución de ATL. Para ejecutar una transformación hace falta indicar los siguientes parámetros: el modelo de entrada (un fichero XMI) y su metamodelo (un fichero Ecore), el nombre del fichero del modelo de salida y su metamodelo (un fichero Ecore), y el nombre del fichero ATL que contiene la función de transformación.

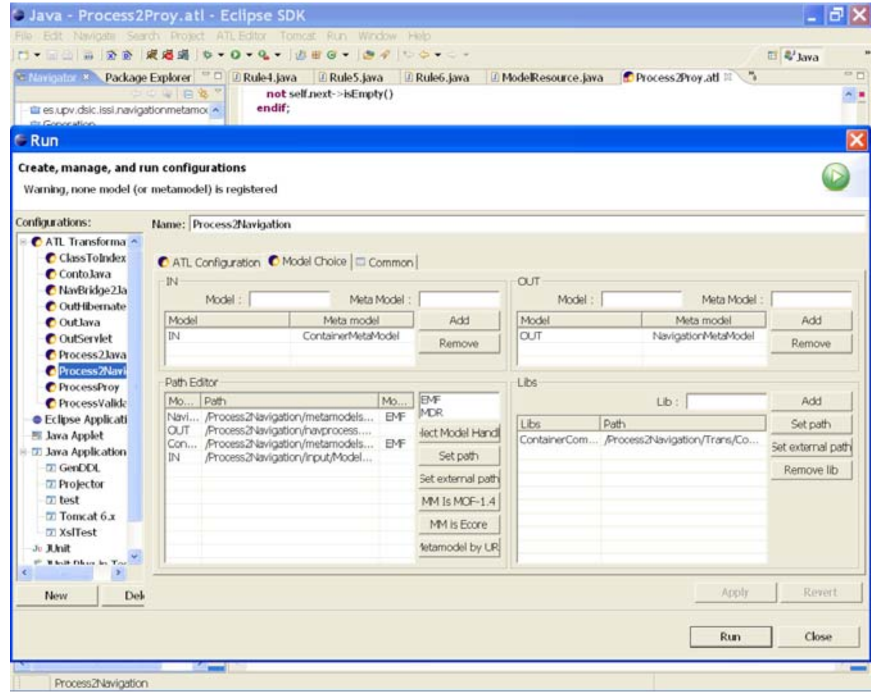


Figura 8.5: Consola de ejecución de ATL

La figura 8.5 muestra la configuración necesaria para ejecutar la transformación llamada *Process2Navigation*, correspondiente al fichero atl con el mismo nombre. El modelo de entrada es identificado como *IN* y es una instancia de *ContainerMetaModel* (la unión del modelo de proceso y estructural). El modelo de salida es identificado como *OUT* y es una instancia

de *NavigationMetaModel*. En la sección llamada *path editor* se indica la localización de los ficheros correspondientes al modelo de entrada y los metamodelos de entrada y salida. También, se indica la ruta del fichero del modelo de salida, que corresponde al resultado de la transformación. Una vez configurada la transformación, se puede ejecutar al oprimir el botón de *run*. El resultado de la transformación se puede observar al abrir el fichero de salida.

Las reglas programadas difieren muy poco de las mostradas en el Apéndice A. Sus diferencias radican en algunos detalles que oscurecerían su definición y explicación. Por ejemplo, en la mayoría de las reglas se ha omitido el nombre del metamodelo de los elementos origen y destino, el cual debe antecederlos siempre.

Otro punto a tener en cuenta durante la ejecución de una transformación es la verificación de valores nulos en las referencias. Es decir, si se navega a través de una asociación vacía se produciría un error en tiempo de ejecución. En el caso de la cardinalidad 0..N se verifica que no está vacía con la función *size()*, y en el caso de la cardinalidad 0..1 que no sea nula con la función *oclIsUndefined()*. Estas revisiones, que sí son parte del programa, no se muestran en la especificación.

El punto de intersección de los modelos de proceso y estructural son los objetos dato y las clases. Este punto representó un obstáculo por que no fue posible efectuar las transformaciones por separado, debido a limitantes en el manejo de referencias a elementos en otros ficheros. Es decir, un modelo de proceso podría tener una referencia a una clase localizada en otro fichero con el modelo estructural. Eso es posible en EMF, pero, el manejo interno de ese tipo de referencias no está especificado en ATL ni QVT, por lo cual se tuvo que definir a ambos metamodelos en un sólo fichero.

El mayor cambio, debido a lo anterior, fue la creación de una nueva metaclase llamada *ModelContainer*, la cual puede contener modelos de proceso y estructurales. Todos los elementos del metamodelo descienden de la metaclase *Element*. Por ello hubo que renombrar algunas metaclases que aparecen en ambos metamodelos y cambiar la definición de reglas de transformación y consultas.

8.6. Generación de código

La generación de código se hace mediante consultas ATL. La diferencia entre una consulta y una transformación es que una consulta recibe un metamodelo y un modelo de entrada, y como salida genera una cadena de caracteres. La salida se construye a partir de *helpers* en OCL. El listado 8.4 muestra un *helper* que genera el código de una clase (JClass) del metamodelo JavaMetaModel. Se observa que el resultado concatena la salida diversas consultas OCL para generar la cadena final. Para otras clases de los metamodelos la técnica de generación de código es similar.

Listado 8.4: Generación de código con ATL

```
helper def : printClass(cc : JavaMetaModel!JClass) : String =
    cc.visibility.toString() + ' class ' + ' ' + cc.name +
        if not cc.supertype.oclIsUndefined() then
          ' extends ' + cc.supertype.name
        else
          ''
        endif
        + '{' + '\n' +
        cc.feature->select(f | f.oclIsTypeOf(JavaMetaModel!
            Attribute))
        ->iterate(i; acc : String = '' | acc + i.toString()) +
        cc.feature->select(f | f.oclIsTypeOf(JavaMetaModel!
            Attribute))
        ->iterate(i; acc : String = '' | acc + i.getMethod()) +
        cc.feature->select(f | f.oclIsTypeOf(JavaMetaModel!
            Attribute))
        ->iterate(i; acc : String = '' | acc + i.setMethod()) +
        thisModule.constructor(cc) +
        '}' + '\n'
        else '' endif
else '' endif;
```

8.7. Modelos específicos de la plataforma

En esta sección se explican los PSM utilizados para implementar los casos de estudio. Los PSM representan alguna tecnología que permite la ejecutabilidad de los modelos de mayor nivel. Los modelos PSM se generan durante el último paso del ciclo de vida de MDHDM.

La arquitectura del código generado se muestra en la figura 8.6. La arquitectura corresponde a un *model-view-controller* (mvc). El *ControlServlet* recibe todas las peticiones desde los navegadores. El componente *Control* se encarga de gestionar el estado de la aplicación, y efectúa las peticiones a la interfaz del componente que implementa el proceso (*ProcessInterface*) y al que gestiona la presentación (*AUIServlet*). El *AUIServlet* se encarga de generar las IAU, e interactúa con los proveedores de datos (*DataServlet*) que corresponden a las instancias de los elementos del modelo navegacional estructural.

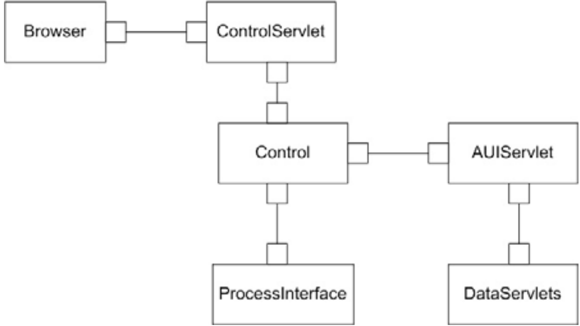


Figura 8.6: Arquitectura del código generado

8.7.1. Modelo de persistencia

El metamodelo al que se transformaron los modelos estructural y navegacional se muestra en la figura 8.7, éste representa a una arquitectura mvc y a un conjunto de clases cuyas instancias serán mostradas en la capa de presentación del mvc.

Se ha elegido como modelo de persistencia para el prototipo al mane-

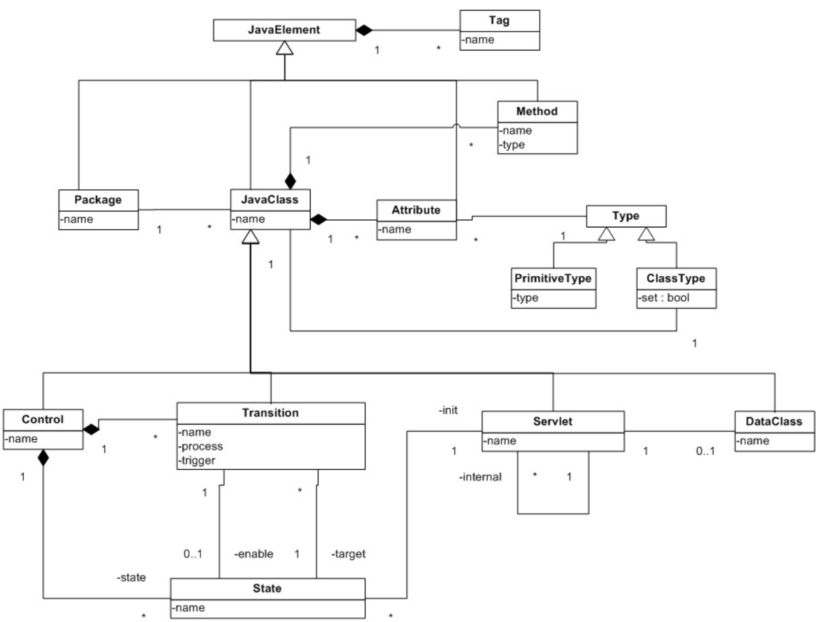


Figura 8.7: Metamodelo Java y MVC

jador objeto-relacional Hibernate [Hib] [BK04]. En el modelo estructural se etiquetan las clases persistentes con la marca *persistent*. Se ha definido un metamodelo de clases Java. Cada clase se transforma en una clase Java (*JavaClass*) (ver figura 8.7). Las asociaciones con cardinalidad 1-1 se transforman en una referencia (atributo *ClassType*) en cada una de las clases; las asociaciones 1-N se transforman en una referencia y un conjunto de referencias (haciendo el atributo *set = true*); y las asociaciones M-N se convierten en dos conjuntos de referencias.

Una vez construido el modelo Java se genera, mediante la técnica de la sección 8.6, la especificación de las clases en Java y los ficheros HBL, éstos últimos son documentos XML que especifican la correspondencia entre las clases Java y las tablas de una base de datos relacional. Hibernate a partir de estos ficheros puede crear la definición de base de datos relacional para manejadores como MySQL [TW06], Oracle [Lon04], y otros.

Respecto al código Java generado se efectúa lo siguiente; una clase tiene dos constructores, uno sin parámetros que sirve para crear nuevos objetos y

otro que recibe un identificador. Los dos tienen llamadas al API de Hibernate para recuperar el estado de la base de datos. Los métodos para leer y escribir los atributos (*set y get*) no requieren de ninguna modificación.

Otro método proporciona el documento XML con el estado del objeto. Este método es la base para construir a las instancias de los objetos navegacionales. Las clases tienen métodos estáticos para recuperar los objetos que están relacionados con otro, uno por cada asociación 1-N y dos para las M-N. Estos métodos son la base para obtener los datos de las instancias navegacionales correspondientes a los índices.

8.7.2. Modelo navegacional

El modelo navegacional se ha transformado a un conjunto de *servlets* [HC01] que proporcionan las instancias de datos de las clases navegacionales e índices como documentos XML. Cada servlet está asociado a un *DataClass* que contiene los datos del objeto navegacional. De cada clase navegacional se ha generado un *servlet* que recibe un identificador de objeto (a través del url), el cual es recibido por el *DataClass* que invoca al constructor de la clase persistente; pasándole tal identificador. Si tiene composiciones entonces invoca a otro *servlet*proveedor de datos y añade ese documento XML al documento que da como resultado (ver figura 8.8).

El caso de los índices es similar, salvo que éstos invocan al método estático de la clase persistente que proporciona los objetos que están relacionados con el objeto del identificador a través de una asociación.

Los enlaces de proceso y las vistas de tarea se usan para definir los estados y transiciones de la clase *Control* del *Model-View-Controller* (MVC) [Ree79] de la aplicación.

8.7.3. Modelo de presentación

Este PSM se encarga de mostrar las interfaces abstractas de usuario de acuerdo al estado del proceso. En la interfaz el estado del proceso está determinado por la vista de tarea activa. Las clases principales de la aplicación se muestran en la figura 8.9, estas conforman un patrón MVC.

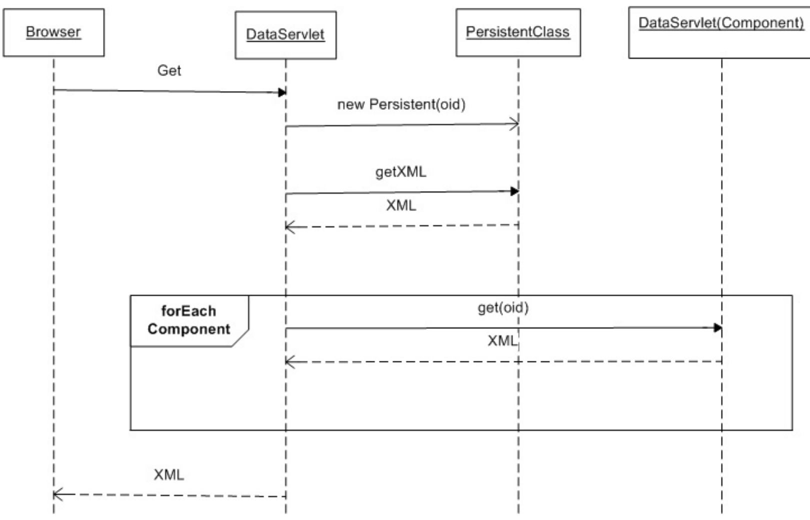


Figura 8.8: Proveedor de datos

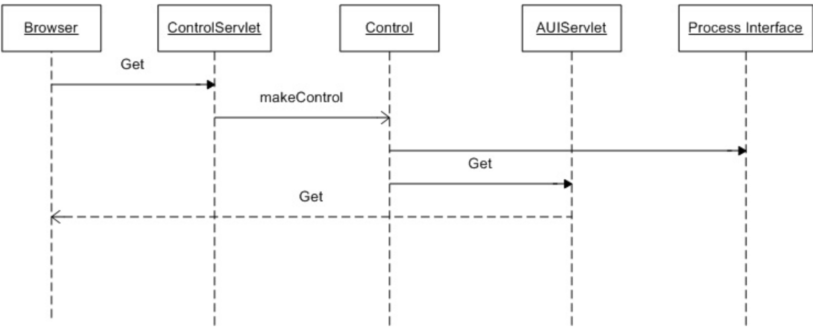


Figura 8.9: Clases del PSM de presentación

La clase *ControlServlet* recibe las peticiones de los navegadores de los usuarios, y reenvía la petición a la clase *Control* que cambia el estado de la interfaz y que interactúa con las actividades del proceso. Una vez que se ha cambiado el estado del proceso y de la interfaz, la navegación se redirige al *AUIservlet* que presenta las interfaces abstractas de usuario (ver figura 8.10).

El *servlet* llamando *ControlServlet* es el que recibe todas las peticiones de los navegadores de los usuarios. Este verifica que la sesión contenga una

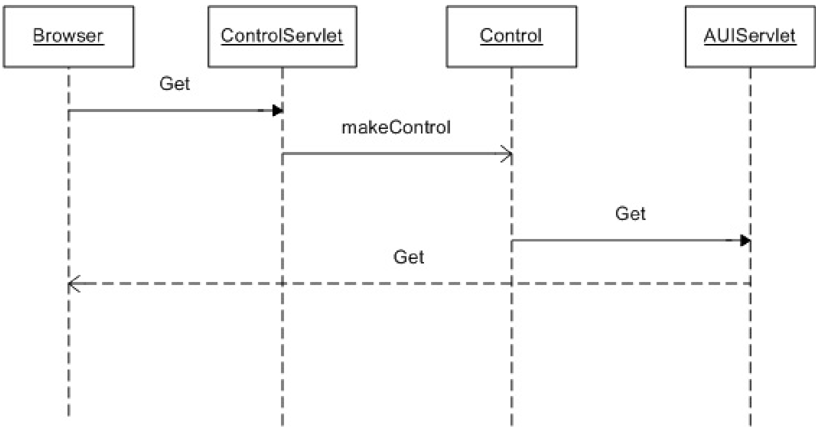


Figura 8.10: Interacción del PSM de presentación

instancia de la clase *Control*. Si no lo tiene entonces crea una instancia, y la guarda en la sesión de navegación. Después invoca el método *makeControl* del objeto instancia de *Control* que se encuentra guardado en la sesión.

La clase *Control* tiene como método principal a *makeControl*. Si el estado de la clase está en inicial entonces busca en los metadatos del modelo cuál es la vista de tarea inicial, la establece como estado inicial de la interfaz. Después obtiene el nodo inicial de la vista de tarea, que es almacenado en la variable *target*, y la redirige la navegación al *AUIservlet*.

Si no está en el estado inicial entonces verifica si el *Request* proviene de un enlace de proceso. Primero verifica que el enlace de proceso pueda ser activado en esa vista de tarea, después obtiene la vista de tarea destino del enlace de proceso. Si se necesita interactuar con una actividad del proceso entonces los datos incluidos en el *Request* sirven como entrada de dicha actividad, después, se redirige la navegación al nodo inicial de la vista de tarea destino. Si el *Request* no contiene una petición de un enlace de proceso entonces se redirige la navegación al nodo destino indicado sin cambiar el estado del proceso ni de la interfaz.

El *AUIservlet* se encarga de presentar las interfaces abstractas de usuario que están almacenadas como ficheros XML. Inicialmente obtiene el nombre del nodo destino, y entonces lee el documento XML que contiene la interfaz

abstracta. De la instancia de la clase *Control* que está en sesión obtiene la vista de tarea activa y los enlaces de proceso que debe mostrar, así como, los nodos dónde están activos los enlaces de proceso. Después, añade los enlaces de proceso a la sección correspondiente del documento XML, es decir, si un enlace de proceso se activa en el nodo X, entonces en el documento XML busca los nodos X y agrega el enlace de proceso a ese nodo. Además, este paso lo repite para la vista de tarea activa y para las vistas de tarea contenedoras que hubiera.

8.8. Conclusiones

En este capítulo se han presentado las tecnologías empleadas para ejecutar los diferentes pasos del método. Los principales soportes tecnológicos son: EMF que permite la definición de metamodelos y modelos MOF (Ecore); y ATL que permite ejecutar las funciones de transformación de modelos. También se utilizó ATL para verificar los modelos a través de transformaciones y como mecanismo de generación de código de los PSM. Finalmente, se presentaron los PSM utilizados en la implementación del prototipo y la arquitectura de éste.

Capítulo 9

Casos de estudio

9.1. Introducción

En este capítulo se ilustra el método MDHDM utilizando dos casos de estudio: una librería electrónica y un sistema de gestión de artículos. En cada uno se muestran los siguientes pasos de MDHDM: modelado conceptual (modelos de proceso y estructural), proyección por actor y diseño navegacional, e implementación. El caso de la librería electrónica es presentado en la sección 9.2, y el caso del sistema de gestión de artículos es mostrado en la sección 9.3. Se han elegido estos ejemplos porque son los que se presentan habitualmente en la literatura relacionada. Por último, las conclusiones del capítulo se dan en la sección 9.5.

9.2. Librería electrónica

En este ejemplo se modela una librería virtual, ésta contiene un catálogo de libros de donde el usuario puede seleccionar los libros que desea añadir a su carro de compras. Cuando el usuario tiene los libros que quiere, puede cerrar la compra y con ello finaliza el proceso. El proceso consta sólo de las actividades efectuadas por los compradores de libros, es decir, en este caso particular únicamente participa un actor.

9.2.1. Modelo de proceso

El primer paso del método es la creación del modelo de proceso, si asumimos que existen los casos de uso presentados en el capítulo 5. La figura 9.1 muestra al modelo BPMN del proceso de compra electrónica de libros. Éste consta de tres subprocesos: selección de libros, modificación del carro de compras y el cierre de la compra. El proceso inicia con la selección de libros. Después de la selección de un libro, se puede añadir éste al carro de compras o proceder a cerrar la compra si el carro de compras no está vacío. A partir del subproceso de modificación del carro de compras se puede cerrar la compra o continuar la selección de libros. El cierre de la compra finaliza el proceso.

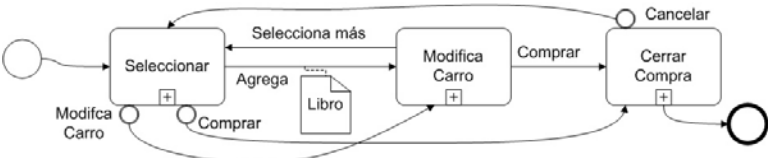


Figura 9.1: Librería electrónica

El subproceso de selección de libros (ver figura 9.2) inicia al introducir los datos de búsqueda de un libro en un formulario. Después se efectúa la búsqueda y se presenta la lista de libros encontrados. Un libro de la lista puede ser seleccionado por el usuario y agregado al carro de compras. Los eventos intermedios modificar y comprar (en el borde inferior del subproceso), representan eventos iniciados por el usuario. Se ha explicado que su semántica es similar a la de un xor diferido después de cada actividad manual. En este caso los eventos intermedios sirven para llevar el proceso a los subprocesos de modificar el carro de compras o al cierre de la compra en cualquier momento.

El subproceso de modificación del carro de compras se muestra en la figura 9.3. El usuario puede cambiar las cantidades de libros o vaciar el carro de compras.

El subproceso de cerrar la compra (ver figura 9.4) inicialmente requiere que el usuario provea un nombre de usuario y una contraseña y que el ca-

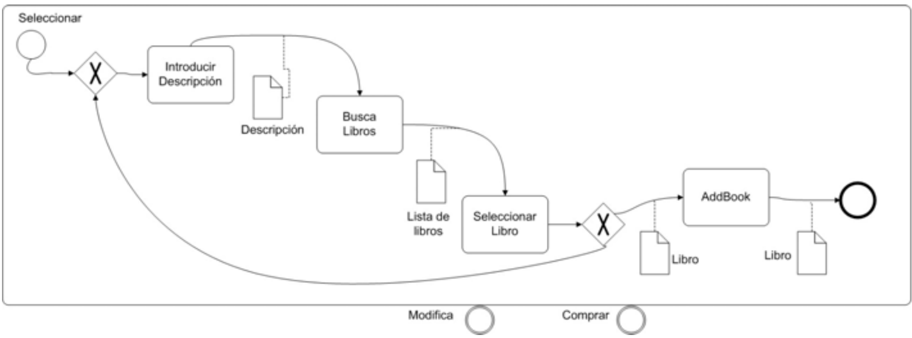


Figura 9.2: Subproceso de selección de libros

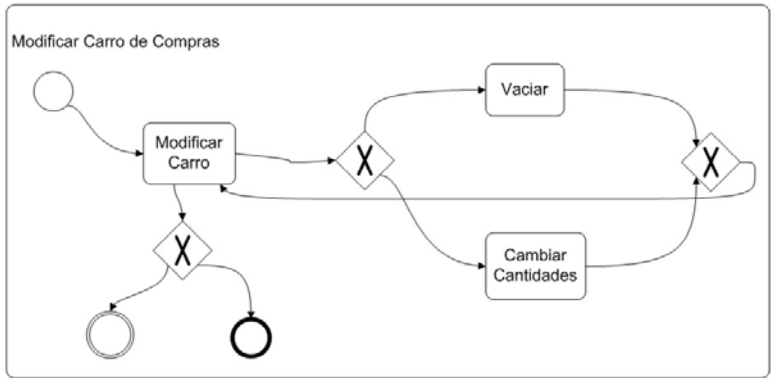


Figura 9.3: Subproceso de modificación del carro de compras

rro de compras no esté vacío. Si no es un usuario registrado, entonces debe proporcionar sus datos (por ejemplo, nombre, dirección, correo-e, etc.). A continuación, el usuario debe seleccionar la dirección de envío o indicar una nueva, en caso de no tener ninguna registrada, y proporcionar información sobre el medio de pago escogido. En el siguiente paso el usuario debe confirmar el pedido; en ese momento, se le debe mostrar el carro de compras y el precio total de la compra. Después de la confirmación, se le debe otorgar al usuario un número de pedido. El usuario puede cancelar la compra en cualquier paso del proceso, excepto después de haber confirmado el pedido.

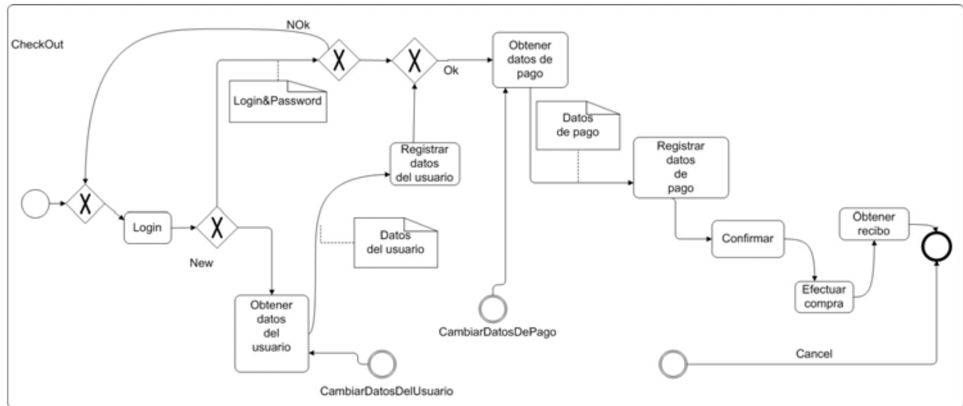


Figura 9.4: Subproceso del cierre de compra

9.2.2. Modelo estructural

El modelo estructural de la librería electrónica se muestra en la figura 9.5. Un libro puede ser escrito por n autores, y cuenta con una revisión; además, puede estar relacionado con varios items de carros de compra. Una orden está asociada a un carro de compras, un usuario, una dirección de envío y una tarjeta de crédito. Las direcciones y las tarjetas de crédito pertenecen a un sólo usuario y éste puede tener varias.

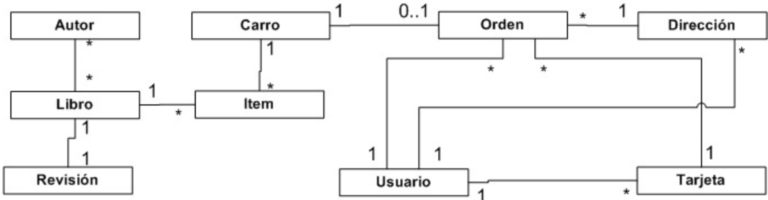


Figura 9.5: Modelo estructural de la librería electrónica

9.2.3. Modelo navegacional fuerte

A continuación se describen las correspondencias usadas para generar el modelo navegacional fuerte que se muestra en la figura 9.6 (los nodos iniciales y de activación de las vista de tarea no son mostrados).

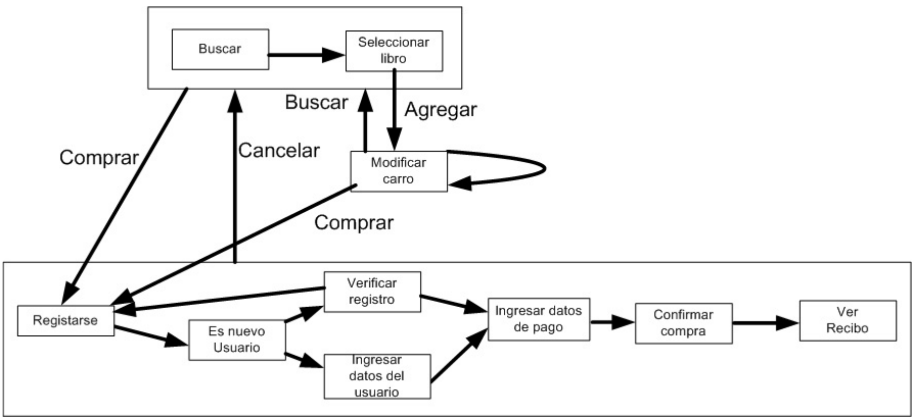


Figura 9.6: Modelo navegacional fuerte de la librería

El subproceso de selección de un libro corresponde a una vista de tarea con enlaces de proceso para modificar el carro de compras o cerrar la compra (los cuales tienen guardas condicionales). La actividad de ingresar la descripción del libro es reemplazada por una vista de tarea que tiene como nodo inicial un formulario, y la actividad de selección del libro por otra que inicia con un índice, a partir del cual, se puede agregar un libro al carro de compras. El nodo de activación del enlace de proceso es la clase navegacional libro, es decir, se puede agregar un libro desde cualquier objeto navegacional libro o desde el índice de resultados de búsqueda.

El carro de compras es una variable de proceso, y la tarea de edición y visualización del carro de compras se transformó en una vista de tarea que tiene como nodo inicial el carro de compras. Esta vista permite editar su contenido y tiene los siguientes enlaces de proceso: uno para vaciar el carro de compras, otro para modificarlo, uno más para continuar seleccionando libros, y otro para cerrar la compra.

El subproceso de cerrar la compra corresponde a un grupo de nodos que comparten el enlace de proceso de cancelación. Cada tarea manual se transforma en una vista de tarea que inicia con un formulario. La tarea de registro se transforma en un formulario, que está conectado a un nodo de redirección. Éste se deriva de un xor-gateway automático, el cual redirige la

navegación al nodo de registro si los datos fueron incorrectos, o al nodo de captura de la dirección de envío y de la tarjeta de crédito en caso contrario. La actividad de captura de la dirección de envío y de la tarjeta de crédito es una tarea compuesta que permite la obtención de un objeto dato desde un formulario, o desde la lista de selección de las direcciones y tarjetas de crédito usadas anteriormente. La tarea de confirmación permite la visualización de la variable de proceso carro de compras (En el diagrama 9.6 la vista de tarea de confirmación se traslapa con otra vista de tarea que tiene el mismo nodo inicial, lo cual no tiene ningún significado adicional). Todos los nodos del subproceso de cerrar la compra muestran el enlace de proceso de cancelación, excepto el nodo que muestra la orden generada al confirmar la compra.

A continuación se detallan todas las vistas de tarea del modelo navegacional:

Vista de tarea:	Selección (Vista de tarea inicial)
Nodo inicial:	Buscar libro

Enlace de proceso	Nodo	Vista de tarea destino
Comprar	Cualquiera	Cerrar la compra
Modificar carro	Cualquiera	Modificar carro

Vista de tarea:	Buscar libro
Vista padre:	Selección
Nodo inicial:	Formulario de búsqueda

Enlace de proceso	Nodo	Vista de tarea destino
Buscar	Formulario de búsqueda	Seleccionar libro

Vista de tarea:	Seleccionar libro
Vista padre:	Selección
Nodo inicial:	Índice de resultados

Enlace de proceso	Nodo	Vista de tarea destino
Agregar libro	Índice de resultados Clase navegacional libro	Modificar carro

Vista de tarea:	Modificar carro
Nodo inicial:	Carro de compras

Enlace de proceso	Nodo	Vista de tarea destino
Modificar	Carro de compras	Modificar carro
Vaciar	Carro de compras	Modificar carro
Comprar	[Si el carro de compras tiene productos] Cualquiera	Cerrar compra

Vista de tarea:	Cerrar la compra
Nodo inicial:	Vista de tarea Registrarse

Enlace de proceso	Nodo	Vista de tarea destino
Cancelar		Selección

Vista de tarea:	Registrarse
Vista padre:	Cerrar la compra
Nodo inicial:	Formulario de registro

Enlace de proceso	Nodo	Vista de tarea destino
Registrarse	Formulario de registro	Verificar registro

Vista de tarea:	Verificar registro
Vista padre:	Cerrar la compra
Nodo inicial:	

Enlace de proceso	Nodo	Vista de tarea destino
Nuevo usuario		Registro de usuario
Pago		Registro de pago
Fallo		Registrarse

Vista de tarea:	Registro pago
Vista padre:	Cerrar la compra
Nodo inicial:	Formulario de pago

Enlace de proceso	Nodo	Vista de tarea destino
Registrar pago	Formulario de pago	Confirmar

Vista de tarea:	Confirmar
Vista padre:	Cerrar la compra
Nodo inicial:	Carro de compras

Enlace de proceso	Nodo	Vista de tarea destino
Comprar	Carro de compras	Recibo

Vista de tarea:	Recibo
Nodo inicial:	Carro de compras

Si se reemplaza las vistas de tarea por sus nodos iniciales se obtiene el modelo navegacional de la figura 9.7.

9.2.4. Modelo navegacional débil

Por su parte, las clases y asociaciones del modelo estructural, de la figura 9.5, se reflejan en el modelo navegacional, de la figura 9.8. Las asociaciones 1-N y M-N se han reemplazado por índices, y en algunos casos se ha agrupado la información en un sólo nodo mediante composiciones. Las composiciones deben indicarse a través del marcado del modelo estructural.

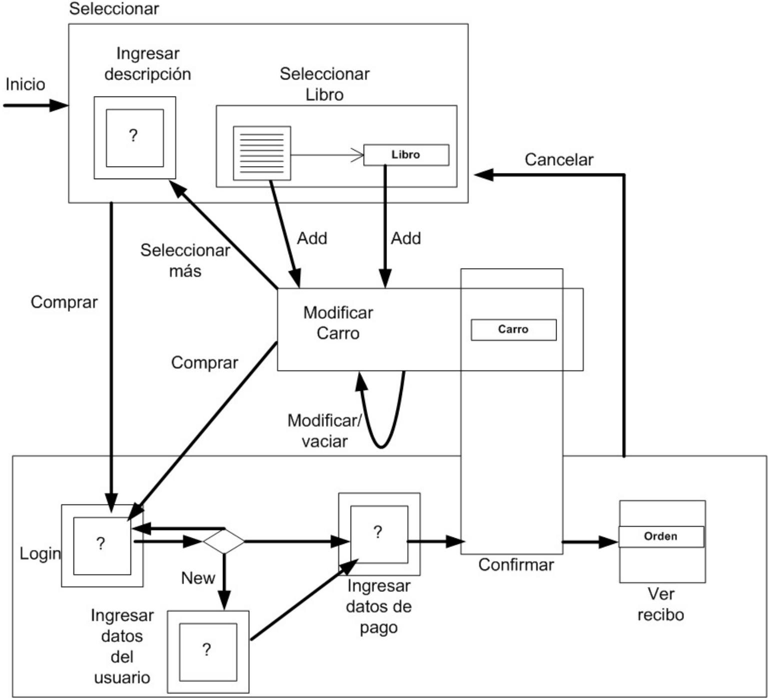


Figura 9.7: Modelo navegacional fuerte de la librería-II

9.3. Sistema de revisión de artículos

En este ejemplo se modela un sistema de revisión de artículos que incluye distintos tipos de actores. Los autores envían artículos que son sometidos a un proceso de revisión por parte de los revisores, cuando éste ha finalizado entonces pueden ver el resultado de la evaluación del artículo. En el caso de que un artículo es aceptado, entonces los autores pueden enviar la versión final.

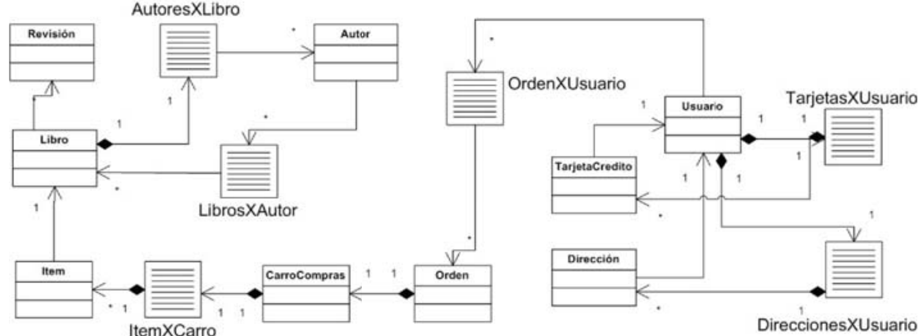


Figura 9.8: Modelo navegacional débil

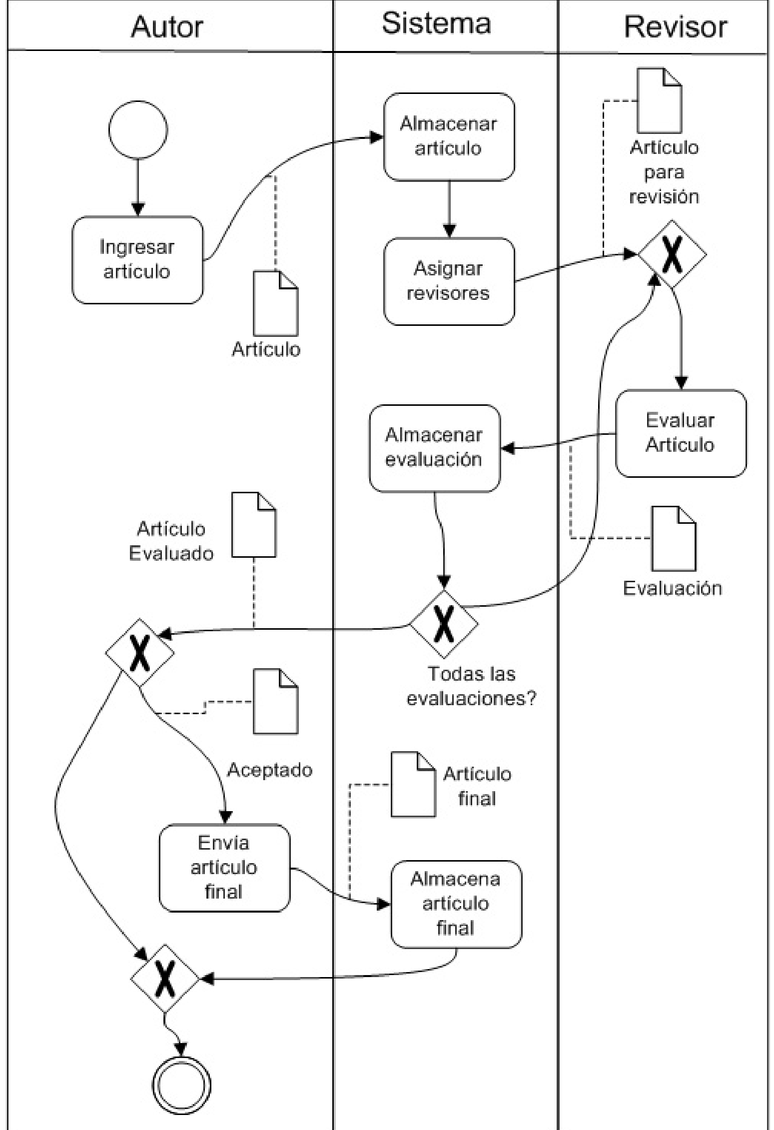


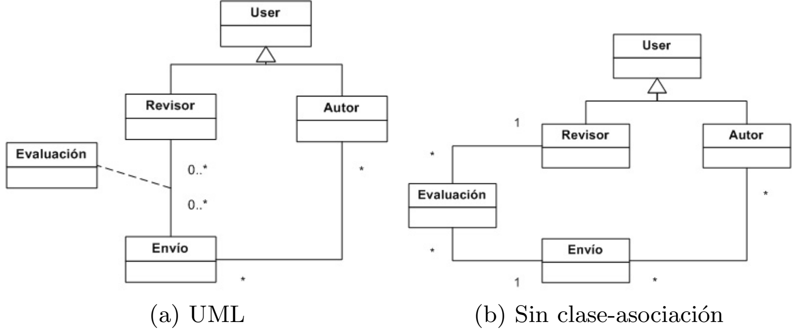
Figura 9.9: Modelo de proceso del sistema de revisión de artículos

9.3.1. Modelo de proceso

El proceso de la figura 9.9 se ejecuta después de que un usuario se ha identificado como autor. El autor registra un artículo, espera su revisión, y si es aceptado entonces puede enviar la versión definitiva. En la figura se pueden ver las actividades asignadas a cada actor dentro de su *calle*. El *ítem* que se procesa a lo largo del flujo de trabajo son los artículos. En el proceso también se pueden observar las actividades automáticas que corresponden al sistema en otra *calle*.

9.3.2. Modelo estructural

La figura 9.10 muestra el modelo estructural del sistema de revisión de artículos. El primer modelo (9.10a) contiene una clase asociación que es reemplazada por asociaciones binarias en el segundo (9.10b). Un artículo tiene n evaluaciones, y cada evaluación es efectuada por un revisor y se refiere a un artículo específico. Del modelo de proceso proceden las clases artículo y evaluación que son dos objetos dato.



(a) UML (b) Sin clase-asociación

Figura 9.10: Modelo conceptual del sistema de revisión de artículos

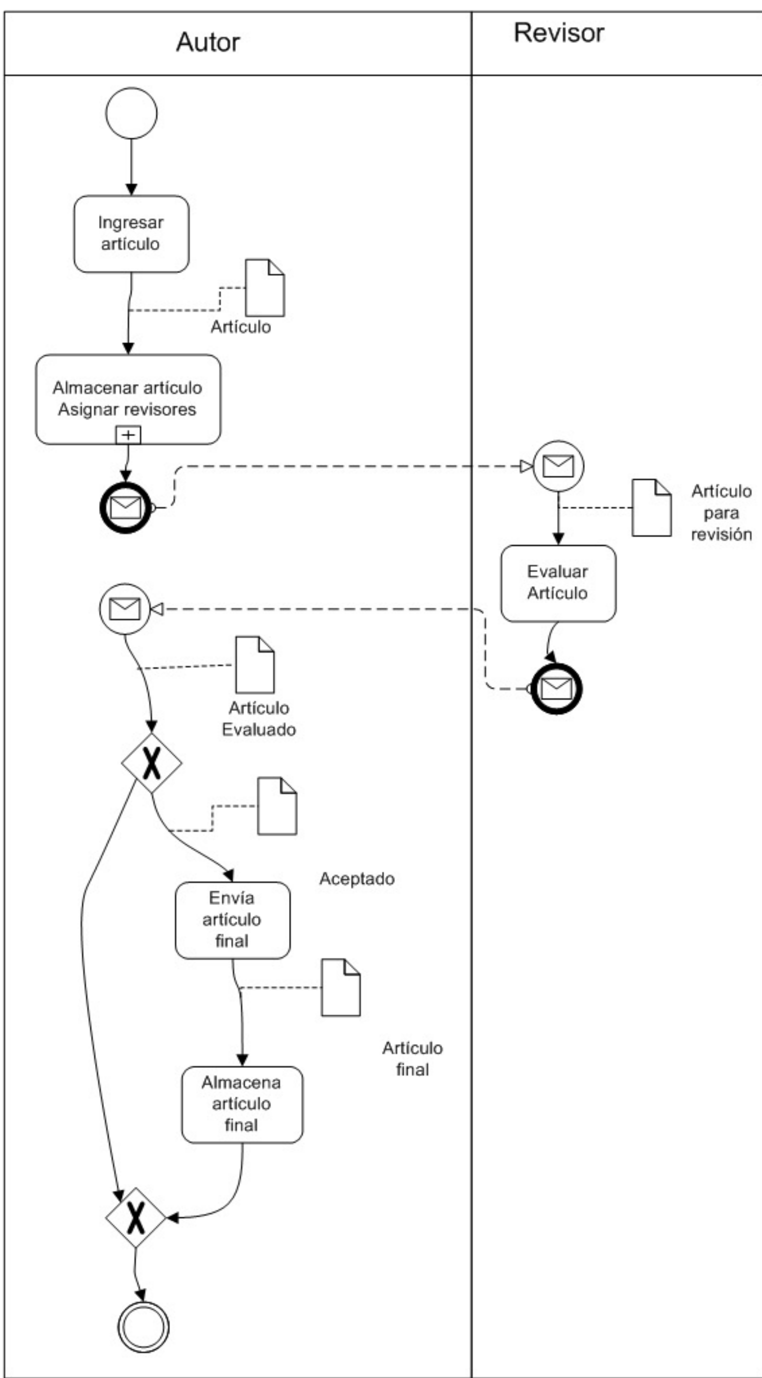


Figura 9.11: Proyecciones del sistema de revisión de artículos

9.3.3. Proyecciones

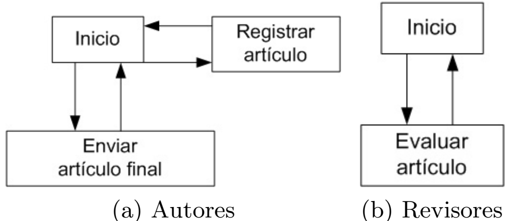
En este caso se efectúan dos proyecciones, una para los autores y otra para los revisores. La figura 9.11 muestra la vista que tiene del flujo de trabajo cada actor. Los pasos que se aplicaron para determinar la proyección para los autores se explican a continuación. Las actividades automáticas almacenar artículo y asignar revisores se agruparon en un subproceso. Después a la actividad manual evaluar artículo (efectuada por los revisores), se le añade la marca de sincronizado. La actividad evaluar artículo es seguida por la actividad automática almacenar evaluación, ambas son agrupadas en un subproceso con la marca de sincronizado. Este subproceso está rodeado por dos *gateways* que efectúan un ciclo, este grupo también es reducido a un subproceso sincronizado. El subproceso derivado de evaluar artículo queda conectado a un *gateway xor* de ramificación, que finalmente sustituye a ambas y que tiene la marca de sincronización.

La proyección muestra el conjunto de actividades automáticas y manuales que no tienen dependencia de otros actores. Las secciones del proceso que dependen de actividades manuales ejecutados por otros se han reemplazado por un *gateway xor* con la marca de sincronizado.

La vista que tienen del flujo de trabajo los revisores, se construye a través de la siguiente proyección: las primeras actividades de proceso se reemplazan por un subproceso con la etiqueta de sincronizado debido a que dependen de la actividad manual de envío, es decir, sustituye desde asignar artículo hasta asignar revisores. Este subproceso quedaría conectado al primer *gateway xor*, por lo cual se puede efectuar su unión y sustituirlos por un *gateway xor* de unión con la etiqueta de sincronizado. El camino de retorno del ciclo se puede eliminar por quedar directamente conectado a un *xor-gateway* de unión. El primer *gateway xor* de sincronización etiquetado quedaría conectado al nodo inicial por lo cual puede ser reemplazado sólo por el nodo inicial. La última sección del proceso depende de las actividades efectuadas por otros actores, por lo que se reemplaza por un subproceso etiquetado como sincronizado. Éste queda conectado al *gateway xor* de ramificación que efectúa el ciclo, por lo cual es reemplazado y eliminado por quedar conectado al nodo final.

9.3.4. Modelo navegacional fuerte

El modelo navegacional fuerte de los autores se muestra en la figura 9.12a.



(a) Autores (b) Revisores

Figura 9.12: Modelos navegacionales fuertes

El punto de entrada para la actividad capturar artículo es desde el menú inicial debido a que no tiene dependencias de otras tareas ni de objetos dato. Debido a que no hay otra actividad subsecuente, el enlace de proceso lleva a una vista de tarea, que desciende de la vista de tarea inicial, pero que tiene como nodo de inicio a la clase navegacional artículo. En cambio la actividad de enviar un artículo final tiene como puntos de entrada un índice con los artículos evaluados o desde la clase navegacional artículo si está aceptado. A continuación se presenta el diseño de las vistas de tarea.

Vista de tarea:	Inicial
Nodo inicial:	Inicio (Home)

Enlace de proceso	Nodo	Vista de tarea destino
Ingresar Artículo	Inicio	Ingresar artículo
Aceptados	Inicio	Aceptado

Vista de tarea:	Ingresar artículo
Nodo inicial:	Formulario para alta de artículos

Enlace de proceso	Nodo	Vista de tarea destino
Registrar	Formulario para alta	Inicial

Vista de tarea:	Aceptado
Nodo inicial:	Índice de artículos aceptados

Enlace de proceso	Nodo	Vista de tarea destino
Versión Final	Índice de artículos aceptados Artículo aceptado	Inicial Versión Final

Vista de tarea:	Versión final	
Nodo inicial:	Formulario para ingreso de la versión final	
Enlace de proceso	Nodo	Vista de tarea destino
Registrar Versión Final	Formulario para ingreso de la versión final	Inicial

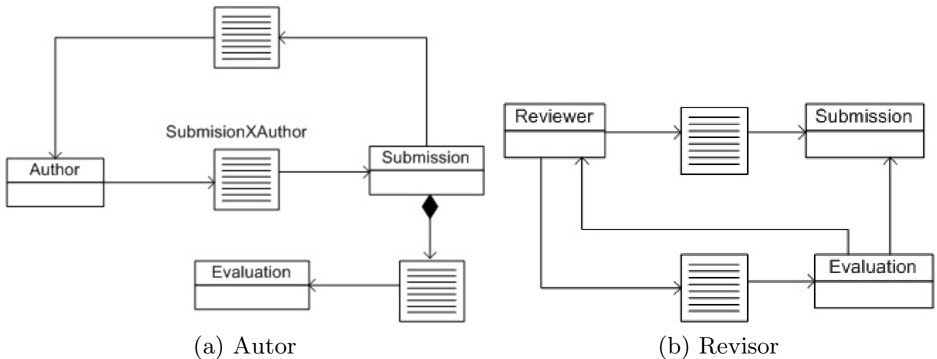
Vista de tarea:	Inicio Revisores	
Nodo inicial:	Índice artículos por revisar	
Enlace de proceso	Nodo	Vista de tarea destino
Evaluar	Formulario de evaluación	Inicio

Vista de tarea:	Evaluar	
Nodo inicial:	Formulario de evaluación	
Enlace de proceso	Nodo	Vista de tarea destino
Evaluado	Formulario de evaluación	Inicio Revisores

9.3.5. Modelo navegacional débil

El modelo estructural debe proyectarse para cada tipo de usuario del sistema, es decir, para los revisores y para los autores. A continuación se explican las marcas aplicadas al modelo estructural. En el caso de los autores tienen prohibido navegar hacia los datos de los revisores, es decir, se aplica la marca de *Omitido* a la clase revisor. Además se quiere mostrar la lista de autores, y las evaluaciones en el mismo nodo que el artículo, por lo cual se aplica la marca de composición a la asociación M-N entre artículo y autor, en el extremo relacionado con la clase artículo. A la asociación 1-N entre artículo y evaluación también se le aplica la marca de composición. El modelo navegacional resultante se muestra en la figura 9.13a. Si se considera que la revisión es una revisión a ciegas, sin conocer los datos de los autores, entonces se tiene que evitar que los revisores vean la clase autor. Para conseguirlo se aplica la marca *Omitido* a la clase autor. Tampoco se permite la navegación entre la asociación artículo y evaluación, porque no se quieren mostrar las evaluaciones de otros autores.

Al reemplazar las vistas de tarea por sus nodos de inicio del modelo navegacional débil se obtienen los siguientes modelos navegacionales:



(a) Autor (b) Revisor

Figura 9.13: Modelo navegacional débil del sistema de revisión de artículos

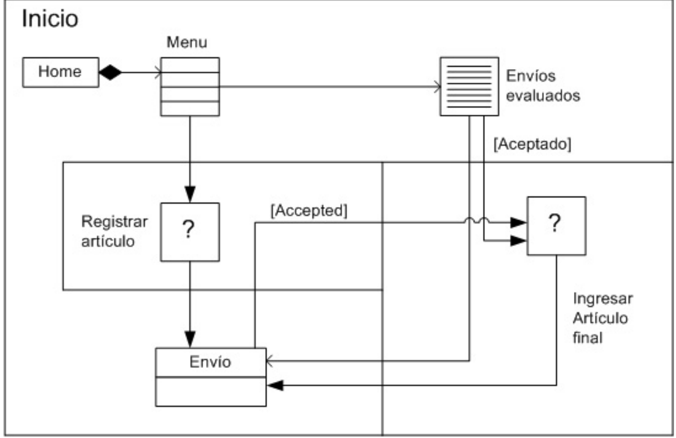


Figura 9.14: Modelo navegacional fuerte para los autores

9.4. Implementación

La implementación depende de la definición de modelos PSM. El usuario puede generar sus propios PSM. Hacia el cual tendría que transformar el modelo navegacional, e inclusive, si hiciera falta información para generar el código puede recibir, adicionalmente, a los modelos de proceso y estructural. Es decir, el método puede ser extendido para cualquier número de PSM, logrando con ello el propósito de interoperabilidad de MDE. Las técnicas de generación de código a seguir son las mencionadas en el capítulo 8. Además, se pueden emplear los PSMs presentados en el capítulo 8.

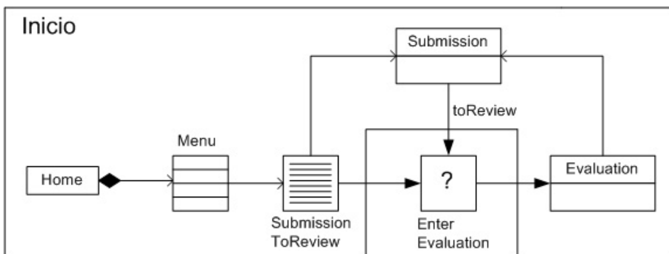


Figura 9.15: Modelo navegacional fuerte para los revisores

9.5. Conclusiones

En este capítulo se han presentado dos casos de estudio, y se han desarrollado paso a paso los diversos modelos del método (excepto el de presentación). Se han construidos los modelos y efectuado las transformaciones a partir de las especificaciones presentadas en capítulos anteriores. Los ejemplos muestran claramente que estos sistemas tienen dos niveles de navegación. Los modelos navegacionales obtenidos sirven como especificación para el desarrollo final del sistema.

Los casos muestran que el método es sencillo de aplicar. Además, muestran que un proceso BPMN puede ser efectuado por una aplicación hipermedia, sin que sea necesario contar como soporte tecnológico con un sistema de gestión de flujo de trabajo, si se cuenta con una especificación adecuada. Los dos casos son procesos simples, pero esto no anula lo dicho. Otra de las ventajas es que queda bien definido que la presencia de enlaces fuertes depende del estado del proceso y del nodo visitado. Entre las desventajas del método se puede mencionar que depende del correcto modelado del proceso y de la estructura de la información.

Parte IV

Conclusiones

Capítulo 10

Conclusiones y trabajo futuro

En la tesis se ha presentado MDHDM (Model Driven Hypermedia Development Method), un método que se basa en el modelado de procesos y en la ingeniería dirigida por modelos para el desarrollo de aplicaciones hipermedia y *web*. El objetivo principal del método es modelar navegaciones complejas que solucionan el problema de acceso a la información y la coordinación, y que son más realistas que las obtenidas a partir de modelos estructurales. En la hipótesis se planteó que si el modelo navegacional se obtiene a partir del flujo de control y de datos de un modelo de proceso, entonces se podría conseguir.

Durante el trabajo realizado se ha analizado los métodos de desarrollo de hipermedia y *web*. Se ha estudiado MDE y seleccionado un marco de trabajo MDA para desarrollar el método. Se ha especificado el ciclo de vida del método, y se han definido los modelos, metamodelos y transformaciones de cada una de sus fases. En las fases del modelado conceptual como en el navegacional se tiene como factor más relevante al modelo de proceso. Se ha aplicado el método a casos de estudio y se ha encontrado a través de sus resultados que se comprueba la hipótesis.

También se ha mostrado que MDE puede ser utilizado en el desarrollo de aplicaciones hipermedia. La especificación precisa de modelos y reglas

de transformación se abordan desde el paradigma MDE. Se ha optado por la implementación de MDE basada en MDA, porque a diferencia de otros enfoques como *Software Factories* [CKJ07], soporta la definición de modelos y transformaciones con lenguajes declarativos como OCL, ATL, y QVT. Además, MDA provee la posibilidad de definir transformaciones entre modelos que son independientes de la tecnología.

Las características particulares de MDHDM lo hace diferente a otros métodos, presentando algunas ventajas frente a ellos. En cuanto a los métodos que sólo se basan en el modelo estructural, obtienen un modelo navegacional insuficiente para modelar la funcionalidad de un sistema. Los métodos basados en el modelado de casos de uso para determinar los enlaces que reflejan la funcionalidad, modelan implícitamente un proceso, y esa no es la mejor de hacerlo. Los métodos que contienen un proceso y un modelo estructural consideran a los modelos navegacionales resultantes de ellos como excluyentes. Además, sólo algunos de los métodos siguen MDE porque sus transformaciones y modelos carecen de definiciones precisas.

En el resto del capítulo se presentan las contribuciones, trabajo futuro, y publicaciones.

10.1. Contribuciones

A continuación se describen con mayor detalle las contribuciones de la tesis.

- *Método de desarrollo de hipermedia.*

  MDHDM se compone de modelos que permiten realizar el análisis conceptual, el diseño navegacional, el diseño de la presentación, e implementación, con el objetivo de facilitar el desarrollo de hipermedia dirigido por modelos.

  El modelado conceptual tiene como pilar al modelo de proceso, y junto con él debe definirse el modelo estructural de clases.

  En el diseño navegacional la especificación del modelo navegacional es generada de forma semi-automática. Este modelo es obtenido a par-

tir del modelo de proceso y el estructural. El modelo de proceso es proyectado para obtener la vista de cada actor. De las proyecciones, a través de una transformación de modelos, se obtiene el modelo navegacional fuerte. Adicionalmente, el modelo estructural debe marcarse para indicar algunas decisiones de diseño de hipermedia. Del modelo estructural se obtiene, mediante una transformación de modelos, el modelo navegacional débil. El modelo navegacional final es la unión de los dos modelos navegacionales.

A partir del modelo navegacional se genera el modelo de presentación que se compone de vistas abstractas de usuario. Finalmente, en la implementación, a partir de los diversos modelos generados se pasa a algunos modelos PSM a partir de los cuales se genera código mediante plantillas.

- *Proyección del modelo de proceso.*
  Se determinaron las reglas de proyección para obtener la vista que un actor tiene sobre el proceso.

- *Modelo de navegación.*
  En MDHDM la navegación en un sistema hipermedia se caracteriza por ocurrir en dos espacios distintos: el de ejecución del proceso y el de datos. El modelo navegacional fuerte representa la navegación en el espacio del proceso y se compone principalmente de vistas de tarea y enlaces fuertes. El modelo de navegación débil representa el espacio de datos. La navegación en el modelo navegacional débil ocurre de forma interna a la navegación en el modelo navegacional fuerte. Es decir, si mediante la navegación se llega un nodo que proporciona los datos necesarios para completar una tarea, entonces los enlaces de proceso se activan, y se puede navegar a través de las vistas de tarea. Por tanto, el hilo conductor de la navegación es el proceso, y no las relaciones estructurales del modelo conceptual.

- *Metamodelo de navegación.*
  Se ha definido el metamodelo de navegación que permite definir a los

modelos navegacionales fuerte y débil, y a los elementos que éstos contienen: vistas de tarea, enlaces fuertes y débiles, clases navegacionales y estructuras de acceso.

- *Correspondencias entre el modelo de proceso y el navegacional.*
  Se han especificado de forma detallada las correspondencias entre los elementos del modelo de proceso y el modelo navegacional, teniendo en cuenta el flujo de control y el flujo de datos.

- *Correspondencias complejas entre el modelo de estructural y el navegacional.*
  Se han definido correspondencias entre los elementos del modelo de estructural y el modelo navegacional que tienen en cuenta a los caminos de longitud mayor que 1 a través de las asociaciones del modelo estructural.

- *Definición de marcas de los modelos.*
  Se han definido las marcas de los modelos de proceso y estructural aplicables a los diferentes elementos que éstos contienen. Es decir, que la marcas reflejan algunas decisiones de diseño que son consideradas durante la obtención del modelo de navegación.

- *Interfaces Abstractas de Usuario independientes de la tecnología.*
  Se ha definido un modelo y metamodelo para representar IAU independientes de la tecnología, y que permiten vincular a los elementos de la interfaz con los datos a presentar. Para cada elemento del modelo navegacional se ha especificado una IAU por omisión, la cuál es generada de forma automática mediante plantillas de generación de código.

- *Aplicación de MDE/MDA al campo de hipermedia*
  Se presentaron algunos espacios tecnológicos de MDA de los cuales se ha efectuado una comparativa, y el resultado de ello fue el uso de la combinación: MOF, ATL y XMI como plataforma de transformación de modelos.

Los modelos se han definido mediante MOF y restricciones OCL. Se definieron los modelos PIM y PSM necesarios para un proceso de desarrollo de hipermedia basado en MDE. De modo que los modelos del ciclo de vida cuentan con un metamodelo, y el paso de un modelo a otro se efectúa mediante transformaciones de modelos en ATL.

10.2. Trabajo futuro

MDHDM combina el modelado de hipermedia y aplicaciones *web* basadas en el proceso y la ingeniería dirigida por modelos, ambos son campos de investigación recientes, y abiertos a mejoras entre las que se pueden mencionar las siguientes.

Ningún lenguaje de modelado es suficientemente expresivo para modelar el mundo real, de ahí que se propongan lenguajes específicos de dominio. Sin embargo, éstos aún tienen carencias para expresar las variantes de la realidad. Por lo cual es necesario saber hasta dónde llega la capacidad de las técnicas de modelado propuestas.

En el área de modelado:

- Hace falta medir los modelos y comprobar su calidad. La medición de los modelos se puede efectuar utilizando transformaciones de modelos, utilizando métricas expresadas en OCL.
- Probar el método en el desarrollo de aplicaciones reales, lo cual serviría para verificar su viabilidad y confirmar las ventajas observadas en las pruebas de laboratorio.

Desde el punto de vista de hipermedia, las mejoras son relativas a la presentación de la información, y al uso de otros paradigmas de hipermedia más ricos que el *modelo de hipermedia centrado en los documentos*.

En el área hipermedia y presentación:

- Permitir al usuario visualizar la semántica de los enlaces, de modo que conozca hacia qué nodo navega y por qué navega hacia él.

- Incorporar una vista hipermedia de la estructura de la aplicación, para que los actores dispongan de una vista global de la proyección de sus tareas, y tengan mayor información contextual.

- Considerar las capacidades visuales y espaciales de las personas para percibir la información en la presentación de los datos.

- Ejecutar pruebas de usabilidad a los sistemas creados a partir de los modelos generados por el método, para comprobar si son fáciles de usar y satisfacen a los usuarios.

En cuanto a MDE las mejoras son relativas a los avances que se van descubriendo en esa área y a su posible incorporación al método. Además, referente a la implementación, se advierte que las herramientas MDE están sujetas a presiones de cambios tecnológicos en relación al desarrollo de PSMs. Inclusive, en el caso de aplicaciones *web*, éstas generalmente integran una gama muy diversa de tecnologías que deberían interoperar entre ellas.

En el área de MDE e implementación:

- Reescribir las transformaciones presentadas en el lenguaje QVT, dado que ahora se cuenta con herramientas para ello.

- Incorporar la gestión de modelos al método para contar con versiones de los modelos creados, y con trazabilidad en las transformaciones entre éstos.

- Se propone analizar mediante líneas de producto la variabilidad a la que están sujetos los modelos PSM de la etapa de implementación. De modo que, la presión ejercida por los cambios tecnológicos pueda ser absorbida, y se cuente con un conjunto adecuado de PSMs.

De las mejoras mencionadas, un primer trabajo que se está explorando es la aplicación del paradigma del hipertexto espacial para la presentación de modelo navegacionales [SA08].

10.3. Publicaciones

Las contribuciones realizadas durante el desarrollo de esta tesis se han plasmado en las siguientes publicaciones:

Journals

- C. Solís, J. H. Canós, M. Llavador, y M. C. Penadés (2007). De modelos de proceso a modelos navegacionales. *IEEE Latin America Transactions 5, July, 2007, pp. 238–244. ISSN 1548-0992.*

Conferencias Internacionales

- C. Solís, N. Ali (2008). ShyWiki-A Spatial Hypertext Wiki. In *Proceedings of the 2008 International Symposium on Wikis.* ACM. ISBN 978-1-60558-128-3.
- J.H. Canós, C. Solís, y M. C. Penadés (2007). Strong links vs. weak links: making processes prevail over structure in navigational design. In *HT '07: Proceedings of the 18th conference on Hypertext and hypermedia*, pp. 139–140. ACM. ISBN 978-1-59593-820-6.
- C. Solís, J. H. Canós, y M. C. Penadés y M. Llavador (2006). A model-driven hypermedia development method. In *Proceedings of WWW/Internet*, pp. 321–328. IADIS. ISBN 972-8924-19-4.

Conferencias Nacionales

- C. Solís, J. H. Canós, M. Llavador, y M.C. Penadés (2006). De modelos de proceso a modelos navegacionales. In *XI Jornadas de Ingeniería del Software y Bases de Datos*, pp. 44–53. ISBN 84-95999-99-4.

Talleres Nacionales

- C. Solís y J. H. Canós (2006). Un método de desarrollo de hipermedia (demo). In *V Jornadas de Trabajo DYNAMICA*. Universidad Politécnica de Valencia. ISBN 84-690-2623-2.

- C. Solís, M. C. Penadés, J. H. Canós, y M. Llavador (2005). Un método de desarrollo de hipermedia dirigido por modelos. In *III Jornadas de Trabajo Proyecto DYNAMICA*

Además de las publicaciones relacionadas con la tesis, se tienen otras efectuadas en trabajos de colaboración del grupo de investigación de Ingeniería de Software y Sistemas de Información (ISSI).

Journals

- J. H. Canós, M. Llavador, E. Ruiz, y C. Solís (2004). A service-oriented approach to bibliography management. *D-Lib Magazine 10. ISSN 1082-9873.*

Conferencias Internacionales

- M. Llavador, P. Letelier, M. C. Penadés, J. H. Canós, M. Borges, y C. Solís (2006). Precise yet flexible specification of emergency resolution procedures. In *Proceedings of the ISCRAM*, pp. 110–120. Tilburg University y New Jersey Institute. ISBN 90-9020601-9.

Conferencias Nacionales

- M. Llavador, J. H. Canós, P. Letelier, y C. Solís (2006). McGen: un entorno para la generación automática de compiladores de modelos específicos de dominio. In *XI Jornadas de Ingeniería del Software y Bases de Datos*, pp. 205–214. ISBN 84-95999-99-4.

- M. Llavador, P. Letelier, M. C. Penadés, J. H. Canós, M. Borges, y C. Solís (2005). Un enfoque orientado a procesos para la especificación de planes de emergencia. In *X Jornadas de Ingeniería del Software y Bases de Datos*, pp. 171–178. ISBN 84-9732-434-X.

- J. H. Canós, M. Llavador, E. Ruiz, y C. Solís (2004). Una visión orientada a servicios de la gestión de bibliografía. In *IX Jornadas de Ingeniería del Software y Bases de Datos*, pp. 387–398. ISBN 84-688-8983-0.

Talleres Internacionales

- N. Ali, C. Solís, I. Ramos (2008). Comparing Architecture Description Languages for Mobile Software Systems. In *SAM '08: Proceedings of the First International Workshop on Software Architectures and Mobility collocated with ICSE'08*, pp. 33–38. ACM. ISBN 978-1-60558-194-1.

Talleres Nacionales

- J.H. Canós, M. Llavador, C. Solís, P. Letelier, M. C. Penadés, y M. Borges (2006). Coordinación y acceso a información en gestión de emergencias. In *V Jornadas de Trabajo DYNAMICA*, pp. 103 – 108. Universidad Politécnica de Valencia. ISBN 84-690-2623-2.
- M. Llavador, J. H. Canós, M. Borges, P. Letelier, M. C. Penadés, y C. Solís (2005). Codificación de procesos en conectores prisma: aplicación a los sistemas de gestión de emergencias. In *III Jornadas de Trabajo Proyecto DYNAMICA*

Bibliografía

[AH02] W. Aalst and A. H. M. Hofstede. Workflow patterns: On the expressive power of (petri-net-based) workflow languages. In *Proc. of the Fourth International Workshop on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 1–20, 2002.

[AH04] W. Aalst and K. Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2004.

[BBM03] F. Budinsky, S.A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[BC89] K. Beck and W. Cunningham. A laboratory for teaching object oriented thinking. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–6, New York, NY, USA, 1989. ACM.

[BCC+03] M. Brambilla, S. Ceri, S. Comai, P. Fraternali, and I. Manolescu. Specification and design of workflow-driven hypertexts. *J. Web Eng.*, 1(2):163–182, 2003.

[BCFM00] A. Bongio, S. Ceri, P. Fraternali, and A. Maurino. Modeling data entry and operations in WebML. In *WebDB (Informal Proceedings)*, pages 87–92, 2000.

[BCFM06] M. Brambilla, S. Ceri, P. Fraternali, and I. Manolescu. Process modeling in web applications. *ACM Trans. Softw. Eng. Methodol.*, 15(4):360–409, 2006.

[BCR05] A. Boronat, J.A. Carsí, and I. Ramos. An algebraic baseline for automatic transformations in mda. *Electr. Notes Theor. Comput. Sci.*, 127(3):31–47, 2005.

[Be03] Blazewicz and et.al. *Handbook of Data Management in Information Systems*. Springer, 2003.

[Ber02] M. Bernstein. Storyspace 1. In *HYPERTEXT '02: Proceedings of the thirteenth ACM conference on Hypertext and hypermedia*, pages 172–181, New York, NY, USA, 2002. ACM.

[BFH06] P. Barna, F. Frasincar, and G. Houben. A workflow-driven design of web information systems. In *ICWE '06: Proceedings of the 6th international conference on Web Engineering*, pages 321–328, New York, NY, USA, 2006. ACM Press.

[BJT05] J. Bezivin, F. Jouault, and D. Touzet. Principles, standards and tools for model engineering. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 28–29, Washington, DC, USA, 2005. IEEE Computer Society.

[BK04] C. Bauer and G. King. *Hibernate in Action (In Action series)*. Manning Publications, 1 edition, 8 2004.

[BKO+02] D. Boswell, B. King, I. Oeschger, P. Collins, and E. Murphy. *Creating Applications with Mozilla*. OŔeilly, 2002.

[BL99] T. Berners-Lee. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. Harper San Francisco, 1999.

[Bor08] A. Boronat. *MOMENT: A Formal Framework for model management*. PhD thesis, Universidad Politécnica de Valencia, 2008.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Software Development Process*. Addison-Wesley, 1999.

[Bus45] Vannevar Bush. As we may think. *The Atlantic Monthly*, 176(1):101–108, 1945.

[CAJ04] J.H. Canos, G. Alonso, and J. Jaen. A multimedia approach to the efficient implementation and use of emergency plans. *IEEE MultiMedia*, 11(3):106–110, 2004.

[CDE+02] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J.F. Quesada. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2):187–243, 2002.

[CFB00] S. Ceri, P. Fraternali, and A. Bongio. Web modeling language (WebML): a modeling language for designing web sites. *Computer Networks (Amsterdam, Netherlands: 1999)*, 33(1–6):137–157, 2000.

[Che85] P. Chen. Database design based on entity and realtionship. In *Principles of Database Design (I)*, pages 174–210. 1985.

[CIdLS93] D. Cowan, R. Ierusalimschy, C.J. de Lucena, and T.M. Stepien. Abstract data views. *Structured Programming*, 14(1):1–14, 1993.

[CK04] J. Carroll and G. Klyne. Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation, W3C, 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[CKJ07] S. Cook, S. Kent, and G. Jones. *Domain Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.

[CL99] L. Constantine and L. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage Centered Design*. Addison Wesley, 1999.

[Cla99] J. Clark. XSL transformations (XSLT) version 1.0. W3C Recommendation http://www.w3.org/TR/1999/REC-xslt-19991116, W3C, 1999.

[CMLP03] J. H. Canós Cerdá, J. Jaén Martínez, J. C. Lorente, and J. Pérez. Building safety systems with dynamic disseminations of multimedia digital objects. *D-Lib Magazine*, 9(1), 2003.

[CMV03] P. Cáceres, E. Marcos, and B. Vela. A MDA-Based Approach for Web Information System Development. In *Workshop in Software Model Engineering (WiSME) in UML'2003.*, 2003.

[DC99] S. DeRose and J. Clark. XML path language (XPath) version 1.0. W3C Recommendation http://www.w3.org/TR/1999/REC-xpath-19991116, W3C, 1999.

[DMA05] Paloma Díaz, Susana Montero, and Ignacio Aedo. Modelling hypermedia and web applications: the ariadne development method. *Inf. Syst.*, 30(8):649–673, 2005.

[Dub03] M. Dubinko. *XForms Essentials.* OŔeilly, 2003.

[Ell79] C.A. Ellis. Information control nets: A mathematical model of office information flow. In *Proc. of the Conference on Simulation, Measurement and Modeling of Computer Systems*, pages 225–240. ACM Press, 1979.

[FJ05] M.D. Del Fabro and F. Jouault. Model transformation and weaving in the amma platform. In *Proceedings of the Generative and Transformational Techniques in Software Engineering (GTTSE'05), Workshop*, pages 71–77, Braga, Portugal, 2005. Centro de Ciências e Tecnologias de Computação, Departemento de Informatica, Universidade do Minho.

[FVR+03] J. Fons, P. Valderas, M. Ruiz, G. Rojas, and O. Pastor. OOWS: A Method to Develop Web Applications from Web-Oriented

Conceptual Models. In *Proc. of the 7th World MultiConference on Systemics, Cybernetics and Informatics*, volume 1, 2003.

[GGC03] I. Garrigós, J. Gómez, and C. Cachero. Modelling adaptive web applications. In *ICWI*, pages 813–816, 2003.

[GMP95] F. Garzotto, L. Mainetti, and Paolo Paolini. Hypermedia design, analysis, and evaluation issues. *Commun. ACM*, 38(8):74–86, 1995.

[GPS93] F. Garzotto, P. Paolini, and D. Schwabe. HDM a model-based approach to hypertext application design. *ACM Trans. Inf. Syst.*, 11(1):1–26, 1993.

[Gro03a] Object Management Group. Meta-Object Facility 1.4. Technical Report formal/02-04-03, OMG, 2003.

[Gro03b] Object Management Group. Meta-Object Facility 1.4. Technical Report formal/02-04-03, OMG, 2003.

[Gro04] Object Management Group. Unified Modeling Language v.2.0. Technical Report formal/05-07-04, OMG, 2004.

[Gro05] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Technical Report ptc/05-11-01, OMG, 2005.

[Gro06] Object Management Group. Business Process Modeling Notation 1.0. Technical Report dtc/06-01-01, OMG, 2006.

[Gro07] Object Management Group. Software Process Engineering Metamodel (SPEM) 2.0 . Technical Report ptc/07-11-01, OMG, 2007.

[HC01] J. Hunter and W. Crawford. *Java Servlet Programming.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[Hib] Hibernate. http://www.hibernate.org.

[HJR99] A. Le Hors, I. Jacobs, and D. Raggett. HTML 4.01 specification. W3C Recommendation http://www.w3.org/TR/1999/REC-html401-19991224, W3C, 1999.

[HK01] R. Hennicker and N. Koch. Systematic design of web applications with UML. In *Unified Modeling Language: Systems Analysis, Design and Development Issues*, pages 1–20, 2001.

[HMT87] F. G. Halasz, T. P. Moran, and R. H. Trigg. Notecards in a nutshell. *SIGCHI Bull.*, 17(SI):45–52, 1987.

[Hol04] S. Holzner. *Eclipse*. O'Reilly, 2004.

[ISB95] T. Isakowitz, E.A. Stohr, and P. Balasubramanian. RMM: a methodology for structured hypermedia design. *Commun. ACM*, 38(8):34–44, 1995.

[JAB+06] F. Jouault, F. Allilaire, J. Bezivin, I. Kurtev, and P. Valduriez. Atl: a qvt-like transformation language. In *OOPSLA Companion*, pages 719–720, 2006.

[Jac92] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional, 1992.

[JK06] F. Jouault and Ivan Kurtev. On the architectural alignment of atl and qvt. In *SAC*, pages 1188–1195, 2006.

[KBJV06] I. Kurtev, J. Bezivin, F. Jouault, and P. Valduriez. Model-based dsl frameworks. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 602–616, New York, NY, USA, 2006. ACM Press.

[Ken02] S. Kent. Model Driven Engineering. In *Proceedings of IFM 2002*, LNCS 2335, pages 286–298. Springer-Verlag, unknown 2002.

[Kim95] W. Kim, editor. *Modern Database Systems: The Object Model, Interoperability, and Beyond*. ACM Press and Addison-Wesley, 1995.

[KKCM04] N. Koch, A. Kraus, C. Cachero, and S. Meliá. Integration of business processes in web application models. *J. Web Eng.*, 3(1):22–49, 2004.

[Koc06] N. Koch. Transformation techniques in the model-driven development process of UWE. In *ICWE'06: Workshop proceedings of the sixth international conference on Web Engineering*, page 3. ACM Press, 2006.

[Kru98] P. Kruchten. *Rational Unified Process, The: An Introduction*. Addison Wesley Professional, 1998.

[KtHB00] B. Kiepuszewski, A.H.M. ter Hofstede, and C. Bussler. On structured workflow modelling. In *Conference on Advanced Information Systems Engineering*, pages 431–445, 2000.

[KWB03] A.G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Lan92] G.P. Landow. *Hypertext: the convergence of contemporary critical theory and technology*. Johns Hopkins University Press, Baltimore, MD, USA, 1992.

[Lan94] D.B. Lange. An object-oriented design method for hypermedia information systems. In *HICSS (3)*, pages 366–375, 1994.

[LC06] M. Llavador and J.H. Canós. XSMapper: a Service-oriented Utility for XML Schema Transformation. *ERCIM News*, January(64):58–59, 2006.

[LLP+06] M. Llavador, P. Letelier, M.C. Penadés, J.H. Canós, M. Borges, and C. Solís. Precise yet flexible specification of emergency

resolution procedures. In *Proceedings of 3rd International Conference on Information Systems for Crisis Response and Management*, pages 110–120, 2006.

[Lon04] K. Loney. *Oracle Database 10g: The Complete Reference (Osborne ORACLE Press Series)*. McGraw-Hill Osborne Media, 1 edition, 5 2004.

[Low99] David Lowe. *Hypermedia and the Web: An Engineering Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[LR99] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, 1999.

[LRSP98] P. Letelier, I. Ramos, P. Sánchez, and O. Pastor. *OASIS 3.0 Un enfoque formal para el moelado orientado a objetos*. Servicio de publicaciones de la Universidad Politécnica de Valencia, 1998.

[MCdC04] E. Marcos, P. Cáceres, and V. de Castro. Modeling the navigation with services. In *Proceedings of the IADIS International Conference WWW/Internet 2004*, pages 723–730, 2004.

[Mem99] Workflow Management Colition Members. Workflow standard process definition metamodel. Technical Report WFMC-TC-1016, WfMC, 1999.

[MM01] J. Miller and J. Mukerji. MDA Guide. Technical Report ormsc/01-07-01, Object Management Group (OMG), 2001.

[MM03] J. Miller and J. Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[Nel65] T. H. Nelson. Complex information processing: a file structure for the complex, the changing and the indeterminate. In *Proceedings of the 1965 20th national conference*, pages 84–100. ACM, 1965.

[Nie99] J.Ñielsen. *Designing Web Usability : The Practice of Simplicity*. Peachpit Press, 1999.

[NS01] J.Ñoll and W. Scacchi. Specifying process-oriented hypertext for organizational computing. *J. Networking and Computing Applications*, 24(1):39–61, 2001.

[OHK98] R. Ogawa, H. Harada, and A. Kaneko. Scenario-based hypermedia: A model and a system. In *Thirty-First Hawaii International Conference on System Sciences*, volume 2, pages 47–56, 1998.

[OJ00] M. Otter and H. Johnson. Lost in hyperspace: metrics and mental models. *Interacting with Computers*, 13(1):1–40, September 2000.

[Omo] Omondo. http://www.omondo.com.

[Ope06] Oasis Open. Web Services Business Process Execution Language Version 2.0. Technical Report wsbpel-specification-draft-01, Oasis Open, 2006.

[Pas92] O. Pastor. *Diseño y desarrollo de un entorno de producción automatica de Software Basado en el modelo orientado a objetos*. Tesis doctoral en informática, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, 1992.

[Pen02] M.C. Penadés. *Una aproximación metodológica al desarrollo de flujos de trabajo*. PhD thesis, Universidad Politécnica de Valencia, 2002.

[PMM97] F. Paterno, Cristiano Mancini, and Silvia Meniconi. Concurtasktrees: A diagrammatic notation for specifying task models. In *INTERACT '97: Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, pages 362–369, London, UK, UK, 1997. Chapman & Hall, Ltd.

[Ree79] T. M. H. Reenskaug. Models-Views-Controllers. Technical report, Xerox, 1979. http://heim.ifi.uio.no/ trygver/1979/mvc-2/1979-12-MVC.pdf.

[RSL03] G. Rossi, H. A. Schmid, and F. Lyardet. Customizing business processes in web applications. In *EC-Web*, pages 359–368, 2003.

[Rum91] J. Rumbaugh. *Object Oriented Modeling and Design*. Prentice Hall, 1991.

[SA08] C. Solís and N. Ali. ShyWiki-a spatial hypertext wiki. In *Proceedings of the 2008 International Symposium on Wikis*. ACM, 2008.

[SD05] H.A. Schmid and O. Donnerhak. OOHDMDA - an MDA approach for OOHDM. In *ICWE*, pages 569–574, 2005.

[Sei03] E. Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, 2003.

[Sel03] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[SHG06] J. Schulz-Hofen and S. Golega. Generating web applications from process models. In *ICWE '06: Workshop proceedings of the sixth international conference on Web Engineering*, page 6, New York, NY, USA, 2006. ACM Press.

[SK03] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[SMMP+06] C. M. Sperberg-McQueen, E. Maler, J. Paoli, F. Yergeau, and T. Bray. Extensible markup language (XML) 1.0 (fourth edition). W3C Recommendation http://www.w3.org/TR/2006/REC-xml-20060816, W3C, 2006.

[Som04] I. Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004.

[SR95a] D. Schwabe and G. Rossi. Building hypermedia applications as navigational views of information models. In *HICSS (3)*, pages 231–240, 1995.

[SR95b] D. Schwabe and G. Rossi. The object-oriented hypermedia design model. *Commun. ACM*, 38(8):45–46, 1995.

[SR98] D. Schwabe and G. Rossi. An object oriented approach to web-based applications design. *TAPOS*, 4(4):207–225, 1998.

[SRJ96] D. Schwabe, G. Rossi, and S.D. Junqueira. Systematic hypermedia application design with OOHDM. In *Hypertext*, pages 116–128, 1996.

[SvdAA99] A.P. Sheth, W. van der Aalst, and I.B. Arpinar. Processes driving the networked economy. *IEEE Concurrency*, 7(3):18–31, 1999.

[TC03] O. De Troyer and S. Casteleyn. Modeling complex processes for web applications using wsdm. In *Proceedings of the Third International Workshop on Web-Oriented Software Technologies*, 2003.

[TL98] O. De Troyer and C. J. Leune. WSDM: A user centered design method for web sites. *Computer Networks*, 30(1-7):85–94, 1998.

[TW06] S. Tahaghoghi and H. Williams. *Learning MySQL (Learning)*. O'Reilly Media, Inc., 11 2006.

[VFP05] P. Valderas, J. Fons, , and V. Pelechano. Using task descriptions for the specification of web application requirements. In *Workshop on Requirements Engineering*, 2005.

[vHM04] F. van Harmelen and D. L. McGuinness. OWL web ontology language overview. W3C Recommendation

http://www.w3.org/TR/2004/REC-owl-features-20040210/, W3C, 2004.

[W3C95] W3C. Hypertext terms. Technical report, W3C, 1995. http://www.w3.org/pub/WWW/Terms.html.

[WAD+06] P. Wohed, W. Aalst, M. Dumas, A.H.M. Hofstede, and N. Russell. On the suitability of BPMN for business process modelling. In *Business Process Management*, pages 161–176, 2006.

[WfM02] WfMC. XML Process Definition Language (XPDL), 2002.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[You88] E. Yourdon. *Modern Structured Analysis*. Prentice Hall, 1988.

[Zis77] M. Zisman. *Representation, Specification and Automation of Office Procedures*. PhD thesis, University of Pennsylvania, 1977.

Apéndice A

Transformaciones

Transformación 1.1: Comprueba que un elemento tiene una etiqueta

```
helper context def : hasTag(name:String): Boolean =
   self.tagged->exists(x | x.name = name);
```

Transformación 1.2: Comprueba que un elemento tiene una etiqueta y un valor determinados

```
helper context def : tagValue(name:String, value:String):
    Boolean =
   self.tagged->exists(x | x.name = name and x.value = value)
       ;
```

A.1. Transformaciones del modelo de proceso al navegacional

Transformación 1.3: Busca el sucesor de un nodo.

```
helper getSuccesor(t: Tarea): Set{Task} =
    Si t es una tarea Manual Entonces
      Inicia Selección de Caso t
             caso es de seleccion
                        devuelve Set{índice}
         caso es de captura
                        devuelve Set{formulario}
```

```
        caso es de edición/visualización
           devuelve Set{clase navegacional}
               Fin Selección de Caso
       Si no
          Inicia Selección de Caso t
                       caso es un Or-gateway de ramificación
                            no diferido
                          devuelve Set{nodo de redirección}
                       caso es un Or-gateway de ramificación
                            diferido
                          devuelve Set{}
                       caso es un And-gateway de
                           ramificación
                          Sucesores = Set{}
                          Por cada x en t.sucesores
                            Sucesores = Sucesores unión
                                BuscaSucesor(x)
                          devuelve sucesores
                       caso es un And-gateway de
                           sincronización
                          devuelve BuscaSucesor(t.sucesores
                              ->primero())
                       caso es un Or-gateway de
                           sincronización
                          devuelve BuscaSucesor(t.sucesores
                              ->primero())
           caso subproceso
              devuelve BuscaSucesor(t.initial)
           caso evento final o de sincronización
              devuelve Set{}
          Fin Selección de Caso
   Fin si
```

Transformación 1.4: Añade un enlace de proceso a un nodo

```
rule ProcessLink (
       source: ContainerMetaModel!Task,
          target: Set{ContainerMetaModel!Task} ) {
       to
        nc : ProcessLink (
                   name   <-  nn.name + '_to_' +  tt.name,
```

```
              model  <-  bb.owner.top()
         ),
        tv1 : Tagged (
                  name  <- 'trigger',
                  value <- bb.name,
            owner <- nc
        ),
        tv2 : Tagged (
                name         <- 'enable',
                value        <- enable.name,
          modelElement <-      nc
        ),
        nc1 : NavigableEnd (
                     navigable  <- true,
                     min        <- '1',
                     max        <- '1',
                     link       <- nc,
                     target     <- resolveTemp(source,'
                        taskview')
         ),
        nc2 : distinct NavigableEnd foreach(x in target) (
                     navigable <- true,
                     min       <- '1',
                     max       <- '1',
                     link       <- nc,
                     target    <- resolveTemp(x,'taskview
                        ')
         )
}
```

Transformación 1.5: De modelo de proceso a navegacional

```
rule Workflow2Model{
  from A:WorkFlow
  to
   nModel : NavigationalModel (
      name  <- A.name
   )
   using{
      begin: Task = getSuccesor(A.implementedBy.initial);
  }
```

```
  to
    Model : NavigationalModel (
      name  <- A.name
    ),
        Tagged (
          name  <- 'Start',
      modelElement <- begin
         )
}
```

Transformación 1.6: De modelo de proceso a navegacional

```
rule Workflow2Model{
  from A:WorkFlow
  (
    not A.hasTag('Anonymous')
  )
  using{
      begin: Task = getSuccesor(A.implementedBy.initial);
  }
  to
    Model : NavigationalModel (
      name  <- A.name
    ),
    home : Query(
          name <- 'Home'
        ),
        asoc : NavigableAssociation(
        ),
    ne1 : NavigableEnd(
           target    <- Query,
           navigable <- true,
           link      <- asoc
        ),
        ne2 : NavigableEnd(
           target    <- resolveTemp(begin,'navnode'),
           navigable <- false,
           link      <- asoc
        ),
        Tagged (
            name  <- 'Start',
```

```
        modelElement <- begin
         )
  }
```

Transformación 1.7: De un nodo de proceso con la etiqueta de coordinado

```
 rule Model2Model{
   from A:Task (
     A.hasTag('Coordinado')
   )
   to
    navnode : Index (
     name  <- A.name,
         class <- resolveTemp(getSuccesor(A.getWorkFlow().
             initial), 'taskview'),
         condition <- A.getOutputDataObject().condition,
    ),
    n4  : NavigableAssociation(
    ),
    n2  : NavigableEnd(
       target <- navnode,
           link   <- n4
    ),
    n3  : NavigableEnd(
       target <- resolveTemp(A.getWorkFlow().initial, 'Home'),
           link   <- n4
    )
    do{
     ProcessLink (
           resolveTemp(getSuccesor(A.getWorkFlow().initial), '
               taskview'),
           resolveTemp(A.getOutputDataObject(),'taskview'),
           Set{ navnode }
         );
    }
  }
```

Transformación 1.8: De las tareas de captura de datos

```
 rule Model2Model{
   from A:Activity(
```

```
    A.manual
        and
    A.hasTag('DataInput')
        and
        A.hasOutputDataObject()
 )
 to
 taskview : TaskView(
        name    <- A.name,
        initial <- navnode
  ),
  navnode   : NavigationalClass(
       name     <- A.name,
       className <- A.getOutputDataObject()
  )
  do{
   ProcessLink(taskview, getSuccesor(taskview), Set{navnode
       });
  }
}
```

Transformación 1.9: De las tareas de selección

```
rule Model2Model{
from A:Activity
(
  A.manual
  and
  A.hasTag('Selection')
  and
  A.hasOutputDataObject()
)
to
  taskview : TaskView(
          name    <- A.name,
          initial <- navnode
       ),
   navnode : Index (
     name      <- A.name,
     class     <- A.getOutputDataObject(),
         multiple  <- A.tagValue('Selection','Multiple')
```

```
    )
  do{
   ProcessLink (taskview, getSuccesor(taskview),Set{navnode,
       A.getOutputDataObject()});
  }
}
```

Transformación 1.10: De las tareas de edición/visualización

```
rule Model2Model{
from A:Activity
(
  A.manual
  and
  (
    A.hasTag('Edition')
    or
        A.hasTag('Visualization')
  )
  and
  A.hasOutputDataObject()
)
to
   taskview : TaskView(
          name    <- A.name,
          initial <- navnode
       ),
   navnode: conditionalSection (
     condition     <- A.tagValue('Condition'),
     class         <- A.getOutputDataObject()
   )
       n2 : taggedValue(
         name  <- 'Edition',
         value <- A.name,
         owner <- A.getOutputDataObject()
       )
       do {
    ProcessLink(taskview, getSuccesor(taskview), Set{navnode
        });
       }
}
```

Transformación 1.11: Obtiene el elemento previo de la colección de predecesores

```
helper context Task def : prev() : Task =
   self.previous->first();
```

Transformación 1.12: Verifica si el gateway-Or es diferido

```
helper context Task def : afterDeferredOr() : AssociationEnd
    =
let
   prev: Task = self.previous->first()
 in
   prev.manual and prev.hasTag('deferred-or');
```

Transformación 1.13: De un gateway deferred-Or a un menú con enlaces de proceso

```
 rule Model2Model{
 from A:OrSplit
 (
   A.afterDeferredOr()
 )
 using{
   iterador: Sequence = A.next;
 }
 to
   menu {
           name <- A.name,
    },
    n4: NavigableAssociation(
        ),
    n2:  NavigableEnd (
           target    <- A.prev(),
           navigable <- true,
           composite <- true,
           link      <- n4
        ),
        n3:  NavigableEnd (
           target    <- taskview,
           navigable <- false,
```

```
            composite <- false,
            link      <- n4
         ),
     n5 : distinct ProcessLink foreach ( x in iterador )(
                 enable       <- Set{menu}
         ),
         n6 : distinct NavigableEnd foreach( x in iterador )(
            target    <- resolveTemp(getSuccesor(x),'taskview
               '),
            navigable <- false,
            composite <- false,
            link      <- n5
         ),
     n7 : distinct NavigableEnd foreach( x in iterador )(
            target    <- resolverTemp(A.prev(),'taskview'),
            navigable <- true,
            composite <- false,
            link      <- n5
         )
```

Transformación 1.14: Un gateway-Or no diferido es transformado en una hiper-arista dirigida.

```
  rule Model2Model{
  from A:OrSplit
  (
    not A.afterDeferredOr()
  )
  to
     n1 : TaskView (
        name <- A.name
     ),
         n2 : taggedValue(
           name  <- 'Xor',
           owner <- n1
         )
     n5 : distinct ProcessLink foreach ( x in iterador )(
                 param        <- x.name,
                 trigger      <- A.name,
                 enable       <- Sequence{A.prev().name}
         ),
```

```
        n6 : distinct NavigableEnd foreach( x in iterador )(
           target    <- resolveTemp(getSuccesor(x),'taskview
              '),
           navigable <- false,
           composite <- false,
           link      <- n5
        ),
    n7 : distinct NavigableEnd foreach( x in iterador )(
           target    <- n1,
           navigable <- true,
           composite <- false,
           link      <- n5
        )
```

Transformación 1.15: De un gateway And

```
 rule Model2Model{
 from A:AndSplit
 (
 A.next
 ->forAll( x | x.hasManualSucesor())
 )
 using{
 sucs: Set(Task) = A.next->collect( x | x.getSuccesor() )
 }
 to
        plink :  processLink (
         ),
        n1 : distinct NavigableEnd foreach( x in  sucs )(
          target    <- resolveTemp(getSuccesor(x),'taskview')
              ,
          navigable <- false,
          composite <- false
        ),
        n2 : distinct NavigableEnd foreach( x in  sucs )(
         target    <- plink,
         navigable <- true,
         composite <- false
        )
}
```

A.2. Transformaciones del modelo conceptual al navegacional

Transformación 1.16: De una clase

```
rule Class2NClass{
from A: Class
to
 n1 : NavigationalClass (
   name <- A.name
 )
}
```

Transformación 1.17: De un atributo

```
rule Attribute2NAttribute{
from A: Attribute
to
 n1 : NAttribute (
   name  <- A.name,
       class <- A.class,
       type  <- A.type
 )
}
```

Para facilitar la obtención de los extremos de una asociación se han definido dos consultas OCL (transformación 1.18).

Transformación 1.18: Obtención de los extremos de una asociación

```
helper context Association def : firstAE: AssociationEnd =
self.associationEnd->first();

helper context Association def : lastAE : AssociationEnd =
self.associationEnd->last();
```

Transformación 1.19: De una asociación con cardinalidad 1-1

```
rule Association2NAssociation{
from A: Association
(
```

```
        firstAE.max = 1 and lastAE.max = 1
    )
    to
     n3 : NAssociation (
     ),
     n1 : NavigableEnd (
       min    <- firstAE.min,
           max    <- 1,
           target <- firstAE.target,
           link   <- n3
     ),
     n2 : NavigableEnd (
       min <- lastAE.min,
           max <- 1,
           target <-lastAE.target,
           link <- n3
     )
    }
```

Transformación 1.20: De una asociación con cardinalidad 1-N

```
helper context Association def : oneToN():
 Boolean =
(firstAE.max = 1 and lastAE.max = -1) or
(lastAE.max = 1 and firstAE.max = -1);

helper context Association def : oneside: AssociationEnd =
self.associationEnd->select(x|x.max=-1)->first();

helper context Association def : nside: AssociationEnd =
self.associationEnd->select(x|x.max=1)->first();

rule Association2NAssociation{
   from A: Association
   (
      A.oneToN
   )
   to
    //Estructura de acceso
        //y dos asociaciones navegables
    AccessStructure : Index (
```

```
),
    n3 : NAssociation (
),
n1 : NavigableEnd (
  min <- 1,
      max <- 1,
      navigable <-false,
      target <-oneside.target,
      link   <- n3
),
n2 : NavigableEnd (
  min    <- 1,
      max    <- 1,
      target <- AccessStructure
      navigable <- true,
      link   <- n3
),
    n6 : NAssociation (
),
    n4 : NavigableEnd (
  min <- 1,
      max <- 1,
      navigable <-false,
      target <- AccessStructure,
      link   <- n6
),
n5 : NavigableEnd (
  min    <- 1,
      max    <- -1,  //N
      navigable <- true,
      target <- nside.target,
      link   <- n6
),
//Más el enlace 1-1
    n9 : NAssociation (
)
    n7 : NavigableEnd (
  min <- 0,
      max <- 1,
```

```
          navigable <-false,
          target <-oneside.target,
          link   <- n9
    ),
    n8 : NavigableEnd (
      min     <- 0,
          max    <- 1,
          target <- nside.target
          navigable <- true,
          link   <- n9
    )
   }
```

Transformación 1.21: Transformaciones de atributos, clases y asociaciones no omitidos.

```
rule Attribute2NAttribute{
from A: Attribute(
    A.notHasTag('Omitted')
 )

...
--Igual con las clases
...

rule Association2NAssociation{
 from A: Association
 (
   not A.associationEnd
          ->exists( x |
                          x.target.hasTag('Omitted'))
 )
 ...
```

Transformación 1.22: Verifica que una clase tiene una sola fusión.

```
helper context Association def : hasFusion(): Boolean =
  self.associationEnd->one( x | x.hasTag('fusion'))

helper context Class def : hasFusion():
 Boolean =
```

```
    let asf: Set(Association) =
          self.associationEnd
          ->collect(Association)
          ->select(x|x.hasFusion())
    in
           asf->one(x|x.hasFusion())
           and asf->first().firstAE = self
```

Transformación 1.23: Verifica que una clase participa sólo en una asociación navegable y que sea el segundo extremo de la asociación.

```
  helper context Class def : isFusionable():
   Boolean =
    let asf: Set(NAssociation) =
          self.associationEnd
          ->collect(Association)
          ->select(x|x.hasFusion())
    in
           asf->one(x|x.hasFusion)
           and asf->first().lastAE = self
```

Transformación 1.24: Obtiene la clase donde deben agregarse los atributos de otra.

```
  helper context Class def : getFusionClass():
   Boolean =
    let asf: Set(Association) =
          self.associationEnd
          ->collect(Association)
          ->select(x|x.hasFusion())
    in
           asf->first()->firstAE
           and asf->lastAE = self
```

La transformación de un atributo considerando la fusión de clases se muestra en la transformación 1.25.

Transformación 1.25: De un atributo de una clase con fusión.

```
   rule Attribute2NAttribute{
    from A: Attribute(
        A.class.isFusionable()
```

```
  )
  to
   n1 : NAttribute (
     name  <- A.name,
         class <- A.class.getFusionClass(),
         type  <- A.type
   )
}
```

A la regla de transformación de clases sólo se necesita agregarle la verificación de que la clase no es fusionable, debido a que los atributos transformados se añaden a la clase correcta.

Transformación 1.26: De una clase sin fusión

```
rule Class2NClass{
  from A: Class
  (
    not A.isFusionable()
  )
  to
   n1 : NClass (
     name <- A.name
   )
  }
```

Transformación 1.27: Verifica que el destino de un extremo de asociación es fusionable

```
helper context AssociationEnd def : targetFusionable() :
    Boolean =
if target.oclIsTypeOf(Class) then
  if target.oclAsType(Class).isFusionable() then
     target.oclAsType(Class).getFusionClass()
  else
     target
  endif
else
  target
endif
```

Transformación 1.28: De una asociación con cardinalidad 1-1 considerando la fusión.

```
rule Association2NAssociation{
from A: Association
(
    firstAE.max = 1 and lastAE.max = 1
)
to
 n3 : NAssociation (
 )
 n1 : NavigableEnd (
   min    <- firstAE.min,
       max    <- 1,
       target <- thisModule.targetFusionable(firstAE),
       link   <- n3
 ),
 n2 : NavigableEnd (
   min <- lastAE.min,
       max <- 1,
       target <-thisModule.targetFusionable(lastAE),
       link   <- n3
 )
}
```

Transformación 1.29: Transformación de una asociación 1-N considerando la marca VisitaGuiada.

```
rule Association2NAssociation{
   from A: Association
   (
      A.oneToN
   )
   using {
           VisitaGuiada: Set(Boolean) =
                if A.hasTag('Tour') then
                           Set{true}
                        else
                           Set{}
                        endif
           Indice: Set(Boolean) =
```

```
                if not A.hasTag('Tour') then
                        Set{true}
                    else
                        Set{}
                    endif
        }
    to
     AccessStructure : distinct Tour foreach(x in VisitaGuiada
         )(
            ...
     ),
         AccessStructure : distinct Index foreach(x in Indice
            )(
            ...
     ),
     ...
```

La marca *CircularTour* afecta a las reglas de transformación 1-N y M-N del mismo modo que la marca *Tour*. La marca *CircularTour* añade la etiqueta de *Circular* a la visita guiada generada.

```
 Circular : distinct Tag foreach(x in CircularTour )(
            name  <- 'Circular',
            owner <- AccessStrcuture
     )
```

Transformación 1.31: Transformación de una asociación 1-1 considerando la marca NoNavegable.

```
    rule Association2NAssociation{
    from A: Association
    (
       A.notHasTag('Omitted')
            and
           A.notHasTag('Fusion')
        and
       A.firstAE.max = 1
            and
           A.lastAE.max = 1
    )
    to
```

```
 n3 : NAssociation (
 ),
 n1 : NavigableEnd (
   min       <- firstAE.min,
       max       <- 1,
       target    <- thisModule.targetFusionable(firstAE.
           target),
       navigable <- not A.firstAE.hasTag('notNavigable'),
       link      <- n3
 ),
 n2 : NavigableEnd (
   min       <- lastAE.min,
       max       <- 1,
       target    <-thisModule.targetFusionable(lastAE.
           target),
       navigable <-not A.lastAE.hasTag('notNavigable'),
       link      <- n3
 )
}
```

Transformación 1.32: Transformación de una asociación 1-N considerando la marca NoNavegable y Fusión.

```
rule Association2NAssociation{
   from A: Association
   (
      A.oneToN
   )
   using{
     osnavigable: Set(Boolean) =
               if A.oneside.hasTag('notNavigable') then
                       Set{}
           else
               Set{true}
           endif;
     nsnavigable: Set(Boolean) =
               if A.nside.hasTag('notNavigable') then
                       Set{}
           else
               Set{true}
           endif;
```

```
}
to
 //Estructura de acceso
     //y dos asociaciones navegables
 AccessStructure : Index (
 ),
     n3 : NAssociation (
 ),
 n1 : distinct NavigableEnd foreach in (x in nsnavigable )
     (
   min <- 1,
       max <- 1,
       navigable <-false,
       target    <-thisModule.targetFusionable(oneside.
           target),
       link      <- n3
 ),
 n2 : distinct NavigableEnd foreach in (x in nsnavigable )
     (
   min    <- 1,
       max    <- 1,
       target <- AccessStructure
       navigable <- true,
       link      <- n3
 ),
     n6 : distinct NAssociation foreach in (x in
         nsnavigable )(
 ),
     n4 : distinct NavigableEnd foreach in (x in
         nsnavigable )(
   min <- 1,
       max <- 1,
       navigable <-false,
       target <- AccessStructure,
       link   <- n6
 ),
 n5 : distinct NavigableEnd foreach in (x in nsnavigable )
     (
   min    <- 1,
       max    <- -1,  //N
```

```
            navigable <- true,
            target <- thisModule.targetFusionable(nside.target)
               ,
            link   <- n6
      ),
      //Más el enlace 1-1
          n9 : distinct NAssociation foreach in (x in
              osnavigable )(
      ),
          n7 : distinct NavigableEnd foreach in (x in
              osnavigable )(
        min <- 0,
            max <- 1,
            navigable <-false,
            target <-thisModule.targetFusionable(oneside.target
               ),
            link   <- n9
      ),
      n8 : distinct NavigableEnd foreach in (x in osnavigable )
         (
        min    <- 0,
            max    <- 1,
            target <- thisModule.targetFusionable(nside.target)
            navigable <- true,
            link   <- n9
      )
     }
```

Transformación 1.33: Transformación del modelo estructural y creación el nodo inicial.

```
rule Model2Model{
 from A: ConceptualModel
 (
  not A.hasTag('Home')
 )
 to
   n1 : NavigationalModel (
     name  <- A.name
   )
}
```

```
rule Model2Model{
 from A: ConceptualModel
 (
  A.hasTag('Home')
 )
 using{
        Clases: Set(Class) =
                    A.elements->
            select( x | x.oclIsTypeOf(Class) and
                    not x.hasTag('NotInHome'))
 }
 to
   n1 : NavigationalModel (
     name  <- A.name,
   ),
       n2 : Menu(
     name  <- A.name,
   ),
       n3 : distinct Index foreach(x in Clases) (
     name  <- A.name
   )
       ...
       Se crea una asociación 1-1 entre cada índice y el
           menú
       ...
       Se crea una asociación 1-1 con composición entre
       el índice y el nodo Home
}
```

Apéndice B

Verificaciones

B.1. Verificación del modelo de proceso

```
module ContainerMetaModel2Problem;
create OUT : Problem from IN :ContainerMetaModel;

--Gets the previous nodes of a Task eliminating events
helper context ContainerMetaModel!Task def : priorTasks :
    Sequence(ContainerMetaModel!Task) =
         self.prior->select(x|
     not x.oclIsKindOf(ContainerMetaModel!Event)).asSet().
         asSequence();

--Gets the next nodes of a Task eliminating events
helper context ContainerMetaModel!Task def : nextTasks :
    Sequence(ContainerMetaModel!Task) =
         self.next->select(x|
     not x.oclIsKindOf(ContainerMetaModel!Event)).asSet().
         asSequence();

--Gets the next nodes of a Task eliminating events
helper context ContainerMetaModel!Process def :isWorkflow :
    Boolean =
not  self.implements.oclIsUndefined();
```

```
rule ConnectChecking {
        from
                aa :ContainerMetaModel!Process
        using {
            disconnected : Set(Boolean) =
                            if  (  (aa.priorTasks->size() = 0
                                    and
                                   aa.invinitial.oclIsUndefined
                                       ()) or
                                  (aa.nextTasks->size() = 0 and
                                     aa.invend.oclIsUndefined())
                                     )
                                              and   aa.
                                                  implements.
                                                  oclIsUndefined
                                                  () then
                                                       Set{true
                                                           }
                                                           else
                                                           Set{}
                                                  endif;
            withOutEnd : Set(Boolean) = if aa.end.
                oclIsUndefined()
                        then     Set{true} else
                        Set{} endif ;
            withOutBegin : Set(Boolean) = if aa.initial.
                oclIsUndefined()
                                    then Set{true}
                                    else Set{} endif ;
        }
        to
         --A subprocess must be connected unless it is the
             implementer of a workflow
          aao : distinct Problem!Problem foreach ( x in
              disconnected )
          (
                severity     <- #Error,
                location     <- aa.name + ' Type: ' + aa.
                    oclType().toString(),
                description  <- 'The node is disconnected'
```

```
    ),
        --A subprocess must have an end node
        bbo : distinct Problem!Problem foreach ( x in
            withOutEnd )
        (
            severity     <- #Warning,
            location     <- aa.name + ' Type: ' + aa.
                oclType().toString(),
            description  <- 'The process must have one
                end node'
        ),
        --A subprocess must have a start node
        cco : distinct Problem!Problem foreach ( x in
            withOutBegin )
        (
            severity     <- #Warning,
            location     <- aa.name + ' Type: ' + aa.
                oclType().toString(),
            description  <- 'The process must have one
                start node'
        )
}


--The start node can not have predecesors
--It can only if it is an Or-Join
rule StartChecking {
        from
                bb :ContainerMetaModel!Task
        (
                not bb.oclIsKindOf(ContainerMetaModel!Process
                    )
        )
        using {
                startCheck      : Set(Boolean) =
                --The initial node of a proess can not have
                    predecesur unless it is a join node.
                        if
                      (not bb.invinitial.oclIsUndefined())
                        and bb.prior->size() >= 1
```

```
            and ( not bb.oclIsTypeOf(
                ContainerMetaModel!PartialJoin)
                ) then
            Set{true}
      else
         Set{}
          endif;
--The end node  of a process can not have
    sucesors unless it is a split node
endCheck        : Set(Boolean) =
          if  (not bb.invend.oclIsUndefined())
               and bb.next->size() >= 1 then
            Set{true}
      else
         Set{}
          endif;
reachableFromStart       : Set(Boolean) =
--If it is the star node then checks that is
    can reach all the nodes in the subprocess
   if not bb.invinitial.oclIsUndefined() then
       if not bb.reacheableFromStart() then
              Set{true}
       else
          Set{}
       endif
   else
            Set{}
   endif;
reachEndNode  : Set(Boolean) =
--If a node it is not the end node then it
    must reach the end node of its subprocess
   if (not bb.oclIsTypeOf(ContainerMetaModel!
       Event)) and  bb.invend.oclIsUndefined()
        then
          if not bb.ReachTheEndNode() then
                Set{true}
          else
             Set{}
          endif
   else
```

```
                  Set{}
            endif;
}
to
  bbo : distinct Problem!Problem foreach ( x in
      startCheck )
  (
        severity     <- #Error,
        location     <- bb.name + ' Type: ' + bb.
            oclType().toString(),
        description  <- 'The start node can not have
            predecesors'
  ),
  eeo : distinct Problem!Problem foreach ( x in
      endCheck )
  (
        severity    <- #Error,
        location    <- bb.name + ' Type: ' + bb.
            oclType().toString(),
        description <- 'The end node can not have
            sucesors'
  ),
  ffo : distinct Problem!Problem foreach ( x in
      reachableFromStart )
  (
        severity    <- #Error,
        location    <- bb.name + ' Type: ' + bb.
            oclType().toString(),
        description <- 'It is not reachable from the
            start node'
  ),
  ggo : distinct Problem!Problem foreach ( x in
      reachEndNode )
  (
        severity    <- #Error,
        location    <- bb.name + ' Type: ' + bb.
            oclType().toString(),
        description <- 'The node can not reach the
            end node'
  )
```

```
}

--Gets the sucesors of a Node
--Initial call: Set{Node}, Empty Set: Set{}
helper def: sucesorA(origen: Set(ContainerMetaModel!Task),
    delta: Set(ContainerMetaModel!Task)) : Set(
    ContainerMetaModel!Task) =
         let
            sucesores : Set(ContainerMetaModel!Task) =
           origen->collect(n|n.next)->flatten()->select( s |
               origen->excludes(s) ).asSet()
         in
       if sucesores = Set{} then
              origen.asSet()
           else
              origen.union( thisModule.sucesorA( origen.union(
                  delta), sucesores) ).asSet()
           endif;

--Checks that all the nodes can be reached from the start
    node

helper context ContainerMetaModel!Task def:
    reacheableFromStart() : Set(ContainerMetaModel!Task) =
let
   sucesores : Set(ContainerMetaModel!Task) = thisModule.
       sucesorA( Set{self}, Set{})
in
   self.invinitial.owns->select( x |
         not x.oclIsTypeOf(ContainerMetaModel!Event)  ). asSet
             ()
         ->forAll(x |  sucesores.includes(x)  );

--Checks that all the nodes can reach the end node
helper context ContainerMetaModel!Task def: ReachTheEndNode()
     : Boolean =
let
   sucesores : Set(ContainerMetaModel!Task) = thisModule.
       sucesorA(self.next, Set{})
in
```

```
    if self.owner.oclIsUndefined() then
           false
        else
                if self.owner.end.oclIsUndefined() then
                    false
                else
                        sucesores->includes(self.owner.end)
                endif
        endif;

--A  split can not be connected directly and ONLY to a join
    node
rule EndNodeReachable2 {
        from
                hh :ContainerMetaModel!Split
        (
                hh.next->forAll(e|e.oclIsKindOf(
                    ContainerMetaModel!Join)) and
                hh.next->asSet()->size() = 1
        )
        to
          hho : Problem!Problem
          (
                severity    <- #Error,
                location    <- hh.name,
                description <- 'A Split node can not be
                    directly connected to a Join node'
          )
}
```

B.2. Verificación del modelo conceptual

```
module ConceptualMetaModel2Problem;
create OUT : Problem from IN : ConceptualMetaModel;

helper def : emptyString(s: String) : Boolean =
 if s.oclIsUndefined() then
    true
 else
        if s = '' then
```

```
      true
    else
          false
        endif
 endif;

--A class can not have attributes with the same name
rule ClassFeatureChecking {
        from
                aa : ContainerMetaModel!CClass
                (
                        let atts: Sequence(ContainerMetaModel
                            !Attribute) = aa.feature->select(x
                             | x.oclIsKindOf(
                            ContainerMetaModel!Attribute))
            in
                        not atts->forAll( c1 | atts->forAll(
                            c2 | c1 <> c2 implies c1.name <>
                            c2.name))
                )
        to
          aao : Problem!Problem
          (
                severity     <- #Error,
                location     <- 'Class ' + aa.name,
                description  <- 'Has attributes with the same
                     name'
          )
}

--Two classes can not have the same name
rule PackageChecking1 {
        from
                bb : ContainerMetaModel!Package
                (
                        let atts: Sequence(ContainerMetaModel
                            !CClass) = bb.elements->select(x |
                             x.oclIsKindOf(ContainerMetaModel!
                            CClass))
                        in
```

```
                        not atts->forAll( c1 | atts->forAll(
                            c2 | c1 <> c2 implies c1.name <>
                            c2.name))
                )
        to
          bbo : Problem!Problem
          (
                severity     <- #Error,
                location     <- 'Package ' + bb.name,
                description  <- 'Has classes with the same
                    name'
          )
}

--Two associations can not have the same name
rule PackageChecking2 {
        from
                cc : ContainerMetaModel!Package
                (
                        let atts: Sequence(ContainerMetaModel
                            !Association) = cc.elements->
                            select(x | x.oclIsKindOf(
                            ContainerMetaModel!Association))
                        in
                        not atts->forAll( c1 | atts->forAll(
                            c2 | c1 <> c2 implies c1.name <>
                            c2.name))
                )
        to
          cco : Problem!Problem
          (
                severity     <- #Error,
                location     <- 'Package '  + cc.name,
                description  <- 'Has associations with the
                    same name'
          )
}

--The associations must have exactly two association ends
rule AssociationChecking1 {
```

```
        from
                dd : ContainerMetaModel!Association
                (
                        dd.associationEnd->size() <> 2
                )
        to
          ddo : Problem!Problem
          (
                severity     <- #Error,
                location     <- 'Association '  + dd.name,
                description  <- 'The association must have
                    two association ends'
          )
}

--The associations must relate two classes
rule AssociationChecking2 {
        from
                ff : ContainerMetaModel!Association
                (
                        ff.associationEnd->exists(x | x.
                            participant.oclIsUndefined() =
                            true )
                )
        to
      ffo : Problem!Problem
          (
                severity     <- #Error,
                location     <- 'Association '  + ff.name,
                description  <- 'The association must relate
                    two classes'
          )
}

--The generalizations must have super and sub class
rule GeneralizationChecking1 {
        from
                gg : ContainerMetaModel!Generalization
                (
                        gg.supClass.oclIsUndefined() = true
```

```
                        or gg.subClass.oclIsUndefined() =
                        true
                )
        to
      ggo : Problem!Problem
          (
                severity      <- #Error,
                location      <- 'Generalization '  + gg.name,
                description   <- 'Must have superclass and
                    subclass'
          )
}

--Can not be two identical generalizations
rule GeneralizationChecking2 {
        from
                gg : ContainerMetaModel!Generalization
                (
                        (not gg.supClass.oclIsUndefined())
                            and (not gg.subClass.
                            oclIsUndefined()) and
                        ( ContainerMetaModel!Generalization.
                            allInstances()->
                          exists(x | x.supClass = gg.supClass
                               and x.subClass = gg.subClass
                              and x <> gg))
                )
        to
      ggo : Problem!Problem
          (
                severity      <- #Error,
                location      <- 'Generalization '  + gg.name,
                description   <- 'is happening twice'
          )
}


--The generalizations can not make a cycle
--Initial call: self = origen
helper context ContainerMetaModel!CClass def: superClasses(
```

```
    origen: ContainerMetaModel!CClass) : Set(
    ContainerMetaModel!CClass) =
      if self.supertype->isEmpty() then
              Set{}
          else
                 let supercs : Set(ContainerMetaModel!CClass)
                      = self.supertype->collect(x|x.supClass).
                     asSet()
                 in
                    supercs.union(
                      supercs->select( y | y <> origen)->
                          collect(z | z.superClasses(origen))
                          ->flatten().asSet() )
          endif;

rule GeneralizationChecking3 {
        from
                gg : ContainerMetaModel!CClass
                (
                        gg.superClasses(gg)->includes(gg)
                )
        to
      ggo : Problem!Problem
          (
                severity     <- #Error,
                location     <- 'Class '  + gg.name,
                description  <- 'has a generalization cycle'
          )
}

--The AssociationEnds must have a unique name within the
    Association.
rule AssociationEndChecking1 {
        from
                hh : ContainerMetaModel!Association
                (
                        not hh.associationEnd->forAll(x| hh.
                            associationEnd->forAll(y| x <> y
                            implies x.name <> y.name ))
                )
```

```
        to
      hho : Problem!Problem
          (
                severity     <- #Error,
                location     <- 'Association ' + hh.name,
                description  <- 'The association ends must
                    have unique names'
          )
}

--Only one AssociationEnd can be composite
rule AssociationEndChecking2 {
        from
                hh1 : ContainerMetaModel!Association
                (
                        hh1.associationEnd->select(x|x.
                            composite = true)->size() > 1
                )
        to
      hh1o : Problem!Problem
          (
                severity     <- #Error,
                location     <- 'Association ' + hh1.name,
                description  <- 'Has two composite
                    association ends'
          )
}

--The tag names can not be the same within an element
rule TagChecking {
        from
                ii : ContainerMetaModel!ConceptualElement
            (
                not ii.tagged->forAll(x| ii.tagged->forAll(y|
                     x <> y implies x.name <> y.name ))
            )
        to
      iio : Problem!Problem
          (
                severity     <- #Error,
```

```
                location      <- 'Element ' + ii.oclType().
                    name + ' name: ' + ii.name,
                description   <- 'The tag names must be
                    different'
            )
}

--The associationEnd names can not be equal to the class'
    attribute names
rule AssociationEndChecking3 {
        from
                hh2 : ContainerMetaModel!CClass
                (
                        not hh2.associationEnd->forAll(y |
                            hh2.feature->select(x | x.
                            oclIsKindOf(ContainerMetaModel!
                            Attribute))->forAll(z|z.name <> y.
                            name))
                )
        to
            hh2o : Problem!Problem
            (
                severity      <- #Error,
                location      <- 'Class ' + hh2.name,
                description   <- 'Has one associationEnd name
                    equal to one atribute name'
            )
}
```

Apéndice C

Plantillas de generación de las interfaces abstractas de usuario

C.1. Plantillas de generación de los esquemas XML

Listado C.1: Plantilla del esquema de una clase navegacional

```
helper context NavigableClass def: Schema() : String =
<xs:element name="$self.name$" type="xs:string">
<xs:complexType> las
  <xs:element name="oid"  type="xs:string"/>
  $
   self.features->oclIsTypeOf(Attribute)
   ->iterate( x, acc: String = "" | x.Schema() + "\n" )

  self.navigableEnd
   ->select(x|x.navigable and x.composite)
   ->select(x.opposite.target)
   ->iterate( x, acc: String = '' | acc + x.Schema() )
  $
</xs:complexType>
<xs:element/>
```

Listado C.2: Plantilla del esquema de un atributo

```
helper context Attribute def: Schema() : String =
   <xs:element name="$self.name$" type="xs:$self.type$"/>
```

Listado C.3: Esquema de la visita guiada autor

```
helper context Tour def: Schema() : String =
<xs:element name="$self.name$">
<xs:complexType>
<xs:sequence>
   <xs:element previous type="xs:string">
   <xs:element next type="xs:string">
   <xs:element ref="
     $self.navigableEnd->
            select(x|x.navigable
                 and x.opposite.target.oclIsTypeOf(
                     NavigableClass))->
                 flatten()->name$" maxOccurs="1" minOccurs
                     ="1"/>
</xs:sequence>
</xs:complexType>
</xs:element>
```

C.2. Interfaces abstractas por omisión

Listado C.4: Interfaz abstracta de una clase navegacional

```
helper context NavigableClass def: aui() : String =
<interface>
     <label text="$self.name$"/>
         $
           self.auiBody()
     $
  </window>
</interface>

helper context NavigableClass def: auiBody() : String =
 //Generates the attributes
 self.features->oclIsTypeOf(Attribute)
   ->iterate( x, acc: String = "" | x.aui() + "\n" )
```

```
 //Generates the associations
 self.navigableEnd( x | x.navigable and (not composite))
   ->iterate( x, acc: String = "" | x.aui() + "\n" )
```

Listado C.5: Interfaz abstracta de un atributo

```
helper context Attribute def: aui() : String =
  <aui:match select="//$self.name$">
    <label>
         <aui:atribute name ="text" value="."/>
    </label>
  <aui:match>
```

Listado C.6: Interfaz abstracta de una asociación

```
helper context NavigableEnd def: aui() : String =
<link>
 <name>$self.target.name$</name>
 <target>$self.target.name$</target>
   <params>
                <oid><aui:value select="/oid"></oid>
   <params/>
</link>
```

Listado C.7: Interfaz abstracta de una visita guiada

```
helper context Tour def: aui() : String =
$
let tourTarget =
     self.navigableEnd->
         select(x|
                x.navigable and
                        x.opposite.target.oclIsTypeOf(
                           NavigableClass))
         ->flatten()->first()
in
$
<interface name="$self.name$">
  <window>
    <label text="$tourTarget.name$"/>
        $
```

```
          tourTarget.auiBody
     $
     <aui:match select="//previous">
      <link>
           <name>Previous</name>
           <target>$self.name$</target>
       <params>
         <oid><aui:value select="//previous"></oid>
       <params/>
      </link>
         </aui:match>
         <aui:match select="//next">
      <link>
           <name>Next/name>
           <target>$self.name$</target>
       <params>
         <oid><aui:value select="//next"></oid>
       <params/>
      </link>
         </aui:match>
  </window>
</interface>
```

Listado C.8: Interfaz abstracta de un índice

```
helper context Index def: aui() : String =
$
let indexTarget =
self.navigableEnd
->select(x|x.navigable
  and x.opposite.target.oclIsTypeOf(NavigableClass))
  ->flatten()->first()
in
$
<interface name="$self.name$">
   <window>
     <aui:match select="/*/"$indexTarget.name$>
           <group align="vertical">
       <link>
             <name><aui:value select="./name"></name>
             <target>$indexTarget.name$</target>
```

```
            <params>
             <oid><aui:value select="./oid"></oid>
            </params>
         </link>
             $
             indexTarget.features->oclIsTypeOf(Attribute)
              ->iterate( x, acc: String = "" | acc +
             $
              <label>
               <aui:atribute name ="$x.name$" value="./$x.name$
                   "/>
          </label>
             $
              )
             $
         </group>
       </aui:match>
    </window>
  </interface>
```