UNIVERSIDAD DE EXTREMADURA

TESIS DOCTORAL

UN PROCESO DE MODERNIZACIÓN DE APLICACIONES WEB A RIA

ROBERTO RODRÍGUEZ ECHEVERRÍA

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS INFORMÁTICOS Y TELEMÁTICOS

2013

Conformidad de los directores:

Fdo: Dr. Pedro José Clemente Martín
Titular de Universidad
Departamento de Ingeniería de Sistemas Informáticos y Telemáticos
Universidad de Extremadura

Fdo: Dr. José María Conejero Manzano
Contratado Doctor
Departamento de Ingeniería de Sistemas Informáticos y Telemáticos
Universidad de Extremadura

A mi madre, siempre un ejemplo de esfuerzo y superación.

A Alex y Bea, que dan sentido a todos los esfuerzos.

A Luz, compañera infatigable en cada uno de ellos.

Agradecimientos

Aunque suene a tópico, es perceptivo comenzar hablando del tiempo invertido, del antes y el después de finalizar un esfuerzo de la magnitud del desarrollo de una tesis doctoral. En este sentido, mi travesía personal ha sido más larga de lo normal y, por lo tanto, mi mochila vuelve cargada de innumerables experiencias. Entre estas experiencias me gustaría quedarme con todas aquellas que me han hecho crecer como persona y como investigador, sean buenas y no tan buenas. Sirvan estos agradecimientos para reconocer el papel fundamental que habéis jugado en la adquisición y disfrute (también sufrimiento, como no puede ser de otra forma) de estas experiencias.

Agradecer a mis directores actuales de tesis, Pedro José Clemente y José María Conejero, el tiempo empleado, sus consejos y su apoyo, que han sido claves para finalizar esta travesía con éxito.

Agradecer el apoyo y los recursos brindados por todos los miembros del grupo de investigación Quercus SEG y, en especial, a su director Juan Hernández. Un enorme agradecimiento a sus miembros más implicados con el desarrollo del proyecto MIGRARIA: María Dolores Villalobos, Víctor M. Pavón y Fernando Macías. Hemos conseguido crear un pequeño, pero potente, equipo de trabajo del que cada día me siento más satisfecho. En este sentido, agradezco a Fernando Sánchez Figueroa que, como investigador principal del proyecto MIGRARIA, haya creído en este equipo y haya puesto los medios necesarios para su formación y mantenimiento.

Agradecer, también, el apoyo y ánimo recibidos por parte de los miembros del Departamento de Ingeniería de Sistemas Informáticos y Telemáticos. En especial, a Alberto Gómez y Adolfo Lozano que me han enseñado mucho estos últimos años, aunque ellos lo niegen. También, a Encarna Sosa por sus ánimos, por sus trabajos dentro del proyecto MIGRARIA y por su sentido del humor único. A mi antiguo compañero de despacho y amigo Álvaro Prieto con el que he compartido muchas de estas experiencias y que siempre me echa una mano con la burocracia. Y, finalmente, a Pilar Bachiller que ha jugado un papel fundamental en el desarrollo de esta tesis al hacer posible que Álvaro y yo dejásemos de compartir despacho.

Agradecer a mi equipo de la oficina técnica del Campus Virtual que me ha estado mimando durante este último año para permitirme terminar este trabajo finalmente.

Finalmente, quiero agradecer a mi entorno personal más cercano todos los ánimos recibidos y toda la ayuda prestada, y comprometerme en devolverles el tiempo robado por esta tesis. A Luz, a Alejandro y a

Beatriz que han soportado de manera directa los momentos más duros de trabajo y han sabido cuidarme para superarlos juntos. A mi hermana, Susana, que lleva todo un año animándome para que deposite la tesis. A mis padres, Jerusalén y Roberto, por su apoyo y por su constante labor de abuelos. Y, también, a los otros abuelos, Álvaro y Ángela por su continua disponibilidad para cuidar de sus nietos.

Posiblemente haya olvidado mencionar a muchos otros que aportaron directa o indirectamente en el desarrollo de este trabajo. Vaya desde aquí mi agradecimiento más sincero para todos ellos.

Resumen

Desde sus inicios a principios de la década de los 90, la World Wide Web (la web) no ha parado de evolucionar cada vez a un ritmo más elevado. Nuevos avances tecnológicos y nuevas formas de interacción sociales, junto al empuje del interés empresarial, se han ido sucediendo a lo largo de estas dos últimas décadas dando lugar a nuevos conceptos de la web, como la web 2.0 o la más reciente web móvil. Las Aplicaciones de Internet Ricas (Rich Internet Applications, RIA) comprenden un conjunto de tecnologías y conceptos que han impulsado el desarrollo de estas redefiniciones de la web.

Esta imparable evolución tanto tecnológica como social conlleva la necesidad de que empresas y organizaciones realicen un esfuerzo constante de adaptación y redefinición de sus aplicaciones web para intentar mantener a sus usuarios o atraer nuevos. En la última década, el desarrollo de aplicaciones web se ha apoyado en el uso de frameworks que se han usado para reducir notablemente los costes de desarrollo y mantenimiento de las mismas, permitiendo almacenar y gestionar el conocimiento relacionado con el desarrollo de este tipo de aplicaciones. Esta aproximación, sin embargo, presenta como principal inconveniente su alto grado de acoplamiento con una determinada pila tecnológica de implementación, que limita o complica la adaptación de la aplicación desarrollada a nuevos avances o su migración a otras tecnologías. Con el objetivo de paliar esta problemática, el consorcio *Object Management Group* (OMG) ha creado la *Architecture-Driven Modernization Task Force* (ADMTF) para crear especificaciones y promocionar el consenso industrial en la modernización de aplicaciones existentes. Este grupo propone el uso de técnicas y herramientas propias de la Ingeniería Dirigida por Modelos (IDM) con el objetivo de formalizar y estandarizar estos procesos de modernización, como se deriva de su principal aportación: el estándar *Knowledge Discovery Metamodel* (KDM). Siguiendo esta línea de trabajo, esta tesis define un proceso de modernización independiente de la tecnología, sistemático, reutilizable y dirigido por modelos para el escenario de modernización que tiene por origen una aplicación web heredada a partir de la cual se desea generar un nuevo interfaz de acceso mediante un cliente RIA. Las principales contribuciones de esta tesis se pueden resumir en:

1. **La definición de un proceso de modernización de aplicaciones web heredadas a RIAs**. Este proceso está formado por las tres fases principales de un proceso de reingeniería software: el proceso de ingeniería inversa, la reestructuración o refinado y el proceso de ingeniería directa. Y se caracteriza por estar especialmente definido para el dominio de aplicaciones web heredadas desa-

rrolladas mediante un framework de aplicaciones web basado en el patrón MVC, y por contemplar métodos, técnicas y herramientas propias de la IDM, en general, y de la Ingeniería Web Dirigida por Modelos (IWDM), en particular.

2. **La definición de una notación conceptual para aplicaciones web heredadas desarrolladas mediante un framework conforme al patrón MVC**. Esta notación está específicamente diseñada para procesos de ingeniería inversa y permite independizar al proceso de modernización de la tecnología de implementación final. Entre sus objetivos se contempla que sea sencillo de entender y utilizar por parte de un ingeniero con experiencia en el desarrollo de aplicaciones web mediante frameworks conformes al patrón MVC. Concretamente, se ha definido el metamodelo MIGRARIA MVC conforme a e-core y usando las herramientas del proyecto Eclipse Modeling Framework.

3. **La definición del proceso de ingeniería inversa como una cadena de transformaciones**. Esta cadena de transformaciones consta de dos etapas fundamentales. La primera consiste en la generación de modelos dependientes de la tecnología a partir de los diferentes artefactos software que componen la aplicación web heredada. Mientras que la segunda consiste en la generación de un modelo conforme al metamodelo MIGRARIA MVC a partir de los modelos dependientes de la tecnología mediante la definición de reglas de transformación modelo a modelo en ATL, que implementan el proceso de análisis estático de la aplicación existente.

4. **La definición de cadenas de transformación dentro de las etapas de reestructuración e ingeniería directa**. Dentro del resto de partes del proceso de modernización, se han realizado varios trabajos siguiendo diferentes estrategias para valorar su viabilidad. De esta manera, se ha definido, por ejemplo, un proceso de transformación de MIGRARIA MVC a WebML con el objetivo de poder utilizar las herramientas disponibles para esa aproximación dentro del proceso de reestructuración. Adicionalmente, también se ha implementado un proceso de generación de código que permite obtener una capa de servicios REST a partir de la aplicación heredada.

How complex or simple a structure is depends critically upon the way we describe it. Most of the complex structures found in the world are enormously redundant, and we can use this redundancy to simplify their description. But to use it, to achieve this simplification, we must find the right representation.

Simon H.A. - The architecture of complexity. In: The Sciences of the Artificial, MIT Press, 1969. (Reprinted in 1981)

Abstract

Since its origin in early nineties, the World Wide Web (the web) has been in constant evolution. New technologies and new ways of social interaction, together with the growing industry interest, has been shaping the original concept of the web to produce new realizations, as the web 2.0 and the more recent mobile web. Rich Internet Applications involve a collection of technologies and principles that have fueled the development of those web redefinitions.

Its ongoing technological and social evolution requires that companies and organizations make a constant effort of adaptation to modernize their web applications to keep their customers or to attract new ones. During the last decade, the development of web applications has been supported by specific frameworks to reduce notably the costs of development and maintenance. Those frameworks have also fostered the conservation and management of the knowledge related to the development of such kind of applications. That approach, however, presents as main drawback its high coupling with a concrete technology stack, which makes the adaptation of the developed application or its migration to new technologies a really complex process. To alleviate that situation, the *Object Management Group* (OMG) consortium started the *Architecture-Driven Modernization Task Force* (ADMTF) to define standars and to promote industrial consensus in the modernization of existing applications. That group advocates for the application of methods and tools defined by Model-Driven Engineering (MDE) to formalize and standardize those modernization processes, as it is derived from its main contribution: the *Knowledge Discovery Metamodel* (KDM) standard. Aligned with that guideline, this thesis defines a technology-independent, sistematic, reusable, and model-driven modernization process to derive RIA clients from legacy web applications. The main contributions of this work may be summarized as follows:

1. **The definition of a modernization process to derive RIA from legacy web applications**. That process is compound by the three characteristic stages of a software reengineering process: reverse engineering, reestructuring, and forward engineering. In this case, the process has been specifically defined within the domain of legacy web applications developed by means of a web application development framework based on the Model-View-Controller (MVC) pattern. And it has also been conceived to use methods and tools from the MDE, in general, and the Model-Driven Web Engineering (MDWE), in particular.

2. **The definition of a conceptual notation for applications of the domain**. Such notation has been specifically designed to support reverse engineering and to decouple the modernization process from the involved technology platforms. It has also been taken into account its understandability by software engineers with experience in the development of web applications by means of frameworks based on the MVC pattern. In concrete, the MIGRARIA MVC metamodel has been defined from Ecore and by means of the tools of the Eclipse Modeling Framework.

3. **The definition of a reverse engineering process as a model transformation chain**. That transformation chain contains two main stages. The first one consists on the definition of text to model transformations to generate technology-dependent models from the software artifacts of the legacy application. And the second one consists on the extraction of a MIGRARIA MVC model from the models previously generated by means of static analysis.

4. **The definition of model transformation chains to realize reestructuring and forward engineering**. Within the latter stages of the modernization process, different approaches has been implemented to evaluate its viability. In that sense, it has been defined, by example, a transformation to derive a WebML representation of the legacy application starting from its MIGRARIA MVC model. That way it is feasible to leverage the toolkit related to WebML in the reestructuring process. Additionally, a code generation process to provide the legacy application with ist corresponding REST API has been successfully implemented.

How complex or simple a structure is depends critically upon the way we describe it. Most of the complex structures found in the world are enormously redundant, and we can use this redundancy to simplify their description. But to use it, to achieve this simplification, we must find the right representation.

Simon H.A. - The architecture of complexity. In: The Sciences of the Artificial, MIT Press, 1969. (Reprinted in 1981)

Índice general

Resumen VII

Abstract IX

I Introducción y Antecendentes 1

1. Introducción 3
1.1. Motivación . . . 3
1.1.1. Desarrollo de Aplicaciones Web . . . 5
1.1.1.1. Tecnologías . . . 5
1.1.1.2. Frameworks . . . 6
1.1.1.3. Ingeniería Web Dirigida por Modelos . . . 6
1.2. Resumen de la propuesta . . . 7
1.2.1. El proyecto MIGRARIA . . . 7
1.2.2. Dominio de aplicación . . . 7
1.2.3. Objetivos de esta Tesis . . . 8
1.2.4. Metodología . . . 9
1.2.5. Propuesta . . . 10
1.2.5.1. Proceso de modernización . . . 10
1.2.5.2. Ingeniería inversa dirigida por modelos . . . 12
1.2.5.3. Estrategias de generación RIA . . . 13
1.3. Estructura de la Tesis . . . 14

2. Desarrollo Dirigido por Modelos de Aplicaciones de Internet Ricas 19
2.1. Características de una Aplicación de Internet Rica . . . 19
2.1.1. Metodología de investigación . . . 20
2.1.2. Características de las RIAs . . . 21
2.1.3. Propuestas de desarrollo RIA . . . 23
2.1.3.1. WebML-RIA . . . 23
2.1.3.2. OOHDM – RIA . . . 24
2.1.3.3. SHDM RIA . . . 25
2.1.3.4. RUX-Method (2007) . . . 25
2.1.3.5. UsiXML . . . 26
2.1.3.6. OOH4RIA . . . 26
2.1.3.7. OOWS 2.0 . . . 27
2.1.3.8. ADRIA – Abstract Design of Rich Internet Applications . . . 28
2.1.3.9. IAML . . . 29
2.1.3.10. UWE y RIA Patterns . . . 30
2.1.3.11. UWE-R . . . 30
2.2. Lecciones aprendidas . . . 31

3. Modernización de Aplicaciones Web **35**
3.1. Mantenimiento Software . . . 35
3.2. Reingeniería Software . . . 37
3.2.1. Ingeniería Inversa . . . 38
3.2.2. Ingeniería Inversa de Sistemas Heredados . . . 39
3.2.3. El Ciclo de Vida de la Reingeniería . . . 40
3.3. Modernización Software Dirigida por Modelos . . . 41
3.3.1. Modernización Dirigida por Arquitectura . . . 43
3.3.2. Trabajos de modernización . . . 44
3.3.3. Herramientas de modernización . . . 47
3.4. Ingeniería Inversa de Aplicaciones Web . . . 48
3.4.1. Reverse Engineering to RMM . . . 50
3.4.2. WANDA . . . 50
3.4.3. WARE . . . 51
3.4.4. Reverse Engineering to UWA . . . 51
3.4.5. ASP.NET . . . 51
3.4.6. ReWeb . . . 52
3.4.7. VAQUISTA . . . 52
3.4.8. TERESA . . . 53
3.4.9. Revangie . . . 53
3.4.10. JSPick . . . 53
3.5. Lecciones aprendidas . . . 53

II Propuesta **57**

4. Proceso de Modernización **59**
4.1. Dominio . . . 59
4.2. Objetivos de una modernización . . . 61
4.3. Un marco de referencia para procesos de modernización . . . 62
4.3.1. Ingeniería inversa . . . 62
4.3.2. Refinado y reestructuración . . . 64
4.3.3. Ingeniería directa . . . 65
4.4. Un proceso de modernización . . . 65
4.4.1. Extracción de información . . . 67
4.4.2. Representación del conocimiento . . . 68
4.4.3. Refinado RIA . . . 68
4.4.4. Generación de código . . . 70
4.5. Resumen . . . 71

5. Representación Conceptual **73**
5.1. Necesidad de una representación . . . 73
5.2. Especificación de la notación . . . 74
5.2.1. Modelo . . . 75
5.2.1.1. Notación . . . 76
5.2.2. Vista . . . 79
5.2.2.1. Objetos de presentación . . . 80
5.2.2.2. Atributos de los objetos de presentación . . . 81
5.2.2.3. Peticiones . . . 83
5.2.2.4. Notación . . . 85
5.2.3. Controlador . . . 90
5.2.3.1. Relación petición-controlador-respuesta . . . 91
5.2.3.2. Mapeo entre elementos de la Vista y elementos del Modelo . . . 92
5.2.3.3. Invocación a operaciones, paso de parámetros e instancias . . . 93

5.2.3.4. Notación . . . 95
5.3. Resumen . . . 100

6. Generación de un Modelo Conceptual 101
6.1. Ingeniería Inversa . . . 101
6.2. Generación de los modelos dependientes de la tecnología . . . 102
6.2.1. Extensión Struts-JSP . . . 104
6.2.2. Extensión Struts-Config . . . 106
6.2.3. Extensión Hibernate . . . 107
6.2.4. Resultados del proceso . . . 108
6.3. Generación del modelo MIGRARIA-MVC . . . 109
6.3.1. Reglas para el Modelo . . . 110
6.3.1.1. Reglas para objetos y atributos . . . 110
6.3.1.2. Reglas para relaciones . . . 111
6.3.1.3. Reglas para operaciones . . . 112
6.3.2. Reglas para la Vista . . . 113
6.3.2.1. Reglas para objetos y atributos de presentación de salida . . . 114
6.3.2.2. Reglas para objetos y atributos de presentación de entrada/salida . . . 115
6.3.2.3. Reglas para peticiones . . . 117
6.3.2.4. Reglas para colecciones . . . 119
6.3.3. Reglas para el Controlador . . . 121
6.3.3.1. Controladores y flujos de control . . . 123
6.3.3.2. Reglas para generar flujos de control, invocación y respuesta . . . 124
6.3.3.3. Reglas para generar las instancias de controlador . . . 125
6.3.3.4. Reglas para generar llamadas a operaciones . . . 127
6.3.3.5. Reglas para generar mapeos entre datos . . . 130
6.4. Resumen . . . 134

7. Caso de estudio 137
7.1. Introducción . . . 137
7.2. Sistema de Revisión para Conferencias . . . 138
7.2.1. Motivación . . . 138
7.2.2. Descripción base . . . 139
7.2.2.1. Actores . . . 139
7.2.2.2. Funciones . . . 140
7.2.3. Desarrollo . . . 142
7.2.3.1. Pila tecnológica . . . 142
7.2.3.2. Metodología y planificación . . . 143
7.2.4. Reglas de negocio o convenios . . . 144
7.2.4.1. Modelo . . . 145
7.2.4.2. Vista . . . 145
7.2.4.3. Controlador . . . 146
7.2.5. Datos de complejidad del sistema . . . 146
7.2.5.1. Modelo . . . 147
7.2.5.2. Vista . . . 147
7.2.5.3. Controlador . . . 148
7.2.5.4. Métricas adicionales . . . 148
7.3. Alcance de la representación conceptual . . . 150
7.3.1. Modelo . . . 151
7.3.2. Vista . . . 153
7.3.3. Controlador . . . 158
7.3.4. Limitaciones . . . 162
7.4. Generación de una representación conceptual . . . 164
7.4.1. Modelo MIGRARIA MVC . . . 165

7.4.1.1. Modelo . . . 166
7.4.1.2. Vista . . . 168
7.4.1.3. Controlador . . . 168
7.4.2. Limitaciones . . . 170
7.4.2.1. Modelo . . . 170
7.4.2.2. Vista . . . 173
7.4.2.3. Controlador . . . 173
7.5. Resumen . . . 174

8. Estrategias de Generación RIA **177**
8.1. Introducción . . . 177
8.2. Proyección a una propuesta IWDM . . . 178
8.2.1. Proyección a WebML . . . 178
8.2.1.1. Modelo de datos . . . 179
8.2.1.2. Modelo de hipertexto . . . 179
8.3. Definición de patrones RIA en WebML . . . 182
8.3.1. Patrones de Sincronización . . . 183
8.3.2. Modelado de los Patrones de Sincronización . . . 184
8.4. Selección de patrones RIA . . . 185
8.4.1. Mapeo entre requisitos y características RIA . . . 186
8.4.2. Mapeo entre patrones y características RIA . . . 186
8.4.3. Mapeo entre requisitos y patrones . . . 188
8.5. Aplicación de los patrones RIA seleccionados . . . 189
8.5.1. Metamodelo RIA . . . 190
8.5.2. Metamodelo de Weaving . . . 191
8.5.3. El proceso de composición . . . 192
8.6. Generación de API REST . . . 194
8.6.1. Especificación de la capa REST dependiente del Framework . . . 195
8.6.2. Especificación de la API REST dependiente de la aplicación . . . 197
8.6.2.1. Identificación de recursos . . . 198
8.6.2.2. Generación de URIs REST . . . 199
8.6.2.3. Identificación de Operaciones Disponibles . . . 200
8.6.2.4. Generación de los mapeos a peticiones REST . . . 200
8.6.2.5. Delegación del procesado de peticiones REST a las Actions Heredadas de Struts . . . 201
8.6.2.6. Generación de la configuración de la capa REST: especificación de la API REST . . . 202
8.7. Resumen . . . 203

III Reflexión y Resultados **205**

9. Conclusiones y Trabajos Futuros **207**
9.1. Resumen de las principales contribuciones . . . 207
9.2. Discusión . . . 209
9.3. Conclusiones . . . 213
9.3.1. Conclusiones del uso de IDM en la modernización . . . 213
9.3.1.1. Ventajas . . . 213
9.3.1.2. Inconvenientes . . . 216
9.3.2. Conclusiones del desarrollo . . . 217
9.4. Trabajo futuro . . . 218
9.4.1. Evolución de la propuesta . . . 219
9.4.2. Extensión de la propuesta . . . 219

10. Publicaciones **221**
10.1. Revistas indexadas . . . 221
10.2. Congresos . . . 221

Bibliografía **225**

Anexos **235**

A. Reglas de transformación en ATL **237**
A.1. Reglas para el Modelo . . . 237
A.1.1. Reglas para objetos y atributos . . . 237
A.1.2. Reglas para relaciones . . . 239
A.1.3. Reglas para operaciones . . . 241
A.2. Reglas para la Vista . . . 243
A.2.1. Reglas para páginas . . . 243
A.2.2. Reglas para objetos y atributos de salida . . . 243
A.2.3. Reglas para objetos y atributos de entrada/salida . . . 244
A.2.4. Reglas para peticiones . . . 246
A.2.5. Reglas para colecciones . . . 248
A.3. Reglas para el Controlador . . . 250
A.3.1. Reglas para flujos de control y respuesta . . . 250
A.3.2. Reglas para las instancias de controlador . . . 251
A.3.3. Reglas para llamadas a operaciones . . . 252
A.3.3.1. Reglas para los parámetros de llamadas a operaciones . . . 254
A.3.3.2. Reglas para los retornos de llamadas a operaciones . . . 255
A.3.4. Reglas para mapeos entre datos . . . 257

B. Entorno de Modernización **263**
B.1. Necesidad . . . 263
B.2. MoDisco Discoverers . . . 264
B.2.1. Procedimiento de construcción de un discoverer . . . 264
B.2.2. Discoverer StrutsConfig . . . 265

Índice de figuras

1.1. Estimación del crecimiento del acceso a internet móvil . . . 4
1.2. Clientes RIA y móviles . . . 5
1.3. Arquitectura del Sistema Objetivo . . . 8
1.4. Resumen del marco del proceso MIGRARIA . . . 11
1.5. Extracto del metamodelo MIGRARIA MVC . . . 13
1.6. Resumen del Proceso de Ingeniería Inversa . . . 14

2.1. Línea de tiempo de las propuestas RIA . . . 21
2.2. Evolución de las propuestas IWDM . . . 33

3.1. ISO/IEC 14764-00 Proceso de mantenimiento software . . . 36
3.2. Etapas de un proceso de ingeniería inversa . . . 38
3.3. Modelo de herradura del instituto SEI . . . 40
3.4. Esquema base de un proceso de reingeniería dirigido por modelos . . . 42
3.5. Modelo de herradura de ADM . . . 44
3.6. Coste del proyecto en función de su tamaño (Fleurey *et al.* 2007) . . . 46
3.7. Vista general de MoDisco . . . 48

4.1. Arquitectura del Sistema Objetivo . . . 62
4.2. Resumen del marco del proceso MIGRARIA . . . 63
4.3. Resumen del proceso de modernización . . . 66
4.4. Proceso de identificación de características y patrones RIA a partir de requisitos . . . 70

5.1. Extracto del metamodelo MIGRARIA MVC . . . 74
5.2. Metamodelo MVC Model . . . 76
5.3. DataObject *Conference* . . . 77
5.4. Detalles de DataObject *Conference* . . . 78
5.5. Detalles del DataObject *Conference* . . . 79
5.6. Ejemplo página Web . . . 80
5.7. Categorías de atributos de objetos de presentación . . . 81
5.8. Ejemplos de peticiones . . . 83
5.9. Peticiones sobre conjuntos de objetos . . . 85
5.10. Peticiones sobre colección de objetos . . . 85
5.11. Metamodelo MVC View . . . 86
5.12. Ejemplo entidad *Page* . . . 87
5.13. Ejemplo entidades PresentationObject y PresentationAttribute . . . 88
5.14. Ejemplo entidades Request y PresentationCollection . . . 89
5.15. Ejemplo controlador en Struts . . . 91
5.16. Metamodelo MVC Controller . . . 94
5.17. Ejemplo de flujo de control . . . 96
5.18. Ejemplo de ciclo petición-controlador-repuesta . . . 96
5.19. Ejemplo de mapeos . . . 97
5.20. Ejemplo de llamadas a operaciones . . . 99

6.1. Fase de ingeniería inversa de MIGRARIA . . . 101

6.2. Ingeniería Inversa Dirigida por Modelos en el Servidor . . . 102
6.3. Extensiones de MoDisco . . . 103
6.4. Struts JSP taglibs metamodel . . . 105
6.5. Extracto del metamodelo Struts-Config . . . 106
6.6. Extracto del metamodelo Hibernate . . . 108
6.7. Generación de objeto de datos . . . 111
6.8. Generación de relaciones entre objetos de datos . . . 112
6.9. Generación de operaciones de los objetos de datos . . . 113
6.10. Generación de objeto salida con atributo simple . . . 115
6.11. Generación de objeto de salida con atributo complejo . . . 116
6.12. Generación de objeto de entrada/salida y sus atributos . . . 118
6.13. Generación de peticiones . . . 120
6.14. Generación de colección de salida . . . 121
6.15. Generación de *ControlFlow* . . . 126
6.16. Generación de *ControllerInstance* . . . 128
6.17. Generación de *DataOperationCall* . . . 131
6.18. Generación de mapeo de entrada . . . 133

7.1. Modelo de datos del CRS . . . 151
7.2. Objetos de datos del subsistema (modelo MVC) . . . 152
7.3. Detalles objeto de datos Paper . . . 153
7.4. Código java de la interfaz PaperDAO . . . 153
7.5. Página listado de artículos enviados . . . 154
7.6. Página de envío de un nuevo artículo . . . 154
7.7. Página de edición de un artículo enviado . . . 155
7.8. Página listado de artículos enviados . . . 156
7.9. Modelos de las páginas de gestión de envío de artículos . . . 156
7.10. Modelos de las páginas de gestión de envío de artículos . . . 157
7.11. Modelos páginas de gestión de envío de artículos . . . 158
7.12. Modelos de controladores para la gestión de envío de artículos . . . 160
7.13. Modelo CRS PaperListAction . . . 161
7.14. Modelo CRS PaperDetailsAction . . . 162
7.15. Modelo CRS PaperCreateAction . . . 163
7.16. Número de instancias del modelo MIGRARIA MVC generado . . . 165
7.17. Resumen resultados proceso ingeniería inversa . . . 166
7.18. Gráfico comparativo Modelo origen y generado . . . 167
7.19. Gráfico comparativo Modelo origen y generado (detalle) . . . 167
7.20. Gráfico comparativo Vista origen y generada . . . 169
7.21. Gráfico comparativo Controlador origen y generado . . . 172

8.1. Modelo de Datos de WebML . . . 179
8.2. Operación de creación en WebML . . . 181
8.3. Operaciones de lectura en WebML . . . 181
8.4. Escenarios de sincronización a partir de las dimensiones origen y sujeto . . . 183
8.5. Patrón de Notificación de Evento de Datos . . . 185
8.6. Replicación de una colección de datos iniciada por el cliente . . . 185
8.7. Esquema de la propuesta de composición (weaving) de modelos . . . 190
8.8. Metamodelo WebML extendido con características RIA . . . 191
8.9. Extracto del WMM . . . 192
8.10. Esquema del proceso de generación de API REST . . . 195
8.11. Extensión de Struts v1.3.X para soporte REST . . . 196
8.12. Identificación de recursos en un modelo MIGRARIA-MVC . . . 198
8.13. Identificación de operaciones en un modelo MIGRARIA-MVC . . . 199
8.14. Mapeo de JSON de entrada *form bean* . . . 202

8.15. Mapeo de objeto de salida a JSON 203
8.16. Fichero de configuración de la API REST 203

Índice de tablas

2.1. Categorización y referencias de características RIA 32

3.1. Comparativa de propuestas de ingeniería inversa de aplicaciones web 54

4.1. Planes de acción de ingeniería inversa 63

5.1. Categorías de peticiones según el nivel de origen 84

6.1. Etiquetas de campos de control de Struts 1.X 116

7.1. Número de archivos y de líneas de código de cada componente 147
7.2. Datos cuantitativos de los objetos de datos del sistema 147
7.3. Datos principales de las JSPs 148
7.4. Datos principales de los controladores 149
7.5. Datos de los objetos de datos generados 166
7.6. Datos de las páginas generadas 169
7.7. Datos de los controladores generados 171

8.1. Matriz de trazabilidad de requisitos 186
8.2. Matriz de trazabilidad de patrones 187
8.3. Matriz de correspondencia requisitos-patrones 188
8.4. Combinaciones de DRR y DPR 189
8.5. Recursos y relaciones REST 198
8.6. Resumen del resultado del proceso de configuración REST 201

Parte I

Introducción y Antecendentes

Capítulo 1

Introducción

1.1. Motivación

Cada vez más la Web está siendo soportada por tecnologías post HTML/HTTP, entre las que están jugando un papel determinante las denominadas Rich Internet Applications (Fraternali *et al.* 2010). El término RIA se refiere a una familia heterogénea de soluciones caracterizadas por el objetivo común de añadir nuevas capacidades a la Web tradicional. Las RIA combinan la ligera arquitectura de distribución de la Web con la interactividad y el poder de computación de las aplicaciones de escritorio, con beneficio en todos los elementos de una aplicación Web (datos, lógica de negocio, comunicaciones y presentación) (Preciado *et al.* 2005).

Desde el punto de vista de datos las RIA soportan almacenamiento del lado del cliente en función de la tecnología y el dispositivo. Respecto a la lógica de negocio, las RIA también permiten mover parte de la computación al lado del cliente, de tal forma que se permiten modos de funcionamiento desconectados. En cuanto a comunicación se refiere, en RIA puede ser iniciada tanto por el cliente como por el servidor. Esta bi-direccionalidad elimina el tráfico innecesario típico de aplicaciones de cliente ligero. Pero sobre todo, la popularidad de las RIA viene dada por sus capacidades de presentación e interacción con el usuario, permitiendo tener un paradigma de página única o single-page. Esto evita refrescos de pantalla completos en cada interacción y permite la carga independiente de elementos individuales en pantalla. La interfaz puede ser también extendida dinámicamente cargando partes de la lógica de presentación (p.e. código JavaScript) y elementos de la interfaz en tiempo de ejecución. La experiencia de usuario puede ser aumentada mediante eventos de interacción extendidos y comportamientos temporales.

En paralelo al fenómeno tecnológico de las RIA, se debe considerar el fenómeno social de la Web 2.0, que ha constituido un aliciente fundamental para la consolidación y despegue de gran parte de las tecnologías RIA relacionadas. Adicionalmente, la irrupción de los dispositivos móviles en internet ha sido tan grande, en los últimos años, que actualmente no se concibe el desarrollo de una aplicación web sin considerar el soporte a este tipo de dispositivos. La Figura 1.1 ilustra la estimación del crecimiento

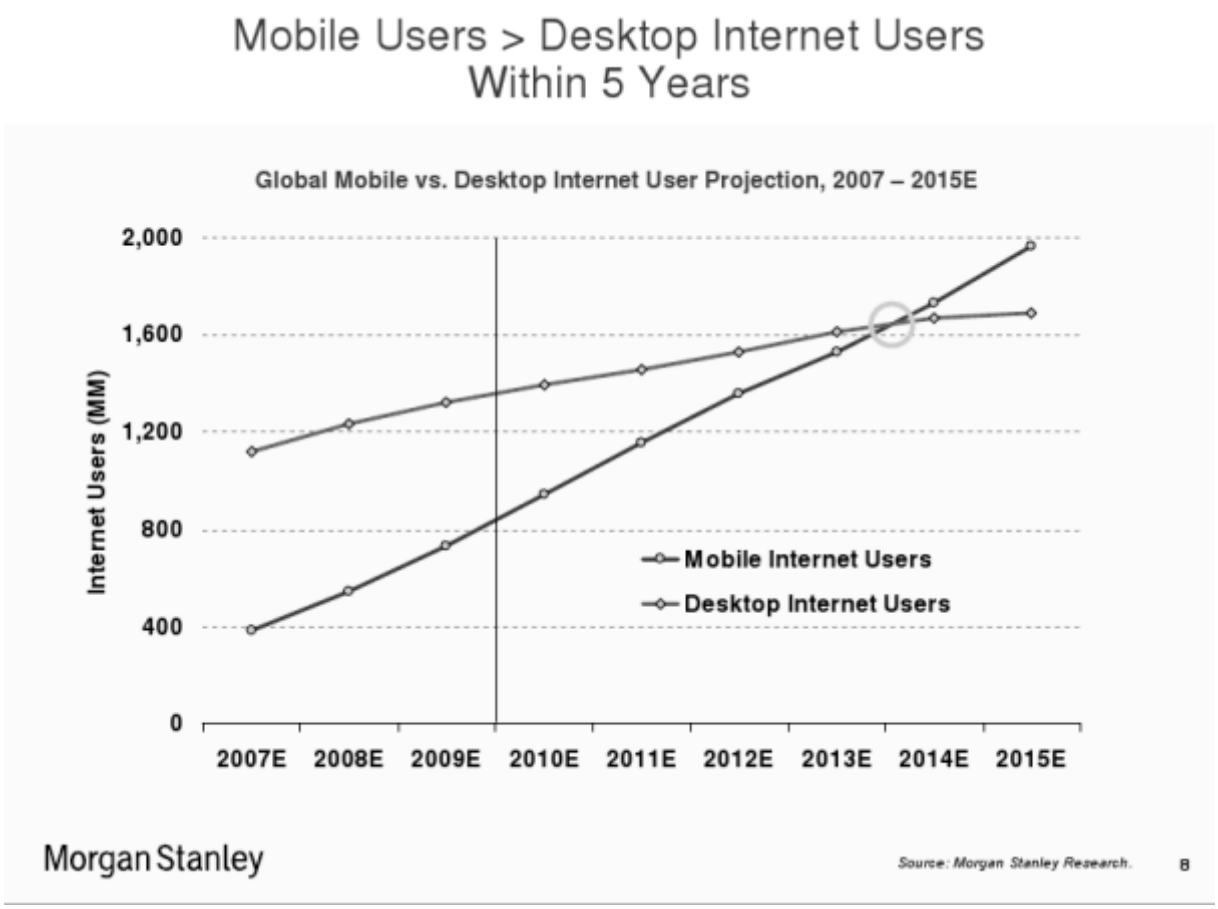
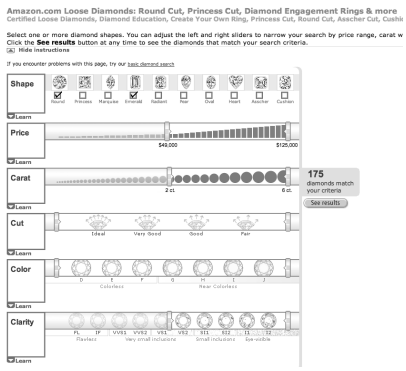


Figura 1.1: Estimación del crecimiento del acceso a internet móvil

del acceso a internet desde ordenadores de escritorio en relación con el del acceso desde dispositivos móviles[1].

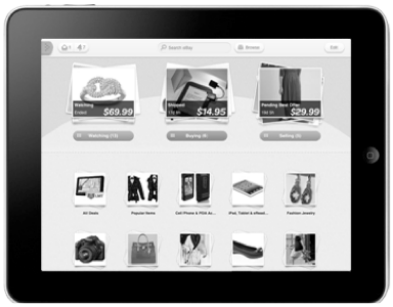
Con el objetivo de aprovechar estas nuevas capacidades tecnológicas para mantener y atraer nuevos clientes, la industria está llevando a cabo constantes procesos de reingeniería de sus aplicaciones web heredadas para producir clientes RIA o clientes móviles. La Figura 1.2 muestra ejemplos de estas prácticas, como son el buscador de diamantes de Amazon desarrollado mediante tecnologías RIA, o el cliente Adobe AIR y la aplicación móvil de eBay. Un gran número de estas aplicaciones web heredadas han sido desarrolladas a partir de Frameworks de Aplicaciones Web (FAW) que han facilitado una reducción de costes y han supuesto un mecanismo de almacenamiento del conocimiento relacionado con el desarrollo web. El principal inconveniente de esta metodología de desarrollo es fundamentalmente el gran acoplamiento de la solución con la tecnología de implementación seleccionada. Esta alta dependencia tecnológica complica futuros procesos de modernización. En muchos casos, estos procesos de modernización se están planteando desde un plano estrictamente tecnológico, dando lugar a soluciones ad-hoc, muy costosas y poco reutilizables en otros escenarios. Por lo tanto, la industria demanda la formalización y estandarización de estos procesos de reingeniería y su independencia tecnológica. En este sentido, la Object Management Group (OMG), mediante la aproximación Modernización Dirigida por Arquitectura

[1] http://www.slideshare.net/fred.zimny/morgan-staneley-internet-trends-ri041210



(a) Buscador de diamantes de Amazon

(b) Cliente Adobe AIR de eBay



(c) Cliente móvil eBay

Figura 1.2: Clientes RIA y móviles

(Architecture-Driven Modernization, ADM) (Ulrich 2010), aboga por la aplicación de técnicas y herramientas propias de la Ingeniería Dirigida por Modelos (IDM) con el objetivo de formalizar y estandarizar estos procesos de modernización.

En conclusión, el problema que intenta resolver esta tesis es la definición de un proceso de modernización independiente de la tecnología, sistemático, reutilizable y dirigido por modelos para el escenario de modernización que tiene por origen una aplicación web heredada a partir de la cual se desea generar una nueva interfaz de acceso mediante un cliente RIA.

1.1.1. Desarrollo de Aplicaciones Web

El desarrollo de aplicaciones Web con tecnología heredada o mediante tecnología RIA se ha abordado desde diferentes aproximaciones. La primera de ellas es una aproximación estrictamente tecnológica en la que se desarrolla a partir de la tecnología disponible sin un mecanismo de reutilización de soluciones comunes en el dominio. La siguiente consiste en una evolución de la primera, en la que se utilizan frameworks como mecanismo de reutilización de soluciones probadas para simplificar el proceso de desarrollo. Ambas aproximaciones de desarrollo conllevan la producción de soluciones ligadas a la tecnología final de implementación. Por su parte, la última alternativa consiste en la aplicación de los principios de la IDM en la Ingeniería Web, dando lugar a la disciplina Ingeniería Web Dirigida por Modelos (IWDM). Esta última alternativa propone aproximaciones para definir una aplicación web de forma independiente a la tecnología final.

1.1.1.1. Tecnologías

El desarrollo de aplicaciones web está sometido a una constante evolución tecnológica. A principios del siglo 21, las tecnologías fundamentalmente se reducían al protocolo HTTP, al lenguaje de marcado HTML y a tecnologías de generación de páginas de forma dinámica en servidor, como por ejemplo las tecnologías Java Server Pages y Servlet dentro de la plataforma Java Enterprise Edition (JEE). Actualmente, se cuenta, sin embargo, con un amplio abanico de tecnologías RIA diversas como AJAX

(Asynchronous Javascript y XML) o el nuevo estándar HTML5, o soluciones basadas en plug-in como Flash[2], OpenLaszlo[3], Silverlight[4] y Flex[5], entre otros. Incluso las RIA pueden ser ejecutadas fuera del navegador como es el caso de AIR[6] o JavaFX[7].

1.1.1.2. Frameworks

Un Framework de Aplicaciones Web (FAW) es una plataforma estructural semi-completa, modular y reusable que puede ser especializada para el desarrollo de aplicaciones web, (Shan & Hua 2006). Estos frameworks normalmente implementan el patrón de diseño Modelo-Vista-Controlador (MVC), en muchas ocasiones en la arquitectura Model 2[8] para el desarrollo de aplicaciones web de petición-respuesta sobre, por ejemplo, la plataforma Java Enterprise Edition. Un gran número de las aplicaciones web heredadas han sido desarrolladas mediante uno de estos frameworks. El framework Jakarta Struts (o Struts 1[9]), cuya primera versión data del 2001, puede ser considerado como uno de los principales precursores de este tipo de frameworks. Y un gran número de aplicaciones web de la primera mitad de esa década se construyeron a partir del mismo.

Durante los últimos años, al igual que ha sucedido con las tecnologías, se ha producido una importante evolución en los frameworks para adoptar los nuevos avances tecnológicos. Dentro del contexto de las tecnologías RIA, cabe destacar la aparición y evolución de los frameworks basados en javascript para la implementación del patrón MVC en cliente, como por ejemplo, Backbone JS[10]

1.1.1.3. Ingeniería Web Dirigida por Modelos

Durante la última decada, la comunidad de Ingeniería Web ha desarrollado propuestas de IDM para el desarrollo de aplicaciones web, dando lugar a la disciplina de Ingeniería Web Dirigida por Modelos (IWDM) (Rossi *et al.* 2008b). Estas propuestas han tratado de elevar el nivel de abstracción en el desarrollo de aplicaciones web con el fin de hacerlo independiente de las tecnologías. De esta manera, al no estar sujetas a la constante evolución tecnológica del desarrollo web, pueden proporcionar menores costes de desarrollo y mantenimiento. Desafortunadamente, la mayoría de las aplicaciones web heredadas fueron desarrolladas antes de que las más prometedoras propuestas de IWDM alcanzasen un grado de madurez suficiente para su aplicación profesional en la industria.

En el seno de la comunidad de Ingeniería Web, durante los últimos años, se ha producido un importante proceso de extensión de las propuestas de desarrollo web dirigidas por modelos con el objetivo de

[2] http://www.adobe.com/products/flash
[3] http://www.openlaszlo.org
[4] http://www.microsoft.com/silverlight
[5] http://www.adobe.com/products/flex
[6] http://www.adobe.com/products/air
[7] http://sun.coom/javafx
[8] http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-jspmvc.html
[9] http://struts.apache.org
[10] http://www.backbone.com

incorporar paulatinamente elementos y características RIA, afectando a todas las capas fundamentales de una aplicación Web: datos y lógica de negocio (Bozzon *et al.* 2006; Comai & Carughi 2007; Carughi *et al.* 2007; Meliá *et al.* 2008; Pérez *et al.* 2008) o presentación (Garrigós *et al.* 2009; Rossi *et al.* 2007; 2008a; Valverde & Pastor 2009; Meliá *et al.* 2010).

1.2. Resumen de la propuesta

En primer lugar, este apartado contextualiza el trabajo realizado dentro del proyecto MIGRARIA[11]. Posteriormente, define el dominio de aplicación de la propuesta, revisa los objetivos planteados y la metodología seguida en el desarrollo del trabajo. Finalmente, realiza una breve exposición de la propuesta.

1.2.1. El proyecto MIGRARIA

El trabajo desarrollado en esta tesis se encuadra dentro del proyecto MIGRARIA (TIN2011-27340 del Ministerio de Ciencia y Educación), en concreto, dentro de su módulo 1: "Modelos y transformaciones para migración hacia RIA".

Como ilustra la Figura 1.3, el objetivo fundamental de esta parte del proyecto consiste en la modernización de una aplicación web heredada en una RIA. Esta modernización consiste en la generación de dos nuevos componentes principales dentro del sistema heredado. Por un lado, un cliente RIA caracterizado por una interfaz de usuario enriquecido (similar a las aplicaciones de escritorio) y por la infraestructura requerida: un almacén de datos local, procesamiento en cliente de la lógica de interfaz y parte de la lógica de negocio y una capa de conexión para definir los modos de operación (conectado o desconectado) y comunicación (síncrona o asíncrona) introducidos por el cliente RIA. Y, por otro lado, una capa de conexión orientada a servicios que permita implementar todos los nuevos escenarios de interacción que demande el nuevo cliente RIA. Finalmente, es importante dejar constancia de que la propuesta no supone la sustitución de la aplicación web heredada, sino la adición de una nueva interfaz para interaccionar con el sistema de información heredado.

1.2.2. Dominio de aplicación

Como dominio fundamental de este proceso de modernización se consideran las aplicaciones web heredadas que cumplan parcial o totalmente el siguiente conjunto de características:

- Son aplicaciones intensivas de datos, cuya misión suele ser definir una interfaz web a un sistema de información preexistente.
- Han sido desarrolladas mediante un framework de aplicaciones web (FAW) conforme al patrón arquitectónico Modelo-Vista-Controlador (MVC).

[11] http://www.unex.es/eweb/migraria

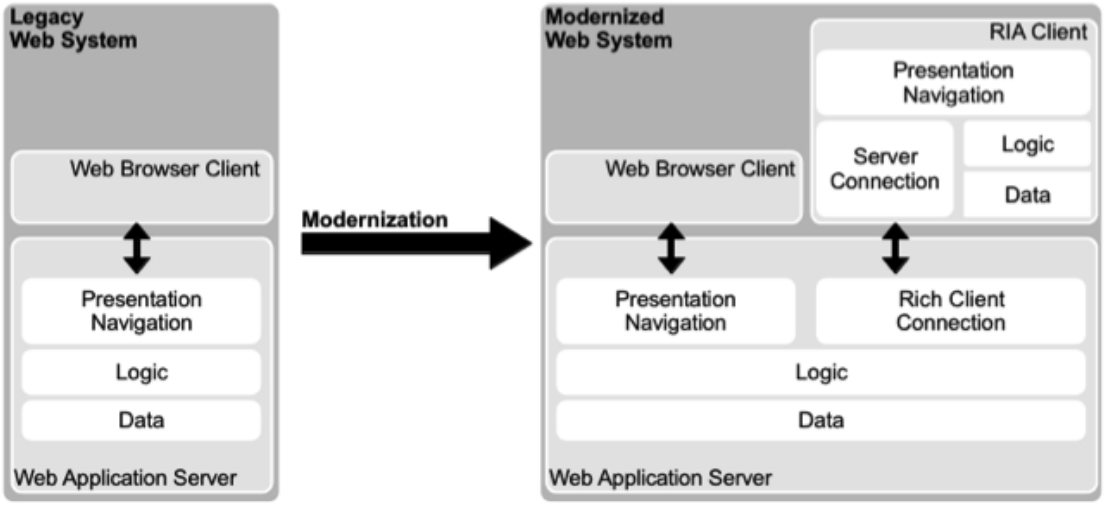


Figura 1.3: Arquitectura del Sistema Objetivo

- Han sido desarrolladas entre los años 2000 y 2005. Y, por lo tanto, las tecnologías y frameworks usados en su implementación están obsoletos o en el fin de su ciclo de vida.
- Han sido desarrolladas antes de que alguna de las propuestas de IWDM presentase la madurez suficiente.
- Se ha usado una colección heterogénea de tecnologías en su desarrollo.
- No existe información actualizada de las mismas, debido a la falta de documentación y a la falta de los desarrolladores originales.
- Han seguido una estrategia de mantenimiento poco definida.

1.2.3. Objetivos de esta Tesis

Como objetivo general (OG) de esta tesis se contempla la definición de un proceso para la modernización de aplicaciones web heredadas a aplicaciones ricas de internet (RIA). Este proceso debe ser independiente de la tecnología, sistemático, reutilizable y estar definido mediante técnicas y herramientas propias de la Ingeniería Dirigida por Modelos (IDM).

Dentro de este objetivo general, se contemplan los siguientes objetivos específicos:

1. (OE-1). Proporcionar la especificación completa de un proceso dirigido por modelos de ingeniería inversa de aplicaciones web. Este proceso es parte fundamental del proceso de modernización. Dentro de este objetivo, podemos considerar los siguientes subobjetivos:

   a) (OE-1.1). Definir una notación conceptual, independiente de la tecnología, específica para la representación de las aplicaciones web heredadas consideradas en el dominio del trabajo.

   b) (OE-1.2). Definir el proceso de ingeniería inversa como una cadena de transformaciones que vayan desde los artefactos software de la aplicación heredada hasta la notación conceptual.

2. (OE-2). Proporcionar estrategias de generación de aplicaciones RIA a partir del modelo generado por el proceso de ingeniería inversa. Dentro de este objetivo, podemos considerar:

   a) (OE-2.2). Generar una representación de la aplicación heredada en una metodología IWDM con soporte RIA.

   b) (OE-2.3). Enriquecer la representación conceptual mediante la aplicación de patrones RIA.

   c) (OE-2.4). Generar código final de la capa de conexión entre el cliente RIA y el sistema heredado.

Dentro del contexto en el que se define, la modernización de aplicaciones web, el proceso debe hacer especial hincapié en la consecución de las siguientes características:

1. Reutilizable. Definido de tal manera que permita su aplicación en otros escenarios de modernización en los que se mantengan constantes las tecnologías utilizadas y las reglas de negocio seguidas en la implementación de la aplicación web heredada.

2. Mantenible. Uso de lenguajes y técnicas ampliamente conocidas en el entorno de la IDM.

3. Sistemático. Definido como una secuencia de pasos semiautomáticos que permitan asesorar al ingenierio de modernización en las acciones a tomar.

1.2.4. Metodología

La metodología seguida en el desarrollo de este trabajo se ha compuesto de los siguientes pasos:

1. Estudio del estado del arte de las principales propuestas IWDM con soporte RIA. Este estudio es necesario para, por un lado, identificar aquellas propuestas más maduras que pueden formar parte del proceso de modernización y, por otro lado, identificar las principales características de las RIAs con el objetivo de poder definir patrones que las recojan.

2. Estudio del estado del arte de las principales propuestas de ingeniería inversa de aplicaciones web, por un lado, y de ingerniería inversa dirigida por modelos, por otro. El conocimiento derivado de este estudio permite definir un proceso de ingeniería inversa de aplicaciones web dirigido por modelos.

3. Definición de un marco de procesos de modernización de aplicaciones web heredadas a RIA. Y definición de un proceso concreto de modernización a partir del mismo.

4. Definición del proceso de ingeniería inversa dirigido por modelos de aplicaciones web heredadas como parte fundamental del proceso de modernización definido. Esta definición debe realizarse sobre técnicas y herramientas propias de la IDM para permitir su aplicación real.

5. Evaluación del proceso definido mediante su aplicación en un caso de estudio. Se ha utilizado un caso de estudio de laboratorio para poder controlar la evolución de la propuesta. Este caso de estudio se ha definido a partir de casos de estudio reales para incorporar en su implementación reglas de negocio y patrones comunes dentro del dominio.

6. Una vez conseguida una versión suficientemente completa del proceso de ingeniería inversa, se han realizado diferentes trabajos para valorar la viabilidad de las estrategias de generación de aplicaciones RIA a partir del modelo resultado del proceso de ingeniería inversa.

1.2.5. Propuesta

A continuación, se presenta un breve resumen del trabajo realizado en esta tesis organizado en los tres temas principales de trabajo abordados:

1. la definición del proceso de modernización,

2. la definición del proceso de ingeniería inversa dirigido por modelos,

3. y la experimentación con estrategias de generación RIA.

1.2.5.1. Proceso de modernización

La Figura 1.4 presenta el marco de referencia del proceso de modernización definido, (Rodríguez-Echeverría *et al.* 2012c;d). Como se puede apreciar, sigue la estructura general de un proceso tradicional de reingeniería software, presentando una organización básica en tres etapas fundamentales:

1. Ingeniería inversa dirigida por modelos. El objetivo de esta etapa consiste en la extracción y representación conceptual del conocimiento de la aplicación web heredada. En este sentido, se definen tres líneas diferentes de acción:

   a) Análisis estáticos y dinámicos en el lado del cliente. Su misión consiste en generar peticiones al servidor que permitan explorar toda la aplicación web heredada y extraer información de las mismas.

   b) Análisis estáticos en el lado del servidor. Su misión consiste en extraer información de los diferentes artefactos software que componen la aplicación web heredada. En el caso de aplicaciones desarrolladas con frameworks basados en el patrón MVC, se deben analizar los diferentes artefactos que permiten especificar la vistas, el modelo y los controladores.

   c) Minería de datos de los registros de ejecución de la aplicación web heredada. Su misión consiste en encontrar información que pueda apoyar decisiones de diseño RIA posteriores dentro del proceso de modernización, como por ejemplo, qué datos o qué lógica de negocio puede moverse a cliente y de qué forma.

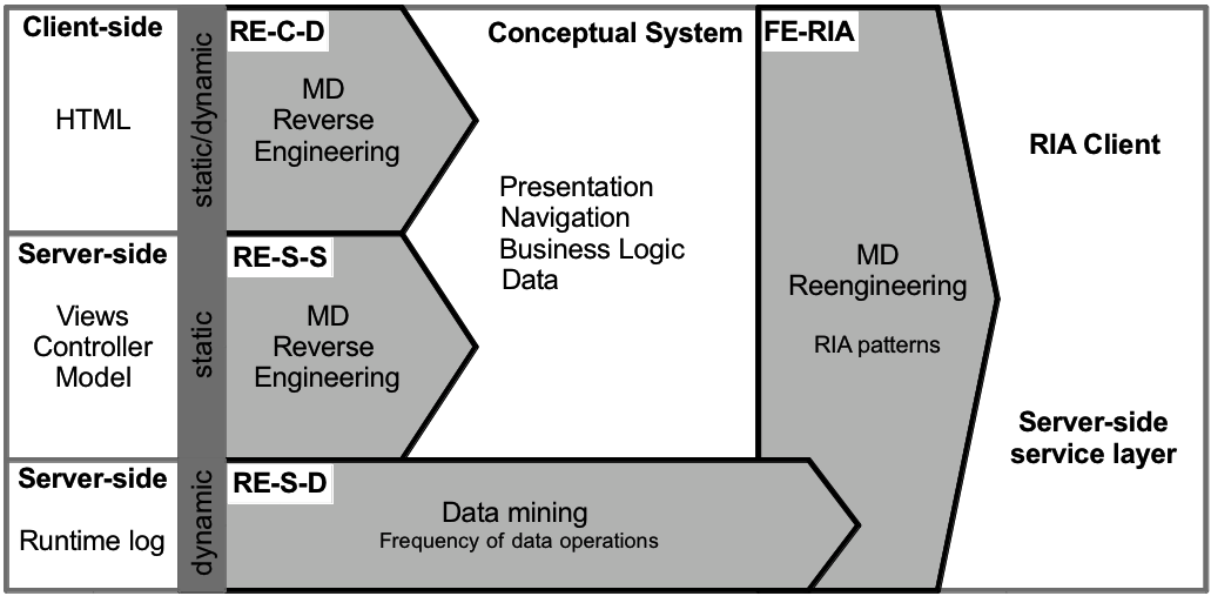


Figura 1.4: Resumen del marco del proceso MIGRARIA

2. Refinado o reestructuración. El objetivo de esta etapa es refinar la representación conceptual para poder utilizarla dentro de un proceso de modernización. Una de sus actividades fundamentales consiste en integrar la información procedente de las diferentes líneas de ingeniería inversa definidas en la etapa anterior. Como alternativa, se puede realizar una transformación de la representación conceptual obtenida a una propuesta IWDM con soporte RIA, si con ello se obtiene alguna ventaja en la generación de código final.

3. Ingeniería directa dirigida por modelos a RIA. A partir de la representación conceptual refinada y de la información adicional extraída por medio del proceso de minería de datos, se lleva a cabo la modernización de la aplicación web heredada. Esta modernización consiste en la aplicación de patrones RIA previamente especificados en la representación conceptual utilizada. Esta aplicación se realiza mediante técnicas de composición de modelos de manera que no se altera la representación conceptual obtenida de la aplicación web heredada, permitiendo la aplicación de diferentes procesos de modernización.

Dentro del ámbito de este trabajo, se ha realizado una instanciación de este marco para definir un proceso de modernización caracterizado por:

- En la etapa de ingeniería inversa se realizan solamente análisis estáticos en el lado del servidor.
- En la etapa de reestructuración se realiza una transformación de la representación conceptual definida a WebML.
- En la etapa de ingeniería directa se realiza la composición del modelo obtenido con los modelos de los patrones RIA seleccionados.

Dentro de este proceso, este trabajo se ha centrado fundamentalmente en la especificación completa del proceso de ingeniería inversa dirigida por modelos en el lado del servidor. Adicionalmente, se han realizado, y se presentan, trabajos dentro de las otras dos etapas fundamentales del proceso.

1.2.5.2. Ingeniería inversa dirigida por modelos

La definición de un proceso de ingeniería inversa de aplicaciones web heredadas requiere una notación objetivo en la que especificar el resultado del proceso. En este caso, el dominio de aplicaciones web heredadas se circunscribe a las aplicaciones desarrolladas con frameworks de aplicaciones web dentro de la plataforma Java Enterprise Edition y basados en el patrón arquitectónico Modelo-Vista-Controlador (MVC). Esta notación objetivo debe ser lo suficientemente conceptual para poder representar aplicaciones generadas con distintos frameworks y lo suficientemente específica para organizarse en torno a los componentes fundamentales definidos por frameworks basados en el patrón MVC.

Con el fin de mantener la propuesta dentro del área de IDM, esta notación se ha definido mediante un metamodelo conforme al metametamodelo ecore (Rodríguez-Echeverría *et al.* 2013a). En su definición, se han utilizado las herramientas definidas dentro del proyecto *Eclipse Modeling Framework*, ya que, dentro de los objetivos de este trabajo se encuentra el de usar tecnologías y herramientas ya existentes. Este metamodelo recibe el nombre de MIGRARIA MVC y se ha organizado en base a los tres componentes fundamentales del patrón MVC: Modelo, Vista y Controlador. La Figura 1.5 muestra un extracto de este metamodelo, centrándose en los elementos fundamentales definidos para cada uno de los componentes MVC. El paquete Modelo proporciona elementos para la representación de objetos de datos, sus atributos, sus relaciones y las operaciones definidas sobre los mismos. El paquete Vista permite representar páginas, como contenedores principales, y objetos de presentación y peticiones, como contenidos principales. Los objetos de presentación, básicamente, contienen un conjunto de atributos, pueden indicar entrada o salida de datos, y pueden ser presentados individualmente o en colecciones. Mientras que las peticiones están caracterizadas por sus parámetros y la ruta, y definen puntos de conexión con los elementos del controlador mediante la asociación petición-manejador. El paquete Controlador proporciona elementos para representar: manejadores de petición, denominados *ControlFlow*; los mapeos definidos entre objetos de presentación y objetos de datos; la respuesta que puede definir una relación con el elemento página objetivo; y la secuencia de llamadas a operaciones realizadas para ejecutar la acción requerida o para obtener los datos pedidos.

En cuanto al proceso de ingeniería inversa (Rodríguez-Echeverría *et al.* 2012a;b; 2013a), se ha definido un proceso dirigido por modelos específico para el dominio y el objetivo de la propuesta. Métodos, técnicas y herramientas propias de la IDM se han utilizado en su definición para hacerlo sistemático, replicable y reutilizable. La Figura 1.6 describe los pasos principales del proceso definido. Este proceso, como se puede apreciar, se organiza como una secuencia de dos pasos:

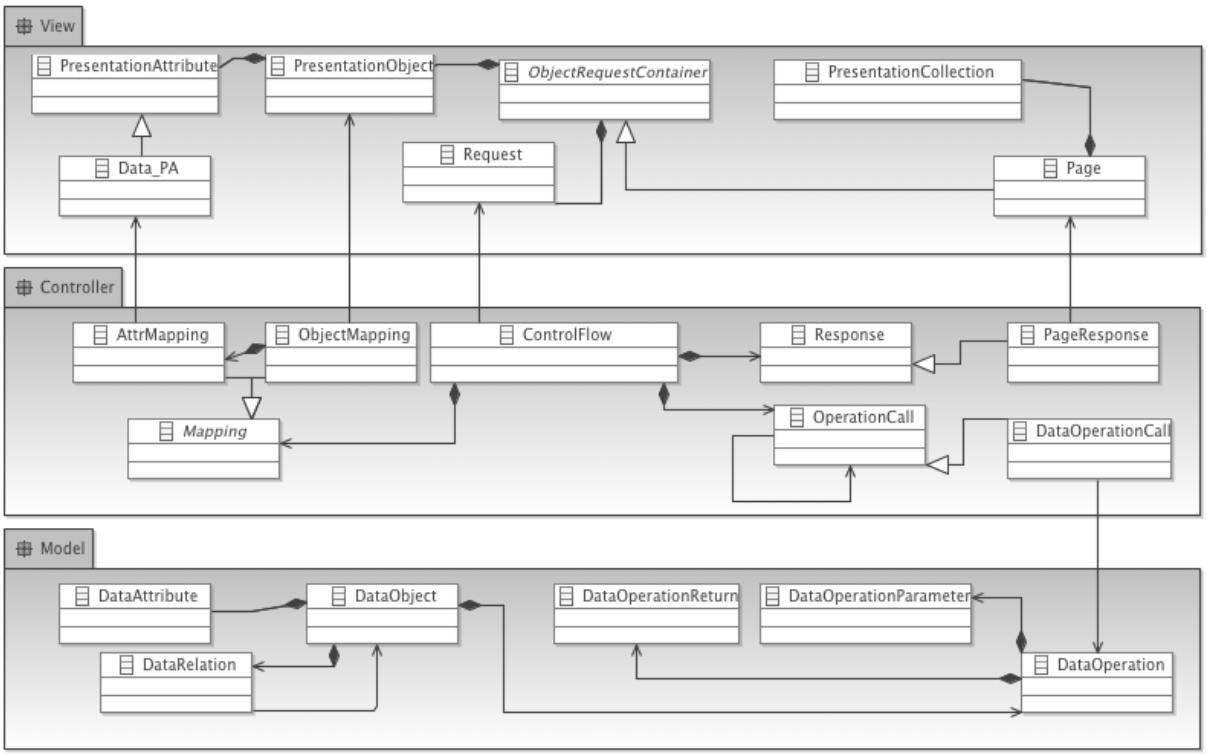


Figura 1.5: Extracto del metamodelo MIGRARIA MVC

1. Extracción de los modelos dependientes de la tecnología. En primer lugar, modelos de representación de los artefactos software que componen la aplicación web heredada son directamente extraídos por medio de transformaciones texto a modelo. Estas transformaciones están definidas tomando como origen las gramáticas de los lenguajes de programación utilizados y como destino los metamodelos de estas gramáticas. El objetivo de este primer paso es proporcionar una representación completa de la aplicación web heredada en un formato que pueda ser procesado uniformemente mediante técnicas y herramientas propias de la IDM. En resumen, pasar de la heterogeneidad del mundo de los lenguajes de programación a la homogeneidad de los modelos. De esta manera, técnicas propias de la ingeniería inversa, como los análisis estáticos, pueden definirse como consultas o transformaciones de modelos.

2. Generación de un modelo MVC conceptual. A partir de los modelos dependientes de la tecnología derivados del paso anterior se genera un modelo MIGRARIA MVC que proporciona una visión integradora de la aplicación web heredada organizada en base a los tres componentes fundamentales del patrón MVC: Modelo, Vista y Controlador. Esta representación conceptual permite independizarse de la tecnología de implementación.

1.2.5.3. Estrategias de generación RIA

Estos trabajos forman parte de las líneas de trabajo seguidas en la etapa de ingeniería directa del proceso marco definido por MIGRARIA. La misión de estos trabajos ha consistido fundamentalmente

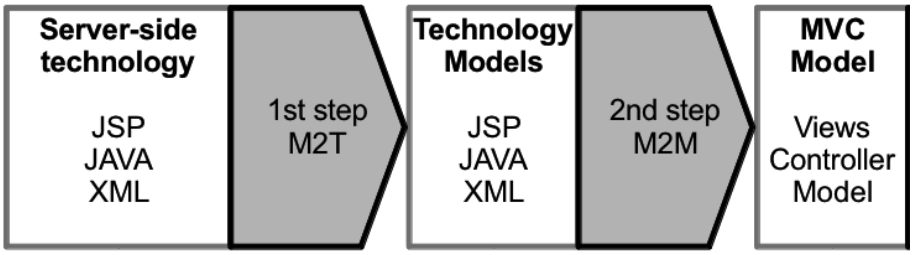


Figura 1.6: Resumen del Proceso de Ingeniería Inversa

en evaluar posibles alternativas en la consecución del objetivo principal del proyecto. Estos trabajos son presentados dentro de esta memoria con fines ilustrativos únicamente, de manera que se pueda obtener una visión del proceso de modernización en su totalidad. No se pretende, sin embargo, realizar una exposición detallada de los mismos.

Dentro del trabajo desarrollado en esta tesis, se ha seguido como estrategia principal la generación de la especificación del sistema RIA final en una propuesta IWDM que cuente con soporte RIA. De manera que, a partir de esta especificación, se define un proceso de enriquecimiento basado en la aplicación de patrones RIA mediante técnicas de composición de modelos. Esta estrategia se alinea con el objetivo de reutilizar técnicas y herramientas ya existentes en la definición y aplicación del proceso. Y ha conllevado la definición de los siguientes pasos fundamentales:

1. La generación de un modelo de la aplicación heredada en WebML a partir del modelo MIGRARIA MVC (Rodríguez-Echeverría *et al.* 2013a).

2. La especificación de patrones de sincronización RIA en WebML (Rodríguez-Echeverría *et al.* 2010; Conejero *et al.* 2011).

3. La selección de patrones RIA en un escenario de modernización (Conejero *et al.* 2013).

4. Y, finalmente, la aplicación de los patrones RIA elegidos sobre el modelo WebML de la aplicación heredada de forma transparente a la misma mediante una propuesta orientada a aspectos (Rodríguez-Echeverría *et al.* 2010; Conejero *et al.* 2011).

Adicionalmente, con el fin de evaluar la notación MIGRARIA MVC como origen para la generación directa de código, se presenta una estrategia para la generación de la capa de conexión mediante la definición de una API[12] REST (Fielding 2000) a partir del modelo MIGRARIA MVC de la aplicación heredada (Rodríguez-Echeverría *et al.* 2013b).

1.3. Estructura de la Tesis

A continuación se presenta un breve resumen de la estructura de esta memoria de tesis para orientar al lector, introduciendo el objetivo principal y contenidos de cada uno de los capítulos de la misma.

[12] Application Programming Interface

Esta memoria de tesis está organizada en las siguientes tres partes fundamentales:

- **Parte I: Introducción y Antecendentes**. En esta parte, en primer lugar, se introduce la motivación del trabajo, su alcance y objetivos. Posteriormente, se realiza un repaso exhaustivo, por un lado, de las principales propuestas de Ingeniería Web Dirigida por Modelos que han sido extendidas para dar soporte al desarrollo de aplicaciones ricas de internet y, por otro lado, se realiza una contextualización a la modernización de aplicaciones y una revisión de las principales propuestas de ingeniería inversa de aplicaciones web. Esta parte está formada por los tres primeros capítulos de la memoria.

- **Parte II: Propuesta**. Esta parte contiene la descripción del trabajo desarrollado y de las principales contribuciones de esta tesis. Primero se presenta el marco de procesos de modernización de aplicaciones web heredadas a aplicaciones ricas de internet propuesto. Después se describe en detalle el proceso de ingeniería inversa dirigido por modelos de aplicaciones web definido dentro de este marco y su aplicación en un caso de estudio. Y, finalmente, se exponen diferentes estrategias seguidas para la generación de una aplicación rica de internet a partir de la especificación conceptual de la aplicación web heredada obtenida como resultado del proceso de ingeniería inversa. Esta parte está formada por los capítulos 4, 5, 6, 7 y 8 de la memoria.

- **Parte III: Conclusión**. Esta última parte está dedicada a razonar y discutir las diferentes partes de la propuesta presentada y su adecuación al problema planteado. Se exponen, finalmente, las principales conclusiones extraídas del desarrollo de este trabajo, las contribuciones principales y las publicaciones realizadas a lo largo del mismo. Esta parte está formada por los dos últimos capítulos de la memoria.

De forma más detallada, esta memoria de tesis esta compuesta de once capítulos, cuya descripción se relaciona a continuación:

- **Capítulo 1: Introducción**. En este capítulo, por un lado, se presenta el problema a resolver y su relevancia para motivar el desarrollo de este trabajo de tesis y, por otro lado, se esboza la solución propuesta y sus principales objetivos. Finalmente, se presenta la estructura de esta memoria de tesis.

- **Capítulo 2: Desarrollo Dirigido por Modelos de Aplicaciones de Internet Ricas**. Este capítulo repasa las principales aproximaciones de desarrollo dirigido por modelos de aplicaciones ricas de internet propuestas dentro de la comunidad de Ingeniería Web. Este repaso se plantea desde dos puntos de vista diferentes: las características RIA soportadas por cada propuesta y las fases del ciclo de vida contempladas en cada propuesta.

- **Capítulo 3: Modernización de Aplicaciones Web**. Este capítulo introduce y sitúa el concepto de modernización software, principalmente desde una perspectiva dirigida por modelos, por un lado,

mientras que, por otro, revisa las principales propuestas de ingeniería inversa de aplicaciones web. De esta manera, se contextualiza el trabajo desarrollado dentro de esta tesis dentro del ámbito de la modernización software desde el punto de vista de la ingeniería dirigida por modelos.

- **Capítulo 4: Proceso de Modernización**. Este capítulo comienza presentando el marco para la definición de procesos de migración de aplicaciones web heredadas a aplicaciones ricas de internet. Y, posteriormente, plantea un proceso de modernización concreto dentro de este marco. Dicho proceso es el que se desarrollará en los capítulos posteriores, teniendo en cuenta que la misión fundamental de esta tesis es definir el proceso de ingeniería inversa.

- **Capítulo 5: Representación Conceptual**. Este capítulo explica con detalle la notación definida en el contexto de este trabajo, denominada MIGRARIA MVC. En este capítulo, se describe, ilustra y razona la notación propuesta para los diferentes componentes marcados por el patrón MVC.

- **Capítulo 6: Generación de un Modelo Conceptual**. Este capítulo presenta el proceso de ingeniería inversa definido como una cadena de transformaciones. Las reglas de transformación se presentan de forma conceptual. Sin embargo, es necesario plantear un contexto tecnológico posible para poder ilustrarlas. Por lo tanto, este capítulo usa el contexto tecnológico base utilizado en este trabajo, cuyo componente principal es el framework de aplicaciones web Struts 1. En primer lugar, se exponen los artefactos, herramientas y procesos definidos para generar modelos dependientes de la tecnología a partir de los diferentes artefactos software de la aplicación web heredada. Posteriormente, se detallan y ejemplifican las reglas de transformación que permiten generar un modelo conforme al metamodelo MIGRARIA MVC a partir de los modelos generados en el paso previo.

- **Capítulo 7: Caso de estudio**. Este capítulo expone el desarrollo de un caso de estudio que permite ilustrar todos los detalles del proceso de modernización definido. La misión fundamental de este capítulo es validar la propuesta presentada en este trabajo. De esta forma, en primer lugar, se presenta con detalle el caso de estudio utilizado. Posteriormente, se ilustra el alcance de la notación propuesta, por un lado, y la cobertura del proceso de ingeniería inversa, por otro.

- **Capítulo 8: Estrategias de Generación RIA**. Este capítulo expone dos estrategias diferentes en la consecución del objetivo principal del proceso de modernización definido. Primero, se presenta una estrategia para la generación de la especificación del sistema RIA final en una propuesta IWDM que cuente con soporte RIA. A partir de esta especificación, se define un proceso de enriquecimiento basado en la aplicación de patrones RIA mediante técnicas de composición de modelos. Posteriormente, se presenta una estrategia para la generación de la capa de conexión mediante la definición de una API REST a partir de la aplicación heredada.

- **Capítulo 9: Conclusiones**. Este capítulo, por un lado, relaciona y discute las principales contribuciones de este trabajo de tesis y, por otro lado, resume las principales conclusiones derivadas del desarrollo de este trabajo y plantea las líneas de trabajo futuro.

- **Capítulo 10: Publicaciones**. Este capítulo recoge un compendio de las principales publicaciones realizadas dentro del ámbito de esta tesis.

Capítulo 2

Desarrollo Dirigido por Modelos de Aplicaciones de Internet Ricas

2.1. Características de una Aplicación de Internet Rica

Las aplicaciones de internet ricas (RIAs) son un fenómeno reciente que ha supuesto un importante impacto en la forma de desarrollar aplicaciones Web, tanto a nivel tecnológico, arquitectónico y metodológico con un claro enfoque en la mejora de la interacción.

Este tipo de aplicaciones ha sido analizado en diferentes trabajos que han permitido identificar sus principales características y componentes. Si bien no existe hasta la fecha una definición ampliamente compartida del concepto Aplicación de Internet Rica, sí se han propuesto categorizaciones basadas en el grado de interacción permitido por las diferentes tecnologías de implementación que colocan a un extremo una aplicación de escritorio y al otro una aplicación Web. En este trabajo nos centramos en las denominadas Web RIAs, que son aplicaciones de internet ricas implementadas sobre tecnologías Web.

Este capítulo repasa las principales aproximaciones de desarrollo dirigido por modelos RIA propuestas dentro de la comunidad de Ingeniería Web. Los objetivos de este capítulo se pueden resumir en:

1. Identificar las principales características de una RIA. La correcta caracterización de las plataformas de destino del proceso de modernización comienza por identificar y entender las principales características y capacidades que introducen las RIA, así como los detalles de su soporte por cada una de las propuestas de IWDM.

2. Analizar comparativamente el soporte de estas características por parte de las principales propuestas de desarrollo. Dentro del proceso de modernización definido se aboga por el uso de herramientas y propuestas ya existentes que puedan aportar valor en alguna de las fases del proceso. En ese sentido, este estudio comparativo permite asistir en la selección de alguna de estas propuestas.

2.1.1. Metodología de investigación

La metodología de investigación seguida para esta parte del trabajo ha consistido en el estudio de las principales características RIA basado en un corpus de artículos que exploran las diferentes propuestas de desarrollo RIA propuestas dentro del dominio académico. Este estudio se ha realizado en tres pasos consecutivos: (1) conformar un corpus suficiente de publicaciones relevantes relacionadas con el desarrollo y especificación de RIAs; (2) extraer el conjunto de características RIA presentado (y cubierto por) cada una de las aproximaciones; y (3) presentar una lista uniforme de características RIA que permita la comparación de las diferentes propuestas.

El primer paso consistió en reunir un corpus suficientemente extenso (más de 70 artículos) de publicaciones relevantes relacionadas con el desarrollo RIA mediante la exploración de las principales conferencias, revistas y libros de la comunidad de Ingeniería Web. Una de las primeras decisiones a tomar consistió en definir el alcance de este estudio mediante el establecimiento de límites temporales históricos y de dominios de investigación. Con respecto a este último, se puede argumentar que otras comunidades de investigación relacionadas con la ingeniería del software debían haber sido consideradas; sin embargo, se decidió acotarlo al ámbito de la Ingeniería Web, primero, porque el enfoque del trabajo son las denominadas RIAs Web y, segundo, para poder simplificar la elaboración de un conjunto común y coherente de características RIA. Con respecto a la limitación temporal, como indica la figura 2.1, se considera el trabajo (Preciado *et al.* 2005) como el hito que marca el inicio de la evolución y extensión de las principales propuestas de Desarrollo Dirigido por Modelos de Aplicaciones Web. En este sentido, sólo se han considerado trabajos publicados desde el 2005.

Con el objetivo de elaborar una lista de las principales características RIA consideradas por las diferentes propuestas estudiadas (11 aproximaciones diferentes), se ha llevado a cabo un profundo estudio de cada una de estas propuestas. Para cada una se han revisado sus publicaciones principales en orden cronológico y se ha estudiado la evolución de las extensiones propuestas para soportar la especificación de una característica RIA determinada.

Aunque las diferentes propuestas estudiadas comparten los mismos objetivos, entre ellas presentan un alto grado de heterogeneidad en los puntos de vista seguidos para cada solución, el subconjunto de características RIA a cubrir, el grado de cobertura (o la granularidad) de cada característica RIA, el nivel de abstracción de la solución aportada, las notaciones usadas y sus semánticas, las herramientas de soporte, las fases del ciclo de vida de desarrollo consideradas, etc. En este contexto, con el objetivo de proporcionar una colección uniforme de características RIA, se ha propuesto su especificación desde un punto de vista conceptual. De esta manera, todas las características RIA que conforman la colección pueden ser consideradas y especificadas de manera muy diversa por parte de las diferentes propuestas.

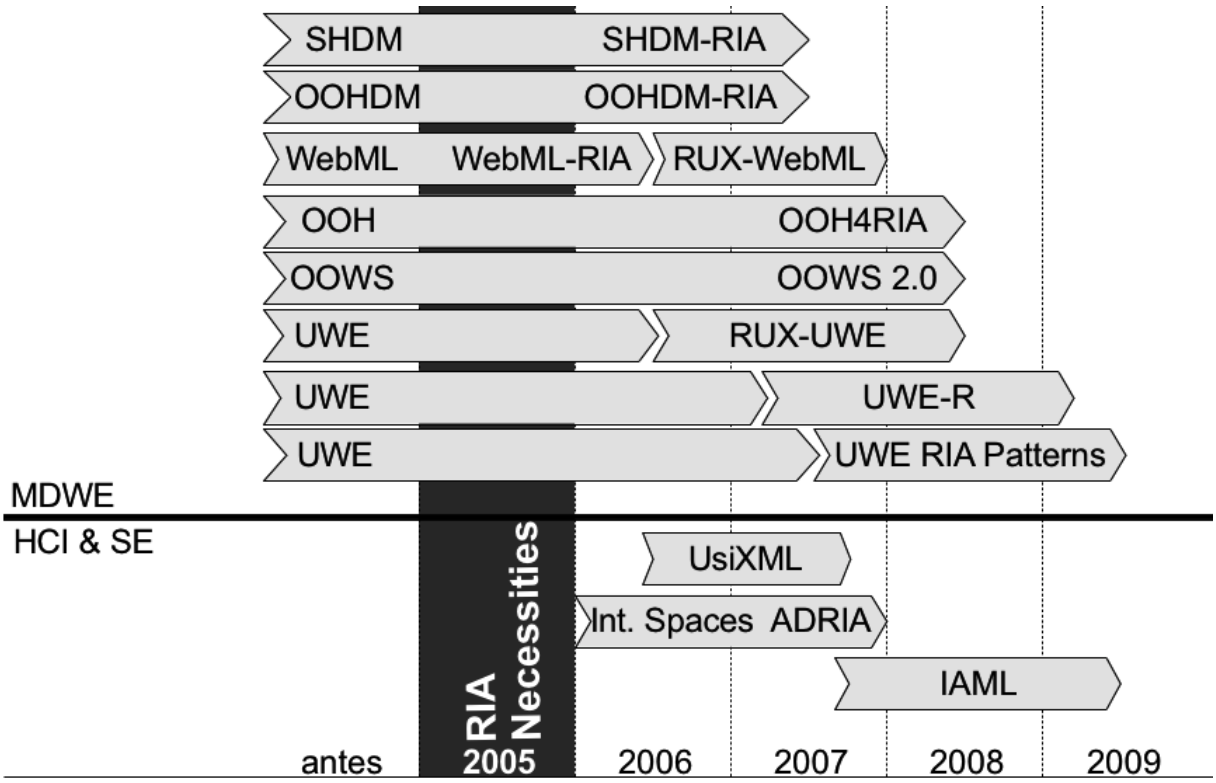


Figura 2.1: Línea de tiempo de las propuestas RIA

2.1.2. Características de las RIAs

Como ya se ha mencionado previamente, el objetivo principal de este estudio consiste en obtener la relación de todas las características RIA soportadas por las diferentes propuestas para construir un glosario común desde un punto de vista conceptual que permita comparar el grado de cobertura de las distintas propuestas. Por lo tanto, se ha realizado una recogida y anotación de características a un alto nivel de abstracción, evitando cuestiones tecnológicas de bajo nivel, e intentando proporcionar una visión unificada de las mismas. A continuación, como resultado de este estudio, se muestra la relación de características RIA que se ha elaborado.

- RF01. Almacenamiento de datos en cliente. Esta característica se refiere a la capacidad del cliente para almacenar datos tanto de forma volátil como permanente. El almacenamiento de datos de forma persistente en cliente se ha convertido en una de las características más demandadas y constituye una de las mayores aportaciones del nuevo estándar HTML5.

- RF02. Múltiples fuentes de datos de naturaleza heterogénea. Las RIAs actuales se conectan a menudo a múltiples proveedores de datos (bases de datos, servicios web, APIs web, etc.) y usan diferentes formatos de datos (datasets, XML[1], JSON[2], etc.).

[1] http://www.w3.org/TR/REC-xml/
[2] http://www.json.org/

- RF03. Soporte multimedia y de animación. Esta característica se refiere a la capacidad del cliente para gestionar animaciones complejas y contenido multimedia correctamente con el objetivo de mejorar la interacción de usuario.

- RF04. Ejecución de lógica en el lado cliente. Esta característica se refiere a la capacidad del cliente para ejecutar parte de la lógica relacionada con la presentación y con el negocio dentro de su propio sistema de ejecución. Junto con la RF01 esta característica puede suponer una considerable reducción de las peticiones a servidor y mejorar la experiencia de usuario y su productividad.

- RF05. Concurrencia y multihilo. Esta característica se refiere a la capacidad del cliente para lanzar simultáneamente diferentes hilos de ejecución. Su ventaja más conocida es permitir mantener una interfaz que responde mientras se realiza una petición al servidor.

- RF06. Interfaz de usuario multidispositivo. Esta característica se refiere a la capacidad de una RIA de ser accedida a través de un amplio rango de terminales heterogéneos. En los últimos años, las RIAs se han extendido al mercado móvil y se han convertido en una de las opciones preferidas para el despliegue de aplicaciones gracias a su mayor independencia de la tecnología del terminal.

- RF07. Paradigma Página Única (Single-Page) o refresco parcial de la página. Esta característica se refiere a la capacidad del cliente a presentar una interfaz de usuario similar al de las aplicaciones de escritorio evitando el ciclo petición-espera-respuesta característico de los clientes Web. Esta característica puede considerarse una consecuencia de las características RF04 y RF12.

- RF08. Componentización de la interfaz de usuario. Esta característica se refiere a la capacidad del cliente para componer su interfaz a partir de una gran variedad de complejos componentes interactivos. Supone la ejecución de lógica de presentación en el cliente.

- RF09. Interacción rica de usuario. Esta característica se refiere a la capacidad del cliente para mejorar la interacción con el usuario mediante la definición explícita de orquestaciones de componentes de interfaz de usuario ricos.

- RF10. Control del sistema de ejecución del cliente. Esta característica se refiere a la capacidad del cliente para usar y controlar parcialmente la funcionalidad de su sistema de ejecución, cambiando su comportamiento por defecto, e.g., el botón Atrás de un navegador web.

- RF11. Inicio de la comunicación en el lado servidor (modelo push). Esta característica se refiere a la capacidad de una RIA a dejar atrás el modelo de comunicación petición-respuesta típico de las aplicaciones Web. En una RIA ambos lados (cliente y servidor) pueden erigirse en iniciadores del proceso de comunicación.

- RF12. Comunicación asíncrona. Esta característica se refiere a la capacidad del cliente para mandar peticiones al servidor sin quedarse bloqueado hasta que se recibe una respuesta. Una RIA puede seguir funcionando normalmente y manejar la respuesta cuando sea necesario.

- RF13. Transferencias de conjuntos de datos del cliente al servidor. Esta característica se refiere a la capacidad del cliente a enviar una colección de datos resultado de un conjunto de acciones de una sola vez al servidor. Con el fin de reducir el número de peticiones al servidor, un cliente RIA puede almacenar los datos producidos por la ejecución normal del proceso y, en un momento determinado, enviar todo este conjunto de datos al servidor de una sola vez.

- RF14. Sincronización entre el cliente y el servidor. Esta característica se refiere a la capacidad de una RIA a mantener la consistencia de datos entre diferentes extremos de la aplicación. Esta es una característica de mayor nivel de abstracción que puede descomponerse en las secuencias de comunicación entre extremos. Esta característica puede verse como una combinación de RF01, RF04 y RF13, y una política de sincronización completa

- RF15. Modo desconectado. Esta característica se refiere a la capacidad del cliente a cambiar de forma transparente su modo de operación entre el modo aislado, sin conexión en vivo con el servidor, y modo conectado, con conexión con el servidor. Se trata también de una característica de más alto nivel que puede verse como la configuración concreta de otras características RIA: RF01, RF04, RF07 y RF14.

2.1.3. Propuestas de desarrollo RIA

Dado que el objetivo es obtener una colección común de las características de Aplicaciones Ricas de Internet, entre las diferentes propuestas de desarrollo dirigido por modelos, se han estudiado, por un lado, las que presentaban un grado de madurez mayor y, por otro lado, las más recientes que presentasen cierta relevancia. En total, se han analizado 11 propuestas distintas: WebML-RIA, OOHDM-RIA, SHDM RIA, OOH4RIA, UWE para RIA, RUX-Method, UWE-R, UsiXML, OOWS 2.0, ADRIA e IAML.

A continuación se realiza una breve descripción de cada una de las propuestas estudiadas siguiendo el orden cronológico de aparición.

2.1.3.1. WebML-RIA

WebML-RIA (Fraternali *et al.* 2010), es la primera propuesta que estudia cómo extender su representación conceptual específicamente diseñada para el desarrollo de aplicaciones web intensivas de datos, WebML en (Ceri *et al.* 2000), para poder soportar la especificación de características RIA. Esta propuesta ha ido evolucionando durante los últimos años, presentando múltiples trabajos relacionados (Bozzon *et al.* 2006; Comai & Carughi 2007; Carughi *et al.* 2007) y posicionándose como una de las propuestas más maduras hasta ahora.

Las principales extensiones del lenguaje incluyen: el almacenamiento (RF01) y la ejecución de lógica (RF04) en el lado cliente; la especificación de un sistema de comunicación basado en eventos, que permite escenarios de comunicación asíncrona (RF12) y notificaciones push (RF11); la definición de páginas con refresco parcial (RF07); y la detallada especificación del modelo de comportamiento para describir la complejidad de las interacciones RIA (RF09). Con todas estas nuevas primitivas junto con la sincronización cliente-servidor basada en transferencias de conjuntos de datos, se puede definir un modo de operación desconectado (RF15) para la aplicación. También permite especificar la conexión a servicios web mediante unidades que procesan y generan XML (RF02). Adicionalmente, en (Bozzon *et al.* 2009; Brambilla *et al.* 2009) presentan extensiones relacionadas con patrones RIA como por ejemplo: *dynamic tooltips on page data* o *field data autocompletion*.

WebML-RIA se centra en el diseño conceptual de la aplicación mediante la especificación de versiones extendidas de su modelo de datos y de hipertexto, que especifican fundamentalmente la estructura de la aplicación, punto de vista estático. Para la especificación del comportamiento, punto de vista dinámico, proponen una extensión del modelo de hipertexto para el soporte de eventos y, además, una extensión del modelo de comportamiento que permite especificar las interacciones entre los elementos que conforman las páginas en el modelo de hipertexto. Por otro lado, el diseño de la presentación se relega a la fase de generación de código, que genera una primera versión basada en plantillas. Por lo tanto, es necesario su tratamiento posterior para obtener una presentación de calidad.

Esta propuesta cuenta con una herramienta profesional para el desarrollo de aplicaciones (WebRatio), que permite disponer de un entorno diseñado para el lenguaje WebML y simplificar las tareas de desarrollo y generación final de código. Para el desarrollo de RIAs se propuso una extensión para generar código en la plataforma RIA OpenLazslo.

Dado que la descripción final de la presentación (interfaz concreto y final) se relega a la fase de generación de código, la propuesta presenta algunas limitaciones en la capacidad de especificar interacciones RIA (RF09) o en la definición y gestión de elementos multimedia (RF03). La definición de la interfaz se especifica solamente a nivel abstracto, centrándose en la composición que establece un comportamiento implícito y predefinido de los elementos.

2.1.3.2. OOHDM – RIA

OOHDM-RIA (Rossi *et al.* 2007) es una extensión ligera de una propuesta conceptual para el desarrollo de aplicaciones Web tradicionales denominada OOHDM (Schwabe *et al.* 1996). Esta extensión se centra en resolver los nuevos requisitos introducidos por la RIA en el diseño de la Interfaz Abstracta. La Vista Abstracta de Datos (Abstract Data View, ADV) es el concepto usado por OOHDM para especificar la estructura y el comportamiento de los objetos de interfaz. Este concepto ha sido extendido para poder representar comportamientos más ricos de interfaz característicos de las RIAs (RF09) mediante el

uso de escenarios de comunicación asíncrona bajo demanda (RF12). OOHDM-RIA también proporciona soporte al paradigma single-page y a la composición basada en componentes ricos de interfaz (RF08).

Un enfoque interesante de esta propuesta consiste en definir un mecanismo para especificar extensiones RIA que luego son integradas con la aplicación web mediante técnicas de *model weaving*, (Rossi *et al.* 2008a), definiendo de esta manera un mecanismo de evolución de aplicaciones web.

Hasta el momento, parece que estas extensiones han sido introducidas a nivel de diseño y no se dispone de una herramienta que dé soporte a esta propuesta. Además, parece que la propuesta se centra exclusivamente en la definición de la interfaz a un nivel abstracto sin entrar a considerar aspectos de más bajo nivel considerados, por ejemplo, en WebML. En este sentido, no especifica ningún mecanismo para dar soporte a la distribución entre cliente y servidor de parte del almacenamiento y de la lógica (RF01, RF04).

2.1.3.3. SHDM RIA

Esta aproximación (Fialho & Schwabe 2007) se define como una extensión de la propuesta de desarrollo web conceptual SHDM (Lima & Schwabe 2003), que es una propuesta dirigida por modelos con origen en la comunidad Interfaz Hombre-Máquina.

Este trabajo se centra en la especificación de animaciones, transiciones de interfaz (RF09), en aplicaciones web. Extiende SHDM para relacionar las animaciones con las semánticas de la aplicación definidas en los modelos SHDM. Una animación de la interfaz consiste en una secuencia de funciones de animación realizadas sobre un conjunto de componentes de interfaz (widgets). Una interfaz se define como una composición de widgets (RF03). Los autores de este trabajo señalan que definir una secuencia correcta de animación es de gran importancia porque influye directamente en la forma en la que el usuario interpreta la transición. Con el fin de determinar la mejor secuencia se utiliza una estructura de animación retórica, basada en la Teoría de la Estructura Retórica (Mann & Thompson 1988), que permite definir el rol de comunicación de cada animación usada durante la transición, resaltando las más importantes.

Esta propuesta proporciona todos los artefactos necesarios para especificar las transiciones animadas en tiempo de diseño. Sin embargo, no parece que tengan una herramienta que dé soporte a la propuesta. En conclusión, es una propuesta muy interesante, sobre todo, desde el punto de vista de Interfaz Hombre-Máquina, pero parece un trabajo no suficientemente maduro.

2.1.3.4. RUX-Method (2007)

Rich User eXperience-Method (RUX-Method), (Linaje *et al.* 2007), es un método dirigido por modelos que da soporte al diseño de interfaces de usuario web multimedia (RF03), multimodal y multidispositivo (RF06), siguiendo el paradigma single-page (RF07).

RUX ha sido específicamente diseñado para el desarrollo de RIAs y se centra fundamentalmente en el soporte y descripción de interacciones ricas. Define tres niveles de descripción de la interfaz (abstracto, concreto y final) en los que se puede especificar las interacciones (RF09) y el comportamiento temporal. RUX da soporte a gran parte de las características RIA relacionadas con la interfaz de usuario. Además, su modelo conceptual puede ser definido encima (como fase final) de otras metodologías de desarrollo web dirigidas por modelos, como por ejemplo UWE o WebML, mediante la especificación de una capa de conexión. Su enfoque centrado principalmente en la descripción de la interfaz de usuario le permite aprovechar las descripciones de datos y de lógica de negocio de otras propuestas.

La herramienta profesional RUX-Tool[3] se encarga de dar soporte al desarrollo de RIAs mediante el método RUX. Esta herramienta permite generar el código final en diferentes plataformas tecnológicas RIA, entre otras: AJAX, OpenLazslo y FLEX.

Entre sus principales aportaciones, se puede destacar su propuesta de separación de la especificación de la interfaz de usuario en tres niveles diferentes, proporcionando un método sistemático de transformación de la especificación del sistema desde el espacio del problema al espacio de la solución. Adicionalmente, cabe señalar que las recomendaciones de la especificación tecnológica WAI ARIA[4] del W3C se han considerado en la definición de la biblioteca de componentes RUX (RF08).

2.1.3.5. UsiXML

UsiXML (Vanderdonckt *et al.* 2004) es una interesante propuesta que proviene de la comunidad Interfaz Hombre-Máquina. Se trata de un lenguaje basado en XML enfocado en la especificación de aspectos de presentación. Básicamente, diferentes versiones de interfaces de usuario se pueden generar mediante la composición de widgets (RF08) que pueden ser adaptados finalmente a entornos heterogéneos (RF06). Esta propuesta sugiere seguir el ciclo de desarrollo definido por la especificación MDA de la OMG[5] para el desarrollo de RIAs mediante la definición de transformaciones entre modelos desde los modelos independientes de la computación (CIM) y los modelos específicos de plataforma (PSM). También, se basa en el diseño multinivel de la interfaz de usuario, según el framework Cameleon.

Esta propuesta se centra fundamentalmente en temas relacionados con el nivel de presentación abstracto. No se tiene constancia de la existencia de una herramienta que dé soporte a esta propuesta.

2.1.3.6. OOH4RIA

OOH4RIA, (Meliá *et al.* 2008), se ha desarrollado como una extensión de una metodología orientada a objetos para el desarrollo sistemático de aplicaciones Web tradicionales llamada OOH, (Cachero Castro 2003). Esta extensión introduce nuevos modelos y transformaciones para dar soporte a características

[3] http://www.homeria.com
[4] http://www.w3.org/WAI/intro/aria
[5] OMG Model Driven Architecture. http://www.omg.org/mda/

RIA como, por ejemplo, capacidades de interacción ricas (RF09) o el paradigma single-page (RF07) mediante la composición de la interfaz de usuario con widgets (RF08). En concreto, esta propuesta permite al desarrollador relacionar el comportamiento de widgets (basados en GWT[6]) mediante una coreografía basada en eventos y la especificación funcional de la navegación con el objetivo de definir la comunicación entre el cliente y el servidor. Además, da soporte a cuestiones relacionadas con la arquitectura del sistema y con la plataforma tecnológica: modo de operación desconectado (RF15), almacenamiento de datos en el lado cliente (RF01), lógica de negocio en el lado cliente (RF04), manejo de eventos en el cliente (RF09), control del navegador (RF10), etc. Esta propuesta no establece un conjunto predefinido de decisiones a nivel de arquitectura o de plataforma tecnológica, sino que permite al desarrollador seleccionar la mejor opción para cada sistema. Algunos trabajos relacionados con esta propuesta (Garrigós *et al.* 2009) presentan una extensión para adaptar la interfaz de una RIA a diferentes dispositivos (RF06).

OOH4RIA proporciona un proceso completo de desarrollo dirigido por modelos que termina con la generación del código de la RIA (parte cliente y parte servidora). La herramienta OIDE[7] da soporte a esta propuesta, permitiendo generar código en las plataformas GWT y Microsoft Silverlight.

Los aspectos de presentación introducidos en la metodología son dependientes de la tecnología, pero la propuesta puede ser reutilizada en dominios con otras tecnologías. Aunque la propuesta no cubre todas las características RIA consideradas en el estudio, sí define claramente un proceso completo de desarrollo y, en particular, las relaciones entre los diferentes modelos y cuáles pueden ser obtenidos automáticamente mediante transformaciones modelo a modelo. Uno de los puntos clave de esta propuesta es la separación completa de la especificación de cuestiones de la arquitectura del sistema y de la plataforma tecnológica, permitiendo un proceso de desarrollo más flexible y la especificación del sistema en el espacio de la solución.

2.1.3.7. OOWS 2.0

OOWS 2.0 (Valverde 2010) es una evolución de OOWS, (Pastor *et al.* 2006), enfocada en el modelado de la interacción en el desarrollo de RIAs. Básicamente, se define una nueva capa conceptual dentro del proceso de desarrollo software dirigido por modelos para la construcción de interfaces RIA encima de un Sistema de Información especificado conceptualmente en base a OO-Method (capa funcional) y OOWS (capa web). Los autores consideran que los escenarios de interacción de usuario en las RIAs son mucho más complejos que en el caso de aplicaciones web; por lo tanto, entienden que se deben definir nuevos conceptos con la suficiente expresividad para especificar esta complejidad añadida. Uno de los objetivos principales de esta propuesta es permanecer independiente de la tecnología RIA final (RF06). En este sentido, basados también en trabajos de la comunidad de Interacción Hombre-Máquina, proponen la descomposición del modelado de la interacción en dos modelos diferentes:

[6]Google Web Toolkit
[7]OOH4RIA Integrated Development Environment

- el modelo abstracto de interacción, que permite especificar la interacción de usuario de forma independiente de la tecnología,
- y el modelo concreto de interacción, que lleva al plano tecnológico el anterior mediante la definición de un metamodelo para cada una de las plataformas tecnológicas de destino. En este modelo se define:
  - el modelado de los widgets (RF08) de cualquier plataforma RIA,
  - la especificación de las relaciones entre los widgets mediante eventos de interacción (RF09),
  - la especificación de la interfaz RIA (RF07) estáticamente (composición) y dinámicamente (comportamiento),
  - y la obtención y almacenamiento de datos desde el servidor (RF01) para minimizar las peticiones.

Esta propuesta proporciona un compilador de modelos que permite generar una interfaz RIA y una fachada a la lógica de negocio en tecnología AJAX y REST, permitiendo su integración con el sistema de información subyacente.

2.1.3.8. ADRIA – Abstract Design of Rich Internet Applications

Esta propuesta está basada en los espacios de interacción (Dolog & Stage 2007), un método de abstracción para el diseño basado en modelos de interfaces de usuario. Dentro de la comunidad de Interacción Hombre-Máquina, los espacios de interacción definen una técnica de diseño dirigida por tareas para derivar modelos de interfaz de usuario. Esta propuesta presenta la aplicación del método espacios de interacción al desarrollo de RIAs, (Dolog & Stage 2009).

A lo largo de todo el proceso de desarrollo propuesto, esta aproximación distingue entre parte cliente y servidora, alojando diferentes objetos en cada una según los espacios de interacción identificados (RF01, RF04). Los métodos de los objetos en la parte cliente (RF08) generan la presentación en el lado cliente (RF07), mientras que los métodos de la parte servidora proporcionan y procesan datos. Con el fin de definir el comportamiento de interacción (RF09) proponen el uso de guías UML[8], que son diagramas de estado que especifican la presentación concurrente de los objetos en cliente y su sincronización basada en eventos con los objetos en servidor (RF13, RF14). A su vez, la descripción del comportamiento de la interfaz se realiza mediante la definición de *splits* y *joins* de sincronización (RF05), una extensión de las guías UML. Como resultado, el modelo de comportamiento del espacio de interacción proporciona una especificación del protocolo de comunicación entre los objetos de la interfaz de usuario y los objetos en servidor mediante la especificación de la inicialización de contenido, los procesos de sincronización, la obtención de datos y la gestión de eventos externos.

[8] http://www.omg.org/spec/UML/2.0/

Esta propuesta propone un ciclo de vida parcial formado por una fase de análisis, un análisis orientado a objetos y enfocado en las tareas del usuario, y por las fases de diseño del proceso de desarrollo RIA. Según los autores, la fase de diseño está compuesta por tres actividades principales que se encargan respectivamente de la estructura, el comportamiento y la apariencia. Esta propuesta no define un método automático de generación de código, aunque proporciona un esbozo de un posible mapeo entre el modelo de comportamiento de los espacios de interacción y su implementación.

Una de sus principales contribuciones consiste en el hecho de que no se centra exclusivamente en aspectos de presentación sino que considera aspectos de comunicación, proponiendo una solución al tema de la sincronización entre cliente y servidor. El punto clave de esta propuesta es el uso homogéneo y coherente de los espacios de interacción como mecanismo de abstracción a lo largo de todo el proceso de diseño. Esta decisión permite delegar la especificación de la apariencia concreta de la interfaz de usuario hasta que la interacción de usuario ha sido completamente diseñada a nivel general.

2.1.3.9. IAML

Los autores proponen la definición de un nuevo lenguaje específico del dominio, denominado Interactive Application Modeling Language (IAML), (Wright & Dietrich 2010), que contempla un conjunto de requisitos RIA fundamentales especificado por ellos mismos en (Wright & Dietrich 2008). Este estudio recoge una lista de 59 requisitos de las RIAs a partir del estudio de 7 aplicaciones del mundo real: gmail, google calendar, google reader, google docs, last.fm, pages y facebook.

Esta propuesta se encuentra en fase de desarrollo. Por lo tanto, para este estudio sólo se han considerado características RIA para las que ya proporcionen soporte. Esta propuesta permite especificar: las operaciones (diagrama de actividad de UML), la estructura de la base de datos (diagrama de clases de UML), eventos (RF09) y condiciones de activación (reglas ECA[9]), las consultas a la base de datos, la navegación y la interfaz de usuario (RF08) parcialmente, así como la sincronización (RF09).

Esta propuesta se centra principalmente en las fases de diseño e implementación (generación de código) del ciclo de vida del desarrollo software. En este sentido, además de definir el lenguaje, en paralelo y de forma iterativa, están desarrollando una cadena de herramientas adecuada para asistir al desarrollador en el diseño e implementación de una RIA. En concreto, usan una colección de tecnologías opensource bien conocidas, como EMF para la definición del lenguaje específico de dominio, GMF para la creación de un editor gráfico y *openArchitectureWare* para la generación de código, usando la terna (PHP, HTML, JavaScript) como tecnologías de destino.

Con respecto a sus principales contribuciones, es interesante reseñar que se trata de una propuesta completamente distinta a las anteriores porque no se propone como una extensión de un trabajo anterior y, por lo tanto, no está limitada de ninguna manera por su origen. También merece la pena destacar la decisión de desarrollar en paralelo y de forma iterativa el lenguaje y la herramienta. Por otro lado, quizás

[9]Event Condition Action

como principal punto débil, se pueda discutir que la base de la propuesta es de alguna manera débil; ya que, consiste en el estudio de requisitos previamente comentados. Este estudio recoge requisitos desde un punto de vista de la interacción, que supone una mezcla de requisitos relacionados con aspectos tecnológicos, funcionales y arquitectónicos. Esta falta de formalización, proporcionando un nivel de abstracción suficiente, puede suponer la introducción de conceptos en el lenguaje diseñado que compliquen su usabilidad y aplicabilidad.

2.1.3.10. UWE y RIA Patterns

Este caso, (Koch *et al.* 2009), presenta otra propuesta de extensión de una metodología de desarrollo de aplicaciones web dirigida por modelos: UML Web Engineering. Su definición a partir de estándares es una de las características más diferenciadoras de UWE, (Kraus *et al.* 2007). Su lenguaje de modelado se define como una extensión de UML 2.0.

UWE fue definido para soportar el despliegue de componentes en el cliente, aunque no fueron concebidos inicialmente para componentes relacionados con el almacenamiento o el procesado de lógica en el cliente. Las extensiones propuestas en esta aproximación conforman un primer paso hacia el soporte de características RIA. Las características RIA seleccionadas son introducidas en la capa de presentación mediante la definición de patrones específicos que son capaces de especificar la interacción (RF09) entre componentes de presentación (RF07, RF08) mediante máquinas de estados, así como su funcionalidad. Como trabajo futuro, los autores proponían la inclusión de otras características RIA como la animación (RF03), la comunicación asíncrona entre cliente y servidor (RF12) y el procesado de lógica en el lado del cliente (RF04).

En cuanto al ciclo de vida de desarrollo, esta propuesta presenta soporte para las fases de análisis, diseño e implementación. Los autores desarrollaron la herramienta MagicUWE para dar soporte a los procesos de desarrollo con UWE y con UWE para RIA. Las transformaciones se definen mediante lenguajes de transformación estándar como QVT o ATL incorporando un perfil de extensión para la especificación de patrones RIA.

Como se puede comprobar, el principal inconveniente de esta propuesta se circunscribe a que la versión actual de este trabajo no da soporte a la gran mayoría de características RIA. Por otro lado, como aportación principal, se puede destacar la definición de los patrones RIA con un alto grado de abstracción favoreciendo su reutilización.

2.1.3.11. UWE-R

Este último caso, (Machado *et al.* 2009), representa otra aproximación distinta de extensión de la propuesta de desarrollo web UWE. UWE-R propone una extensión mucho más ambiciosa que pretende cubrir aspectos de navegación, lógica y presentación. En este sentido, nuevos conceptos han sido introducidos en el lenguaje de modelado mediante la definición de elementos que heredan su estructura

y comportamiento de los elementos originales de UWE. UWE-R utiliza estereotipos para definir gran parte de las extensiones, intentando mantener el máximo grado de compatibilidad con UWE. Ninguna metaclase de UWE se ha modificado.

Con respecto a la navegación, los autores proponen una extensión basada en la definición de tres nuevas metaclases: *RichNavigationClass*, *RichNavigationLink* and *Dialogue*. La metaclase *RichNavigationClass* pretende solucionar dos limitaciones fundamentales de las metaclases *NavigationClass* de UWE. Primero, una *NavigationClass* está unida a la página web que la contiene. Segundo, una *NavigationClass* sólo admite relaciones uno-a-uno con su respectiva clase de contenido (RF07). Por su parte, la metaclase *RichNavigationLink* permite la invocación de procesos remotos o locales (RF09), proporciona continuidad visual (retorno a la misma *RichNavigationClass*), y da apoyo a la especificación de comunicación asíncrona (RF12).

Con respecto a la presentación, básicamente proporcionan nuevos contenedores y elementos de interfaz de usuario más característicos de RIA (RF08).

Con respecto a la lógica de proceso, se centran en el despliegue y en la comunicación. Por un lado, definen estereotipos para poder especificar si un proceso se despliega en servidor o en cliente (RF04). Por otro lado, permiten una definición más detallada de la comunicación y soportan la activación de acciones sin intervención humana mediante la metaclase *AutonomousAction* que permite especificar la activación de una acción en base a un evento, un temporizador o un *callback*. Proponen la utilización de un diagrama de secuencia UML extendido (perfil) y de una extensión de la metaclase *UML Message*, denominada *ControlMessage*, para especificar la comunicación entre el cliente y el servidor. Estas extensiones permiten la introducción de conceptos de nivel de plataforma como, por ejemplo, el protocolo de comunicación (REST, SOAP, etc.), el formato de datos (JSON, XML, etc.) (RF13), o la definición de los tipos de datos (RF02).

Atendiendo al ciclo de vida de desarrollo, sólo se cubre la fase de modelado sin proporcionar ninguna herramienta que dé soporte y permita la generación del código final.

Su inconveniente principal es su inmadurez. Su mayor contribución es la propuesta de una aproximación completa de extensión de UWE para el soporte del desarrollo de RIAs.

2.2. Lecciones aprendidas

En esta sección, se exponen las principales conclusiones del estudio desde dos puntos de vista diferentes: el de las características RIA y el de las propuestas de desarrollo RIA.

La tabla 2.1 presenta una visión comparativa del soporte de características RIA por parte de cada una de las aproximaciones estudiadas. Desde el punto de vista de las características RIA, con el objetivo de realizar un análisis más estructurado, la tabla 2.1 propone una categorización de las características según las capas conceptuales del desarrollo de aplicaciones web (la capa de navegación se considera

Capa	Ref.	WebML	OOHDM	SHDM	RUX	USiXML	OOH4RIA	OOWS 2.0	ADRIA	IAML	UWE & RIA Patterns	UWE-R
Datos	RF01	✓					✓	✓	✓			✓
	RF02	✓										✓
Lógica	RF03			✓	✓			✓				
	RF04	✓					✓		✓			✓
	RF05						✓		✓			
Presentación	RF06				✓		✓	✓				
	RF07	✓	✓		✓		✓	✓	✓	✓		✓
	RF08		✓	✓	✓	✓	✓	✓		✓	✓	
	RF09	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
	RF10						✓					
Comunicación	RF11	✓										✓
	RF12	✓	✓				✓					✓
	RF13	✓					✓		✓			✓
Arquitectura	RF14	✓							✓			
	RF15	✓					✓		✓			

Tabla 2.1: Categorización y referencias de características RIA

incluida dentro de la de presentación). Es necesario resaltar que algunas de las características RIA pueden extenderse entre diferentes capas conceptuales, pero la tabla 2.1 no muestra dicha expansión. También conviene indicar que las dos últimas características RIA consideradas se han separado del resto porque describen aspectos de más alto nivel que el resto. Cada una de estas últimas puede ser especificada mediante la agregación de otras características RIA; sin embargo, al igual que otros autores (Meliá *et al.* 2010), hemos decidido hacerlas explícitas para enfatizar su relevancia en el desarrollo RIA.

Como se desprende de la tabla 2.1, la capa de presentación tiene una especial relevancia en el desarrollo RIA. Las características RF07, RF08 y RF09 son soportadas por la mayoría de las propuestas estudiadas. Estas características están muy relacionadas: RF08 y RF09 son necesarias para obtener RF07. La composición de la interfaz de usuario de una RIA como una colección de componentes de interfaz ricos (widgets) y la especificación de la interacción (comportamiento de la interfaz de usuario), normalmente mediante aproximaciones orientadas a eventos, constituyen los aspectos principales del desarrollo RIA para la mayoría de las propuestas. Este resultado está alineado con el punto de vista tecnológico que considera una RIA como una interfaz de usuario rico (semejante a una interfaz de usuario de escritorio) conectado a una capa remota de servicios.

En el otro extremo, nos encontramos con 5 características (RF02, RF05, RF10, RF11 y RF14) que son soportadas como máximo por dos propuestas. En este caso, RF02 y RF10 expresan aspectos de más bajo nivel que el resto de características y, por lo tanto, no son explícitamente soportadas en muchas de las propuestas que mantienen un nivel más conceptual. Por otro lado, RF05 puede ser tapada por RF07,

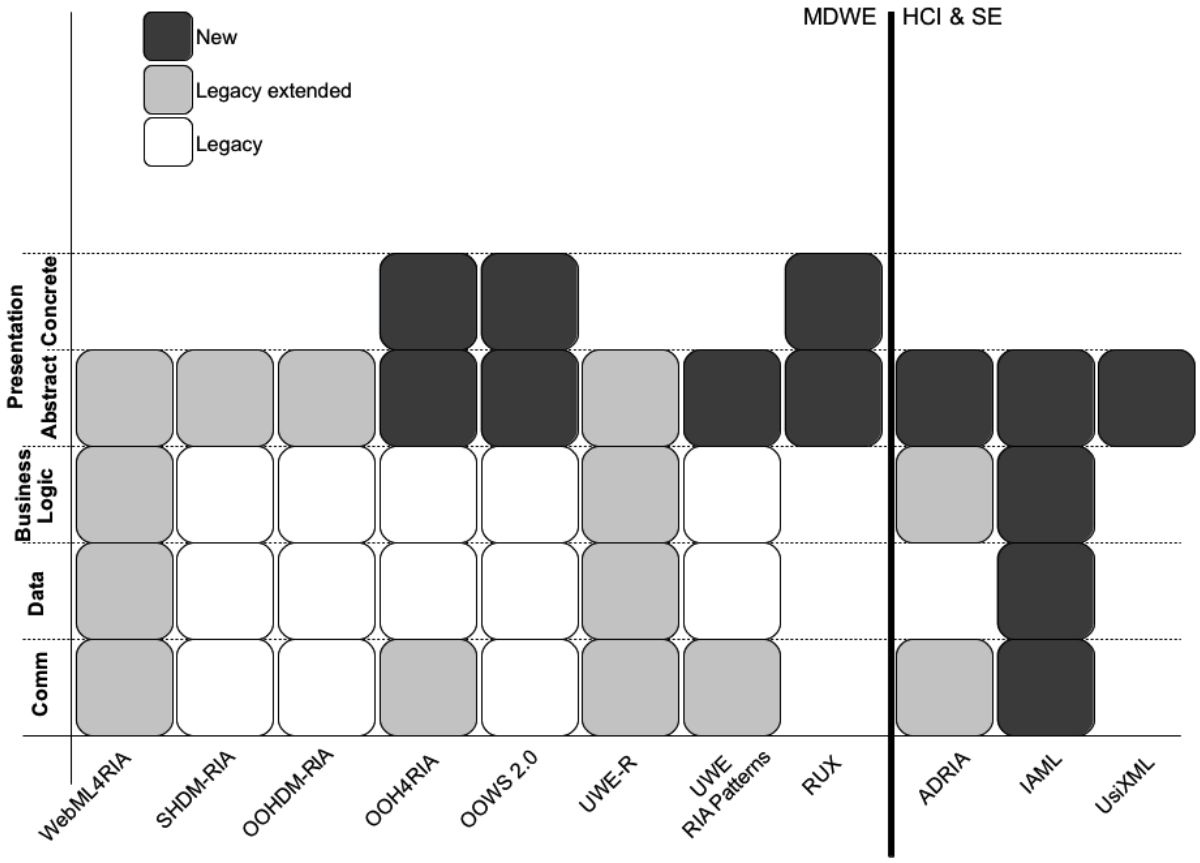


Figura 2.2: Evolución de las propuestas IWDM

puesto que RF07 necesita de RF05 de alguna manera. La misma situación ocurre con RF11 y RF12. Y, finalmente, considerar RF14 como una característica RIA puede no tener sentido para muchas de las propuestas, ya que se mantienen en un plano más conceptual. Esta última característica puede verse, por tanto, como un escenario de aplicación de otras características RIA.

Con respecto a las propuestas RIA estudiadas, se debe tener en cuenta que la mayoría de las propuestas consisten en extensiones de aproximaciones IWDM; por lo tanto, se centran en extender fundamentalmente la capa de presentación para permitir la especificación de características RIA. Atendiendo a la organización en base a las capas web conceptuales, la Figura 2.2 presenta cómo las diferentes propuestas han decidido mantener, modificar o redefinir sus propuestas de modelado para tratar las necesidades introducidas por el desarrollo de RIAs. Como se puede observar, todas las propuestas han necesitado modificar o redefinir los modelos usados para la especificación de aspectos de presentación, pero sólo tres de ellas proponen una definición concreta de la interfaz de usuario RIA.

Actualmente, ninguna de las metodologías estudiadas puede dar soporte al conjunto completo de características RIA consideradas tanto en el diseño como en la generación de código. Como señala la tabla 2.1, las propuestas WebML4RIA y OOH4RIA cubren la mayoría de las características RIA consideradas en este trabajo. Es importante señalar la posibilidad que ofrece la combinación de WebML y RUX, (Linaje *et al.* 2009), dado que proporciona gran parte de las características RIA identificadas. Son, por lo tanto, las propuestas más completas. UWE-R tiene en cuenta también un gran número de características,

trasladando las extensiones realizadas en WebML a UWE; sin embargo, se trata de una propuesta poco madura.

Como conclusión final, se puede resaltar que las diferentes propuestas han realizado un esfuerzo importante para permitir desarrollar RIAs desde un punto de vista conceptual, aunque, en algunos casos, la extensión de metodologías IWDM presenta algunas limitaciones debidas básicamente a que el enfoque del diseño inicial de la propuesta de modelado era un dominio menos complejo, como es el caso de las aplicaciones web. Además, como señalan (Toffetti Carughi *et al.* 2011), cada propuesta IWDM con soporte RIA se enfoca solamente en algunos aspectos específicos, algunas características.

Capítulo 3

Modernización de Aplicaciones Web

3.1. Mantenimiento Software

La gestión de la evolución de los sistemas software es un problema complejo que conlleva la puesta en marcha de diferentes actividades de una forma sistemática y coordinada. A lo largo de la historia de la Ingeniería del Software, se han desarrollado diferentes metodologías para gestionar la evolución de los sistemas software. Una de las claves consiste en disponer de un conocimiento preciso del sistema software que se quiere evolucionar. Por lo tanto, es fundamental disponer de unas metodologías adecuadas de documentación y gestión de cambios.

El estándar IEEE/EIA 12207 para procesos de ciclo de vida del software, ilustrado en la Figura 3.1, considera el mantenimiento como uno de los procesos principales del ciclo de vida, y describe mantenimiento como el proceso continuo de modificación del código y de la documentación de un producto software debida a la aparición de un problema o a la necesidad de mejora. El objetivo es modificar el producto software existente mientras se conserva su integridad. El estándar internacional para el mantenimiento del software, ISO/IEC 14764, define el mantenimiento del software en los mismos términos enfatizando los aspectos preliminares al mantenimiento, como es la planificación.

Un proceso de mantenimiento puede ser realizado para:

- Corregir fallos
- Mejorar el diseño
- Implementar mejoras
- Integrarse con otros sistemas
- Adaptar programas a diferentes dispositivos, plataformas, características del sistema y facilidades de telecomunicación.
- Migrar código heredado.

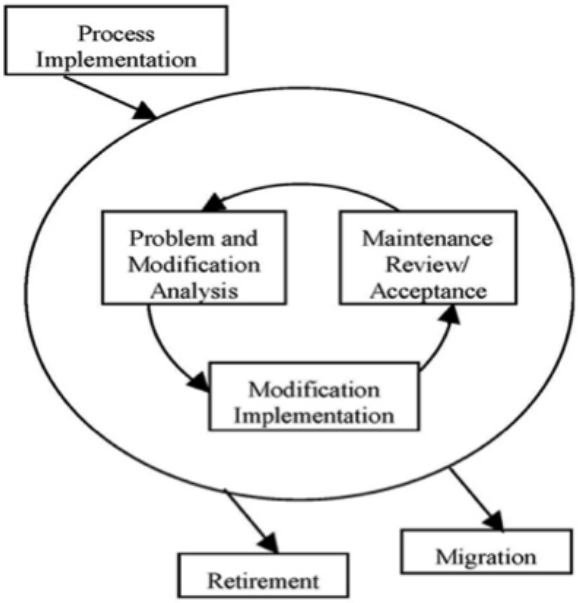


Figura 3.1: ISO/IEC 14764-00 Proceso de mantenimiento software

- Retirar software.

El alcance actual del uso de sistemas de información en empresas, que automatizan la gran mayoría de sus tareas, vuelve a las empresas muy dependientes de sus sistemas de información. Estos sistemas están también sujetos a las leyes Lehman (Lehman *et al.* 1997) del mantenimiento software:

1. La ley del cambio continuo. Un programa que se usa en un entorno real debe cambiar o su utilidad decrecerá de forma progresiva en ese entorno.

2. La ley del incremento de la complejidad. Según va cambiando un programa en evolución, su estructura tiende a hacerse más compleja. Recursos extra deben ser asignados para preservar y simplificar la estructura.

La primera ley proviene del hecho de que el entorno técnico, económico, legal, administrativo y social de una empresa cambia, conllevando la correspondiente necesidad de modificar los sistemas de información de soporte. El problema es que, generalmente, no se dedican suficientes recursos a contrarrestar el efecto de la segunda ley de Lehman. Por lo que un sistema tiende a volverse más y más complejo según pasa el tiempo. (Jacobson 1992) sugiere que el incremento de la complejidad derivada de un cambio en el sistema (o incremento de su entropía) es proporcional a la entropía del sistema antes del cambio. Cuanto más complejo es el sistema a cambiar, más se incrementará su complejidad. El autor utiliza esta idea para sugerir que un sistema de información cuya complejidad no está monitorizada alcanza un punto en el que su reescritura completa se vuelve obligatoria. Dada la gran cantidad del presupuesto software dedicado al mantenimiento (del 60 al 90 % del presupuesto total dependiendo de los autores (Erlikh 2000)), parece obvio que cualquier iniciativa para facilitar el mantenimiento podría ser muy beneficiosa.

Además del incremento de la complejidad, normalmente se puede observar que las modificaciones de un sistema no suelen recogerse en su documentación. Por lo tanto, con el tiempo, la implementación

de un sistemas dejará de estar alineada con su documentación. Esta discrepancia se puede dar tanto a nivel de arquitectura (la diferencia entre la arquitectura diseñada y la implementada (Kazman & Carrière 1997)) como a nivel funcional (nueva funcionalidad de usuario sin documentación de diseño). En estas circunstancias no tiene sentido utilizar la documentación para realizar el mantenimiento de un sistema software.

3.2. Reingeniería Software

En las últimas décadas la reingeniería de software se ha convertido en un importante campo de la ingeniería de software. Independientemente del resultado del proceso de reingeniería sobre un sistema heredado en producción, el ingeniero debe conocer cómo funciona el sistema y cuáles son sus reglas de negocio. Por lo tanto, es necesario extraer suficiente información del sistema heredado para que el objetivo de la reingeniería pueda satisfacerse. La calidad de la información dependerá de la tarea de reingeniería concreta a abordar.

Por un gran número de razones, especialmente económicas, volver a desarrollar un sistema de información desde el principio no parece una opción actualmente (Gall & Weidl 1999). En el caso de la reingeniería de sistemas heredados, una de las posibilidades es reutilizar algunos de los componentes heredados, cuyo funcionamiento es estable y está probado, en un nuevo sistema usando una tecnología de desarrollo moderno.

De hecho, la reingeniería de sistemas software complejos es un tema de investigación muy activo en los últimos años, extendiéndose desde la re-documentación de la arquitectura software, para la comprensión de los sistemas, a la reutilización de componentes software o la reestructuración o modernización de los sistemas para extender su vida útil.

Siguiendo la categorización propuesta por(Chikofsky & Cross 1990), se pueden definir los principales términos utilizados por la literatura de reingeniería como:

- Ingeniería directa (forward engineering). Es la forma tradicional de diseñar sistemas, empezando por los conceptos abstractos y la especificación independiente de la implementación para gradualmente llegar a la implementación en la plataforma destino.

- Ingeniería inversa (reverse engineering). Consiste en un proceso de análisis de un sistema software existente para identificar sus componentes y la relación entre los mismos para generar representaciones del sistema a niveles conceptuales diferentes. En este área, se pueden identificar dos subdominios comunes: re-documentación, en la que se genera o enriquece una representación al mismo nivel de abstracción (como el grafo de llamadas del código fuente), y la recuperación del diseño, en el que se crean modelos de más alto nivel de abstracción. Este último es normalmente un requisito para el entendimiento del programa.

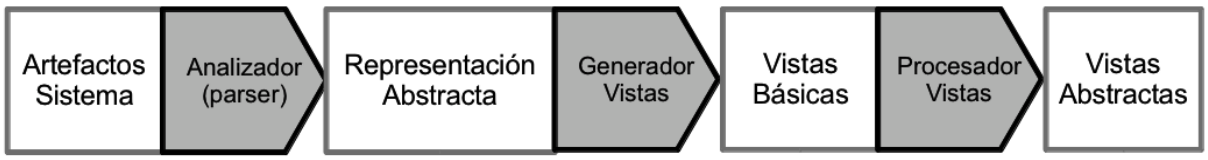


Figura 3.2: Etapas de un proceso de ingeniería inversa

- Reestructuración. Consiste en la transformación de la representación de un sistema a otra representación al mismo nivel conceptual, mientras se mantiene inalterado el comportamiento externo del sistema. Por ejemplo, la reestructuración del código fuente de un sistema para mejorar su legibilidad.

- Reingeniería. Consiste en el análisis y modificación de un sistema software para cambiar su forma e implementarlo en su nueva forma. Por lo general, la reingeniería se compone de dos pasos fundamentales (Chikofsky & Cross 1990): el primero, un proceso de ingeniería inversa para crear una descripción abstracta del sistema, y el segundo, un proceso de ingeniería directa. La reingeniería no se utiliza a menudo para mejorar la mantenibilidad, sino para reemplazar software heredado.

3.2.1. Ingeniería Inversa

La ingeniería inversa es el proceso de analizar software para identificar sus componentes e interrelaciones, y generar representaciones del software en una forma diferente o a un nivel más alto de abstracción. La ingeniería inversa es pasiva. No cambia el sistema software o da como resultado un nuevo sistema software. Un tipo de ingeniería inversa es la re-documentación. Otro tipo es la recuperación del diseño (Dorfman & Thayer 2002).

(Chikofsky & Cross 1990) afirman que “el propósito fundamental de la ingeniería inversa de un sistema software es incrementar el nivel de entendimiento general del sistema tanto para su mantenimiento como para nuevos desarrollos”.

Independientemente del objetivo de un proceso de reingeniería, el primer paso consiste en entender el sistema original, o al menos su estructura. El resultado de este paso puede ser usado para varias tareas desde la evaluación de la conformidad de la arquitectura (Kazman *et al.* 2003) hasta la identificación y extracción de componentes (Bergey *et al.* 2001). El marco fundamental de un proceso de ingeniería inversa se ilustra en la Figura 3.2.

El primer paso consiste en transformar los artefactos del sistema (código fuente, estructura de ficheros y directorios, ficheros de construcción y despliegue) a una representación abstracta que permita ser manipulada. El resultado de este proceso se almacena en algún tipo de repositorio permanente. Este paso conlleva dos ventajas principales:

- Facilitar el procesado posterior de las distintas fuentes de información que son transformadas en un solo formalismo.

- Eliminar las peculiaridades del lenguaje de programación usado en la implementación, al menos hasta cierto grado. De hecho, cuando el paradigma de programación cambia (por ejemplo, de un estilo procedural a un estilo orientado a objetos), este cambio se debe reflejar en la representación abstracta, a menos que el analizador realice algunas transformaciones de paradigma. Nótese que la representación abstracta se encuentra básicamente al mismo nivel de abstracción que el artefacto original, aunque alguna información puede haber sido introducida.

El segundo paso procesa la información contenida en el repositorio de representaciones abstractas para construir las vistas del sistema. Estas vistas representan alguna abstracción de mayor nivel de la información. Normalmente, una sola vista no proporciona suficiente información para entender el sistema.

El tercer paso construye algún tipo de vista abstracta del sistema entendible para un ingeniero mediante la agregación de la información de múltiples vistas básicas. Por ejemplo, el tercer paso puede abstraer alguna representación de la arquitectura del sistema o, alternativamente, calcular métricas software sobre el mismo sistema.

3.2.2. Ingeniería Inversa de Sistemas Heredados

Dado un sistema heredado, diferentes fuentes de información pueden ser usadas en el proceso de ingeniería inversa, entre las cuales podemos encontrar:

- La documentación técnica del sistema: especificación, análisis, diseño, implementación, pruebas y despliegue.
- Los manuales: el de usuario o el de entrenamiento.
- La experiencia de los profesionales implicados en el desarrollo del sistema y su mantenimiento: arquitecto software, diseñador, desarrollador y mantenedor.
- Los artefactos de despliegue del sistema: código fuente con comentarios, estructura de ficheros y directorios, descriptores de despliegue, scripts de construcción y compilación.
- El resto de implicados en el sistema: usuarios y expertos del negocio.

Dependiendo de la tarea de reingeniería, el tipo de información a recuperar puede ser diferente. Por ejemplo, si el objetivo de la reingeniería es mejorar la arquitectura de un sistema, entonces es posible que no se necesiten conocer los detalles de los algoritmos que lo componen. Por otro lado, si el objetivo es extraer las reglas de negocio, entonces se deben analizar los algoritmos y no la arquitectura general del sistema.

En ese sentido, según su objetivo, las diferentes propuestas de reingeniería favorecerán algunas de las fuentes de información en detrimento de otras. Dependiendo de las fuentes de información disponibles,

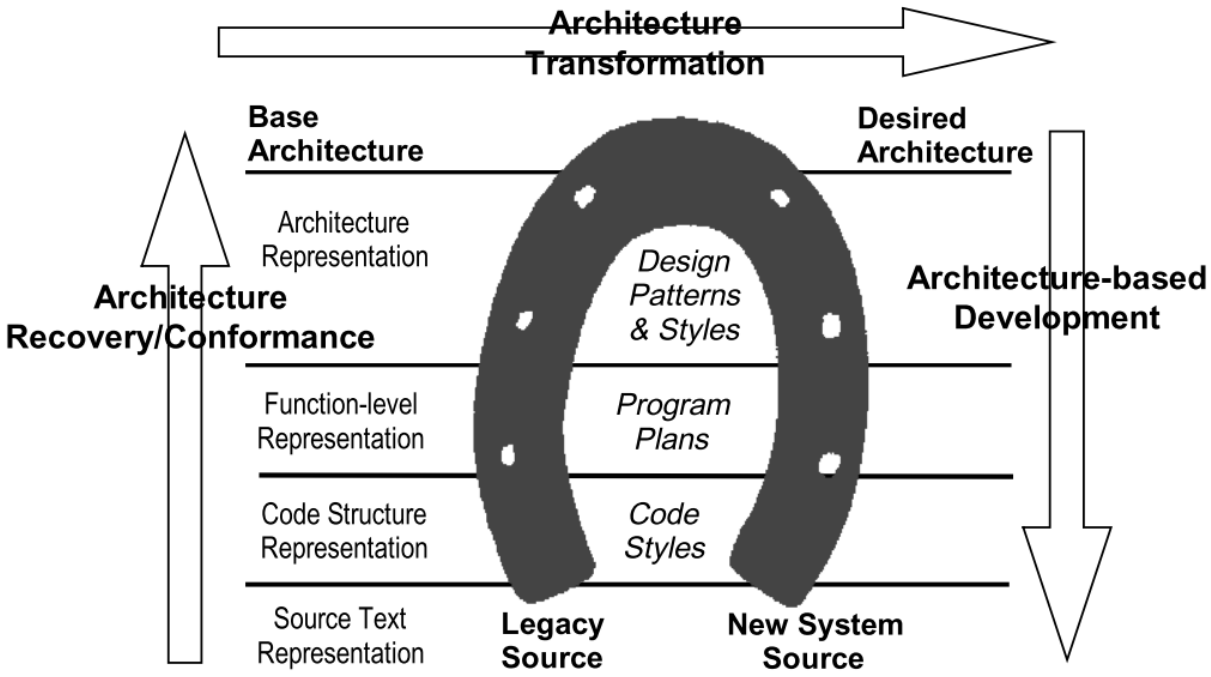


Figura 3.3: Modelo de herradura del instituto SEI

un proceso de reingeniería podría necesitar inferir la información no disponible para alcanzar sus objetivos. De hecho, (Kazman & Carrière 1997) sugieren que el proceso de ingeniería inversa se asemeja al trabajo de un detective: reconstruir la imagen completa de un sistema a partir de la evidencia disponible. Por tanto, la reingeniería está fuertemente conectada al campo de la comprensión de programas que se relaciona a su vez con la noción de complejidad de software. En este sentido, (Banker *et al.* 1993) ha investigado el enlace entre los costes de mantenimiento y la complejidad software (interpretada como la dificultad de comprender un sistema) resultando en una correlación positiva.

3.2.3. El Ciclo de Vida de la Reingeniería

La ingeniería inversa de un sistema software puede perseguir muchos objetivos, entre otros:

- Comprender el sistema actual para facilitar su mantenimiento y reducir la introducción de errores.
- Obtener la especificación del sistema de manera que refleje fielmente al sistema en ejecución.
- Modificar la arquitectura del sistema en ejecución para mejorar sus atributos de calidad.
- Extraer componentes útiles para su reutilización en un sistema nuevo.

En el caso de la reingeniería software basada en la arquitectura, el Software Engineering Institute de la Universidad Carnegie Mellon publicó el famoso modelo de herradura (Bergey *et al.* 1999), ilustrado en la Figura 3.3.

En este modelo, el proceso de reingeniería se realiza en tres pasos fundamentales:

1. Análisis del sistema existente y construcción de un conjunto de descripciones lógicas de su estructura.

2. Transformación de estas descripciones para obtener una nueva estructura mejorada.

3. Reestructuración del sistema para seguir las nuevas descripciones.

En este modelo, se analiza el código fuente y se van generando modelos de nivel de abstracción creciente hasta el nivel de arquitectura. Entonces se modifica la arquitectura del sistema para cumplir con las nuevas especificaciones del sistema o alcanzar los niveles predefinidos de los atributos de calidad. Dado que los atributos de calidad dependen de la arquitectura del sistema, es necesario primero recuperar la arquitectura para poder proponer alguna mejora. Los procesos de ida y vuelta de todos los niveles de la herradura representan la forma de reingeniería más completa. Sin embargo, en la práctica existen dos atajos a través de la herradura. Éstos están representados por las transformaciones en los dos niveles más bajos de abstracción: el nivel de código y el nivel funcional. A nivel de código, estas transformaciones pueden variar desde simples acciones como cambios sintácticos en el código fuente a la migración de un sistema de un lenguaje de programación a otro. Los cambios en la implementación de una función, la adaptación de una función a nuevos requisitos o la adaptación de la interfaz de una función son ejemplos de transformaciones a nivel funcional. Las transformaciones a nivel de arquitectura son representadas, por ejemplo, por cambios en la estructura del sistema, el tipo de componentes e interacciones o la composición funcional de los módulos. Aunque transformaciones de bajo nivel pueden tener lugar sin transformaciones de más alto nivel, las transformaciones de más alto nivel se apoyan en las de más bajo nivel (Bergey *et al.* 1999). Por ejemplo, si se migra un programa monolítico a un sistema con arquitectura orientada a servicios, esta operación se realiza a nivel de arquitectura porque implica cambios de la estructura del sistema. Este cambio fundamental tendrá su impacto en los niveles de código y funcional también. Finalmente, es importante señalar que cuanto más alto sea el nivel en el que se realizan las transformaciones, más alto será el nivel necesario de experiencia de los ingenieros implicados.

La flecha a la izquierda (de abajo a arriba) del modelo de herradura representa la ingeniería inversa del sistema existente. Después de la transformación a nivel de arquitectura, la flecha a la derecha (de arriba a abajo) representa la ingeniería directa del nuevo sistema basado en una arquitectura mejorada.

3.3. Modernización Software Dirigida por Modelos

Desde los primeros años de la pasada década, en los que IDM emergió como una nueva disciplina de la ingeniería del software, la utilidad de los modelos y de las transformaciones de modelos en la ingeniería inversa y modernización de software se hizo evidente, como se señaló en (Favre 2004; Bezivin *et al.* 2007). En el año 2003 el consorcio *Object Management Group* (OMG) crea la *Architecture-Driven Modernization Task Force* (ADMTF) para definir especificaciones y promocionar el consenso industrial

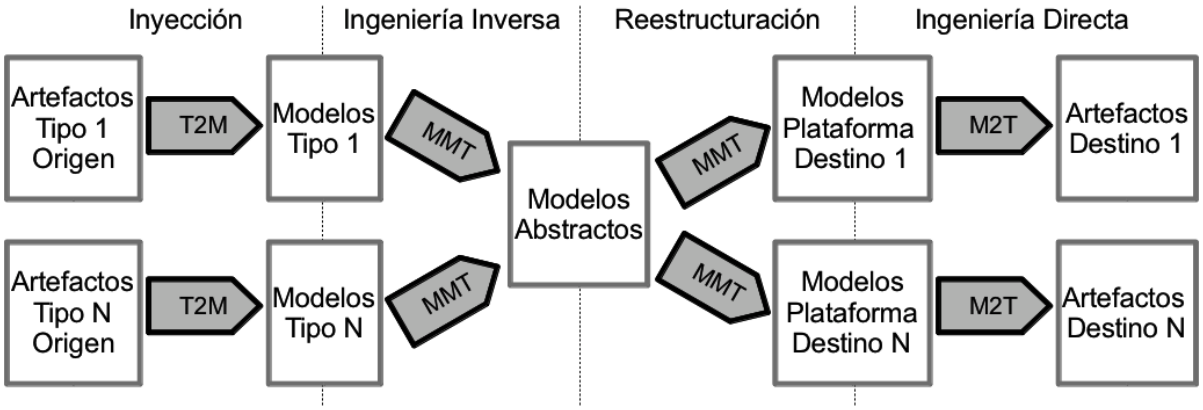


Figura 3.4: Esquema base de un proceso de reingeniería dirigido por modelos

en la modernización de aplicaciones existentes. Este grupo propone el uso de técnicas y herramientas propias de la IDM con el objetivo de formalizar y estandarizar estos procesos de modernización, como se deriva de su principal aportación: el estándar *Knowledge Discovery Metamodel* (KDM).

A diferencia del mantenimiento, la modernización de software implica cambios importantes en la estructura y/o funcionalidad de una aplicación existente. Estos cambios son destinados a mantener el valor estratégico de la aplicación para la empresa. La reingeniería de software es un tipo de modernización destinada a transformar aplicaciones de una forma disciplinada con el propósito de mejorar su calidad (Tilley & Smith 1995). Una migración es un caso especial de reingeniería en el que un sistema software heredado se traslada a una tecnología más actual.

Metamodelado y transformaciones de modelos son las dos técnicas básicas sobre las que se construyen las soluciones IDM (Brambilla *et al.* 2012). En el caso de la reingeniería, las etapas de ingeniería inversa, reestructuración y generación del nuevo sistema son organizadas como una cadena de transformaciones de modelos, como ilustra la Figura 3.4. Antes de aplicar la ingeniería inversa es necesaria una etapa previa para inyectar modelos (T2M) a partir de los artefactos a modernizar (por ejemplo, código fuente o un esquema relacional). Después de esta inyección, se aplica la ingeniería inversa como una cadena de transformaciones modelo a modelo (MMT). La siguiente etapa es la reestructuración, que implica otra cadena de transformaciones MMT, y finalmente se genera el nuevo sistema mediante una cadena que debe tener una transformación modelo a texto (M2T) final. Como indican (Bruneliere *et al.* 2010), uno de los principales retos de la ingeniería inversa proviene del hecho de la gran heterogeneidad existente entre las aplicaciones heredadas a considerar, en contraposición a la IDM en la que los datos son uniformemente representados como modelos. De esta manera, la Ingeniería Inversa Dirigida por Modelos consiste en cambiar el heterogéneo mundo de las tecnologías de implementación por el homogéneo mundo de los modelos. La idea general consiste en recuperar uno o varios modelos de un sistema dado, dependiendo en los puntos de vista requeridos, y trabajar directamente con esos modelos. Cuando el sistema heredado se representa como modelos, se simplifica: (1) la utilización (o reutilización) de metamodelos genéricos y las transformaciones relacionadas para manejarlos; (2) la extensión

del soporte a diferentes tipos de tecnologías (o herramientas) tanto de entrada como de salida del proceso de ingeniería inversa.

3.3.1. Modernización Dirigida por Arquitectura

La modernización dirigida por arquitectura (Architecture-Driven Modernization, ADM), según (Ulrich 2010), es el proceso de entender y evolucionar un sistema software existente con alguno de los siguientes objetivos:

- Mejora de la calidad del software.
- Modificaciones.
- Interoperabilidad.
- Refactorización.
- Reestructuración.
- Reutilización.
- Portabilidad.
- Migración.
- Traducción a otro lenguaje.
- Integración de aplicaciones empresariales.
- Arquitectura orientada a servicios.

Para que las organizaciones puedan aplicar conceptos de transformación con éxito, la modernización debe realizarse desde las tres perspectivas arquitectónicas: (1) tecnológica, (2) aplicación y datos, y (3) de negocio. La modernización dirigida por la arquitectura tecnológica constituye tradicionalmente el tipo de modernización de proyectos más comúnmente aplicado. Se debe al riesgo relacionado con la obsolescencia del lenguaje o la plataforma, el coste de propiedad, la usabilidad del sistema y otros factores que son solucionados mediante un cambio físico. La modernización dirigida por la arquitectura de la aplicación y los datos eleva la escala del paradigma de modernización. Los proyectos en esta categoría generalmente se expanden por múltiples aplicaciones. La modernización a nivel de negocio supone la solución más completa e incorpora modelos de la arquitectura del negocio, arquitecturas de aplicaciones y datos, y arquitecturas técnicas. La Figura 3.5 ilustra la modernización dirigida por la arquitectura del negocio, propuesta por ADM. Este tipo de modernización alinea formalmente las arquitecturas de aplicación y datos con los modelos de negocio que incluyen una combinación de gobernanza, semánticas del negocio, reglas de negocio y procesos de negocio.

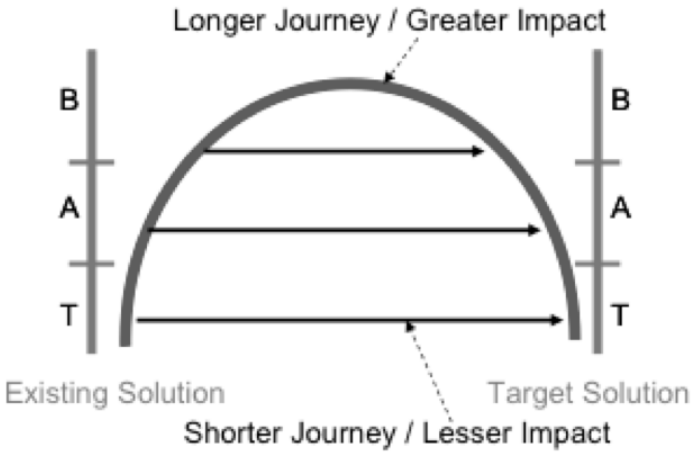


Figura 3.5: Modelo de herradura de ADM

Existen tres elementos fundamentales para cada camino de transformación, independientemente del nivel de impacto arquitectónico. Éstos son:

1. Descubrimiento del conocimiento de la solución existente. Éste se da a diferentes niveles de abstracción y en varios grados de alcance, apropiados para cada uno de los proyectos relacionados.

2. Definición de la arquitectura objetivo. Con el fin de definir una propuesta de transformación, los analistas deben crear una solución objetivo que sirva como marco de trabajo en el que las soluciones existentes pueden ser mapeadas o transformadas.

3. Pasos de transformación que llevan del estado original al estado destino. Las propuestas varían desde las físicas (por ejemplo, migración del lenguaje) a las más abstractas (por ejemplo, mapeo de reglas de negocio a un entorno basado en reglas). Estas rutas de transformación deben sincronizarse tanto vertical (del negocio a la implementación física) como horizontalmente (sistema existente a sistema objetivo).

3.3.2. Trabajos de modernización

Los modelos son útiles como representaciones abstractas de diferentes aspectos de un sistema heredado y las transformaciones de modelos permiten automatizar algunas tareas implicadas en la modernización de tales sistemas. A lo largo de estos años, las técnicas IDM han sido aplicadas en una gran variedad de escenarios de modernización, en especial en migración de aplicaciones .

(Reus *et al.* 2006) presentan una propuesta de reingeniería de sistemas heredados hacia la arquitectura dirigida por modelos (model-driven architecture, MDA). En concreto, los autores aplican su propuesta a la modernización de un sistema de producción de 179 KLOC del dominio de los seguros escrito en PL/SQL. Los principales pasos de su propuesta son:

1. Realizar un análisis del código fuente del sistema heredado según su gramática.

2. Mapear los árboles de sintaxis abstracta obtenidos a un modelo de gramática definido en MetaObject Facility (MOF[1]).

3. Usar transformaciones modelo a modelo (MMT) para convertir el modelo de gramática en un metamodelo genérico, denominado *GenericAST*, en el que la información acerca de los sistemas software se puede almacenar de forma indenpendiente del lenguaje.

4. Mapear los modelos *GenericAST*, mediante transformaciones modelo a modelo, a modelos UML que se pueden usar para la generación de código o para documentación.

Una de las contribuciones de más valor es su colección de lecciones aprendidas, que proporcionan una valiosa información desde el punto de vista de la experiencia. A continuación, se recogen algunas de las más interesantes:

- Tener una imagen clara de lo que se quiere extraer. Las construcciones de interés que aparecen en la entrada deben ser especificadas, de manera que los extractores puedan reconocerlas.

- Conocer los modelos destino con detalle. Incrementa considerablemente la usabilidad de los modelos extraídos.

- Hacer uso de guías de estilo de programación o convenciones de nombres. Cuanta más información específica del sistema se use en la extracción, mejor será la calidad de los modelos iniciales obtenidos.

- Mantener las gramáticas pequeñas y enfocadas. Las gramáticas más pequeñas son más fáciles de mantener. Cuando una gramática completa no está disponible para un proceso de extracción, se debe desarrollar la gramática mínima que contenga la información necesaria para describir la entrada pero no un lenguaje completo.

- Modularizar las gramáticas. Las gramáticas modulares mejoran la reutilización de las partes de las gramáticas de uso común, como las definiciones de sentencias o expresiones. Adicionalmente, se mejora la reutilización de las partes de las transformaciones MMT que se corresponden con las partes de las gramáticas.

- Diseñar gramáticas hacia un metamodelo destino. La gramáticas deben desarrollarse teniendo en mente un metamodelo destino concreto, incluyendo cómo mapear la gramática al metamodelo destino. De esta manera se mejora la reutilización de las partes de las transformaciones MMT.

- Evitar el uso de XMI para almacenar los contenidos de los modelos. Se debe evitar usar XMI para la persistencia de los resultados de la extracción porque es un mecanismo muy lento.

[1] http://www.omg.org/mof/

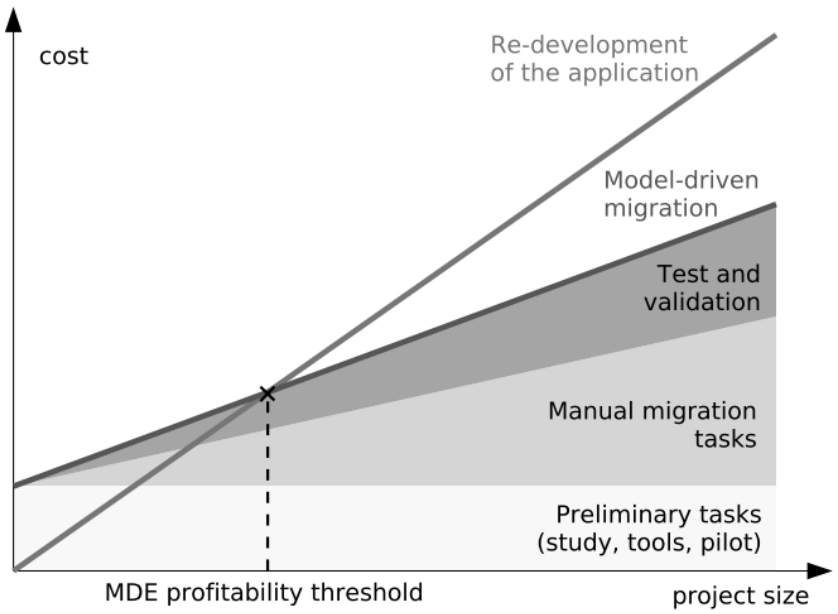


Figura 3.6: Coste del proyecto en función de su tamaño (Fleurey *et al.* 2007)

- Optimizar transformaciones MMT. Las transformaciones MMT pueden convertirse en el cuello de botella del proceso.

(Fleurey *et al.* 2007) presentan un informe del uso de IDM como una propuesta eficiente, flexible y de confianza para un proceso de migración (ingeniería inversa, transformación y generación de código). El trabajo se ilustra con la migración de un sistema bancario de gran escala desde Mainframe a JEE. El proceso incluye el análisis automático de código existente, ingeniería inversa de modelos de alto nivel de abstracción, transformación de modelos a la plataforma destino y generación de código. En el desarrollo de este proceso ha sido necesario definir un conjunto de metamodelos y cadenas de transformaciones para la automatización de todos estos pasos.

Además, los autores discuten cómo el uso de la IDM es más beneficioso económicamente y más eficiente en costes que la migración mediante el re-desarrollo manual. La diferencia más importante entre ambas aproximaciones es el conjunto significativo de tareas preliminares requerido por las técnicas dirigidas por modelos. Los autores discuten sobre la influencia de estas tareas preliminares en la planificación y reducción de costes de los procesos de migración y concluyen que, para proyectos por encima de un tamaño crítico, la aproximación dirigida por modelos es más beneficiosa que el re-desarrollo. De hecho, el esfuerzo a invertir en el desarrollo de herramientas de migración depende principalmente de la complejidad de las plataformas de entrada y salida y no del volumen de código a migrar. La Figura 3.6 presenta una estimación del coste de un proyecto de migración usando una migración dirigida por modelos y un re-desarrollo. En el caso de éste último, el coste es directamente proporcional al tamaño de la aplicación heredada. Mientras que en el caso de la migración dirigida por modelos, existe un coste inicial fijo relacionado con el desarrollo de las herramientas que se complementa con un coste lineal correspondiente a los esfuerzos de migración manual. El gradiente de la función correspondiente a la migración es

menor que el gradiente correspondiente al re-desarrollo porque usar herramientas de migración reduce el esfuerzo manual a realizar. Sin embargo, no es posible estimar un umbral general de beneficio de la migración dirigida por modelos porque depende en última instancia del número de herramientas que se deben desarrollar para cada proyecto.

Finalmente, los autores exponen las principales ventajas y limitaciones del uso de la IDM en la modernización de sistemas heredados. La principal ventaja consiste en la automatización parcial del proceso de reingeniería, que puede permitir una reducción significativa de los costes y la duración de los procesos de reingeniería. La segunda ventaja es que mejora la reutilización entre proyectos de reingeniería. Todas las transformaciones y herramientas desarrolladas para un proceso concreto pueden ser adaptadas a procesos futuros con plataformas similares de entrada y salida. La primera limitación viene dada por el tiempo y coste que suponen las tareas preliminares. Y la segunda limitación consiste en el coste de pruebas, que es un problema general en la migración de software. La principal razón del gran impacto de las tareas de pruebas en el coste se debe a que, para la mayoría de los proyectos de migración, se realizan manualmente.

(Sánchez-Ramón *et al.* 2013) comentan el desarrollo de un proyecto referente a la generación automática de unos wrappers Java para el acceso a lógica de negocio implementada como procedimientos PL/SQL. En el proceso se utilizaron las reglas de codificación y guías para el desarrollo de los wrappers que poseía la empresa. La solución consistió en una cadena de transformaciones de modelos que generaba el código Java de los wrappers a partir del código PL/SQL. Esta cadena comenzaba con una inyección de modelos, seguía con una transformación MMT de los modelos PL/SQL de bajo nivel a unos modelos conformes a un metamodelo que representaba explícitamente las reglas y guías de la empresa, y finalmente se aplicaba una transformación M2T para generar el código Java de los wrappers. Adicionalmente, los autores presentan una interesante relación de ventajas e inconvenientes del uso de IDM en procesos de modernización software que se ha utilizado para la redacción de las conclusiones de esta tesis y, por lo tanto, no se ha incluido aquí. (Cánovas Izquierdo & García Molina 2012) proponen un nuevo lenguaje, Gra2Mol, para la extracción de modelos a partir de código fuente de lenguajes de programación. (Sánchez Ramón *et al.* 2010; Bermudez & García-Molina 2012) presentan una herramienta para asistir en la migración semiautomática de aplicaciones Oracle Forms a Java. Los autores definen una arquitectura dirigida por modelos para automatizar la migración del interfaz de usuario y de la capa de persistencia. Estas arquitecturas son independientes de la plataforma destino, facilitando su reutilización.

3.3.3. Herramientas de modernización

A lo largo de esta última década, han aparecido algunas herramientas de modernización dirigida por modelos. (Perez-Castillo *et al.* 2011) presenta una relación de once herramientas, de las cuáles sólo tres

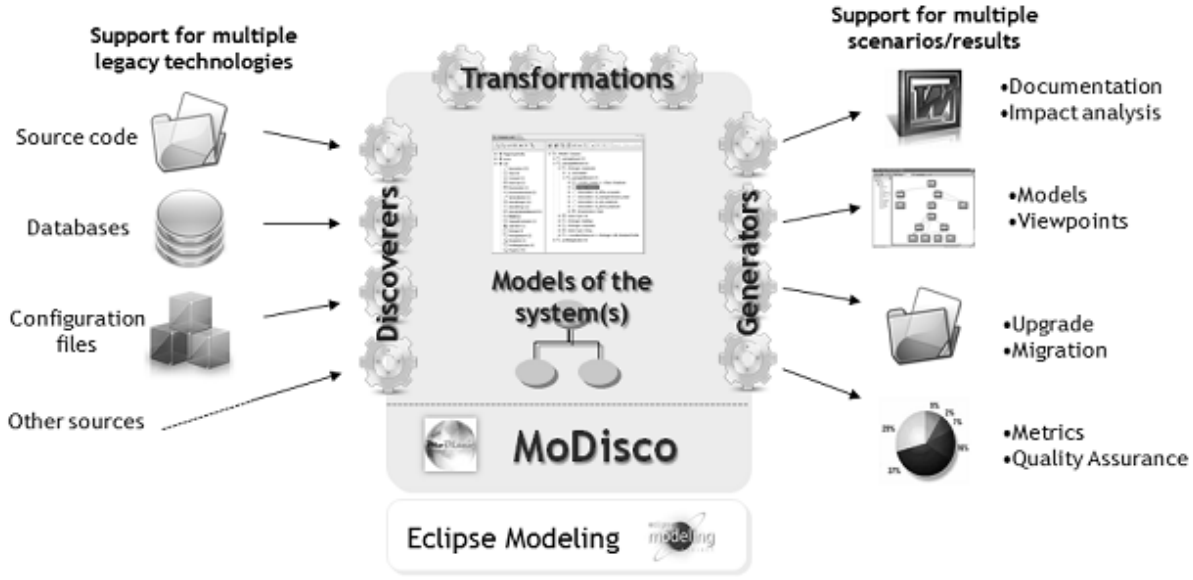


Figura 3.7: Vista general de MoDisco

usan métodos y técnicas propios de la IDM. Entre ellas destaca el proyecto MoDisco[2] (Bruneliere *et al.* 2010) tanto por su planteamiento extensible y genérico como por su licencia no comercial.

MoDisco es un proyecto oficial Eclipse-MDT (Modeling Tools) dedicado a la ingeniería inversa dirigida por modelos de sistemas de información heredados. Proporciona un conjunto de componentes extensibles y reutilizables que conforman un framework genérico para la elaboración de soluciones concretas que den soporte a las especificidades de los diferentes escenarios de ingeniería inversa. Como muestra la Figura 3.7, puede ser usado como parte de procesos de modernización pero también para la re-documentación, evaluación de la calidad o cualquier otro caso de uso que requiera una mejor comprensión de los sistemas existentes.

MoDisco, además, es considerado por la OMG ADMTF como el proveedor de referencia para implementaciones reales de varios de sus estándares: Knowledge Discovery Metamodel (KDM), Software Metrics Metamodel (SMM) y el Abstract Syntax Tree Metamodel (ASTM). Adicionalmente, proporciona un soporte avanzdo de JAVA, JEE y tecnologías XML, incluyendo metamodelos, discoverers (inyectores), navegadores de modelos adaptados y motor de consultas propio.

3.4. Ingeniería Inversa de Aplicaciones Web

El método propuesto en (Di Lucca *et al.* 2001) define todas las tareas necesarias para generar un conjunto de vistas a partir de una aplicación web a diferentes niveles de abstracción. El comportamiento de una aplicación web proviene de los elementos tanto estáticos como dinámicos del sistema, como en el caso de las aplicaciones software tradicionales; por lo tanto, un proceso de ingeniería inversa de aplicaciones web debe comprender las siguientes fases:

[2]http://www.eclipse.org/modisco

- Análisis estático.
- Análisis dinámico.
- Análisis del comportamiento.

Según (Patel *et al.* 2007), generalmente, las actividades de ingeniería inversa de aplicaciones web tienden a concentrarse en el análisis estático de los datos fuente en vez de en el análisis dinámico de la operación de las aplicaciones web. La ingeniería inversa de aplicaciones web requiere procedimientos distintos a los utilizados en otros sistemas software. Las aplicaciones web son sistemas basados en internet y desarrollados con un conjunto específico de tecnologías que no está tradicionalmente presente en otros sistemas software. Otro aspecto distintivo se basa en el hecho de que generalmente el proceso de desarrollo de aplicaciones web suele verse especialmente afectado por períodos de desarrollo demasiado cortos y una frecuencia de actualización (mantenimiento) muy alta, por lo que se suelen sacrificar algunas buenas prácticas del desarrollo software que afectan a la calidad final del propio sistema, como señala (Di Lucca *et al.* 2002). Por lo tanto, como señala (Patel *et al.* 2007), un gran número de propuestas de ingeniería inversa sobre aplicaciones web tienen como objetivo realizar refactorizaciones para mejorar la calidad del sistema final, generar documentación del sistema (no realizada en el proceso de desarrollo) o simplemente realizar una validación del mismo.

Un gran número de las propuestas de ingeniería inversa de aplicaciones web se ha construido sobre propuestas tradicionales de ingeniería inversa contrastadas. Por ejemplo, (Benedusi *et al.* 1992) describen el paradigma de ingeniería inversa denominado GMT (Goals, Models, Tools) que fue usado posteriormente como base de la metodología de ingeniería inversa de aplicaciones web WARE (Di Lucca *et al.* 2001). (Hassan & Holt 2001) describen cómo adaptaron la propuesta Portable Bookshelf Environment (PBS) de ingeniería inversa de sistemas tradicionales para obtener la arquitectura de una aplicación web. (Boldyreff & Kewish 2001) proponen un sistema que explota las ideas tradicionales de la ingeniería inversa para extraer contenido duplicado de la páginas web con el objetivo de reestructurarlas para mejorar su mantenibilidad.

(Tilley & Huang 2001) caracteriza los procesos de ingeniería inversa en base a tres categorías:

1. Datos. La obtención de los datos es normalmente el primer proceso de la ingeniería inversa. Las técnicas usadas para la obtención de los datos incluyen: la exploración del sistema, el estudio de la documentación y la captura de la experiencia. La exploración del sistema a menudo supone el análisis estático del código fuente del sistema heredado. La alternativa consiste en realizar análisis dinámicos que requieren la ejecución del código.

2. Conocimiento. La gestión del conocimiento se refiere a la captura y organización del conocimiento acerca del dominio del problema.

3. Información. La exploración de la información es donde se produce el entendimiento del sistema heredado.

El uso de metodologías de ingeniería inversa diseñadas concretamente para aplicaciones web debe proporcionar un plan completo, determinando qué procesos deben ser realizados mediante herramientas relacionadas para el soporte de los métodos empleados. A continuación, se revisan algunas de las principales propuestas de ingeniería web de aplicaciones web.

3.4.1. Reverse Engineering to RMM

La primera referencia de una propuesta de ingeniería inversa de aplicaciones web se puede atribuir a (Antoniol *et al.* 2000), en concreto, se centran en la reingeniería de sitios web. La metodología basada en RMM de (Antoniol *et al.* 2000) consiste en una propuesta de reingeniería y, por tanto, su primera etapa define un proceso de ingeniería inversa. Los autores describen el proceso usando un módulo de una universidad como ejemplo de una aplicación web. El proceso recupera una vista de la arquitectura a partir del análisis de los enlaces y contenidos de las páginas HTML del sitio web. Esta vista es conforme al modelo de datos de gestión de las relaciones (Relationship Management Data Model, RMDM) de la Metodología de Gestión de Relaciones (Relationship Management Methodology, RMM) para el diseño de aplicaciones hipermedia. El diseño recuperado se utiliza para realizar un proceso de reingeniería del sitio web pasando de un sitio web basado en ficheros a uno basado en un sistema gestor de bases de datos relacional y generalizando su funcionalidad para dar servicio a un mayor número de usuarios.

3.4.2. WANDA

(Antoniol *et al.* 2004) proponen una herramienta para analizar dinámicamente una aplicación web denominada WANDA (Web ApplicatioNs Dynamic Analyzer). La herramienta introduce código script en las páginas HTML originales para almacenar en una base de datos la información recogida durante la ejecución de la aplicación web. Posteriormente, la información de la base de datos se utiliza para generar diagramas UML, como diagramas de componentes, diagramas de despliegue, diagramas de secuencia y diagramas de clases. Los diagramas de componentes y despliegue resaltan la arquitectura multicapa de la aplicación web, así como la interacción con servicios web, sistemas de bases de datos, ficheros, etc. Los diagramas de secuencia muestran la interacción entre el control y las entidades de datos. Los diagramas de clases representan la aplicación según las extensiones de UML de Conallen (Conallen 1999). Dado que WANDA es principalmente una herramienta de análisis dinámico, ha siso necesario extender el modelo UML de Conallen para representar la frecuencia de invocación de cada asociación. De esta manera, los ingenieros de mantenimiento pueden visualizar las interacciones que realmente tienen lugar durante el uso de la aplicación web.

3.4.3. WARE

Uno de los casos mejor documentados lo constituye WARE propuesta por (Di Lucca *et al.* 2001; 2004). Esta metodología está basada en el paradigma GMT de (Benedusi *et al.* 1992) y utiliza como sistema de representación la extensión de UML propuesta por (Conallen 1999), WebUML. La fase de modelado incluye el uso de: (1) diagramas de casos de uso para la especificación de los requisitos funcionales, (2) diagramas de clases para la descripción de la estructura de la aplicación web y (3) diagramas de secuencia para documentar la interacción dinámica. Adicionalmente, (**?**Di Lucca *et al.* 2002) desarrollaron la herramienta WARE para proporcionar soporte a esta metodología.

Inicialmente, esta propuesta sólo daba soporte a la realización de análisis estáticos tanto de las páginas HTML en el lado cliente como de los artefactos software de la aplicación en el lado servidor. Posteriormente, sin embargo, se extendió para proporcionar funcionalidades de análisis dinámico. En (Di Lucca *et al.* 2005), los autores presentan la integración de las herramientas WARE y WANDA. En ese trabajo los autores presentan cómo esas herramientas pueden ser usadas para identificar la equivalencia de clases de grupos de páginas cliente construidas dinámicamente, con el objetivo de mejorar la comprensión de la aplicación web.

3.4.4. Reverse Engineering to UWA

En (Di Lucca *et al.* 2006) se presenta una propuesta de ingeniería inversa de aplicaciones para la extracción de modelos conceptuales centrados en el usuario a partir de aplicaciones web existentes con el fin de re-documentarlas desde un nivel de abstracción mayor y desde el punto de vista del usuario. Los modelos recuperados son representados según la metodología de diseño Ubiquitous Web Application (UWA). Los modelos UWA son capaces de describir la estructura de los contenidos de la aplicación, las relaciones semánticas entre los contenidos, las diferentes vistas de contenidos que la aplicación ofrece a los usuarios, así como los caminos y nodos de navegación usados para presentar el contenido a los usuarios.

La propuesta explota métodos y herramientas de ingeniería inversa existentes para extraer información estructural de detalle a partir de las aplicaciones analizadas y abstraer modelos UWA a partir de la misma. Dado que gran parte del proceso de análisis y abstracción se realiza sobre las páginas HTML en el lado cliente, el método es aplicable a la mayoría de aplicaciones web existentes.

3.4.5. ASP.NET

(Katsimpa *et al.* 2006) describen una metodología de ingeniería inversa de aplicaciones web específicamente desarrollada para aplicaciones creadas usando la tecnología ASP.NET; aunque los autores señalan que puede ser aplicada de manera general. Esta propuesta tiene como objetivo generar una representación de la aplicación web en un nivel más alto de abstracción. El proceso consiste en analizar estáti-

camente los ficheros *aspx* para generar un grafo dirigido acíclico que sirve de representación intermedia a partir de la cual se genera una representación conceptual. En concreto, generan una representación en WebML. Los autores comentan que las razones para la elección de esta representación son su robustez y la existencia de una herramienta de soporte como WebRatio.

Las aplicaciones web basadas en la tecnología ASP están normalmente formadas por una colección de páginas y formularios ASP.NET soportados por código adicional, ficheros de configuración y descripciones de metadatos en ficheros XML. En esta propuesta se realiza un análisis estático mediante el uso de dos analizadores. El primero se aplica a los ficheros *aspx* y produce un árbol por cada página aspx. Posteriormente, con la ayuda de una matriz de correspondencia, se emparejan etiquetas con las unidades WebML apropiadas. De esta manera, se crea una primera estimación de qué unidades contendrá cada página. Finalmente, el segundo analizador procesa los ficheros fuente para localizar la fuente de los datos de cada unidad.

3.4.6. ReWeb

(Ricca & Tonella 2000; 2001) desarrollaron una herramienta para llevar a cabo el análisis de código de una aplicación web mediante la representación de la aplicación como un grafo y la realización de varios tipos de análisis sobre el mismo como de alcanzabilidad, de flujo y detección de patrones. ReWeb puede descargar y analizar aplicaciones web. Se incluyen, además, herramientas de búsqueda y navegación y existe incluso la posibilidad de visualizar la evolución de una aplicación web mediante un código de colores. Los autores también proponen mejorar la información obtenida mediante análisis dinámicos mediante el análisis de los ficheros de registro del servidor.

3.4.7. VAQUISTA

VAQUISTA (reVerse engineering of Applications by Questions, Information Selection, and Transformation Alternatives) (Vanderdonckt *et al.* 2001; Bouillon & Vanderdonckt 2002) propone un proceso de ingeniería inversa del interfaz de usuario de aplicaciones web con el objetivo de facilitar su migración a diferentes plataformas. El proceso definido por VAQUISTA comienza con el análisis estático de cada página HTML para derivar un modelo de presentación que describa los elementos principales de las páginas HTML. Las transformaciones son realizadas mediante el uso de tablas de mapeo. El modelo de presentación está formado por una jerarquía de elementos de presentación u objetos. Los autores identifican cuatro categorías de objetos de presentación:

1. Concrete Interaction Objects (CIOs), representan elementos reales de la página HTML que son atómicos (no pueden descomponerse), como texto, imágenes, animaciones, botones, etc.

2. Abstract Interaction Objects (AIOs), representan abstracciones de los CIOs identificados por su nombre, atributos y otros parámetros.

3. Logical Window (LW), representa el contenedor lógico para AIOs o una ventana física definida en la página HTML.

4. Presentation Unit (PU), representa el entorno de presentación para realizar una tarea interactiva.

3.4.8. TERESA

TERESA (Paganelli & Paterno 2002) es una herramienta de análisis estadístico de código fuente que permite analizar un sitio web con el fin de generar automáticamente el diseño de la lógica de interacción subyacente. Este diseño se representa mediante un modelo de tareas que describe cómo se deben realizar las actividades para alcanzar los objetivos de un usuario. Además, este modelo de tareas puede ser utilizado como entrada de una herramienta para la evaluación automática de la usabilidad de sitios web.

3.4.9. Revangie

Revangie (Draheim *et al.* 2005) propone el uso de técnicas de caja negra para realizar un proceso de ingeniería inversa del interfaz dinámico de una aplicación web sin analizar el código de la misma. Se trata de una propuesta independiente de las tecnologías de implementación de la aplicación web. Según los autores es mucho más sencillo analizar código HTML que el código que genera las páginas HTML. Revangie utiliza un modelo de interfaz de usuario orientado a formularios que usa máquinas de estado bipartitas con tipos para modelar (i) páginas en el cliente y (ii) acciones en el servidor. Los modelos son grafos que incluyen las relaciones entre las páginas y las acciones en el servidor.

3.4.10. JSPick

JSPick (Draheim *et al.* 2003) es una herramienta de ingeniería inversa para capas de presentación basadas en páginas de servidor. Aunque los autores argumentan que su solución es genérica porque sus semánticas formales son independientes del lenguaje de programación utilizado, la herramienta se ha implementado sobre Java Server Pages (JSP). JSPick genera la documentación de la interfaz del sistema en un lenguaje de especificación de fácil lectura. Además, JSPick genera un navegador del interfaz gráfico de usuario para cada sistema. Con este navegador el desarrollador puede examinar el código fuente, los árboles de sintaxis abstracta, la información de tipos y la estructura del sistema mediante la consulta de los diferentes puntos de vista proporcionados. También, permite la detección de fuentes potenciales de error relacionados con la tecnología.

3.5. Lecciones aprendidas

Este capítulo ha permitido definir concretamente la terminología relacionada con los procesos de modernización software y entender que la modernización debe formar parte de la evolución de un sistema

	cliente	servidor	artefacto	estático	dinámico	objetivo	representación	presentación	navegación	modelo	web Site	web app
RE-RMM	X		HTML	X		reingeniería	RMM	X	X		X	
WANDA	X	X	HTML, PHP		X	diseño	Web UML	X	X	X		X
WARE	X	X	HTML	X	X	diseño	UML	X	X	X		X
ReWeb	X		HTML	X		diseño	Grafo	X	X		X	
VAQUISTA	X		HTML	X		migración	M. Presentación	X	X			X
TERESA	X		HTML	X		análisis	M. Tareas	X	X		X	
Revangie	X		HTML	X		diseño	M. Interfaz	X	X			X
JSPick		X	JSP	X		diseño	Propia	X	X			X
RE-UWA	X		HTML	X	X	diseño	UWA	X	X	X		X
ASP.NET		X	ASPX	X		diseño	WebML	X	X			X

Tabla 3.1: Comparativa de propuestas de ingeniería inversa de aplicaciones web

software. Se ha definido con detalle el concepto de reingeniería software, detallando cada una de sus tres fases principales: ingeniería inversa, reestructuración e ingeniería directa.

Posteriormente, se ha revisado la modernización software desde la perspectiva de la Ingeniería Dirigida por Modelos, exponiendo las ventajas que se desprenden del uso de métodos y herramientas dirigidas por modelos para la especificación de procesos de modernización software. Básicamente, mediante la definición de metamodelos y cadenas de transformación se puede lograr la especificación de un proceso de modernización software dirigido por modelos. En este proceso, se pueden reutilizar metamodelos y transformaciones ya definidos con otros objetivos, reduciendo los costes del proceso. El uso de IDM en la modernización presenta claras ventajas, pero también presenta inconvenientes y limitaciones que es necesario entender cómo afectan al proceso de modernización concreto en fases tempranas de su desarrollo.

Por otro lado, dado el dominio de aplicación del trabajo de esta tesis, se ha realizado un estudio de las principales propuestas de ingeniería inversa de aplicaciones web. La Tabla 3.1 presenta un resumen de este estudio. En este resumen se comparan las distintas propuestas analizadas según los siguientes aspectos:

1. En qué lado se realiza el proceso de ingeniería inversa: en el cliente o en el servidor.

2. Tipo de artefacto de la aplicación web heredada que se analiza.

3. Tipo de análisis: estático o dinámico.

4. Objetivo del proceso de ingeniería inversa: diseño, reingeniería, migración, análisis.

5. Representación elegida para la especificación de los modelos generados como resultado del proceso de ingeniería inversa.

6. Capa conceptual de la aplicación web analizada en el proceso de ingeniería inversa: presentación, navegación, modelo.

7. Entrada del proceso: web site o aplicación web

De este estudio, se puede destacar las siguientes conclusiones:

- Las propuestas de ingeniería inversa de aplicaciones web se han propuesto tradicionalmente desde el procesado de las páginas HTML recibidas por el cliente en la interacción con la aplicación web heredada.

- La gran mayoría de propuestas sólo plantea la aplicación de análisis estáticos de los artefactos de la aplicación heredada.

- El objetivo del proceso de ingeniería inversa en la mayor parte de los casos consiste en la obtención del diseño para tareas de re-documentación de la aplicación web heredada.

- La gran mayoría de propuestas se centran en las capas de presentación y navegación, que son las más características de este tipo de aplicaciones. Y no contemplan la extracción del modelo ni de la lógica de negocio asociada.

- La mayoría de las propuestas soporta como entrada del proceso una aplicación web.

- La gran mayoría no utiliza métodos y herramientas propias de la IDM.

En definitiva, todas las propuestas estudiadas aportan gran valor al proceso de modernización a definir en este trabajo de tesis. Sin embargo, la aproximación a seguir en este trabajo de tesis difiere en gran medida de la mayoría de las mismas. La primera diferencia es que como entrada (dominio) se consideran aplicaciones web desarrolladas con un framework de aplicación web basado en el patrón Modelo-Vista-Controlador. Esta diferencia es fundamental porque nuestra propuesta de ingeniería inversa tiene como objetivo representar una aplicación web en base a los conceptos de los frameworks que implementan ese patrón. Y, por lo tanto, el proceso de extracción intentará encontrar estos conceptos en la aplicación web de entrada. La segunda diferencia consiste en el uso de métodos y herramientas de la IDM para la definición de este proceso y, por tanto, la especificación de metamodelos, la creación de cadenas de transformaciones de modelos, así como el desarrollo de inyectores específicos para la tecnología considerada.

Parte II

Propuesta

Capítulo 4

Proceso de Modernización

4.1. Dominio

Como ya se apuntó en la introducción de esta memoria, el dominio de este trabajo está formado por aplicaciones Web heredadas desarrolladas mediante un Framework de Aplicación Web (FAW) dentro de la pila tecnológica definida dentro de la plataforma JEE. La primera versión de esta plataforma (v1.2) data de 1999[1] y ha sido un referente desde entonces en el desarrollo de sistemas empresariales distribuidos y multicapa (multi-tier).

El contexto en el que se encuadra este trabajo se circunscribe al dominio de aplicaciones web desarrolladas mediante un framework que esté basado en el patrón de diseño Model-View-Controller (MVC). Normalmente, estos frameworks proporcionan elementos y estructuras para implementar la interacción entre el usuario y el sistema de información subyacente. De esta manera, siguiendo la terminología MVC, suelen imponer un mecanismo de acceso al sistema de información (acceso al modelo), un mecanismo de composición y presentación de la información al usuario (elementos de las vistas), y un mecanismo de control que permite asociar a cada petición un manejador adecuado y devolver una respuesta al cliente.

Según (Shan & Hua 2006), aunque la gran mayoría de frameworks para el desarrollo de aplicaciones web Java está basado en el patrón MVC, se pueden distinguir cuatro aproximaciones distintas en su diseño:

1. Framework basado en peticiones.
2. Framework basado en componentes.
3. Framework híbrido.
4. Meta frameworks.

Un framework basado en peticiones presenta un diseño muy cercano a la especificación original de CGI: usa controladores y acciones que manejan directamente las peticiones. Las peticiones no tienen estado.

[1] http://en.wikipedia.org/wiki/Java_EE_version_history

Un nivel básico de mantenimiento del estado se alcanza con la introducción de sesiones en el lado del servidor. Los frameworks de este tipo se diferencian entre sí básicamente en la aproximación que siguen, por un lado, para mapear lógica con URLs y, por otro, para encapsular los datos y proporcionárselos a los manejadores.

Los frameworks basados en componentes se caracterizan por abstraer los detalles de la gestión de peticiones y encapsular la lógica en componentes reutilizables, que suelen mantenerse independientes de la tecnología web subyacente. El framework se encarga de mantener el estado en base a los datos contenidos en cada instancia activa de un componente. Suelen contar con algún método de gestión de eventos que hace que su modelo de desarrollo sea muy similar al propuesto por las herramientas de desarrollo de interfaces gráficos de usuario para aplicaciones de escritorio. Los frameworks de este tipo se diferencian entre sí básicamente en su API de componentes y en la forma de integración de los mismos.

Un framework híbrido combina los modelos anteriores (petición y componentes) gestionando el flujo de control y datos en un modelo basado en peticiones. Los desarrolladores se mantienen cercanos a la arquitectura de aplicaciones CGI y tienen un control total sobre las URLs, formularios, parámetros, cookies y pathinfos. Sin embargo, en vez de mapear acciones y controladores directamente a la petición, este tipo de frameworks proporciona un modelo de componentes que permite una gestión de grano más fino: páginas individuales, peticiones interceptadas, portlets y widgets integrables. Los componentes suelen seguir el patrón GoF Composite, de manera que un paquete de componentes se percibe como un nuevo componente. Este tipo de frameworks combina el método de reutilización de los frameworks basados en componentes con el nivel de control de las aproximaciones basadas en peticiones.

Un Meta Framework presenta un conjunto de interfaces centrales para servicios comunes y una arquitectura extensible que permite integrar componentes y servicios. Suelen estar basados en alguna implementación del patrón Inversión del Control [2] para la separación de aspectos que favorece la incorporación flexible de otros frameworks y componentes.

La propuesta de proceso de modernización presentada en este trabajo ha sido diseñada fundamentalmente para aplicaciones desarrolladas con la primera familia de frameworks, aquellos cuyo diseño está basado en peticiones. Y, en este sentido, para la presentación de los métodos, herramientas y resultados desarrollados en este trabajo se ha utilizado como referencia el framework Struts 1[3]. No obstante, la propuesta se presenta en dos planos diferentes: un plano conceptual y un plano de aplicación. Desde el plano conceptual, la propuesta intenta abarcar aplicaciones desarrolladas mediante frameworks de desarrollo web basados en el patrón MVC dentro de las tres primeras familias presentadas en la taxonomía anterior. Y por otro lado, desde el punto de vista de aplicación, se realiza una instanciación de esta propuesta conceptual en el caso de un framework concreto de la primera familia para poder evaluarla y refinarla.

[2] http://martinfowler.com/articles/injection.html
[3] http://struts.apache.org

4.2. Objetivos de una modernización

Siguiendo la especificación de Modernización Dirigida por Arquitectura (Architecture-Driven Modernization, ADM) de la OMG (Ulrich 2010), los objetivos fundamentales de un proceso de modernización se pueden resumir en los siguientes:

- Descubrimiento del conocimiento. El conocimiento que puede ser extraído de una aplicación web heredada puede ser de muy distinta naturaleza según los objetivos perseguidos por dicha extracción. En nuestro caso estamos interesados en recuperar la información que tenga que ver con la definición de la capa web a través de un framework de aplicaciones web. A partir de esta información pretendemos descubrir los patrones de interacción web que aparecen en la aplicación web heredada. Para obtener este conocimiento se deben definir tanto el sistema de representación como los procesos y herramientas de extracción.

- Definición de la arquitectura del sistema objetivo de la modernización. Básicamente, como se ilustra en la figura 4.1, se considera que el sistema final estará formado por dos componentes fundamentales: un cliente RIA y una capa de conexión orientada a servicios en la parte del servidor. Con respecto al primer componente, un cliente RIA puede ser caracterizado conceptualmente por una interfaz de usuario rica, de características similares a los de las aplicaciones de escritorio, y su infraestructura: un almacén de datos en el lado cliente, la posibilidad de ejecutar parte de la lógica en el cliente, y una capa de comunicación para gestionar los diferentes modos de operación y comunicación del cliente RIA con el servidor (offline/online, síncrono/asíncrono). Por otro lado, en la parte del servidor, es necesario especificar una capa de comunicación entre el nuevo cliente RIA y la lógica de negocio original del sistema heredado. Esta nueva capa es la responsable de modularizar todos los aspectos relacionados con los nuevos escenarios de interacción definidos por clientes RIA, i.e. interacciones e intercambio de datos orientados a servicio. Como se puede apreciar en la figura 4.1, nuestra propuesta no supone la sustitución de la aplicación web heredada, sino la adición de un nuevo sistema frontal para interaccionar con el sistema de información subyacente.

- Especificación de los pasos de transformación del sistema original al sistema objetivo. Nuestra propuesta conceptual define un marco de referencia que establece una secuencia de actividades concreta para la transformación de la aplicación web heredada a un sistema que permita la interacción a través de un cliente RIA.

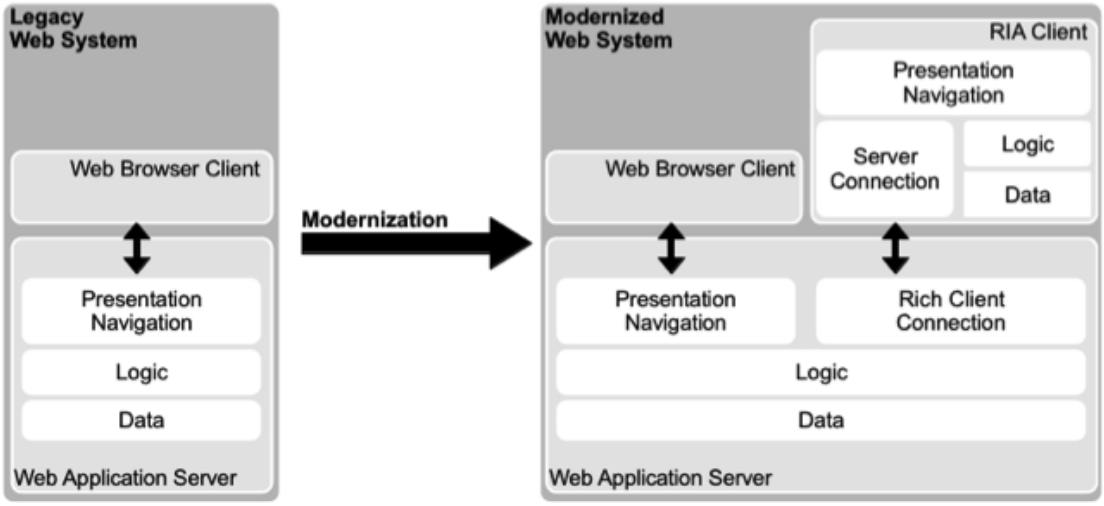


Figura 4.1: Arquitectura del Sistema Objetivo

4.3. Un marco de referencia para procesos de modernización

Como se ha comentado anteriormente, la propuesta de este trabajo se realiza en dos planos diferentes: conceptual y de aplicación. En este apartado, se va a detallar la conceptualización del proceso de modernización en base a la definición de un marco de referencia de proceso que identifica las principales fases y actividades a tener en cuenta, así como sus objetivos, alternativas y dependencias. El marco de referencia define la estructura general de la propuesta y sus fases y actividades principales, favoreciendo la flexibilidad ante posibles adaptaciones del mismo al definir un proceso de modernización concreto en un contexto determinado. Como se presenta en la figura 4.2, el marco de referencia de proceso sigue la estructura general de un proceso tradicional de reingeniería software, presentando una organización básica en tres etapas fundamentales: (1) la etapa de ingeniería inversa; (2) la etapa de refinado o reestructuración; y (3) la etapa de ingeniería directa. La primera etapa tiene como objetivo la extracción y representación conceptual del conocimiento de la aplicación web heredada. La segunda etapa se centra en refinar y mejorar la representación conceptual mediante la integración de la información obtenida en los diferentes procesos de extracción. Mientras que la tercera etapa persigue la generación del sistema final a partir del conocimiento obtenido en la primera y, adicionalmente, los requisitos específicos de la modernización.

En los siguientes apartados, se analiza con más detalle cada una de estas fases.

4.3.1. Ingeniería inversa

Con respecto a la etapa de ingeniería inversa, las actividades fundamentales se han organizado dentro de tres planes de acción diferentes, identificados como *RE-C-D*, *RE-S-S* y *RE-S-D* en la figura 4.2. Como se puede apreciar, el primer plan de acción se desarrolla en la parte cliente, mientras que los otros dos se implementan en la parte servidora. El propósito de estos planes de acción paralelos es obtener infor-

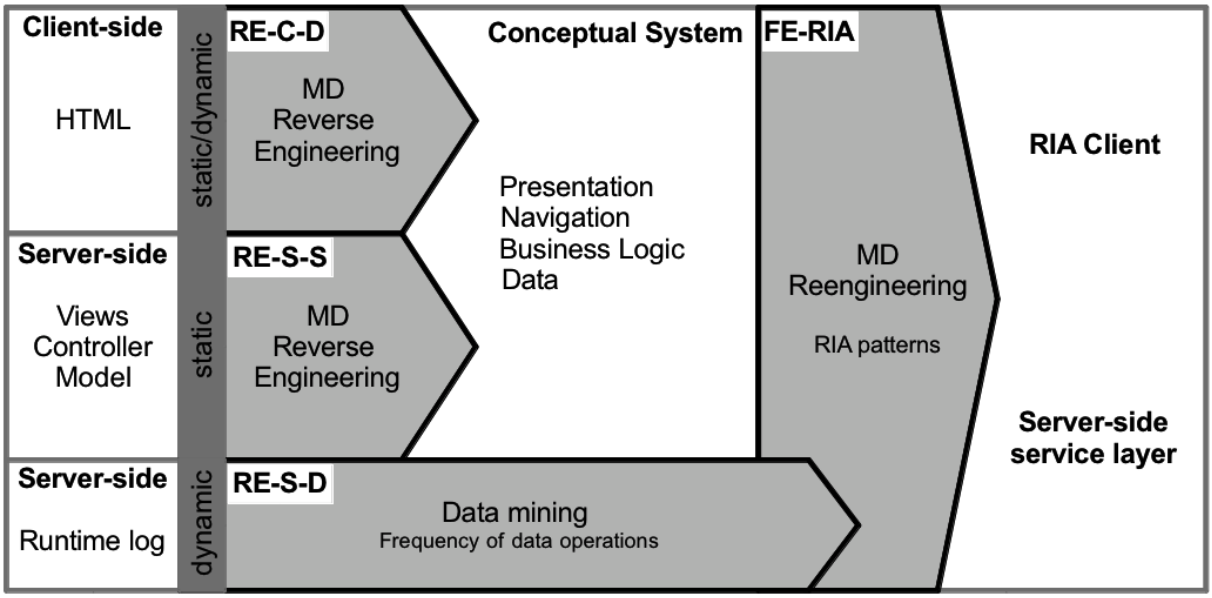


Figura 4.2: Resumen del marco del proceso MIGRARIA

Identificador	Localización	Naturaleza
RE-C-D	Cliente	Dinámico
RE-S-S	Servidor	Estático
RE-S-D	Servidor	Dinámico

Tabla 4.1: Planes de acción de ingeniería inversa

mación del sistema heredado desde perspectivas diferentes pero complementarias. Las dos dimensiones que caracterizan a cada plan de acción definido, por lo tanto, son:

- localización de las actividades, se puede realizar en la parte cliente o en la parte servidora,
- naturaleza de las actividades, fundamentalmente se distingue entre actividades de análisis estático y dinámico.

La tabla 4.1 indica el significado del identificador de cada uno de los planes de acción definidos.

Como señala (Di Lucca *et al.* 2001), el comportamiento de una aplicación web proviene de los elementos tanto estáticos como dinámicos del sistema, como en el caso de las aplicaciones software tradicionales; por lo tanto, un proceso de ingeniería inversa de aplicaciones web debe recuperar:

- la arquitectura estática,
- las interacciones dinámicas,
- y el comportamiento.

Los análisis estáticos no requieren la ejecución de la aplicación. Fundamentalmente, tienen como objetivo recuperar la arquitectura de componentes y sus relaciones estáticas. Este tipo de análisis se suele centrar en el procesamiento de archivos HTML, archivos script, estructuras de directorios, etc. Las páginas web de la aplicación se descomponen en sus elementos base que son identificados y clasificados.

Establecer la relación entre estos elementos y las entidades del modelo de datos es una actividad fundamental del proceso. Las actividades fundamentales que forman parte de un análisis estático son:

- Inventario de recursos y artefactos a analizar.
- Localización de elementos y componentes fundamentales.
- Recuperación de las relaciones entre los elementos.
- Generación de una representación intermedia.

Por su parte, los análisis dinámicos se pueden realizar siguiendo dos estrategias diferentes:

- Realizar el análisis dinámico en paralelo al estático, dos puntos de vista diferentes.
- Realizar el análisis dinámico partiendo de los resultados del estático, complementando a éste.

En cualquiera de las alternativas, el objetivo es recuperar las interacciones dinámicas entre los elementos de la aplicación web en ejecución. Se realiza una monitorización de la ejecución de la aplicación web, o bien mediante instrumentalización de la misma (modificación de la aplicación para registrar interacciones), o bien mediante la captura del tráfico HTTP o mediante el procesamiento de los archivos de registro de ejecución del servidor web y el servidor de aplicaciones. Las actividades base que forman parte de un análisis dinámico son:

- Ejecución de la aplicación web.
- Monitorización de la ejecución.
- Recuperación de la arquitectura de componentes y elementos, si no se dispone de ella antes.
- Representación de las interacciones dinámicas entre los componentes y elementos de la aplicación web.

En este sentido, se definen los tres planes de acción fundamentales en esta etapa apuntados en la tabla 4.1, que conllevan la realización de análisis estáticos y dinámicos para poder capturar el comportamiento de la aplicación web heredada.

4.3.2. Refinado y reestructuración

La etapa de refinado y reestructuración tiene como objetivo general la integración de la información derivada de los diferentes procesos de extracción de información. De manera que se pueda disponer de la información más completa para apoyar el proceso de ingeniería directa.

Otros objetivos específicos de esta etapa son:

- Eliminar redundancias en los modelos conceptuales generados.

- Eliminar ambigüedades en los modelos conceptuales generados.

Dentro de esta etapa se puede definir también un proceso de transformación a una de las propuestas de Ingeniería Web Dirigida por Modelos (IWDM), como parte el proceso de modernización.

4.3.3. Ingeniería directa

Una vez que se dispone de una representación conceptual del conocimiento extraído de la aplicación web heredada, se establece una nueva etapa en el marco de referencia de proceso en la que se pretende derivar la generación del sistema final definido anteriormente (cliente RIA y capa de conexión en servidor) mediante el uso de técnicas, métodos y herramientas propios del desarrollo software dirigido por modelos. Esta tercera etapa recibe el identificador FE-RIA en la figura 4.2. Las actividades fundamentales que pueden formar parte de esta etapa son:

- Identificación de posibles patrones RIA a aplicar en la representación conceptual del sistema. Estos patrones pueden provenir de:
  - la aparición de determinados componentes y relaciones en la representación conceptual,
  - o la introducción de nuevos requisitos para el nuevo sistema no contemplados en el original.
- Refinado de los modelos obtenidos mediante la aplicación de los patrones seleccionados.
- Transformación a una representación relacionada con una plataforma RIA concreta, como por ejemplo Google Web Toolkit[4].
- Generación del código final de la aplicación.

Como se desprende de la complejidad de las actividades relacionadas, esta etapa puede requerir un alto grado de intervención humana. Para mitigar en parte este condicionante, se ha creado el plan de acción RE-S-D cuyo objetivo es proporcionar mayor información sobre las interacciones dinámicas de la aplicación web original de manera que se puedan apoyar las decisiones de introducción de patrones RIA y composición del sistema final. De esta manera, se pretende, por un lado, evitar que la generación del sistema final se base en un simple mapeo uno a uno entre patrones (y componentes) web y RIA; y por otro, proporcionar información suficiente para disminuir el esfuerzo humano a realizar en esta etapa.

4.4. Un proceso de modernización

Una vez definido un marco de referencia para procesos de modernización, este apartado se centra en la descripción del proceso concreto seguido en este trabajo (plano de aplicación). Este proceso instancia, por tanto, una de las posibles versiones de procesos de modernización dentro del rango establecido por el

[4] http://www.gwtproject.org/

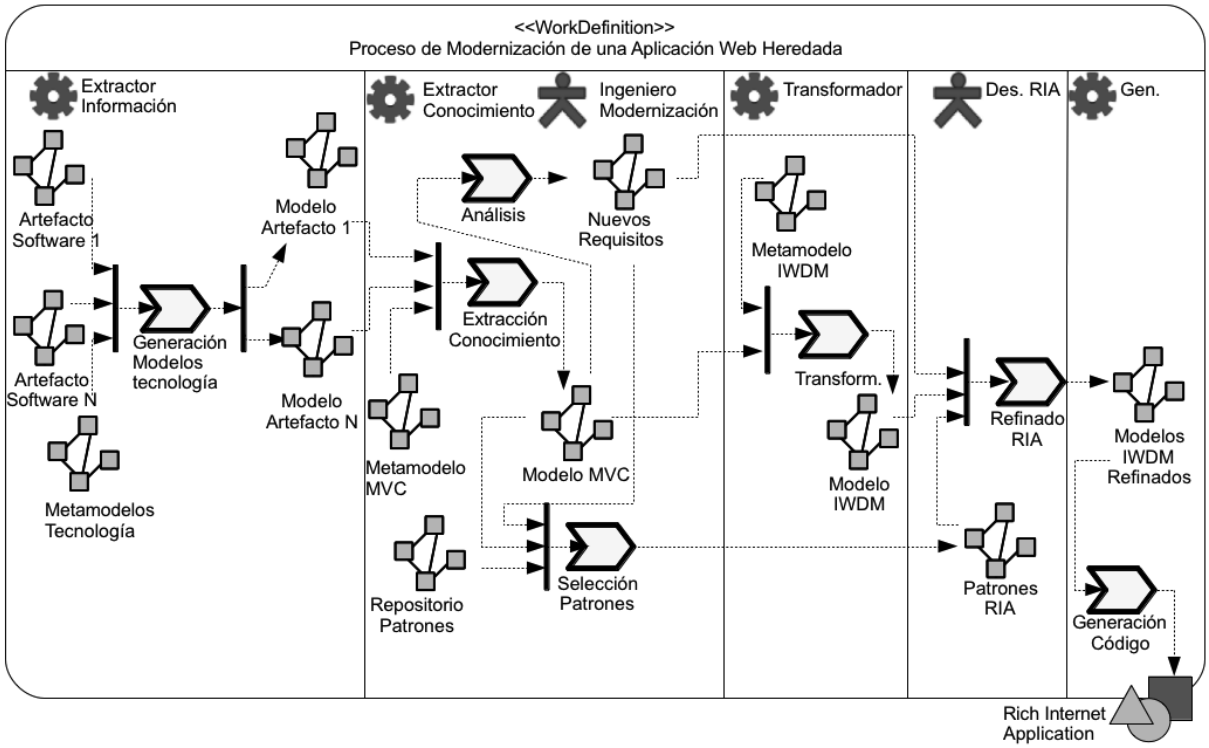


Figura 4.3: Resumen del proceso de modernización

marco de trabajo. Esta versión constituye una primera evolución del proyecto MIGRARIA y, por tanto, la base sobre la que desarrollar versiones más completas de este proceso de modernización.

Como se ha comentado, basado en el marco de trabajo definido en el apartado anterior, el proceso de modernización presentado en este trabajo está organizado en cinco fases secuenciales fundamentales, como aparece representado en la figura 4.3:

1. extracción de la información mediante análisis estáticos de los principales artefactos software que conforman la aplicación web heredada;

2. representación conceptual del conocimiento derivado de esta información dentro del dominio de frameworks de aplicaciones web basados en el patrón MVC (modelos conceptuales);

3. transformación de la representación a una propuesta IWDM;

4. aplicación de patrones RIA a los modelos conceptuales del sistema;

5. y generación de código.

Resulta interesante reseñar que las primeras fases incluyen las actividades que realizan el proceso de ingeniería inversa de la aplicación web heredada; mientras que las últimas conllevan la especificación de las actividades de ingeniería directa para generar el código del sistema final.

Como se aprecia en la figura 4.3, se trata de un proceso semiautomático en el que se intenta reducir en lo posible la necesidad de apoyo humano.

A continuación, se describen las principales actividades de las distintas fases con más detalle.

4.4.1. Extracción de información

Para la extracción de información de la aplicación web heredada se han desarrollado procesos de análisis estático de las mismas. Aunque está previsto realizar análisis dinámicos para complementar la información extraída, se trata de trabajo todavía en desarrollo y, por lo tanto, no se dispone de datos suficientes de evaluación. Este trabajo formará parte de evoluciones futuras del proceso. Por lo tanto, dentro del marco definido anteriormente, sólo hemos seguido una de las alternativas definidas para realizar análisis de la aplicación web con el objetivo de extraer información.

Por otro lado, es necesario apuntar que esta fase es muy dependiente de las tecnologías de implementación usadas en el desarrollo de la aplicación web heredada: lenguajes, frameworks, etc. Por lo tanto, las actividades definidas dentro de la misma ven reducida su reutilización a escenarios en los que se den las mismas tecnologías web. Estos escenarios vienen establecidos fundamentalmente por el FAW usado en el desarrollo de la aplicación web heredada. Si bien, su definición final está muy ligada a la tecnología, su especificación conceptual y organización para constituir un proceso sí representa un activo con un mayor grado de reusabilidad. En este sentido, las principales actividades que conforman esta fase son:

- Generación de modelos dependientes del lenguaje mediante transformaciones de texto a modelo.
- Refinado de los modelos en base a los conceptos definidos en el FAW.

Como ya se ha puesto de manifiesto anteriormente, este trabajo tiene como línea fundamental el uso de métodos, técnicas y herramientas propios de la ingeniería dirigida por modelos. En este sentido, las primeras actividades del proceso de modernización están dedicadas a obtener una representación en base a modelos de los artefactos software que componen la aplicación web heredada y que deben ser analizados. Por lo tanto, una vez realizado el inventario de estos recursos, se realiza una serie de actividades de transformación text2model. De esta manera, pretendemos reducir la complejidad operativa del proceso de modernización al pasar de la heterogeneidad del mundo de las tecnologías de programación web a la homogeneidad del mundo del modelado. Estas actividades son, por tanto, dependientes de la tecnología, por ejemplo del lenguaje JAVA, pero, en general, son independientes del FAW utilizado. De esta manera, estas actividades son reutilizables en escenarios en los que se trabaje con diferentes FAW pero dentro de la misma tecnología de implementación.

Una vez generada esta primera colección de modelos dependientes de la tecnología, se llevan a cabo una serie de actividades de refinado que tienen por objetivo obtener una representación que utilice los conceptos y relaciones definidas en el FAW utilizado. Para ello es necesario definir un conjunto de metamodelos que den soporte a la representación de estos conceptos. Normalmente, estos metamodelos se pueden definir como extensiones de los metamodelos de las tecnologías de implementación, por ejemplo, un metamodelo que extienda el metamodelo de las JavaServer Pages para incluir nuevas etiquetas definidas en el FAW.

4.4.2. Representación del conocimiento

Como se ha comentado en el apartado anterior, la información extraída de la aplicación web es de naturaleza muy variada y se encuentra repartida en diferentes artefactos software resultado de la extracción. En este trabajo, el concepto fundamental con el que se quiere trabajar es la interacción entre cliente y servidor. En este sentido, se ha definido una notación intermedia para poder representar este conocimiento dentro del dominio de aplicaciones web desarrolladas con frameworks web basados en el patrón MVC. Esta conceptualización recibe el nombre de MIGRARIA MVC. En el capítulo 5 se presenta esta notación con más detalle. Tal como sugiere ADM, se usan transformaciones modelo a modelo para obtener de forma automática esta representación a partir de los modelos obtenidos en las actividades de extracción de información.

Esta representación está pensada para escenarios de modernización y, por lo tanto, no posee un nivel de abstracción tan alto como las principales propuestas de Ingeniería Web Dirigida por Modelos (IWDM), cuyos escenarios de aplicación se corresponden con la ingeniería directa. Sin embargo, dadas las ventajas que conlleva el uso de una propuesta madura de IWDM, en este trabajo se ha definido una proyección de la representación basada en el FAW a una de las representaciones definidas en las propuestas IWDM. Entre otras ventajas, que esta proyección aporta al ingeniero de modernización, es interesante señalar las siguientes:

- El sistema estará representado en una notación validada profesionalmente y con un grado de abstracción mayor, permitiendo un grado adicional de refinado en base a la nueva notación.
- Las propuestas IWDM más maduras cuentan con herramientas de soporte del desarrollo, que pueden resultar de utilidad para asistir o automatizar parte de las tareas del ingeniero de modernización.
- Contar con un motor de generación de código a partir de una notación conceptual también puede ser de utilidad a la hora de afrontar la etapa de ingeniería directa del proceso de modernización.

4.4.3. Refinado RIA

Durante esta fase se realiza la reestructuración de los modelos conceptuales obtenidos en la etapa anterior fundamentalmente mediante la aplicación de patrones RIA. En esta propuesta, se genera una representación de la aplicación web heredada según una propuesta de IWDM. Por lo tanto, como primera alternativa de refinado, se pueden usar los elementos RIA que defina la propuesta IWDM utilizada. Sin embargo, para dotar de un grado de sistematización mayor a esta fase se ha decidido seguir una propuesta basada en el reconocimiento de patrones. De esta manera, ante determinados patrones presentes en la representación IWDM de la aplicación heredada, se propone la reestructuración de los modelos afectados con la introducción de los componentes que implementan un determinado patrón RIA. Por lo tanto, un

requisito fundamental en esta fase es disponer de un repositorio de patrones RIA que proporcionen una especificación según la propuesta IWDM elegida. Adicionalmente, en esta fase se puede realizar una especificación adicional de requisitos para el sistema final a generar. Estos requisitos corresponden a funcionalidad o características no funcionales que no son directamente recuperables de la información proporcionada por la fase anterior.

En la versión actual de la propuesta, esta fase debe ser realizada por el ingeniero de modernización a RIA prácticamente en su totalidad, aunque asistido por un conjunto de herramientas de apoyo. Básicamente las actividades a realizar por el ingeniero de modernización son:

- Detección de patrones en los modelos IWDM de la aplicación heredada.
- Selección de un patrón para un caso determinado entre el conjunto de candidatos (opcionalmente, apoyado por una fase de captura de requisitos).
- Reestructuración de los modelos afectados.
- Verificación de la corrección de los modelos reestructurados.

Normalmente, un proceso de modernización conllevará la definición de nuevos requisitos. Por una parte, estos requisitos definirán el alcance del proceso mismo de modernización, estableciendo qué parte del sistema original debe ser modernizada. Mientras que, por otra parte, estos requisitos establecerán las características concretas que debe cumplir el sistema destino, entre otras las características RIA. En este último caso, por tanto, los requisitos especificados servirán para identificar los patrones RIA a aplicar para la obtención del sistema destino.

Para apoyar este proceso de selección de patrones RIA a partir de la especificación de los requisitos del sistema final, dentro de este trabajo, hemos desarrollado una propuesta basada en la utilización de matrices de trazabilidad que permiten relacionar los requisitos de modernización con características y patrones RIA concretos. La figura 4.4 presenta una vista general de este proceso. Como se puede apreciar, este proceso está compuesto de tres actividades principales:

- Identificación de requisitos. Básicamente, en esta actividad, se obtiene una especificación de requisitos del sistema destino. Esta actividad se apoya en la información obtenida a partir de la realización de entrevistas al cliente y, además, de la información disponible del sistema. Esta última puede estar simplemente compuesta de la documentación disponible del sistema. No obstante, es recomendable complementar esta documentación con información obtenida en la fase de ingeniería inversa.
- Construcción de las matrices de trazabilidad de requisitos a partir de los requisitos especificados y de una colección de características RIA conceptuales, que relacionen las características necesarias para satisfacer un determinado requisito.

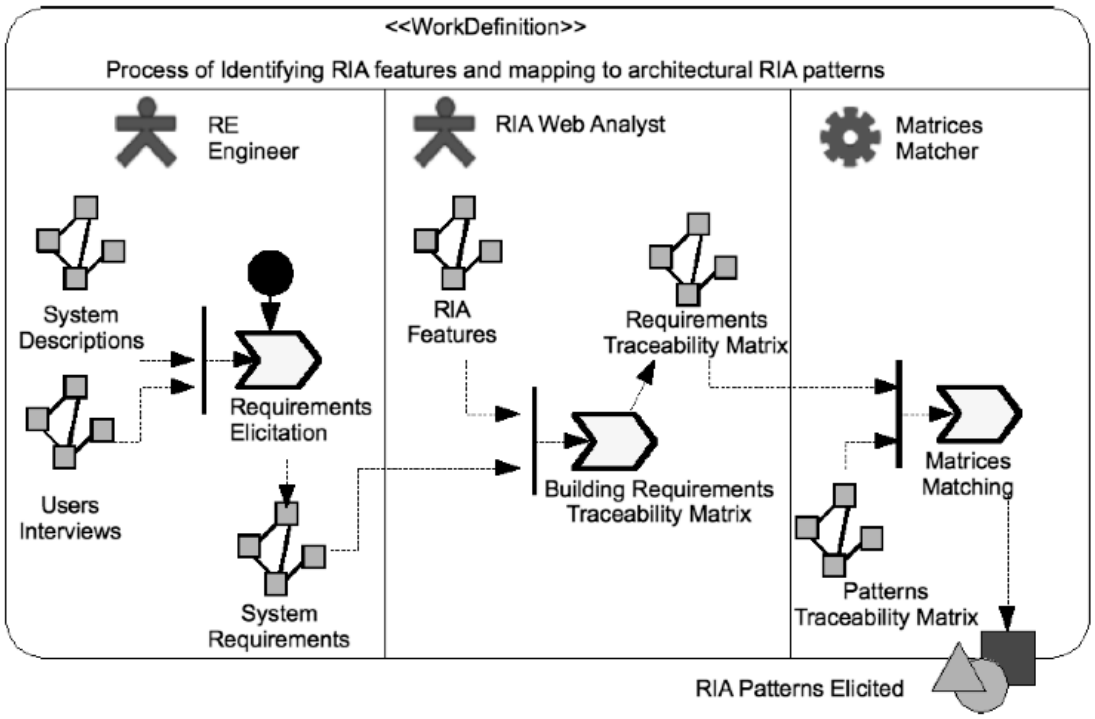


Figura 4.4: Proceso de identificación de características y patrones RIA a partir de requisitos

- Selección de patrones RIA. Partiendo, por un lado, de un repositorio de patrones RIA en el que cada patrón está identificado por el conjunto de características RIA que engloba y, por otro lado, de las matrices generadas en la actividad anterior, se lleva a cabo una actividad de mapeo que devuelve el conjunto de patrones RIA candidatos para satisfacer los requisitos identificados en el sistema final.

4.4.4. Generación de código

Durante esta fase, se generará el código del sistema final. En la Ingeniería Dirigida por Modelos existen muchas alternativas para generar código, las cuales suelen tener en común la presencia de un motor de generación.

En su versión actual y con referencia al cliente RIA, dentro de este trabajo no se ha abordado directamente la construcción de un motor de generación de código, sino que se ha optado por seleccionar y hacer uso de sistemas contrastados que, en la mayoría de los casos, se corresponden con soluciones profesionales.

Algunas de las propuestas IWDM más maduras comparten dos características fundamentales para el proceso de modernización aquí definido:

- han sido extendidas para dar soporte a la especificación de características RIA;
- y proporcionan una herramienta de desarrollo que suele incluir un motor de generación de código final.

En este sentido, sin descartar la idea de desarrollar herramientas propias para la generación de código dentro del proyecto MIGRARIA, en esta primera aproximación se ha decidido optar por herramientas ya presentes que nos asegurasen una cobertura suficiente de nuestro sistema final. A partir del trabajo presentado en el capítulo 2, las opciones disponibles que más encajaban con esta aproximación son:

1. La herramienta RUX-Tool, (Linaje *et al.* 2009), que está basada en en el paradigma de Ingeniería Dirigida por Modelos y da soporte a la especificación y generación automática de código de interfaces de usuario para Rich Internet Applications (RIAs) con capacidades multidispositivo y multiplataforma. Esta herramienta se apoya en otras que le proporcionen la especificación y generación de código de la estructura de contenidos y la lógica de negocio de la aplicación, como por ejemplo, WebRatio [5], que usa como representación la propuesta WebML.

2. La herramienta OOH4RIA[6], que da soporte a la aproximación OOH4RIA basada en el paradigma de Ingeniería Dirigida por Modelos para el desarrollo de Rich Internet Applications (RIAs). Esta herramienta proporciona un proceso de desarrollo basado en un conjunto de modelos y transformaciones que permiten generar el código final de RIAs.

Por otra parte, con respecto a la generación de la capa de conexión entre el nuevo cliente RIA y el servidor, actualmente se ha desarrollado una propuesta que permite su generación automática dentro del paradigma REST (Fielding 2000) a partir de la descripción del sistema heredado.

4.5. Resumen

Este capítulo presenta el marco de referencia para la definición de procesos de modernización de aplicaciones web heredadas a RIA, que se especifica como un proceso de reingeniería con tres etapas fundamentales: ingeniería inversa, reestructuración e ingeniería directa. Posteriormente, se ha especificado un proceso concreto de modernización como instancia de este marco de referencia, haciendo especial hincapié en la descripción de las actividades fundamentales del mismo. Cada una de estas actividades son especificadas con detalle en los siguientes capítulos de la memoria. Una atención especial se les ha dedicado a las actividades de extracción de la información y representación del conocimiento, porque son las actividades fundamentales de la etapa de ingeniería inversa que ocupa gran parte de este trabajo de tesis.

La misión de este capítulo es, por tanto, definir un proceso de modernización de aplicaciones web heredadas a RIA haciendo uso de métodos y herramientas de la IDM dentro del dominio de aplicación definido. Este dominio de aplicación se caracteriza por aplicaciones web heredadas desarrolladas mediante un framework de aplicaciones web basado en peticiones y diseñado a partir del patrón Modelo-Vista-Controlador.

[5] http://www.webratio.com/
[6] http://artemisa.dlsi.ua.es/ooh4ria/

Capítulo 5

Representación Conceptual

> "The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise". Dijkstra

5.1. Necesidad de una representación

Considerando que el objetivo de este trabajo consiste en definir un proceso marco de modernización de aplicaciones Web heredadas a RIA, se estima necesario definir un sistema de representación específico que ocupe un lugar intermedio, y actúe de puente, entre el espacio tecnológico concreto definido por la aplicación web heredada, y el espacio de modelado conceptual, definido por las actuales propuestas IWDM. Se debe tener en cuenta que las propuestas de IWDM están pensadas para la generación de aplicaciones web mediante ingeniería directa (generación de código) y no para ser usadas en escenarios de modernización. Esta representación debe permitir especificar los conceptos y patrones propios del modelado de la interacción Web desde un plano definido por los elementos de los frameworks de aplicaciones Web basados en el patrón de diseño *Model-View-Controller* (MVC). Se establecen, por lo tanto, como requisitos fundamentales para esta especificación los siguientes:

1. Representar la aplicación Web heredada con un gran nivel de detalle, proporcionando una visión concisa, uniforme e integradora de todos los elementos que la definen.

2. Representar elementos propios de frameworks de aplicaciones Web, minimizando la dependencia con tecnología Web.

3. Simplificar la generación de representaciones a un mayor nivel de abstracción, como por ejemplo mediante una propuesta IWDM concreta, aportando una representación con gran nivel de detalle pero independiente de la tecnología final.

4. Simplificar el proceso de modernización de una aplicación Web heredada, proporcionando la información necesaria para la modernización y manteniendo la trazabilidad modelo-código.

5. Mantenerse independiente del framework de aplicaciones web usado para el desarrollo de la aplicación web heredada.

6. Permitir representar parcialmente una aplicación Web heredada, evitando restricciones propias de propuestas de IWDM para ingeniería directa.

Este capítulo presenta la representación conceptual desarrollada en este trabajo, que recibe el nombre de MIGRARIA-MVC. centrándose la descripción de esta notación desde el punto de vista del patrón MVC.

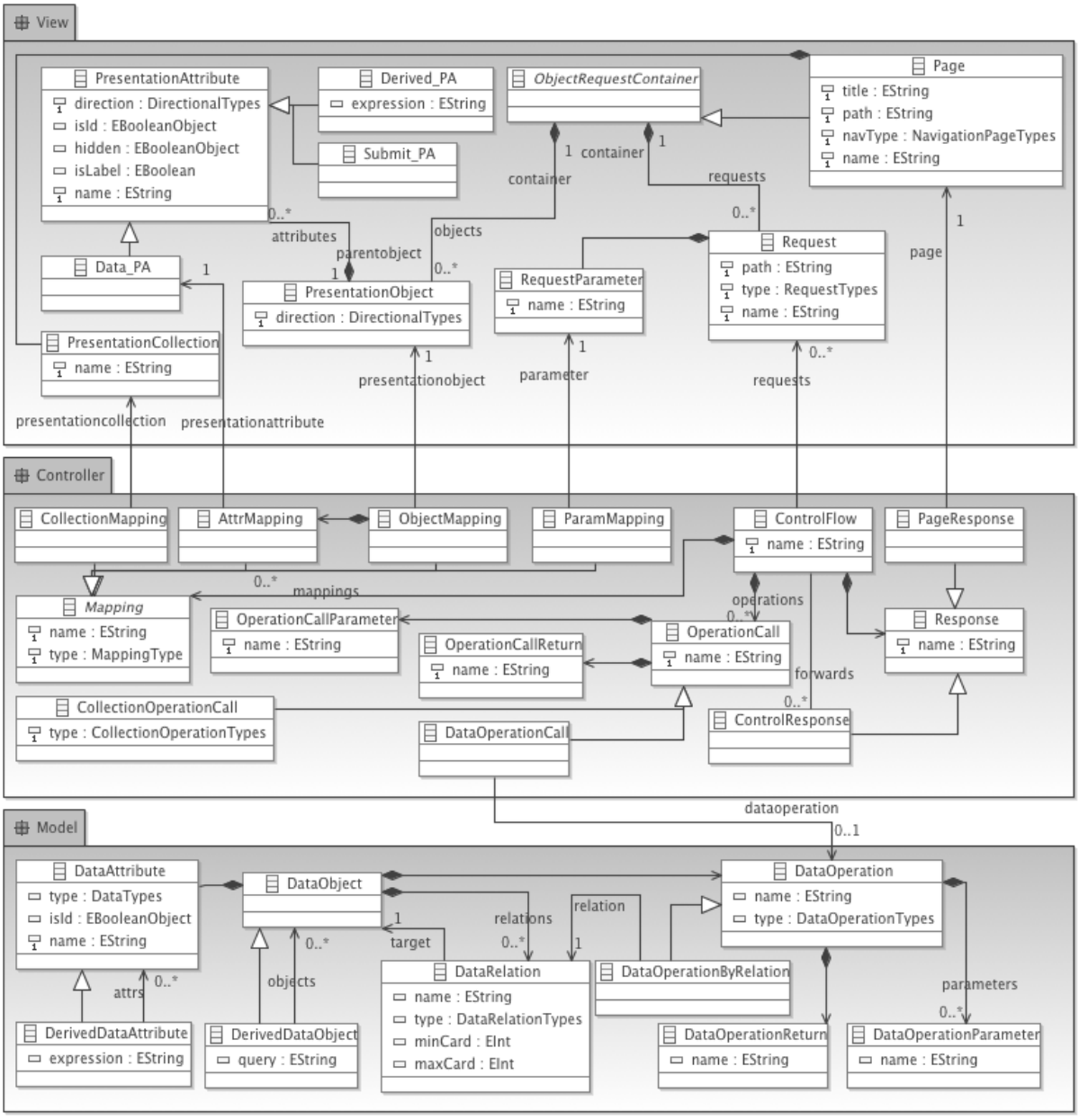


Figura 5.1: Extracto del metamodelo MIGRARIA MVC

5.2. Especificación de la notación

Tomando como referencia el patrón de diseño arquitectónico MVC, base conceptual de la gran mayoría de frameworks de desarrollo Web, la notación propuesta debe ser capaz de especificar todos los

elementos comunes de los tres componentes fundamentales de este patrón en su aplicación al dominio Web, esto es, los elementos del Modelo, los de la Vista y los del Controlador. La figura 5.1 muestra un resumen de la definición de esta notación formado por tres paquetes fundamentales: Modelo, Vista y Controlador. Los siguientes subapartados explican con detalle los elementos que conforman cada uno de estos paquetes.

Para la definición de esta notación de representación conceptual, se ha usado como tecnología base Eclipse Modeling Framework, de manera que los metamodelos definidos son conformes al metametamodelo ecore[1] .

5.2.1. Modelo

Del estudio de las diferentes propuestas IWDM desarrolladas en la última década, que presentan notaciones de gran similitud para la representación de los datos, se pueden identificar los siguientes elementos fundamentales[2]:

- *Objetos de datos*, que permiten definir las entidades fundamentales de datos de la aplicación, su composición y sus relaciones.
- *Atributos de los objetos de datos*, que definen las características de los objetos de datos. Sus tipos suelen corresponderse en gran medida con sistemas de tipos genéricos en diferentes sistemas gestores de bases de datos. En notaciones orientadas a objetos, se usa también la relación de asociación para definir atributos cuyos tipos son otros tipos de objetos de datos.
    - *Atributos derivados*, que permiten representar valores calculados a partir de fórmulas o de consultas realizadas sobre el modelo de datos.
- *Relaciones entre los objetos de datos*, que permiten especificar relaciones entre los objetos de datos. Es posible identificar normalmente dos tipos de relaciones: descriptivas y estructurales. Las primeras permiten definir relaciones dependientes del dominio con una cardinalidad determinada (muchos-a-muchos, uno-a-muchos, etc.). Adicionalmente, algunas propuestas, como (Kraus *et al.* 2007; Cachero Castro 2003), también permiten especificar relaciones de tipo estructural como, por ejemplo: la herencia, para indicar la herencia de características entre objetos de datos; y la agregación, para indicar relaciones uno-a-uno o maestro-detalle entre objetos de datos.
- *Operaciones sobre los objetos de datos*, que permiten especificar tanto operaciones de recuperación de información sobre un objeto o sobre un conjunto de objetos como operaciones de gestión de objetos de datos (crear, borrar, actualizar) y, por otro, operaciones de recuperación de datos definidas sobre relaciones.

[1] http://www.kermeta.org/docs/org.kermeta.ecore.documentation/build/html.chunked/Ecore-MDK/ch02.html
[2] La nomenclatura propuesta por cada propuesta difiere.

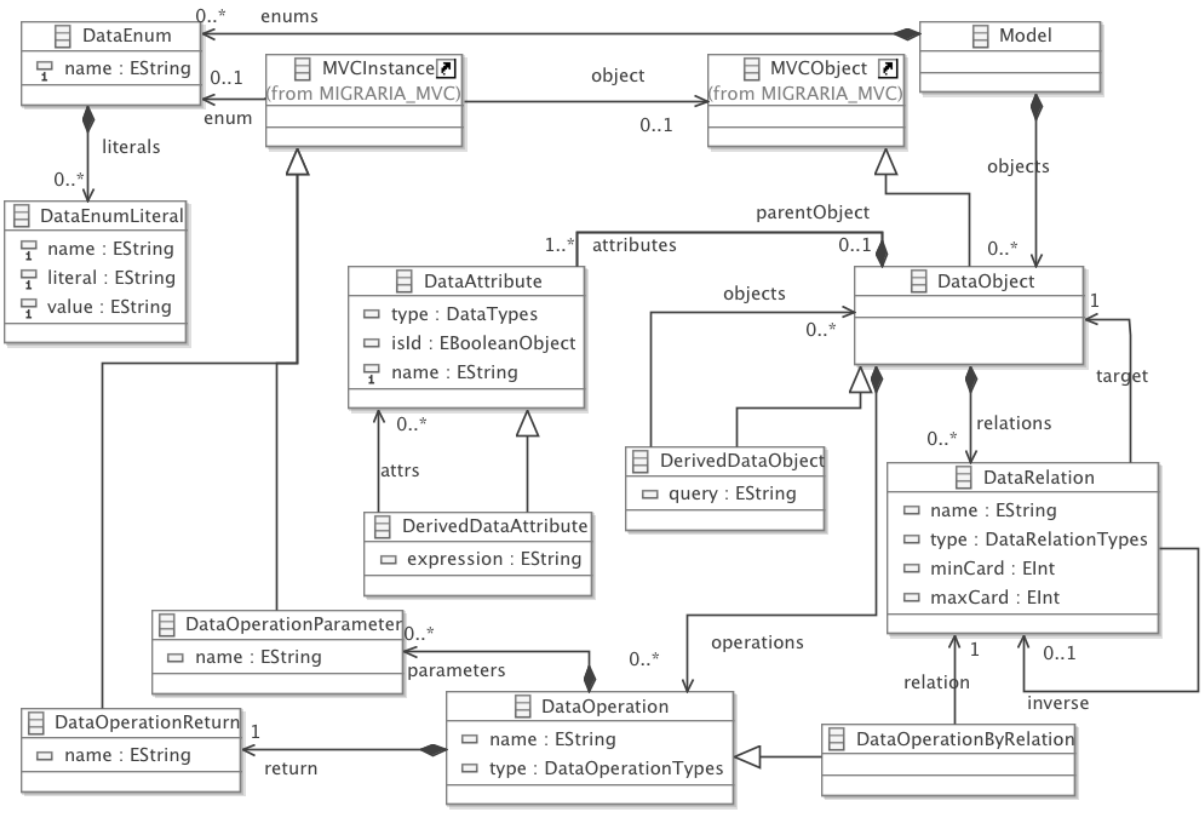


Figura 5.2: Metamodelo MVC Model

Desde un plano tecnológico, es necesario tener en cuenta que es muy común que los frameworks de desarrollo de aplicaciones Web no especifiquen nada sobre cómo definir el componente *Modelo*, delegando esta responsabilidad en otro framework o tecnología. En este sentido, se encuentra ampliamente extendido el escenario definido por un framework de aplicación Web y un framework de persistencia de objetos, que actúe de puente entre la capa Web y el sistema de almacenamiento y gestión de los datos. En este escenario se suelen encontrar soluciones basadas en los patrones *Data Access Object (DAO)* y *Data Transfer Object*, en (Alur *et al.* 2001).

5.2.1.1. Notación

En este caso, al igual que las propuestas IWDM basadas en la orientación a objetos, sugerimos el uso de una versión adaptada del diagrama de clases de UML, simplificado para la especificación del componente *Modelo* de nuestra notación.

Como se puede apreciar en la figura 5.2, la especificación del componente Modelo se puede realizar a través de dos tipos fundamentales de datos: los objetos y los tipos enumerados. Dado que la especificación de tipos enumerados no conlleva ninguna complejidad, a continuación la descripción del metamodelo se va a centrar en los elementos definidos para la especificación de objetos de datos. Como no podía ser de otra forma, el elemento fundamental de este metamodelo es el objeto de datos definido mediante la entidad *DataObjetct* (*DO*). Un *DO* está estructuralmente compuesto por *DataAttributes*

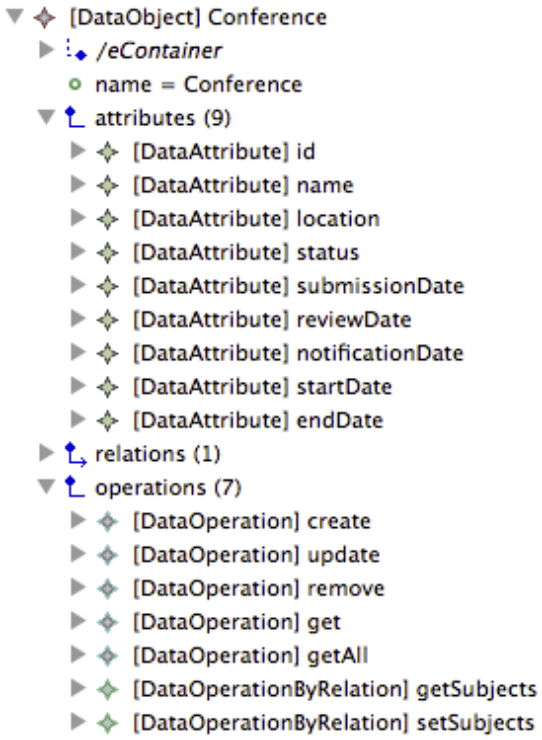
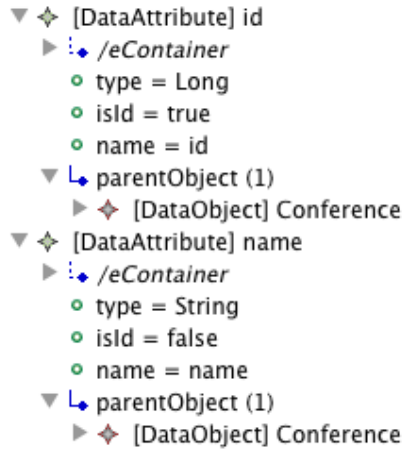


Figura 5.3: DataObject *Conference*

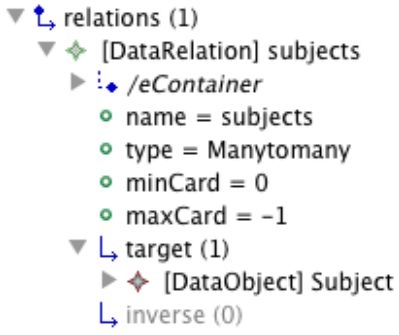
(*DA*), *DataRelations (DR)* y *DataOperations* (*DOp*). La figura 5.3 muestra la especificación del DataObject *Conference* extraído del caso de estudio para ilustrar esta explicación. El caso de estudio consiste en una aplicación web heredada que implementa un sistema de gestión de conferencias y se presenta con más detalle en el Capítulo 7.

Un *DA* permite especificar el nombre y el tipo de un atributo. Adicionalmente, permite especificar si ese atributo constituye una clave primaria para el objeto al que pertenece. La figura 5.4a ilustra los detalles de la especificación de los atributos *id* y *name* del objeto de datos *conference*. El primero es de tipo *Long* y su atributo *isId* contiene el valor *true*, mientras que el segundo es de tipo *String* y su atributo *isId* contiene el valor *false*. En el caso de que el objeto de datos tuviese una clave primaria formada por múltiples atributos, todos ellos deberían tener a *true* su atributo *isId*. Un tipo especial de *DA* está constituido por los *DerivedDataAttributes* (*DDA*), que permiten especificar atributos derivados recogiendo la expresión de generación de los mismos. La diferencia fundamental entre un *DA* y un *DDA* es que el primero especifica un atributo persistente mientras que el segundo especifica un atributo no persistente o volátil de un objeto de datos.

Un *DR* permite especificar una relación existente entre dos objetos de datos, indicando su nombre, los objetos implicados y el tipo de relación. Adicionalmente, permite especificar la relación inversa de una relación. La figura 5.4b ilustra la especificación de la relación muchos-a-muchos que existe entre un objeto *conference* y objetos *subjects*. En este caso, por lo tanto, el atributo *target* referencia al *DO Subject*.



(a) DataAttributes *id* y *name*



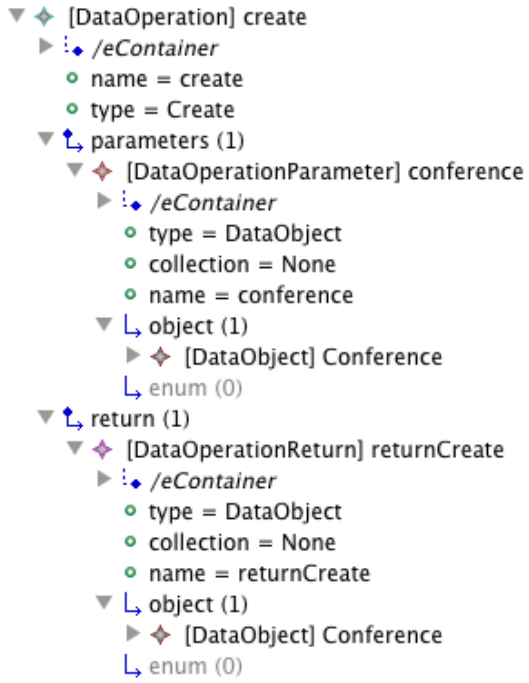
(b) DataRelation *subjects*

Figura 5.4: Detalles de DataObject *Conference*

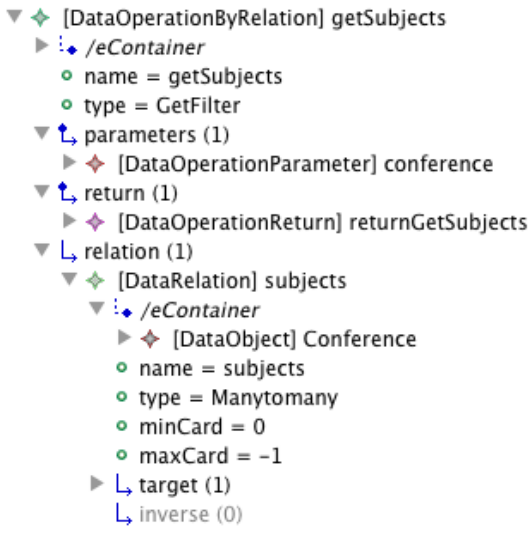
Un *DOp* permite especificar operaciones sobre objetos de datos, indicando su nombre, su tipo, sus parámetros y su retorno. Los tipos de operaciones considerados hacen referencia a las operaciones fundamentales sobre objetos de datos (crear, borrar, actualizar y recuperar) y sobre relaciones de objetos de datos (establecer relación, recuperar datos relacionados). Adicionalmente, se define un tipo de operación, denominada *custom*, para especificar una operación no categorizada, que sirve además como punto de extensión de la especificación de tipos de operación. Desde el plano estructural, una operación está compuesta por sus parámetros y su retorno. En la notación planteada es posible especificar ambos mediante las entidades *DataOperationParameter* (*DOpP*) y *DataOperationReturn* (*DOpR*), respectivamente. Estas entidades permiten especificar instancias de objetos de datos definidos, de tipos primitivos de datos, de datos enumerados y de colecciones de objetos o de tipos primitivos. En el caso de *DOpR* se puede especificar que la operación no especifica un valor de retorno estableciendo el atributo *type* al valor *Void*. La figura 5.5a ilustra con detalle la especificación de la *DOp create* que indica que es del tipo *Create* y contiene la especificación de un parámetro y un valor de retorno. En el caso del parámetro, corresponde con una instancia del objeto de datos *Conference*, definida mediante el valor *DataObject* del atributo *type* y la referencia *object* apuntando al *DO Conference*.

DataOperationByRelation (DOpByR) define un tipo especializado de operación que permite asociar una operación a una relación concreta. Este tipo especializado sirve para especificar operaciones de recuperación de los objetos de datos relacionados con un objeto de datos, como por ejemplo, en el caso de nuestro ejemplo, todos los *subjects* relacionados con un objeto *conference* determinado. La figura 5.5b ilustra la especificación de la *DOpByR getSubjects* definida como de tipo *GetFilter*, que indica recuperación de datos filtrados por una condición, y asociada a la *DR subjects* definida previamente. En este caso, el filtro viene dado por el propio objeto *conference* sobre el que se invoca la operación.

Finalmente, como se puede observar en la figura 5.2, muchos de los atributos de las metaclases definidas no son obligatorios; ya que, el objetivo de estos modelos es poder representar la información que se consiga extraer; por lo tanto, es posible que la disponibilidad de información para los diferentes



(a) DataOperation *create*



(b) DataOperationByRelation *getSubjects*

Figura 5.5: Detalles del DataObject *Conference*

atributos del modelo sea muy variable. Por lo tanto, el metamodelo simplemente intenta establecer un conjunto mínimo de requisitos y restricciones.

5.2.2. Vista

El componente Vista dentro del patrón MVC define los elementos necesarios para la especificación de la interfaz del sistema con el usuario. Teniendo en cuenta que nos encontramos en el dominio de aplicaciones Web, este componente permite la especificación de la estructura y composición de páginas Web. Existe un gran número de tecnologías relacionadas con la generación dinámica de páginas Web, aunque todas ellas comparten una serie de conceptos fundamentales, que son:

- La página es el elemento contenedor fundamental.
- Una página está compuesta fundamentalmente por objetos de presentación y peticiones.
- Una página puede mostrar datos de un objeto de presentación o recoger datos de entrada para crear o modificar un objeto.
- Una página permite definir dos tipos fundamentales de peticiones al servidor: hiperenlaces o envío de formularios.
- El servidor Web normalmente devuelve como respuesta una página Web. Por lo tanto, cada petición obtiene como respuesta una página y, de esta forma, todas las páginas que conforman una aplicación Web están relacionadas mediante un grafo que suele recibir el nombre de mapa de navegación.

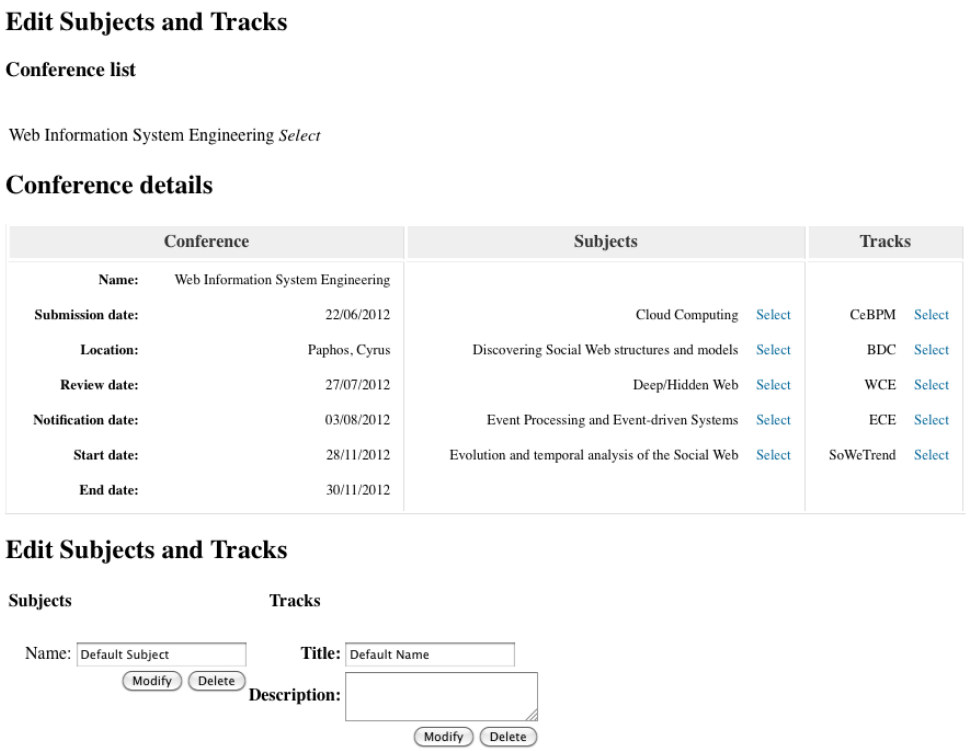


Figura 5.6: Ejemplo página Web

La figura 5.6 muestra una página web de ejemplo en la que aparecen los diferentes conceptos comentados anteriormente. Esta página muestra los detalles de una conferencia (objeto de presentación) y permite introducir (envío de formulario) y editar (hiperenlace) los *subjects* y las *tracks* de la misma.

5.2.2.1. Objetos de presentación

Los objetos de presentación normalmente definen vistas de los objetos de datos del modelo subyacente, de manera que muestran o permiten modificar los datos correspondientes en cada momento. Los objetos de presentación pueden categorizarse, fundamentalmente, según dos dimensiones: la direccionalidad de los datos y el modo de publicación del objeto.

Con respecto a la direccionalidad de los datos, los objetos de presentación se pueden dividir en dos categorías base:

- *Objetos de salida*. Son los más comunes en una aplicación web y se usan para mostrar información al usuario, como por ejemplo los detalles de un producto dentro de un catálogo, o el número de asientos libres en un teatro, etc.

- *Objetos de entrada*. Su misión fundamental consiste en obtener datos del usuario para modificar los objetos de datos de la aplicación web y permiten, fundamentalmente, desde el punto de vista de los datos, crear un objeto, modificar un objeto total o parcialmente, modificar un atributo de un objeto, eliminar un objeto, relacionar objetos o eliminar relaciones entre objetos. Alternativamente,

ID	Title	Authors	Track	Subject	Revisions		
10	Towards a trust-manager service for hybrid clouds	Fatma	CeBPM		0	Review	Change Reviewer
6	Cloud Storage of Artifact Annotations to Support Case Managers in Knowledge-Intensive Business Processes	Marian	CeBPM	Cloud Computing Discovering Social Web structures and models Event Processing and Event-driven Systems	1	Review	Change Reviewer

Figura 5.7: Categorías de atributos de objetos de presentación

también pueden recoger datos para parametrizar operaciones de control que no alteran los objetos de datos del sistema, como por ejemplo, realizar una búsqueda a partir de una cadena.

En cuanto al modo de publicación, es decir, cómo se realiza la composición de la página a partir de los objetos de presentación, se suelen distinguir tres casos básicos:

- Se presenta *un objeto* de forma individual. Normalmente, este modo suele recibir el nombre de detalle o registro. Muy común para presentar los detalles de un producto en un catálogo o para recoger datos de un usuario en un sistema de registro.

- Se presenta una *colección de objetos* del mismo tipo. Sus representaciones web más comunes son una tabla en la que cada fila presente datos de un objeto, como por ejemplo la lista de *subjects* de una determinada conferencia como ilustra la figura 5.6), o campos de control de selección propios de formularios web, como por ejemplo las listas de selección de *Program Comitte Members* presentadas en la figura 5.9. Las colecciones contienen, al menos, un objeto de salida que representa al ítem de la colección y pueden contener objetos de entrada que permitan, por ejemplo, modificar alguno de los atributos del ítem. Además, las colecciones de objetos permiten anidamiento, normalmente uno o varios atributos del objeto ítem presenta a su vez una colección de valores que puede ser tanto de salida como de entrada.

- Se presenta una *composición de objetos* y colecciones que pueden compartir, o no, tipo, pero suelen estar relacionados a partir de una relación definida en el modelo de datos, como por ejemplo, una relación maestro-detalle. Este modo de presentación es muy común. También puede entenderse como el modo por defecto usado para la composición de una página; ya que, engloba a los dos casos anteriores.

5.2.2.2. Atributos de los objetos de presentación

Los objetos de presentación están caracterizados fundamentalmente por sus atributos. Un atributo de un objeto de presentación se caracteriza mediante su nombre, su direccionalidad, su visibilidad y, por supuesto, por el objeto de datos que lo contiene.

La direccionalidad de los datos hace referencia al comportamiento del atributo en la composición de la página. El atributo es de salida si sólo se usa para presentar información en la página. El atributo es de entrada si permite recoger datos del usuario, además de presentar información. Obviamente, un objeto

de salida sólo puede estar formado por atributos de salida. Mientras que los objetos de entrada pueden contener tanto atributos de salida como de entrada.

La visibilidad de un atributo controla si el valor del atributo se muestra al usuario (visible) o si, por el contrario, aparece en el código HTML de la página de respuesta pero no se muestra al usuario (oculto).

El tipo de los atributos está restringido a cadena de caracteres en el caso de atributos de entrada, como consecuencia de las características del protocolo HTTP. Mientras que en el caso de objetos de salida los atributos pueden tener cualquier tipo dentro del sistema de tipos definido, como el que se presenta en el apartado del Modelo. Hay que tener en cuenta que los atributos de los objetos de salida son convenientemente serializados en el momento de procesar la plantilla que define la página web de respuesta; por lo tanto, es la tecnología de generación de páginas web dinámicas la que establece las restricciones a los tipos de atributos. En conclusión, en lo referente al componente Vista, se puede realizar la generalización de que todos los atributos son de tipo cadena de caracteres y, por lo tanto, el tipo de datos de un atributo de un objeto de presentación no necesita ser explícitamente especificado.

Finalmente, desde el punto de vista de los datos, podemos distinguir cuatro categorías de atributos, que aparecen ilustradas en la figura 5.7:

- *Datos*. Son aquellos que presentan un mapeo directo con un atributo del objeto de datos relacionado, como por ejemplo el atributo *Title* en el listado de la figura que se corresponde directamente con el atributo *Title* del objeto de datos *Paper*.

- *Derivados*. Representan valores calculados a partir de fórmulas o de consultas al modelo de datos, como por ejemplo el atributo *Revisions* del listado de la figura 5.7, que indica el número de revisiones que existen para un determinado artículo.

- *Conjunto*. Se presentan al usuario como una lista de valores o como un control que permite seleccionar uno o varios valores. En el caso de selección simple, algunos casos de uso básicos son: elegir un valor entre una lista de valores posibles (tipo enumerado) o establecer una relación *muchos-a-uno* entre objetos de datos. En el caso de selección múltiple, se dan los mismos casos con la diferencia de que se trabaja sobre una relación muchos-a-muchos. En el caso de la figura 5.9b, se presentan ejemplos de salida para los atributos *Track* (selección simple) y *Subjects* (selección múltiple).

- *Control*. Son aquellos que recogen valores utilizados para parametrizar algún servicio o componente, y no se corresponden con atributos de objetos de datos. Por ejemplo, se utilizan en el caso de formularios HTML que presenten varios botones de envío para poder seleccionar el controlador adecuado a cada una de las peticiones derivadas de esos botones, como en el caso del formulario de edición de *Tracks* con los botones *modify* y *delete*, ilustrado por la figura 5.6.

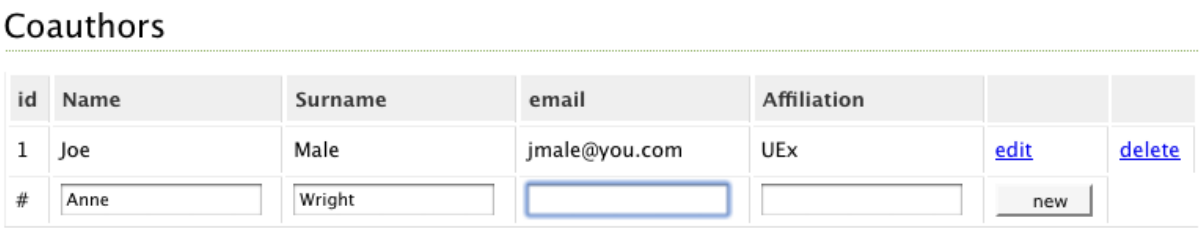


Figura 5.8: Ejemplos de peticiones

5.2.2.3. Peticiones

La interacción fundamental entre un usuario y una aplicación web consiste en realizar una petición al servidor Web, que responde mediante el envío de otra página Web generada para satisfacer la demanda del usuario.

Desde un punto de vista estrictamente tecnológico, existen dos formas fundamentales de definir peticiones en una página Web: (1) mediante la especificación de hiperenlaces, y (2) mediante el envío de formularios. Por lo tanto, se puede realizar una primera categorización de las peticiones:

- *Peticiones mediante enlaces*. A nivel de protocolo HTTP, se traducen en peticiones con el método GET, en las que los parámetros se pasan a través de una cadena de consulta (*query string*) que suele presentar un tamaño máximo limitado. Estos parámetros suelen contener un valor previamente fijado en la generación dinámica de la página Web.

- *Peticiones mediante formularios*. A nivel de protocolo HTTP, se traducen normalmente en peticiones con el método POST, en las que los parámetros se especifican y recogen por medio de un formulario y cuyo envío se suele realizar mediante un control de tipo botón. Permite el uso de diferentes controles HTML para la recogida de datos: campos de entrada de texto, botones radio, áreas de texto, etc.

Desde un plano conceptual, como señalan (Pastor *et al.* 2006), se pueden identificar tres tipos de interacciones Web genéricas:

- *Recuperación de información* del estado de uno o varios objetos de datos, como por ejemplo, la petición realizada para obtener la información de los coautores de un determinado artículo, cuyo resultado se ilustra en la figura 5.8.

- *Ejecución de un servicio* sobre un objeto determinado, como por ejemplo, dar de alta un nuevo coautor a través del botón *new* representado en la figura 5.8.

- *Navegación* a una página que no supone la recuperación de información relacionada con objetos de datos.

Como se puede apreciar, el punto de vista tecnológico pone su foco en la forma de especificar la interacción, mientras que el punto de vista conceptual pone su foco en la misión (objetivo último) de la petición.

	enlace	formulario	parámetros	información	servicio	navegación
página	X			X		X
objeto	X	X	X	X	X	
relación	X	X	X	X	X	
conjunto		X	X	X	X	
colección	X		X	X		

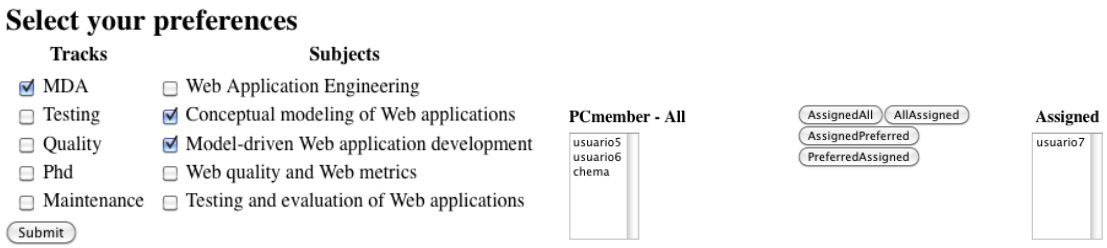
Tabla 5.1: Categorías de peticiones según el nivel de origen

Si bien, identificar la categoría de las peticiones desde un punto de vista tecnológico es inmediato en el análisis estático de una página, no lo es tanto identificar su categoría conceptual. A primera vista, se puede intentar relacionar, por un lado, *recuperación de información* y *navegación* con peticiones mediante enlaces, mientras que, por otro lado, se relaciona *ejecución de servicios* con peticiones mediante *formularios*. Sin embargo, esta aproximación no es válida para tareas de ingeniería inversa; ya que, no es suficientemente representativa. En las aplicaciones web heredadas se pueden encontrar fácilmente casos que la contradicen. Por ejemplo, la figura 5.8 muestra dos peticiones mediante enlace con misiones distintas: por un lado, el enlace *edit* tiene como misión recuperar toda la información de uno de los objetos mostrados en la colección para poder modificarlo, mientras que por otro lado, el enlace *delete* ejecuta un servicio que tiene como resultado la eliminación del objeto de datos relacionado con el objeto de presentación seleccionado.

Tomando como base estos dos extremos, desde un punto de vista intermedio, adecuado a tareas de ingeniería inversa y modernización, interesa definir una categorización de peticiones con mayor grado de detalle que permita tanto especificar su formato como identificar su posible misión. La tabla 5.1 especifica la relación entre estas tres categorías, indicando todos los casos posibles. En este sentido, establecemos una categorización basada en el nivel al que se define la petición:

- *Peticiones a nivel de página*. Se especifican como enlaces sin parámetros y definen una interacción de recuperación de información o de navegación. La distinción de la misión, por tanto, es posible que deba delegarse al proceso de análisis del controlador para poder contar con más información sobre el tratamiento de la petición por parte de la aplicación web.

- *Peticiones a nivel de objeto*. Se pueden especificar como enlaces o como formularios. Como enlaces, llevan como parámetro, al menos, el ID del objeto y su misión puede ser tanto la recuperación de información del objeto como la ejecución de servicios. La distinción entre una u otra misión se debe realizar en el análisis del controlador que se encarga de manejar esa petición. Los enlaces *edit* y *delete* de la figura son un ejemplo de peticiones a nivel de objeto mediante enlaces con misiones diferentes. Como formularios, su misión es la ejecución de un servicio asociado a ese objeto. De forma general, se puede considerar que el formulario a nivel de objeto se utiliza para modificar ese objeto o para crear una nueva instancia de ese tipo de objetos. El primer caso incluirá el ID del objeto a modificar, mientras que el segundo no incluirá ID al tratarse de la invocación del servicio



(a) Checkbox (b) Select fields

Figura 5.9: Peticiones sobre conjuntos de objetos

ID	Title	Revisions	Relevance ▲ ▼	Tech Depth ▲ ▼	Presentation ▲ ▼	Originality ▲ ▼
3	On the Unification Power of Models	1	3.0	3.0	4.0	4.0
4	Recursive Functions of Symbolic ...	0	--	--	--	--

Figura 5.10: Peticiones sobre colección de objetos

crear. La figura 5.8 ilustra este último caso definiendo un formulario que permite la creación de un nuevo objeto *coauthor*, que está formado por cuatro controles de texto y un botón con la etiqueta *new* encargado de enviar la petición al servidor.

- *Peticiones a nivel de relación*. Aparecen en propuestas IWDM para definir navegaciones entre páginas que muestran objetos relacionados. Desde el punto de vista del análisis estático de la página, representa un caso específico de navegación a nivel de objeto, que es complejo diferenciar del caso base; por lo tanto, normalmente se necesita información de la gestión de la petición que hace el componente controlador para poder identificarla.

- *Peticiones sobre conjuntos de objetos* seleccionados de una colección. Se especifican mediante formularios; ya que, es necesario disponer de algún tipo de control que permita seleccionar varios objetos a la vez, como por ejemplo, un campo de selección o mediante botones *checkbox*, tal como ilustra la figura 5.9. Con los objetos seleccionados se suele invocar un servicio, por ejemplo, asociar a un usuario *PC member* sus preferencias en cuanto a *Tracks* y *Subjects* (subfigura 5.9a), o asignar la revisión de un artículo determinado a uno o varios *PC members* (subfigura 5.9b).

- *Peticiones a nivel de colección de objetos* para parametrizar su comportamiento, cambiando el filtro, el orden, la página de objetos mostrada en una paginación, etc. La figura 5.10 muestra una colección de objetos *Paper* que puede ser ordenada en función del valor de sus atributos.

5.2.2.4. Notación

El metamodelo presentado en la figura 5.11 recoge todos los conceptos definidos en los apartados anteriores, permitiendo representar los elementos del componente Vista del patrón MVC con un gran nivel de detalle.

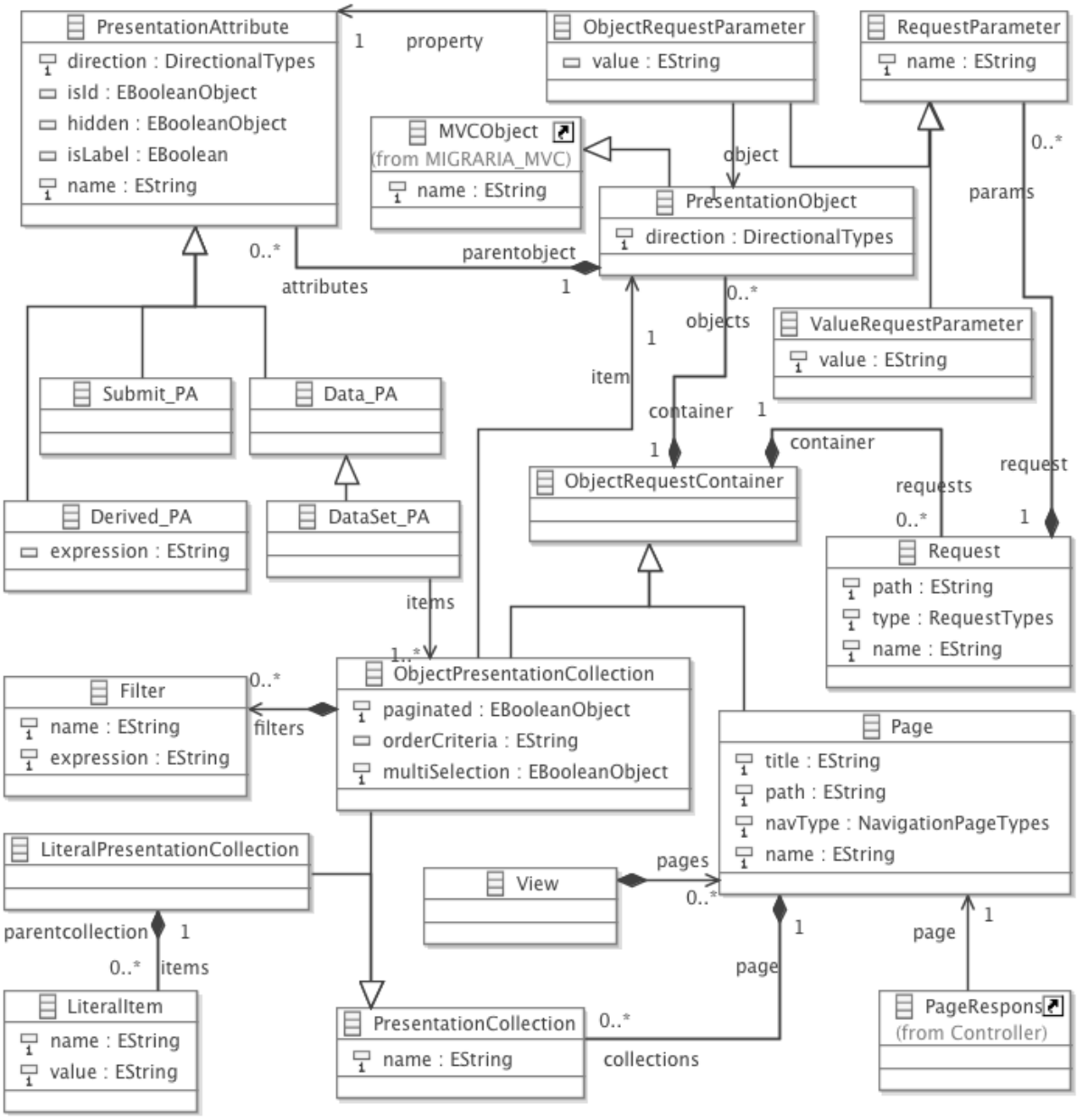


Figura 5.11: Metamodelo MVC View

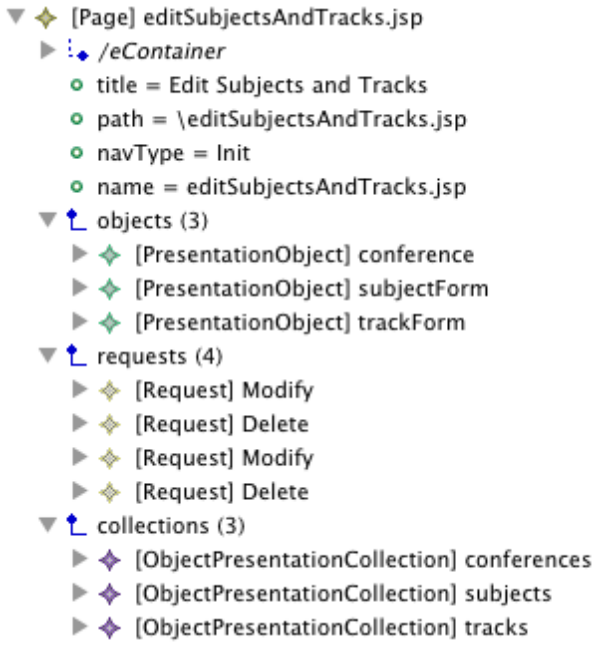


Figura 5.12: Ejemplo entidad *Page*

Como se puede apreciar en la figura 5.11, la especificación del componente Vista se realiza exclusivamente mediante la especificación de los elementos página (*Page*) que lo constituyen. Por lo tanto, los elementos Page constituyen el elemento central de este metamodelo, permitiendo definir la estructura de presentación y de navegación de la aplicación Web. Una página está caracterizada por sus atributos de nombre, título, ruta y tipo de navegación (inicio, menú o secuencia). En lo referente a su composición, una página puede contener objetos de presentación (*PresentationObject, PO*), peticiones (*Request, Rq*) y colecciones de presentación (*PresentationCollection, PC*). La figura 5.12 muestra la representación de la página *editsubjectsandtracks.jsp*, cuya apariencia final se muestra a su vez en la figura 5.6. Como se puede apreciar, esta entidad está compuesta por tres objetos de presentación, cuatro peticiones de objeto y tres colecciones de objetos de presentación.

Un *PO* es una especialización de la entidad *MVCObject* al igual que sucedía con *DataObject*. Un *PO* está simplemente caracterizado por su nombre y su direccionalidad. Y estructuralmente está conformado por un conjunto de atributos (*PresentationAttribute, PA*). Por lo tanto, su misión fundamental consiste en definir el contenedor de un conjunto de atributos. La figura 5.13 muestra la especificación de dos objetos de presentación: uno de salida, objeto *conference*, y otro de entrada, objeto *subjectForm*. El primero representa al objeto *conference* cuya relación con *subjects* y *tracks* se quiere gestionar. Mientras que el segundo representa al objeto *subject* que se quiere editar.

La entidad *PA* permite especificar las principales características de los atributos de los objetos de presentación, como son su nombre (atributo *name*) y direccionalidad (atributo *direction*). También permite especificar características relacionadas con el aspecto de presentación, como son su visibilidad (atributo *hidden*) y su misión dentro de una colección (atributo *isLabel*), así como características relacionadas con los datos, como su pertenencia a la clave primaria del objeto de datos origen del objeto de presentación

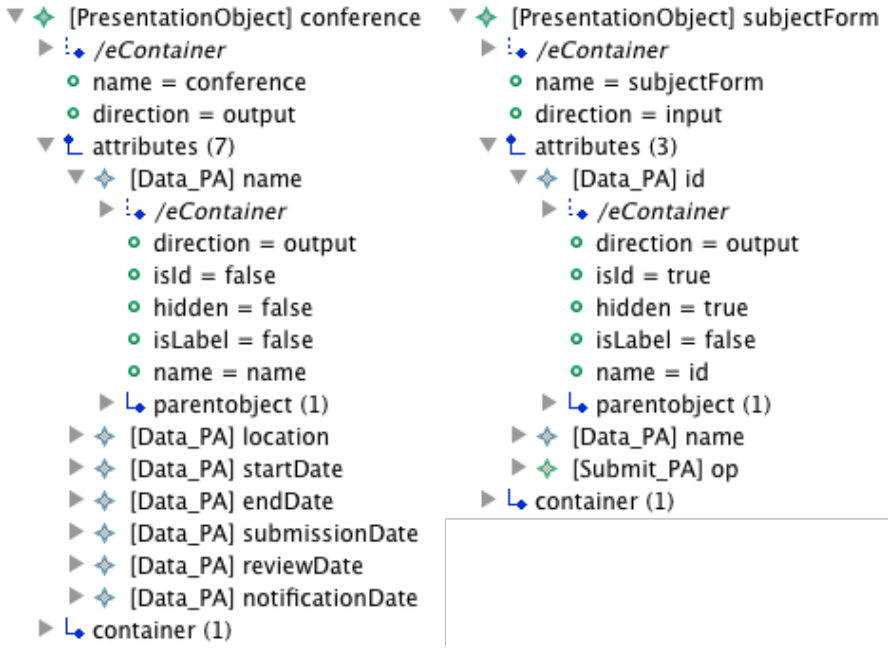


Figura 5.13: Ejemplo entidades PresentationObject y PresentationAttribute

contenedor (atributo *isId*). En la figura 5.13 se comparan los valores de las características del atributo *Data_PA name* del objeto *conference* con las del atributo *Data_PA id* del objeto *subjectForm*.

Con el objetivo de representar la taxonomía de atributos de objetos de presentación introducida anteriormente, se ha definido una jerarquía de atributos de presentación tomando como raíz la entidad *PA*. De esta manera, se definen cuatro nuevas entidades que especializan a ésta:

- *Data_PA*, que permite especificar un atributo simple de un objeto de presentación y que permitirá especificar el mapeo con un atributo de un objeto de datos.

- *Derived_PA*, que permite especificar un atributo derivado de un objeto de presentación que se encuentre definido dentro de la vista. Esta entidad permite recoger la expresión de generación del atributo derivado.

- *DataSet_PA*, que se define como una especialización de *Data_PA* y permite especificar un atributo complejo (puede tomar múltiples valores) de un objeto de presentación. Esta entidad contiene una referencia a una colección de datos, entidad ObjectPresentationCollection, que es la encargada de especificar el conjunto de valores disponible para ese atributo.

- *Submit_PA*, que permite especificar un atributo de un objeto de presentación de entrada cuya misión consiste en parametrizar el manejo de la petición. La figura 5.13 ilustra el uso del tipo de atributo Submit_PA para representar el valor del parámetro (*edit*, o *delete*) de la petición realizada sobre el objeto de entrada *subjecForm*.

Otro de los componentes fundamentales de una página son las peticiones. La entidad *Request* permite especificar cualquiera de los tipos de peticiones discutidos previamente. Básicamente, una petición está

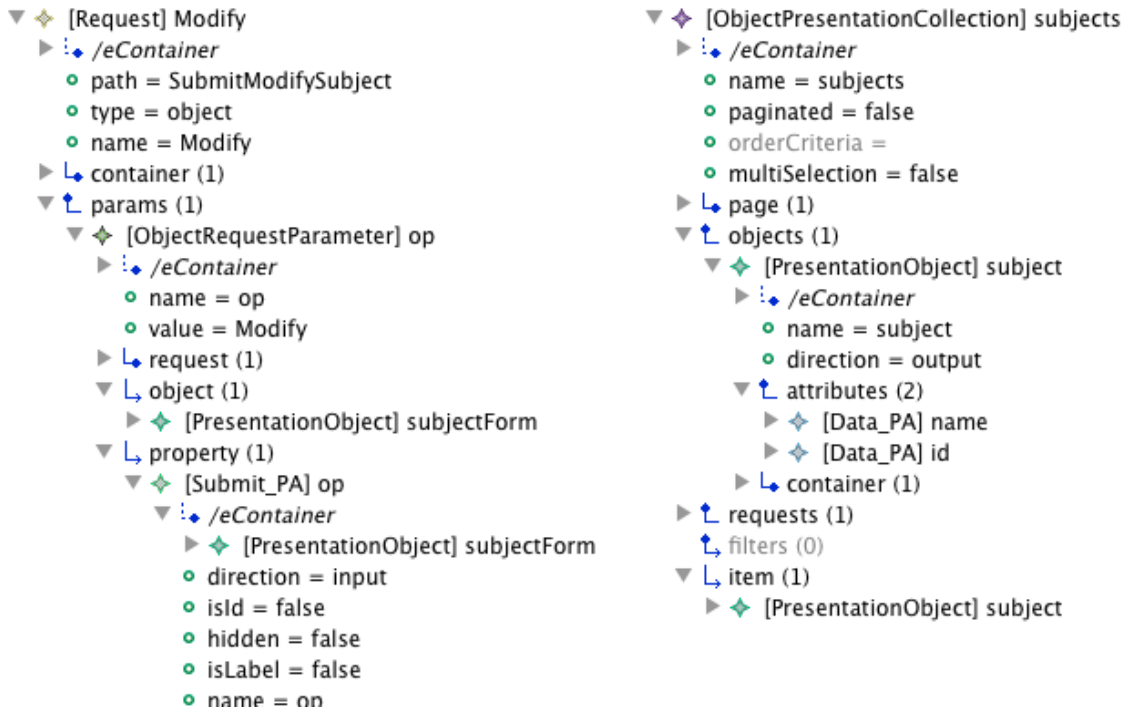


Figura 5.14: Ejemplo entidades Request y PresentationCollection

caracterizada por su nombre, tipo y ruta. El atributo de tipo es el encargado de indicar si se trata de una petición de página, objeto, colección, conjunto o relación. La figura 5.14 ilustra la especificación de la petición *Modify* sobre el objeto de entrada *subjectForm*, definida de tipo *object* y portadora de un parámetro. Por su parte, la ruta (*path*) permite especificar el nombre del manejador de la petición en el controlador (*SubmitModifySubject*). Una *Request* puede contener parámetros que son especificados mediante la jerarquía definida a partir de *RequestParameter*. Esta jerarquía define dos tipos de parámetros básicos:

- Parámetros valor. La entidad *ValueRequestParameter* permite especificar un parámetro cuyo valor viene especificado literalmente en la vista. Este valor se recoge en el atributo *value* de la entidad.

- Parámetros objeto. La entidad *ObjectRequestParameter* permite especificar un parámetro cuyo valor se obtiene a partir de la propiedad de un objeto. Las referencias *object* y *property* permiten indicar el objeto de presentación asociado y su atributo, respectivamente. En el caso de tratarse de un objeto de presentación de entrada, el atributo a referenciar será el que representa el botón de envío que define la petición a especificar con la entidad *Request*. Este último caso se muestra en la figura 5.14 mediante la especificación del parámetro *op* que se relaciona con el objeto de entrada *subjectForm* y su atributo *op* de tipo *Submit_PA*.

El último componente de una página, que ha sido considerado, consiste en las colecciones de presentación, entidad *PresentationCollection*, que permiten especificar colecciones tanto de valores literales como de objetos de datos. Las colecciones se usan, por un lado, para generar listados de objetos de salida

y, por otro lado, para mostrar múltiples valores en atributos multivaluados de tipo *DataSet_PA*. Al igual que en el caso de las peticiones, se ha definido una pequeña jerarquía para distinguir entre:

- Colecciones de presentación de valores literales. La entidad *LiteralPresentationCollection* permite especificar colecciones que han sido literalmente definidas en la vista mediante la relación de sus elementos. Estos elementos son especificados mediante la entidad *LiteralItem* que permite recoger el nombre (etiqueta) y valor de cada elemento.

- Colecciones de presentación de objetos. La entidad *ObjectPresentationCollection* permite especificar colecciones que han sido definidas a partir de un conjunto de objetos del mismo tipo, representado mediante la referencia *item* a un objeto de presentación. Esta entidad, como se puede apreciar en la figura 5.11, se ha definido como subclase, por un lado, de *PresentationCollection* y, por otro, de *ObjectRequestContainer*. Esta segunda relación de especialización se debe a que esta entidad define también un contenedor de objetos de presentación y de peticiones, al igual que la página. En este caso, la referencia *objects* permite representar al objeto de presentación ítem y a los objetos de entrada relacionados con él; mientras que, la referencia *requests* representa las peticiones relacionadas con el objeto de presentación *ítem*. Adicionalmente, las entidades de este tipo permiten recoger aspectos relacionados con la presentación de los elementos de la colección, como son, la paginación, el criterio de ordenación y la capacidad de selección múltiple de valores. La figura 5.14 ilustra la especificación de la colección de presentación de objetos *subjects* que está fundamentalmente formada por objetos *subject*, ítem de la colección, y la petición asociada a la selección de un determinado *subject* (no mostrada por simplicidad).

Las propuestas de IWDM varían en el grado de abstracción que utilizan para definir los conceptos principales de la capa de presentación de una aplicación Web. Por ejemplo, WebML usa el concepto *Page* como elemento contenedor principal, mientras que OOH utiliza el concepto *NavigationClass* y otras, incluso, van un paso más allá en el nivel de abstracción y usan el concepto *Node* como elemento de composición fundamental. En este caso, se ha elegido una notación más cercana a la tecnología de desarrollo debido a que el objetivo es obtener una representación de la aplicación Web heredada mediante técnicas de ingeniería inversa y no especificar una nueva aplicación Web desde cero y generar su código mediante técnicas de ingeniería directa.

5.2.3. Controlador

El componente controlador tiene como misión fundamental atender las peticiones realizadas desde la parte cliente, devolviendo una página web de respuesta. El framework de aplicaciones web suele definir, por un lado, un mecanismo de mapeo petición-controlador y, por otro lado, una serie de clases fundamentales que permiten organizar la lógica relacionada con el componente controlador. De esta manera, el framework dirige la petición al controlador adecuado y se encarga de proporcionarle en un

```
24 public class ModifyAndDeleteSubjectsAndTracks extends Action{
25
26     public ActionForward execute(ActionMapping mapping, ActionForm form,
27             HttpServletRequest request, HttpServletResponse response)
28             throws Exception {
29
30         Long confId = new Long(request.getParameter("confId"));
31         ConferenceDAO confDAO = HibernateConferenceDAO.getInstance();
32         Conference conf = confDAO.get(confId);
33
34         request.setAttribute("conference", conf);
35         request.setAttribute("conferences", confDAO.getAll());
36
37         Set<Subject> subjects = conf.getSubjects();
38         request.setAttribute( "subjects", subjects );
39
40         TrackDAO trDAO = HibernateTrackDAO.getInstance();
41         Collection<Track> tracks = trDAO.getAll(conf);
42         request.setAttribute("tracks", tracks);
43
44         return mapping.findForward("success");
45
46     }
47
48 }
```

Figura 5.15: Ejemplo controlador en Struts

formato manejable los datos de entrada que vienen en la petición, así como los datos relativos al contexto de ejecución como por ejemplo los datos de sesión. La lógica del controlador, por su parte, se encarga básicamente de: (1) procesar los datos de entrada, (2) realizar la operación definida para esa petición, y (3) responder al cliente con el envío de una nueva página web que puede contener los datos resultado de la operación realizada. La figura 5.15 ilustra la definición de un controlador en el framework de desarrollo web Struts 1. Este controlador es el responsable de recuperar los datos necesarios para generar la página presentada en la figura 5.6 y se usará a lo largo de este apartado para ilustrar los detalles de la notación definida para el componente Controlador.

Considerando el objetivo de obtener una representación lo más completa posible de la aplicación web heredada para realizar un proceso de modernización, la información fundamental a extraer del componente controlador estaría constituida por:

- Relación petición-controlador-respuesta.
- Mapeo entre elementos de la Vista y elementos del Modelo.
- Invocación a operaciones y paso de parámetros.

5.2.3.1. Relación petición-controlador-respuesta

Relacionar una determinada petición con la lógica concreta que la atiende es un paso fundamental para poder entender qué acciones realiza esa petición y qué respuesta se obtiene.

Para poder obtener, por lo tanto, una representación completa de la navegación y funcionalidad de una aplicación Web es requisito indispensable poder relacionar la petición con el controlador concreto que la atiende. En general, los frameworks de desarrollo web se encargan de proporcionar una representación para indicar esta relación y un mecanismo para derivar cada petición al controlador deseado. Sin embargo, en muchos casos, la granularidad de la especificación de la petición no es suficientemente fina y se delega, en última instancia, en la lógica del controlador la diferenciación entre peticiones semejantes pero distintas, como por ejemplo puede ocurrir en las peticiones generadas por distintos botones *submit* de un mismo formulario. Por lo tanto, para relacionar una petición con la lógica concreta que la atiende, en algunos casos, se debe primero dividir la lógica del controlador en los distintos flujos de control que lo componen, cada uno de los cuales se encarga de atender una petición distinta. Aunque esta división se abordará en el siguiente capítulo de esta tesis, es necesario dejar constancia de este problema derivado de la granuralidad del tratamiento de las peticiones.

La respuesta normal de un controlador consiste en devolver la página generada a partir de la población de una plantilla con los datos que haya recuperado o generado. Por lo tanto, se puede simplificar y establecer que la respuesta normal de un controlador lo constituye una página. En este caso, los frameworks de desarrollo web también pueden definir algún mecanismo que permita desacoplar la lógica del controlador de la plantilla utilizada para la respuesta, como sucede con Struts que permite definir en su configuración un mapeo nombre-plantilla que es posible alterar sin cambiar el código del controlador. Este tipo de mecanismos que mejoran la flexibilidad y extensibilidad de la aplicación complica, por otra parte, su entendimiento y la extracción de información.

Otro comportamiento posible de un controlador es delegar la respuesta a otro controlador. Esta situación es muy común en el procesamiento de los datos de entrada de un formulario, de manera que un controlador se encarga de realizar las operaciones relacionadas con la entrada de datos y delega en otro controlador la responsabilidad de recuperar la información para generar la página de respuesta. Por lo tanto, la ejecución de algunos controladores no responde a la atención directa a una petición, sino más bien a una atención delegada.

5.2.3.2. Mapeo entre elementos de la Vista y elementos del Modelo

En su forma más sencilla, podemos considerar al controlador como la lógica que se encarga de recoger los datos que le llegan de una petición y generar los datos que componen la página web de respuesta. En este sentido, siguiendo la notación propuesta, es necesario identificar, por un lado, los objetos y atributos de presentación que llegan con la petición y conforman la entrada al controlador, y por otro lado, los objetos y atributos de presentación generados para poblar la página de respuesta a la petición.

Una de las acciones más comunes en la lógica de un controlador consiste en mapear los datos de entrada en forma de objeto de presentación a un objeto de datos, realizando las conversiones de tipos

necesarias, para poder operar posteriormente con ese objeto de datos. Se pueden detectar tres tipos de mapeo fundamentales:

- Mapeo 1:1. Este mapeo indica que un objeto de presentación se corresponde con un solo objeto de datos, conformando una vista total o parcial sobre ese objeto de datos. Este mapeo corresponde al caso más sencillo de mapeo, siendo muy característico en operaciones de entrada de datos: creación o modificación de una entidad de datos.

- Mapeo 1:N. Este mapeo indica que un objeto de presentación se corresponde con varios objetos de datos, conformando una vista conjunta de los datos relacionados de los diferentes objetos de datos. Este mapeo normalmente se corresponde con objetos de presentación de salida en los que se muestra información relacionada de varios objetos de datos. Sin embargo, puede aparecer también en el caso de objetos de presentación de entrada que contienen un atributo cuyos valores deben ser seleccionados dentro de un rango definido por otro objeto de datos. Este mapeo, por lo tanto, puede sugerir la existencia de una relación entre los objetos de datos implicados.

- No existe mapeo definido. En este caso el objeto de presentación no se corresponde con ningún objeto de datos. Por lo tanto, ese objeto de presentación se utiliza para mostrar información que no se almacena de manera persistente en el sistema, por ejemplo, mostrar la hora actual, o para recoger datos que parametrizan operaciones pero que no están representados en el modelo.

5.2.3.3. Invocación a operaciones, paso de parámetros e instancias

Por un lado, una parte fundamental de un controlador está constituida por la invocación de operaciones de la lógica de negocio, que tienen como resultado realizar modificaciones en los datos del Modelo o recuperar datos para poder generar nuevas páginas de respuesta. Este trabajo se centra fundamentalmente en la llamada a operaciones definidas sobre los objetos de datos del Modelo, que son suficientemente representativas para poder especificar aplicaciones Web intensivas de datos. La secuencia de llamadas, el paso de parámetros entre llamadas, los objetos de datos sobre los que se realiza la operación y los valores de retorno obtenidos, que servirán para poblar páginas de respuesta o como parámetro de otra llamada, constituye información fundamental para poder obtener una representación suficiente de la aplicación web que se desea modernizar.

Por otro lado, a diferencia de la situación encontrada en los artefactos que definen la Vista o el Modelo, marcada por el carácter declarativo y centrado en la especificación de la estructura, los artefactos que definen el Controlador se caracterizan por su orientación imperativa enfocada en la especificación del comportamiento. Por lo tanto, mientras que la especificación de la Vista y el Modelo conlleva la definición de los objetos que las componen estructuralmente, la especificación del controlador conlleva la definición de instancias a partir de los tipos de datos definidos en el sistema, que mantienen los valores necesarios para definir el comportamiento, como por ejemplo el paso de parámetros a una operación.

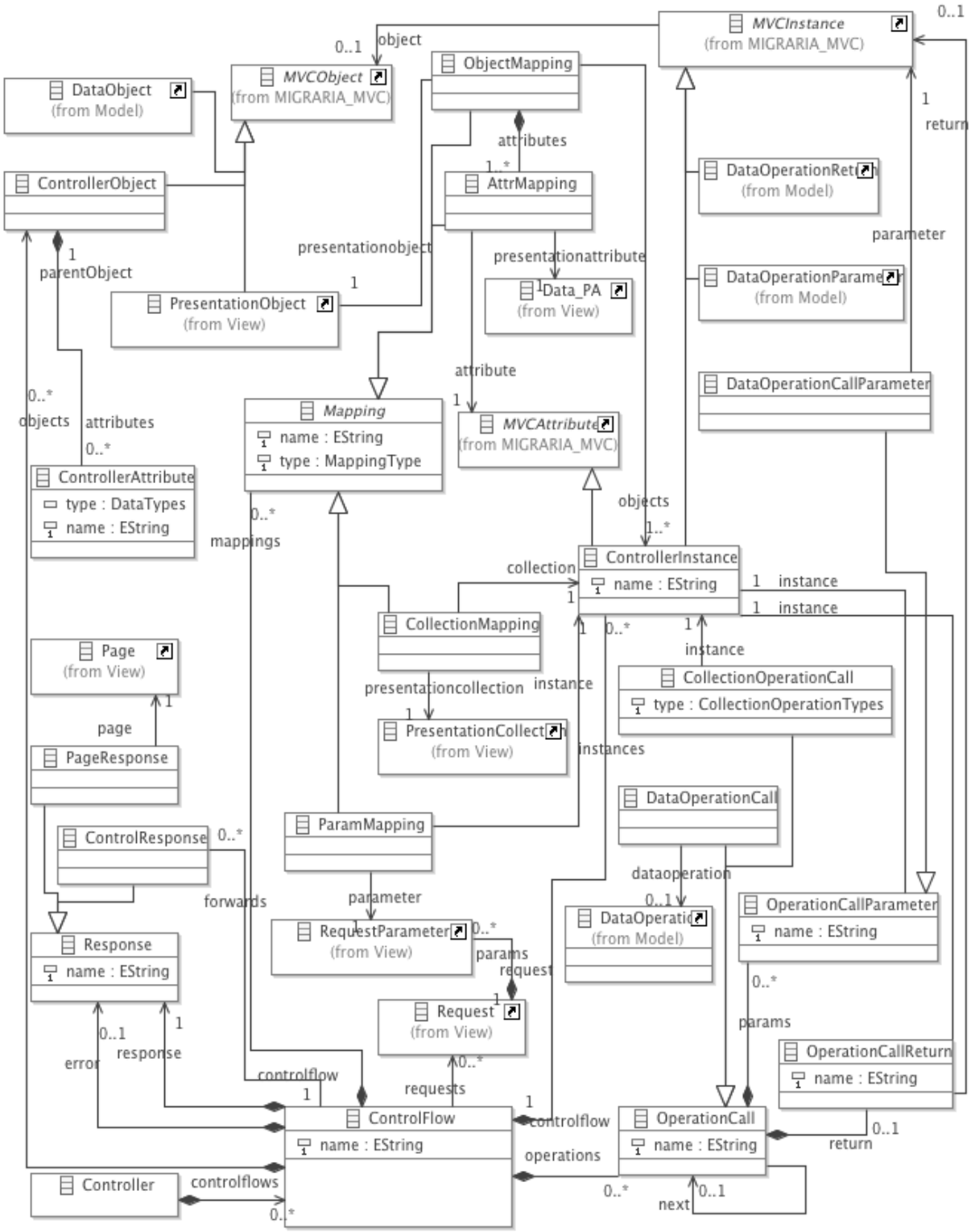


Figura 5.16: Metamodelo MVC Controller

5.2.3.4. Notación

El metamodelo presentado en la figura 5.16 recoge todos los conceptos definidos en los apartados anteriores, permitiendo representar con suficiente detalle para el objetivo de este trabajo los elementos del componente Controlador del patrón MVC.

Como se puede apreciar en la figura 5.16, el elemento fundamental de este metamodelo es la entidad *ControlFlow* que representa de manera genérica un flujo de control que atiende una determinada petición del cliente. Un modelo del componente Controlador de MVC estará definido, por lo tanto, a partir de una colección de elementos *ControlFlow*.

La entidad *ControlFlow* permite relacionar y organizar la información asociada a la atención de una petición, siendo básicamente: los objetos de entrada y salida, los mapeos entre la presentación y los datos, y la secuencia de operaciones invocadas dentro de la lógica de control asociada para atender la petición. Por lo tanto, presenta una estructura compleja formada fundamentalmente por los siguientes tipos de elementos: entidades *ControllerInstance*, entidades *Mapping* y entidades *OperationCall*. Además, la notación permite especificar el flujo petición-controlador-respuesta mediante la definición de las referencias *requests*, *forwards* y *response*. La figura 5.17 ilustra la representación del flujo de control encargado de realizar las operaciones necesarias para mostrar como respuesta la página presentada en la figura 5.6. En esta representación se pueden apreciar los elementos relacionados con la generación de esa página, como son los mapeos del objeto *conference* y de las colecciones *conferences*, *subjects* y *tracks*, así como las operaciones utilizadas para recuperar los datos de esos objetos y colecciones, que son *confDAO.get*, *confDAO.getAll*, *conf.getSubjects* y *trDAO.getAll*, respectivamente.

La respuesta de un flujo de control está representada por la entidad *Response*. Se contemplan dos tipos de respuestas, representadas por las entidades *PageResponse* y *ControlResponse*. La entidad *PageResponse* permite representar la respuesta formada por una página definida en la vista mediante la referencia *page*. Mientras que la entidad *ControlResponse* permite representar la delegación de la respuesta a otro flujo de control mediante la referencia *controlflow*. La Figura 5.18a muestra los detalles relacionados con la respuesta de este flujo de control. En este caso, se trata de un *PageResponse* que devuelve la página generada a partir de la plantilla *editSubjectAndTracks.jsp*.

En el caso de la petición, también podemos encontrar dos situaciones diferentes. La primera se caracteriza porque el inicio de la ejecución del flujo de control está especificado en la vista mediante una entidad *Request*, indicando que el flujo de control se ha iniciado desde la vista. La segunda situación se caracteriza porque el inicio de la ejecución del flujo de control se especifica mediante una entidad *ControlResponse* en el controlador, indicando que un flujo de control previo ha delegado la respuesta a la petición original en el flujo de control que se va a iniciar. La Figura 5.18b ilustra esta segunda situación, indicando que el flujo de control *SelectSubjectToModify* ha delegado la respuesta en el flujo de control actual.

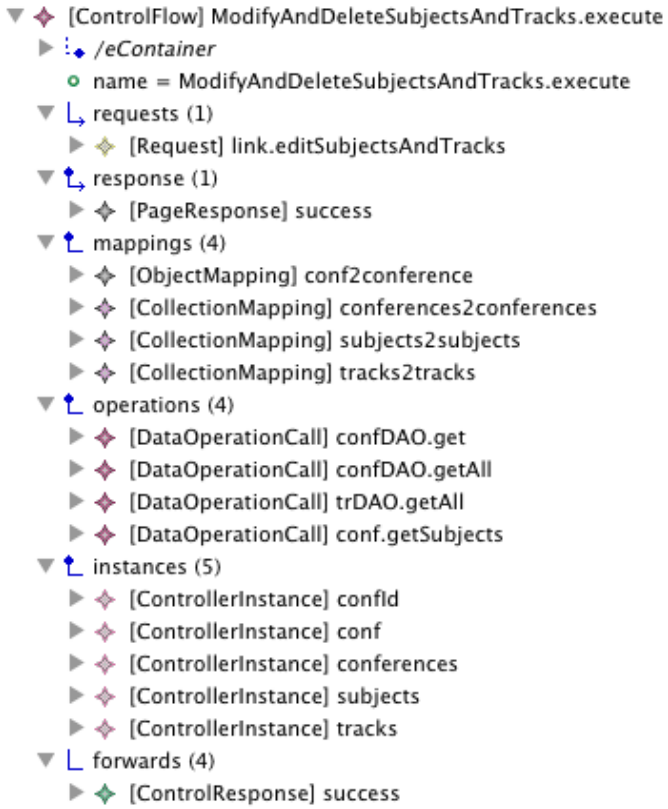
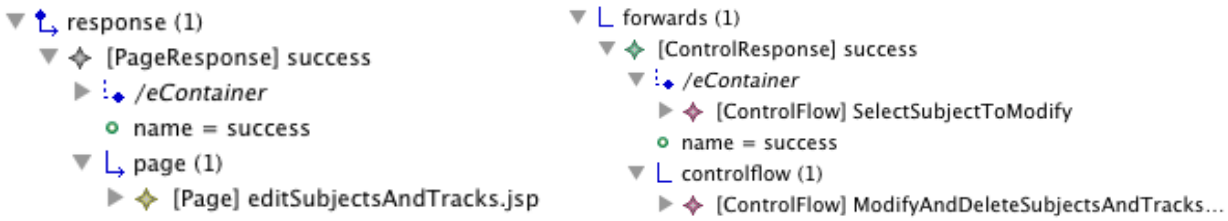


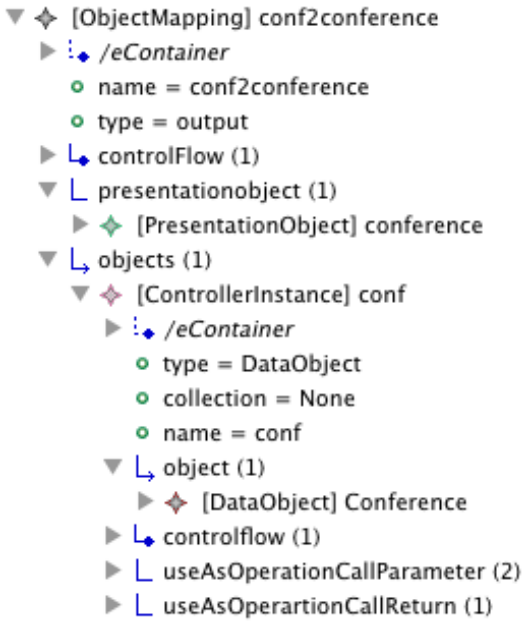
Figura 5.17: Ejemplo de flujo de control



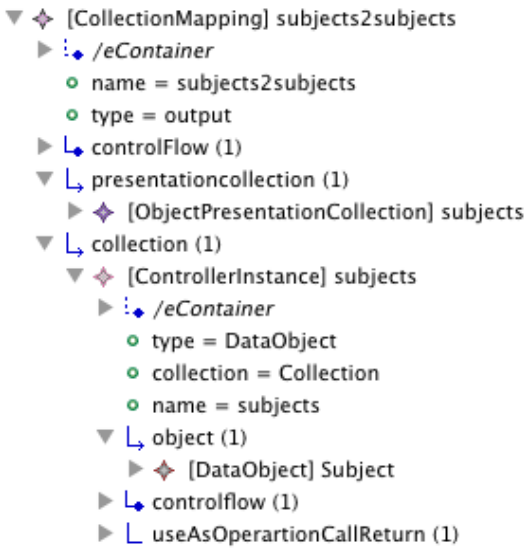
(a) Respuesta

(b) Petición

Figura 5.18: Ejemplo de ciclo petición-controlador-repuesta



(a) Mapeo de objetos



(b) Mapeo de colecciones

Figura 5.19: Ejemplo de mapeos

Las entidades *ControllerInstance* permiten representar cualquier instancia de un determinado objeto de datos (*DataObject*), de un tipo primitivo o de una colección de alguno de esos dos tipos que aparezca en el ámbito definido por el *ControlFlow*. Esta entidad es fundamental para poder representar tanto el mapeo de objetos de la vista y el modelo como el paso de parámetros entre la secuencia de operaciones invocadas. Como ilustra la Figura 5.16, tanto los mapeos como los parámetros y retorno de operaciones presentan una referencia a esta entidad.

Las entidades *Mapping* son las encargadas de especificar los mapeos tanto de entrada como de salida que existen entre los elementos definidos en las vistas y los objetos de datos originales a partir de los que están definidos. Los mapeos se pueden producir entre distintos elementos, básicamente se pueden considerar los siguientes mapeos según los tipos de elementos de la vista:

- Mapeo objeto de presentación con objeto de datos.
- Mapeo atributo de objeto de presentación con atributo de objeto de datos.
- Mapeo de colección de objetos de presentación con colección de objetos de datos generada en el controlador.

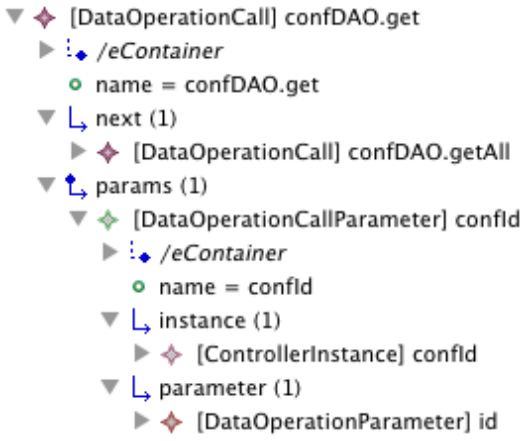
Esta categorización se ha utilizado para definir la jerarquía de mapeos con raíz en la entidad *Mapping* que aparece en la notación. Esta jerarquía está formada por las entidades *ObjectMapping*, *AttrMapping* y *CollectionMapping*, que permiten representar instancias de los mapeos comentados anteriormente. Por otro lado, es necesario señalar que la notación admite otros tipos de mapeo no relacionados con elementos del Modelo para poder especificar el uso de parámetros de peticiones (*ParamMapping*) o el paso de parámetros entre invocaciones a operaciones, entre otros.

Los mapeos permiten especificar si son de entrada o de salida. Los mapeos de entrada representan la entrada de datos al sistema, mientras que los mapeos de salida especifican los datos necesarios para generar la vista de respuesta a la petición. La combinación de la direccionalidad de los mapeos con la direccionalidad de los objetos de presentación nos da las siguientes posibilidades:

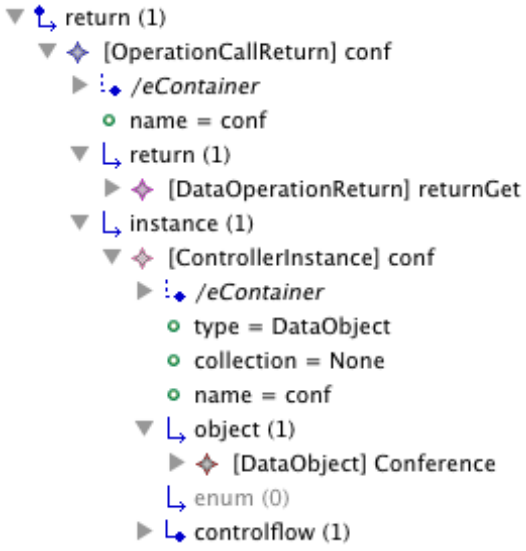
- Mapeo de entrada
  - Mapeo entre un objeto de presentación de entrada y objeto de datos. Permite representar la entrada de datos en la aplicación mediante el envío de los datos definidos en un formulario relacionados con los atributos del objeto de datos.
  - Mapeo entre un objeto de presentación de salida y objeto de datos. Esta situación no es posible. Un objeto de presentación de salida sólo puede aparecer en mapeos de salida.
- Mapeo de salida
  - Mapeo entre un objeto de presentación de entrada y objeto de datos. Permite representar la población de los campos de un formulario para la edición de los datos relacionados con los atributos del objeto de datos.
  - Mapeo entre un objeto de presentación de salida y objeto de datos. Permite especificar los atributos del objeto de datos que serán utilizados para la generación de la vista de respuesta.

La figura 5.19a presenta el mapeo de salida del objeto de presentación conference con el objeto de datos *conference*. En este caso, no se especifican mapeos entre los atributos de los objetos porque se pueden deducir de los atributos del objeto de presentación al ser un objeto de salida y un mapeo de salida. Por su parte, la figura 5.19b presenta el mapeo de salida de la colección de presentación *subjects* con la colección de tipo conjunto definida por la instancia de controlador *subjects* a partir del objeto de datos *Subject*.

Las entidades *OperationCall* son las encargadas de representar la secuencia de operaciones invocada dentro del *ControlFlow*. Básicamente, un elemento *OperationCall* está caracterizado por su nombre, por sus parámetros y por su retorno. La llamada a una operación puede tener varios parámetros y puede contener o no retorno. Para representar la secuencia se utiliza la referencia *next* que permite relacionar un determinado elemento *OperationCall* con el siguiente dentro de la secuencia de llamadas a operaciones definida. La entidad *OperationCallParameter* es la encargada de representar un parámetro en la llamada a una operación y permite especificar el elemento *ControllerInstance* que es usado como parámetro en esa invocación. La entidad *OperationCallReturn* es la encargada de representar el retorno de la llamada a una operación y permite relacionar el retorno especificado en la definición de la operación con el elemento *ControllerInstance* que actúa como receptor de ese retorno. En esta notación, se han



(a) Llamada a operación de datos



(b) Llamada a operación de colección

Figura 5.20: Ejemplo de llamadas a operaciones

definido dos tipos diferentes de llamadas a operación definidas por las entidades *DataOperationCall* y *CollectionOperationCall* que especializan a la entidad *OperationCall*.

Por una parte, una entidad *DataOperationCall* permite representar la invocación de una de las operaciones sobre los objetos de datos definidas en el componente Modelo, definiendo una referencia (*dataoperation*) al elemento que representa esa operación concreta. Para este tipo de operaciones se ha definido, además, un tipo especial de parámetro, *DataOperationCallParameter*, que extiende a *OperationCallParameter* mediante la definición de la referencia *parameter* que permite relacionar el elemento que representa al parámetro en la definición de la operación con la instancia del controlador que se pasa como parámetro. Las figuras 5.20a y 5.20b ilustran la representación de la llamada a operación sobre objeto de datos *confDAO.get*, que contiene un parámetro y devuelve un valor de retorno. Por un lado, la figura 5.20a presenta los detalles del parámetro *confId*, que consiste en un elemento *DataOperationCallParameter* que permite relacionar la instancia de controlador *confId* con el parámetro de la operación de datos (*DataOperationParameter*) *id*. Por otro lado, la figura 5.20b presenta los detalles del valor de retorno *conf*, que consiste en un elemento *OperationCallReturn* que permite establecer la relación entre la instancia de controlador *conf*, que es una instancia del objeto de datos *Conference* y que almacenará el resultado de la operación, con el elemento *DataOperationReturn returnGet*. Finalmente, en la Figura 5.20a, también se puede apreciar que la siguiente llamada de operación en la secuencia la constituye el elemento *DataOperationCall confDAO.getAll*.

Por otra parte, una entidad *CollectionOperationCall* permite representar la invocación de una operación sobre una colección de datos, definiendo una referencia (*instance*) a la instancia del controlador que representa la colección sobre la que se realiza la acción. El atributo *type* permite indicar el tipo de operación que se realiza sobre la colección: añadir, eliminar, vaciar y retener. Este atributo, junto con el

parámetro asociado a la llamada a la operación, permiten indicar si la operación se debe realizar sobre un objeto o sobre un conjunto de objetos.

Adicionalmente, se han incluido las entidades *ControllerObject* y *ControllerAttribute* que permiten representar un tipo de datos definido en el controlador y que no está relacionado a priori con ningún objeto de datos.

5.3. Resumen

Este capítulo presenta con detalle la notación definida para la actividad de representación del conocimiento de la etapa de ingeniería inversa dentro del proceso de modernización definido en el proyecto MIGRARIA. Esta notación, denominada MIGRARIA MVC, se ha definido específicamente para la representación de conocimiento derivado de un proceso de ingeniería inversa sobre una aplicación web heredada desarrollada con un framework de aplicaciones web basado en el patrón Modelo-Vista-Controlador. La notación definida es independiente tanto de la tecnología web como del framework utilizado en el desarrollo de la aplicación heredada. Sin embargo, es una notación que representa los conceptos fundamentales de desarrollo de una aplicación web heredada mediante un FAW y que se pretende que resulte sencillo de reconocer y utilizar por parte de los desarrolladores web implicados en el desarrollo y mantenimiento de la aplicación web heredada. Parte de estos desarrolladores estarán posiblemente implicados en el proceso de modernización y es, por tanto, necesario que sean capaces de entender y manejar la notación con el menor esfuerzo posible.

La notación se ha definido mediante un metamodelo ecore con el objetivo de poder utilizar todas las herramientas relacionadas con el proyecto EMF. Este metamodelo está organizado en tres paquetes, uno por cada componente del patrón MVC. El capítulo introduce con detalle los elementos de cada uno de estos paquetes y su semántica.

Capítulo 6

Generación de un Modelo Conceptual

> "Reverse engineering is the process of analyzing software to identify the software's components and their interrelationships and to create representations of the software in another form or at higher levels of abstraction." SWEBOK IEEE 2004

6.1. Ingeniería Inversa

La generación del modelo conceptual a partir de la aplicación web heredada supone la ejecución de un proceso de ingeniería inversa, que se corresponde con la primera fase del proceso de modernización propuesto por MIGRARIA. La figura 6.1 muestra los planes de extracción de información definidos, que conllevan la aplicación de técnicas de análisis estático y dinámico de los artefactos software presentes en cliente y en servidor.

Con respecto a la fase de ingeniería inversa, el trabajo presentado en esta tesis se centra en la aplicación de técnicas de análisis estático de los artefactos software de la aplicación presentes en el lado del servidor, es decir, el código fuente de la aplicación. El objetivo de este proceso es generar una representación de la aplicación web heredada en la notación MIGRARIA MVC definida en el capítulo anterior. La

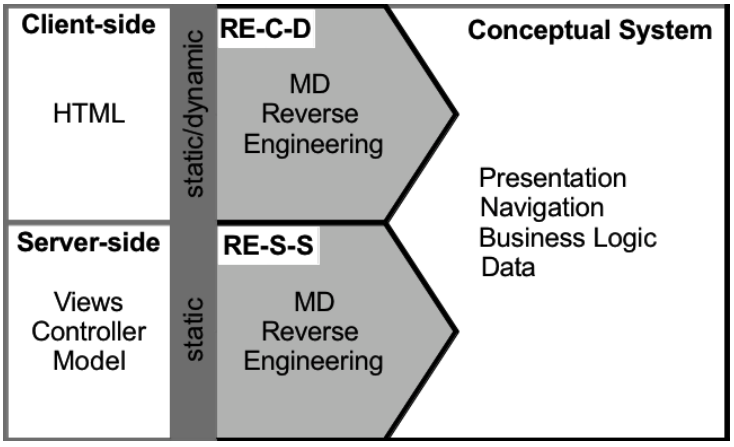


Figura 6.1: Fase de ingeniería inversa de MIGRARIA

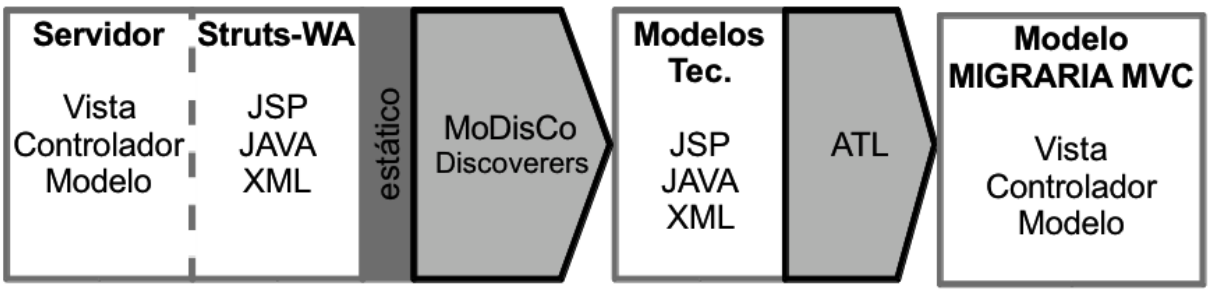


Figura 6.2: Ingeniería Inversa Dirigida por Modelos en el Servidor

figura 6.2 ilustra los diferentes pasos implicados en el proceso de ingeniería inversa dirigida por modelos que se ha definido. Básicamente, se pueden considerar los siguientes pasos:

1. Generación de los modelos dependientes de la tecnología. En el caso de estudio usado en este trabajo, se ha utilizado Struts 1.3 como framework que especifica la tecnología de desarrollo de la aplicación web. Por lo tanto, fundamentalmente se consideran las siguientes tecnologías: Java Server Pages (JSP), XML y Java. Y se deberán generar modelos a partir de los artefactos software definidos con estas tecnologías.

2. Generación del modelo MIGRARIA MVC. A partir de los modelos dependientes de la tecnología, se realiza un proceso de transformación de modelo a modelo con el fin de obtener una representación conceptual de la aplicación web heredada, siguiendo la notación propuesta en este trabajo. De esta manera, se consigue integrar en un solo modelo toda la información relacionada con la aplicación web heredada desde el punto de vista del patrón MVC.

3. Proyección a una propuesta de ingeniería web dirigida por modelos (no mostrado en la figura), (Rossi *et al.* 2008b). Como paso opcional del proceso de generación de una representación conceptual, se establece la posibilidad de generar una representación de la aplicación web heredada en alguna de las propuestas de IWDM más conocidas, como por ejemplo, WebML o OOH4RIA. Esta proyección es totalmente opcional, pero puede suponer una alternativa interesante en alguno de los escenarios de modernización considerados. Este último paso será presentado en el capítulo 8.

6.2. Generación de los modelos dependientes de la tecnología

El proceso de ingeniería inversa empieza por la generación de los modelos dependientes de la tecnología a partir de los artefactos software de la aplicación web heredada presentes en el servidor. Por lo tanto, este proceso de generación se puede definir como una transformación de texto a modelo. Con el objetivo de sistematizar esta transformación, se ha usado MoDisco (Bruneliere *et al.* 2010): un proyecto de Eclipse enfocado en la modernización de sistemas heredados. Este proyecto proporciona una serie

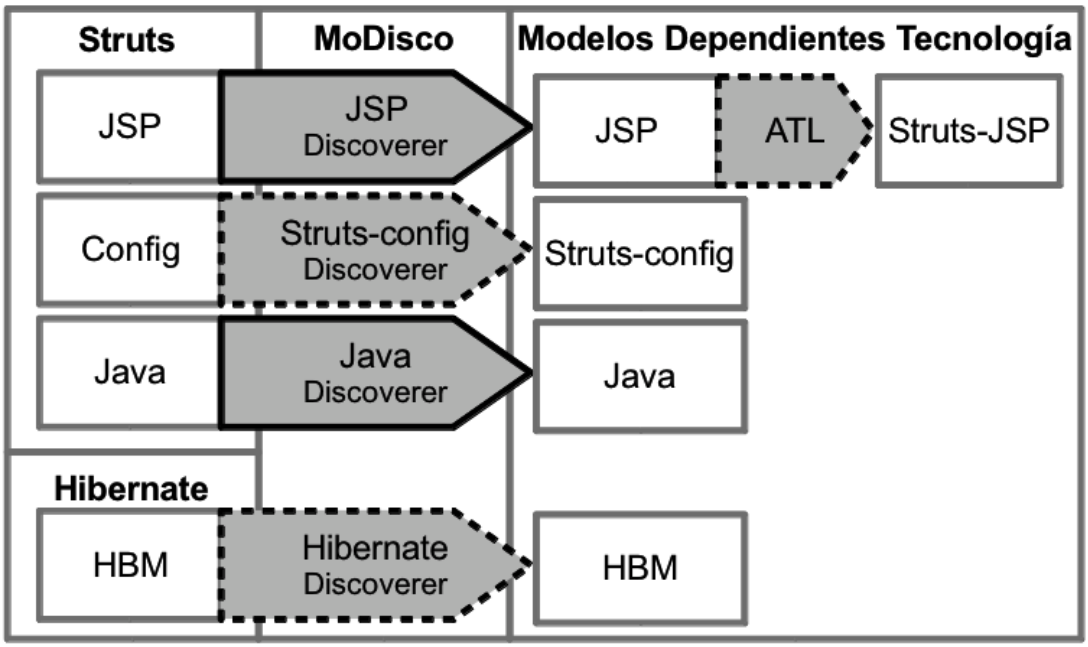


Figura 6.3: Extensiones de MoDisco

de subsistemas, denominados *discoverers*, que permiten realizar la transformación de texto a modelo de diferentes tipos de artefactos software, entre otros: JavaServer Pages, XML y JAVA.

Las tecnologías relacionadas con el framework web de referencia, Struts 1, están basadas en JSP, XML y JAVA. Por un lado, Struts proporciona varias bibliotecas de etiquetas que extienden a la biblioteca estándar de etiquetas para JSP[1]. Por otro lado, la configuración de Struts que define fundamentalmente los mapeos petición-acción se especifica en una sintaxis definida en XML. Y, finalmente, para definir el código de los controladores se utiliza el lenguaje de programación JAVA con las clases definidas por el framework Struts. Por lo tanto, aunque los discoverers de MoDisco pueden ayudar en la generación de los modelos dependientes de la tecnología, es necesario extenderlos para que los modelos puedan representar las particularidades introducidas por el framework Struts.

En este sentido, como ilustra la figura 6.3, ha sido necesario desarrollar tres extensiones fundamentales (resaltadas con líneas discontinuas): una para generar un modelo JSP que incluya las etiquetas definidas por Struts; y otra para generar un modelo que represente un archivo de configuración de Struts. La primera se ha definido como una transformación modelo a modelo que toma como entrada el modelo JSP generado por el discoverer JSP de MoDisco. Por su parte, la segunda ha consistido en la implementación de un nuevo discoverer para poder procesar archivos de configuración de Struts convenientemente. En ambos casos, ha sido necesaria la definición de metamodelos (Struts-JSP y Struts-config) que recojan los conceptos fundamentales del framework Struts. Adicionalmente, se ha desarrollado una tercera extensión para dar soporte a los ficheros de configuración del gestor de persistencia Hibernate[2], siguiendo la misma aproximación que para el fichero de configuración.

[1] http://struts.apache.org/1.x/struts-taglib/index.html
[2] http://www.Hibernate.org/

Por su parte, para la generación del modelo a partir del código JAVA se ha utilizado directamente el metamodelo y el *discoverer* que proporciona MoDisco, que permite representar el árbol de sintaxis abstracta del código (Abstract Syntax Tree).

A continuación se proporciona una descripción de estas dos extensiones, sus metamodelos y sus procesos de generación de los modelos conformes a ellos.

6.2.1. Extensión Struts-JSP

El metamodelo *Struts-JSP*, figura 6.4, define los elementos necesarios para poder representar con detalle todas y cada una de las etiquetas introducidas por las bibliotecas de etiquetas de Struts (*html*, *logic* y *bean*). La misión de este metamodelo es fundamentalmente facilitar la extracción de información del modelo de JSP generado por MoDisco, simplificando la detección de la semántica asociada a las etiquetas introducidas por Struts, como por ejemplo, la relación de composición que existe entre la etiqueta que define un formulario y las etiquetas que definen los campos de control del mismo.

Este metamodelo se ha definido como una extensión del metamodelo *JSP* utilizado por MoDisco siguiendo las directrices del mecanismo de extensión definido para las librerías JavaServerPages Standard Tag Library *(JSTL)* [3], que consiste en crear nuevas etiquetas a partir del elemento base de definición de etiquetas *JSPAction*. En este sentido, se ha definido una taxonomía a partir de esta metaclase del metamodelo de JSP que permite representar las etiquetas introducidas por las bibliotecas de etiquetas de Struts (*html*, *logic* y *bean*). Esta taxonomía está formada por un primer nivel que define una nueva metaclase como raíz de las etiquetas de cada biblioteca: *JSPActionStrutsHTML*, *JSPActionStrutsBean* y *JSPStrutsLogic*. A partir de las mismas, se definen las metaclases que representan a las etiquetas concretas presentes en estas bibliotecas.

Con el objetivo de refinar el modelo JSP generado por MoDisco se ha definido una transformación modelo a modelo que genera una versión extendida de este modelo que incluya las etiquetas propias de Struts como entidades de primer nivel. No es, por tanto, una extensión realizada a MoDisco propiamente dicha, como podría haber sido definir un nuevo *discoverer*. En este caso, tiene más sentido la alternativa seguida porque permite mantener un mayor grado de adaptabilidad frente a la evolución del *discoverer* de MoDisco para *JSP*.

Básicamente, el procedimiento de refinado seguido consiste principalmente en definir las reglas que copien los elementos del modelo origen al modelo destino sin ningún cambio. Posteriormente en el caso de la regla que genera los elementos *JSPAction* se filtran aquellos elementos que representen etiquetas propias de Struts (html:*, logic:* o bean:*). Y, finalmente, se definen las reglas que tratan de manera especial a estos elementos generando sus versiones correspondientes en el modelo extendido conforme al metamodelo Struts-JSP .

[3] http://www.oracle.com/technetwork/java/index-jsp-135995.html

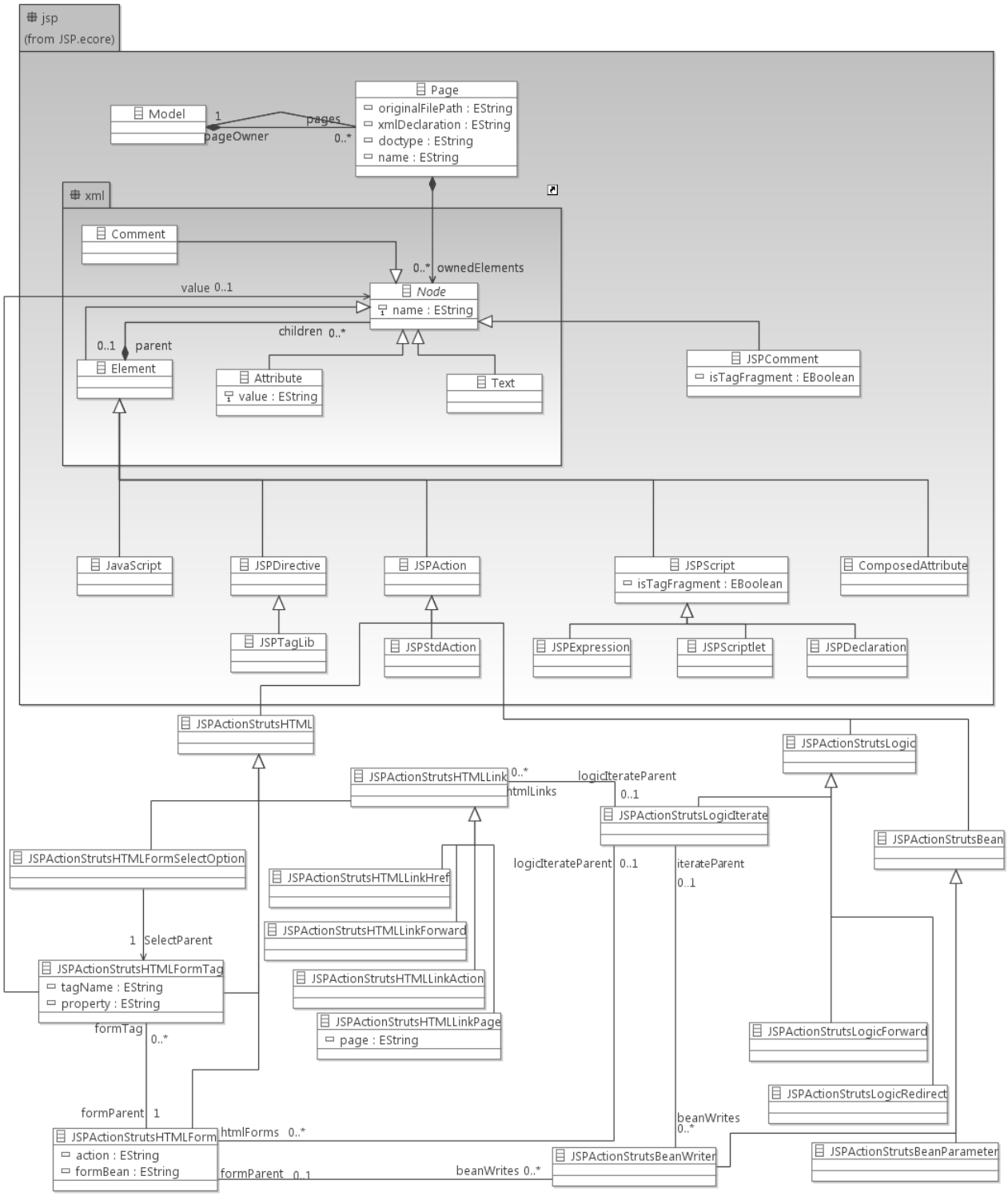


Figura 6.4: Struts JSP taglibs metamodel

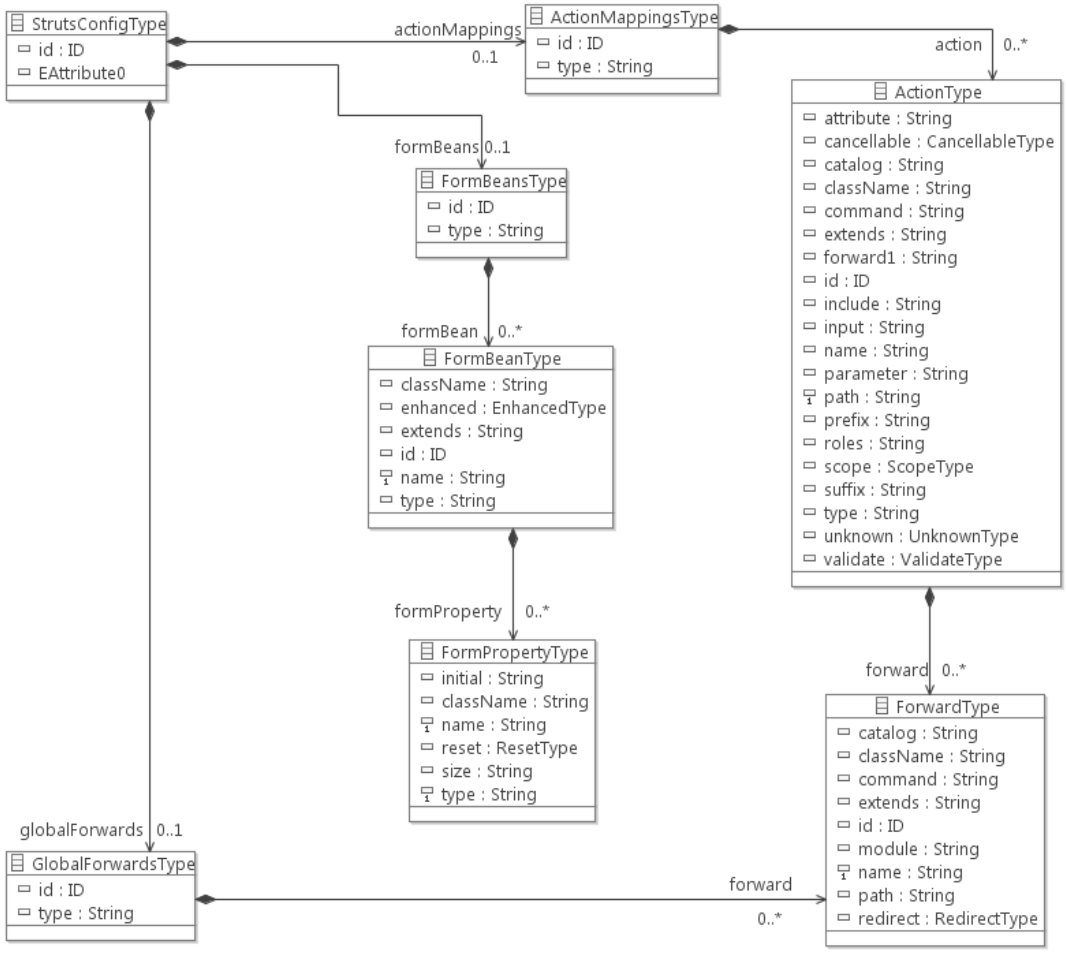


Figura 6.5: Extracto del metamodelo Struts-Config

6.2.2. Extensión Struts-Config

El metamodelo Struts-Config define la estructura de los archivos de configuración de Struts usados para parametrizar el comportamiento del *servlet* que actúa como *Front Controller*. La misión de este metamodelo consiste fundamentalmente en simplificar la definición de las reglas de transformación que procesan, posteriormente, en el proceso de modernización, los modelos que representan archivos de configuración de Struts.

Un archivo de configuración se especifica mediante un lenguaje basado en XML. Por lo tanto, es posible utilizar el metamodelo y *discoverer* para XML que proporciona MoDisco para obtener un modelo del mismo. Sin embargo, este modelo no tenía en cuenta el esquema definido para los archivos de configuración, complicando sobremanera la definición de reglas de transformación posterior. Por ejemplo, por un lado, se debían construir funciones de ayuda encargadas de la búsqueda de los elementos por nombre en lugar de poder definir reglas de transformación por tipo de elemento. Por otro lado, a la hora de obtener los hijos de un elemento se debía llevar a cabo un filtrado debido a los diferentes tipos de hijos.

El esquema de archivo de configuración del framework Struts 1.X ha ido evolucionando a lo largo de las diferentes versiones del mismo, desde la 1.1 hasta la 1.3.10. Por lo tanto, ha sido necesario definir un metamodelo Struts-Config por cada una de estas versiones. La generación de estos metamodelos se ha realizado mediante un proceso automático. Este proceso toma como fuente los archivos DTD (Document Type Definition) que definen la estructura de los archivos de configuración y genera una descripción en formato XSD (XML Schema Definition) mediante cualquier herramienta XML que proporcione esta funcionalidad [4]. EMF (Eclipse Modeling Framework) puede generar automáticamente un metamodelo ecore a partir de un archivo XSD siguiendo las siguientes reglas básicas:

- Los elementos del árbol XML se convierten en objetos en el metamodelo Ecore. El nombre que se asigna a estos elementos se establece utilizando el nombre original del elemento al que se concatena *type*. Por ejemplo en el metamodelo correspondiente a los archivos de configuración de Struts los elementos *Forward* en el metamodelo se denominan *ForwardType*.

- Los atributos que se pueden indicar para un elemento del diagrama XML se establecen como propiedades del objeto destino en el metamodelo.

- Las relaciones de anidamiento de elementos se traducen en relaciones de inclusión.

En la figura 6.5 se muestra un fragmento del metamodelo Struts-Config correspondiente a la versión 1.1 del archivo de configuración de Struts utilizado en el proceso de modernización.

En este caso, la extensión realizada a MoDisco consiste en desarrollar un nuevo *discoverer* que sea capaz de procesar cualquier versión de los archivos de configuración de Struts 1.X. Este *discoverer* se ha construido siguiendo la estructura de los *discoverers* WebApp de MoDisco para el análisis de los archivos de configuración del servidor de aplicaciones (web.xml). Básicamente, estos *discoverers* aprovechan la característica de serialización de EMF que permite la persistencia y carga de modelos en formato XML cuando el metamodelo se ha generado a partir de un archivo XSD.

6.2.3. Extensión Hibernate

Siguiendo la misma aproximación que en el caso del fichero de configuración de Struts, se ha obtenido un metamodelo para poder especificar las configuraciones de mapeo entre los objetos de la aplicación y las tablas del sistema gestor de base de datos mediante el sistema de gestión de persistencia Hibernate. Entre los diferentes modos de especificar este mapeo, hemos considerado el de ficheros en XML de Hibernate. Estos ficheros son conformes a un esquema XML para la especificación de mapeos definido por Hibernate.

Un fichero de configuración de mapeo en Hibernate permite especificar fundamentalmente los siguientes mapeos:

- Objeto con tabla.

[4] Por ejemplo: http://www.xmlutilities.net/

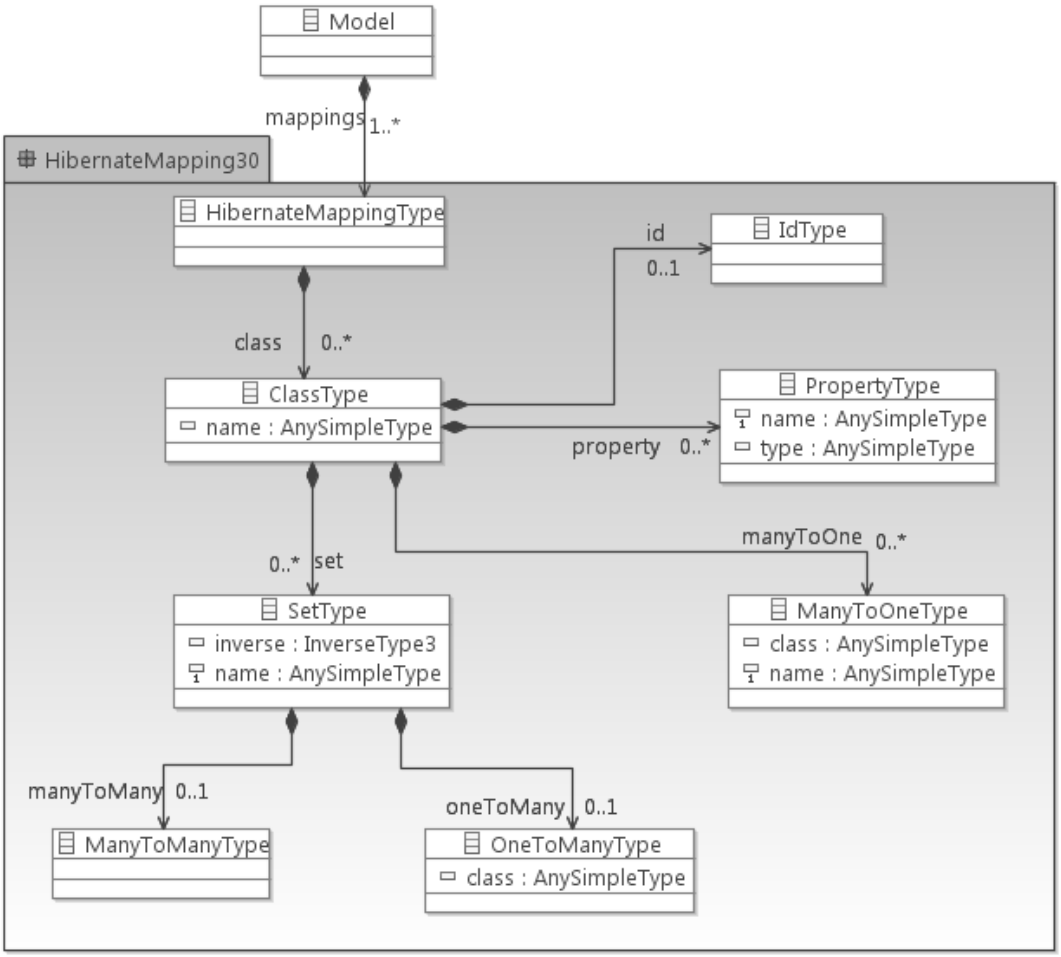


Figura 6.6: Extracto del metamodelo Hibernate

- Atributo de objeto con columna de tabla.
- Relaciones entre objetos con columnas de clave externa.

La figura 6.6 muestra el metamodelo Hibernate que hemos definido para la versión 3.6 de este sistema de gestión de persistencia.

Como se ha comentado anteriormente, para este caso también se ha extendido MoDisco mediante el desarrollo de un nuevo *discoverer* que sea capaz de convertir en modelos los ficheros de configuración de mapeo de Hibernate.

6.2.4. Resultados del proceso

Como resultados de esta primera parte del proceso, se obtienen los modelos dependientes de las tecnologías a partir de los artefactos software de la aplicación web heredada. Para la automatización de este proceso se ha usado, y extendido convenientemente, el proyecto MoDisco. De esta manera, se han creado los modelos dependientes de la tecnología conformes a metamodelos que representan las gramáticas de los lenguajes utilizados para la especificación de los artefactos software origen. Estos

modelos generados serán, como ilustra la Figura 6.2, la entrada de la siguiente parte del proceso que tiene como objetivo la generación de un modelo MIGRARIA MVC de la aplicación web heredada.

6.3. Generación del modelo MIGRARIA-MVC

El proceso de generación del modelo MIGRARIA-MVC a partir de los modelos dependientes de la tecnología se organiza en base a la siguiente secuencia de pasos:

1. Generación del modelo del componente Modelo.
2. Generación del modelo del componente Vista.
3. Generación del modelo del componente Controlador.

En los siguientes apartados se exponen, de manera conceptual, las reglas de transformación básicas para la obtención de un modelo MIGRARIA-MVC a partir de los modelos dependientes de la tecnología obtenidos en el primer paso mediante MoDisco. El Anexo A recoge la implementación de todas las reglas del proceso en ATL. Estas reglas, como ya se ha comentado previamente, deben ser específicas para la tecnología empleada para el desarrollo de la aplicación Web heredada. En este trabajo, se ha usado el framework de aplicaciones Web Struts 1.3.10 y, por lo tanto, se considera la siguiente tecnología relacionada:

- páginas generadas mediante la tecnología JavaServer Pages con etiquetas propias del framework;
- archivo de configuración del controlador frontal especificado mediante XML conforme al esquema definido para este tipo de archivos y la versión determinada del framework;
- y código JAVA para la implementación del código de control de la aplicación dentro de las clases y conceptos definidos por el framework.

Las reglas de transformación son también sensibles a las reglas de negocio[5] concretas usadas en el desarrollo del sistema heredado. La definición de unas reglas de negocio compartidas por todos los miembros del equipo de desarrollo es una práctica común en el desarrollo software. Estas reglas de negocio permiten definir con mayor granularidad las reglas de transformación. De esta manera, las reglas de transformación van a estar parcialmente ligadas a las reglas de negocio seguidas.

A continuación, se exponen estas reglas organizadas por los diferentes componentes del patrón MVC, de manera que se vea reducida la complejidad de su redefinición para otro framework Web diferente que siga el mismo patrón de diseño.

[5]Estilo o convenios de programación

6.3.1. Reglas para el Modelo

La definición de los datos, sus relaciones y las operaciones que se pueden realizar con ellos forman parte del componente Modelo dentro del patrón MVC. La definición de los datos y sus relaciones se puede realizar a diferentes niveles de abstracción y con diferentes tecnologías. El escenario más simple en una aplicación web consiste en definir el modelo de datos en un sistema relacional de gestión de base de datos, mientras que las operaciones son especificadas y convertidas a SQL en el código de la aplicación, que se comunica con el SGBD mediante un driver. Este escenario ha sido recogido por gran parte de los frameworks de persistencia de objetos, simplificando la definición de los datos, las relaciones y las operaciones. Algunos frameworks de aplicaciones web permiten definir todos los elementos relacionados con el Modelo, mientras que otros, como en el caso de Struts, no especifican nada y recomiendan el uso de un framework de persistencia de objetos externo. Éste es, por tanto, el escenario que hemos elegido para este trabajo, seleccionando como solución el framework Hibernate[6] en su versión 3.6, cuyo uso está muy extendido a nivel profesional.

En este trabajo, hemos seguido el siguiente esquema de uso de Hibernate:

- Definir una clase como mapeo de cada tabla de la base de datos, por ejemplo, la clase *Conference* en el caso de estudio. Todas estas clases estarán dentro del paquete <*lwa*>.*model*.
- Definir el mapeo clase-tabla en un archivo XML, que permite indicar el mapeo de atributos y el mapeo de relaciones, por ejemplo, el mapeo *Conference.hbm.xml* en el caso de estudio. Todos estos archivos estarán dentro de una carpeta *mappings*.
- Definir una interfaz Service y una clase que lo implemente, por ejemplo, la interfaz *ConferenceDAO* y la clase *HibernateConferenceDAO* en el caso de estudio. Un service en Hibernate permite definir las operaciones que se realizan sobre los objetos de datos. El concepto Service es una implementación del patrón Data Access Object. Todas estas clases estarán dentro del paquete <*lwa*>.*model.dao*.

Los siguientes apartados ilustran las reglas fundamentales para la generación de objetos y atributos de datos, relaciones entre objetos de datos y operaciones.

6.3.1.1. Reglas para objetos y atributos

Como primer paso de la generación del modelo de datos, se obtienen todos los objetos de datos a partir de las clases que representan entidades del modelo de datos. La generación de los objetos de datos y sus atributos a partir de estas clases no es un proceso complejo. Básicamente, consiste en procesar el fichero de mapeo para extraer el nombre del objeto de datos y sus atributos, además de identificar los atributos que son clave primaria y los tipos concretos de cada uno de los atributos. En un segundo

[6] http://www.Hibernate.org/

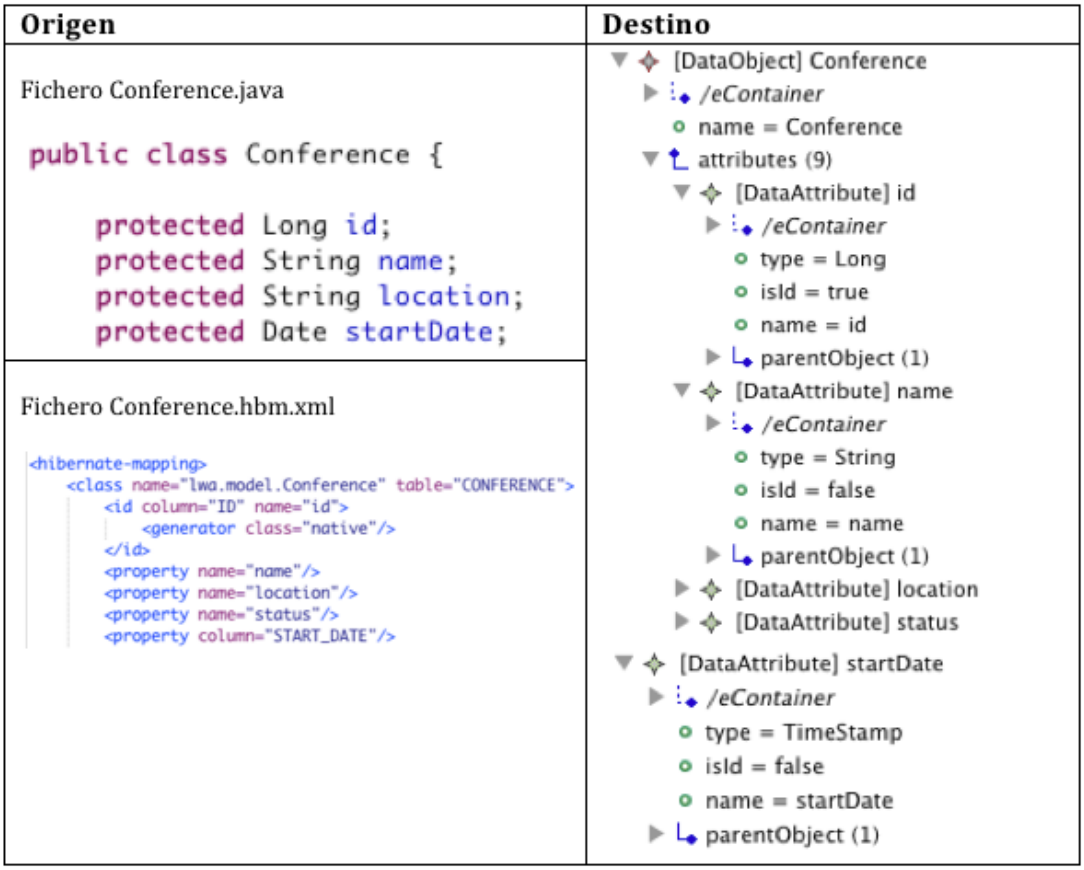


Figura 6.7: Generación de objeto de datos

paso, se procesa la clase asociada para extraer información adicional, como los tipos de los atributos no especificados en el mapeo o los métodos de acceso y modificación de atributos. La figura 6.7[7] representa la generación del objeto de datos Conference a partir de la información de la clase de datos y el fichero de mapeo correspondiente.

Un caso especial de atributo lo constituye el atributo derivado, que no se corresponde directamente con ningún campo de la tabla, sino que se calcula a partir de otros datos.

El resultado de este primer paso, por lo tanto, es un conjunto de objetos de datos no relacionados entre sí.

6.3.1.2. Reglas para relaciones

Una vez generados todos los objetos de datos, se generan las relaciones que existen entre ellos mediante el análisis de los ficheros de mapeo de Hibernate que especifican las relaciones existentes en el modelo de datos y las mapean en atributos concretos de las clases de datos. La figura 6.8 ilustra esta regla para la generación de la relación muchos-a-muchos entre la entidad *Conference* y *Subjects*.

El resultado de este segundo paso es, por lo tanto, un modelo con todos los objetos de datos, sus atributos y relaciones.

[7] Se ha usado como origen extractos de código JAVA, porque su lectura e interpretación es más sencilla que la del modelo JAVA

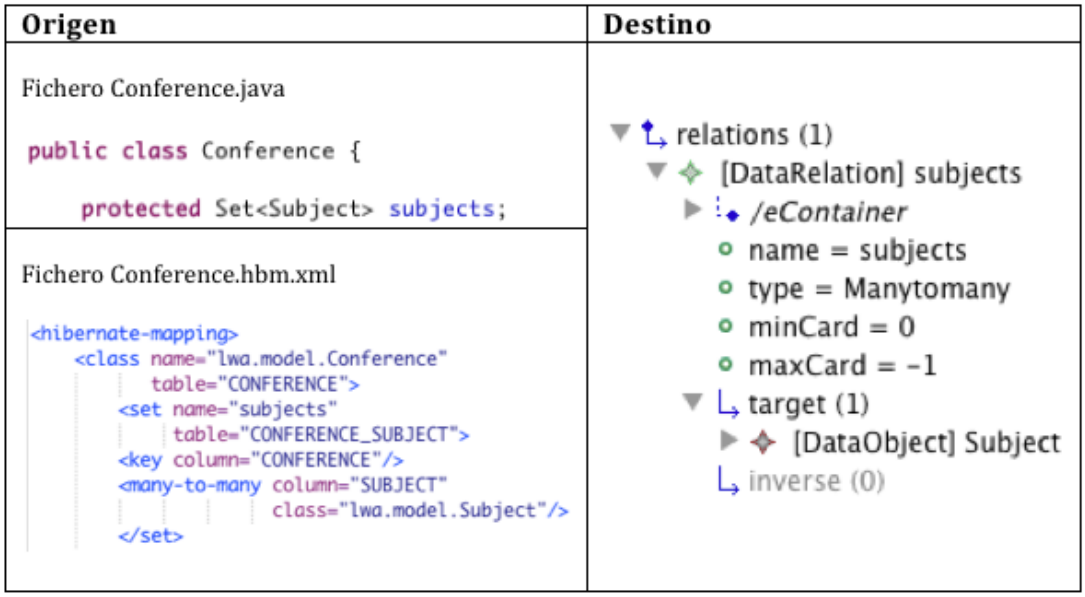


Figura 6.8: Generación de relaciones entre objetos de datos

6.3.1.3. Reglas para operaciones

En este caso, como se especificó en la explicación del metamodelo MIGRARIA MVC, distinguimos entre dos tipos de operaciones: operaciones sobre objetos de datos (DataOperation) y operaciones sobre relaciones entre objetos de datos (DataOperationByRelation). Evidentemente, las reglas definidas para su identificación son distintas.

En el primer caso, para extraer la información relacionada con las operaciones definidas sobre el modelo de datos, es necesario procesar la definición de los métodos especificados en la interfaz DAO para cada uno de los objetos de datos definidos. El objetivo de este proceso consiste en identificar el tipo de operación a realizar, los parámetros de la operación y los datos resultados de la misma. Para identificar el DAO relacionado con una determinada clase de objeto de datos, se ha seguido el convenio *<nombre_clase>DAO* para la especificación del identificador del DAO .

La identificación del tipo de operación se puede realizar mediante dos vías complementarias. Por un lado, el análisis sintáctico del identificador del método a ejecutar del DAO; ya que, incluir el nombre base de la operación dentro de este identificador es un convenio muy extendido. Por otro lado, se puede mantener un mapeo que asocie el tipo de operación realizada a cada uno de los métodos relacionados de la API de Hibernate, por ejemplo, asociar el tipo de operación *create* al método *org.hibernateSession.save*. La figura 6.9 ilustra la primera alternativa.

Por su parte, en el caso de las operaciones derivadas de relaciones, se detecta la presencia de una relación entre dos objetos y qué objeto de datos juega el rol de maestro en esa relación, mediante el análisis del objeto de datos y su mapeo con Hibernate. Adicionalmente, se comprueba la existencia de métodos en el objeto maestro para la recuperación de los objetos producto de esta relación. Por ejemplo, como se ilustra en la Figura 5.3, la presencia de los métodos *getSubjects* y *setSubjects* que actúan sobre

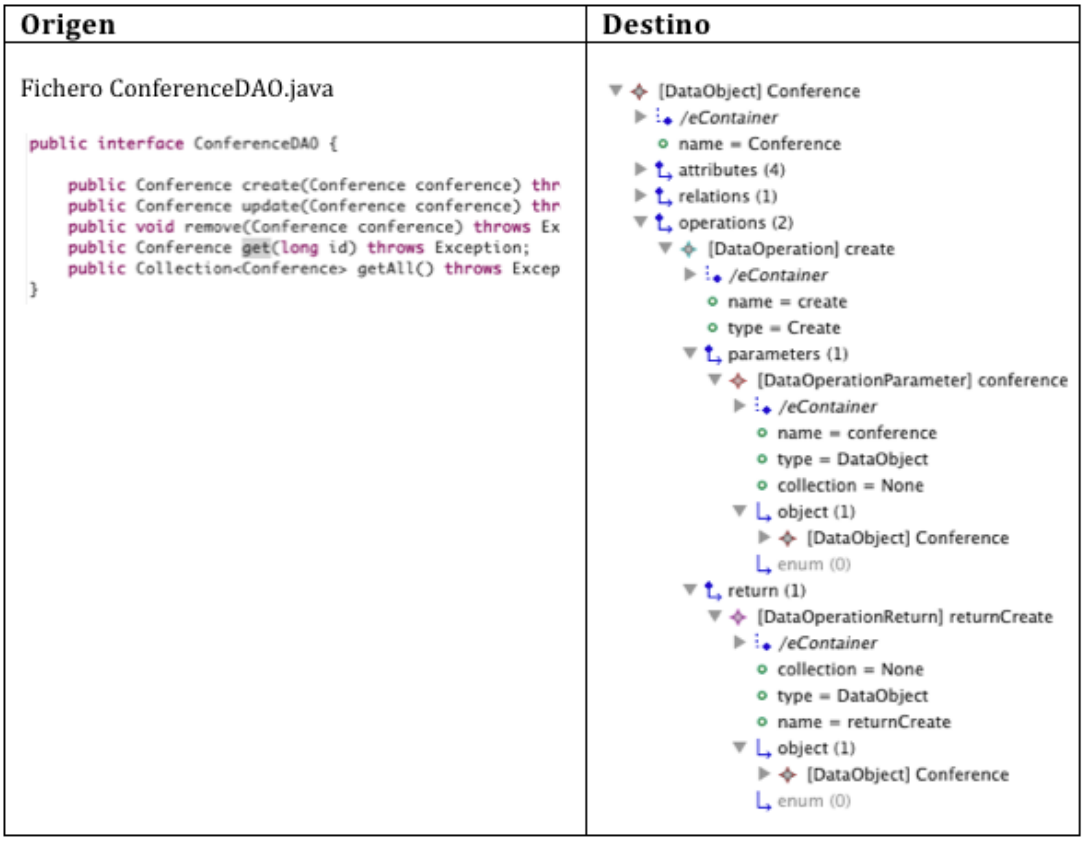


Figura 6.9: Generación de operaciones de los objetos de datos

el atributo *subjects* que contiene los objetos del tipo *Subjects* que están relacionados con un determinado objeto *Conference*.

El resultado de este tercer paso consiste en la descripción completa del componente Modelo del patrón MVC.

6.3.2. Reglas para la Vista

El elemento contenedor principal dentro del componente Vista es la página (elemento *Page*). Por lo tanto, la primera regla a definir es la encargada de generar un elemento de tipo *Page* en nuestro modelo MIGRARIA-MVC. Esta regla simplemente realiza el mapeo de los elementos de tipo *Page* del modelo *Struts-JSP* con el elemento *Page* de nuestro modelo.

La generación de los elementos que componen una página, sin embargo, si presenta un gran número de casos diferentes que conviene repasar con un mayor nivel de detalle. Con el fin de organizar su exposición, las reglas se han organizado en tres grupos diferentes:

- Reglas para objetos y atributos de presentación de salida.
- Reglas para objetos y atributos de presentación de entrada/salida.
- Reglas para peticiones.
- Reglas para colecciones.

6.3.2.1. Reglas para objetos y atributos de presentación de salida

Los objetos de salida no son identificables directamente; ya que, en las JSP sólo se especifica las propiedades del objeto de salida que se quiere mostrar en la página mediante la etiqueta <*bean:write*>. Por lo tanto, el patrón que marca la aparición de un objeto de salida es una colección de etiquetas <*bean:write*> que se refieren a un mismo bean que implementa el objeto de salida. Se pueden dar dos casos base:

1. Las etiquetas <*bean:write*> no están contenidas en ninguna etiqueta contenedora como <*logic:iterate*> o <*html:form*> y, por lo tanto, usan su atributo *name* para referirse al bean que representa el objeto de salida, mientras que su atributo *property* indica la propiedad determinada del bean a mostrar. En este caso, un objeto de presentación estará definido por el conjunto de etiquetas <*bean:write*> que comparten el mismo atributo *name*.

2. Las etiquetas <*bean:write*> están contenidas en alguna etiqueta contenedora como <*logic:iterate*> y, por lo tanto, usan su atributo *name* para referirse a una propiedad del bean indicado en el contenedor.

3. Las etiquetas <*bean:write*> están contenidas en alguna etiqueta contenedora como <*logic:iterate*> o <*html:form*>, pero proporcionan información tanto para su atributo *name* como su atributo *property*. En este caso, no se considera la información proporcionada por el contenedor y se realiza el mismo tratamiento que en el caso 1.

La figura 6.10 ilustra la generación del objeto de presentación de salida para el primero de estos casos. En este ejemplo, dentro de la página bajo análisis se detecta una secuencia de etiquetas <*bean:write*> que comparten el mismo valor para su atributo *name*, en este caso el identificador *conference*. A partir de esta secuencia, se genera el objeto de presentación de salida *conference* y los atributos que lo componen, en este caso: *name*, *submissionDate*, *location*, *reviewDate*, *notificationDate*, *startDate* y *endDate*. Tanto el objeto de presentación como sus atributos son generados con direccionalidad de salida. Al no encontrarse dentro de un formulario, ninguno de estos atributos va a ser un campo oculto. Y, finalmente, establecer si almacenan un valor de clave primaria para ese objeto no puede determinarse con el análisis de los artefactos referentes a la Vista; por lo tanto, la correcta asignación del valor del atributo *isId* es relegada al momento de procesamiento del Controlador.

Un caso aparte lo constituyen los atributos compuestos de un objeto de salida que pueden presentar múltiples valores. En ese caso, el atributo aparecerá representado como una etiqueta <*logic:iterate*> que permita recorrer y mostrar sus múltiples valores. La representación en la JSP de este caso coincide con el caso 2 apuntado anteriormente; por lo tanto, para poder diferenciar entre estos dos casos es necesario incluir información del contexto en el patrón a localizar. En el caso de representar un atributo de un objeto, la etiqueta <*logic:iterate*> estará acompañada por otras etiquetas <*bean:write*> con el mismo valor del atributo *name* que muestren los valores de otras propiedades del mismo objeto. La figura 6.11

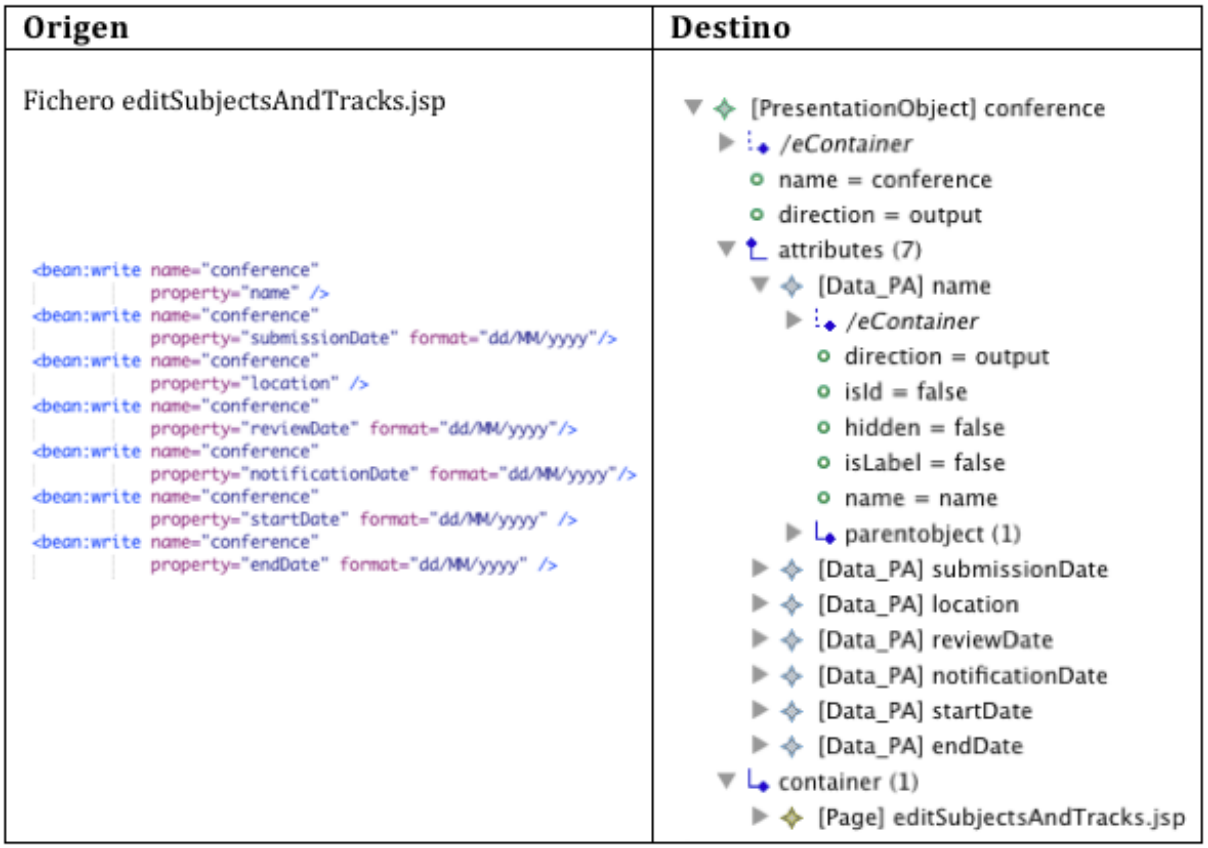


Figura 6.10: Generación de objeto salida con atributo simple

ilustra la generación del atributo complejo *subjects* del objeto de presentación *paperr*. Los valores de la propiedad *subjects* para este objeto se muestran mediante una etiqueta <*logic:iterate*>, que genera el atributo de presentación *subjects* de tipo *DataSet*, cuyos valores se especifican mediante una colección de presentación que contendrá como ítem un objeto *subject* con un atributo *name*, que corresponde con la representación de la etiqueta <*bean:write*> que aparece en la figura.

6.3.2.2. Reglas para objetos y atributos de presentación de entrada/salida

Los objetos y atributos de entrada/salida, al contrario que en el caso de los de salida, sí son directamente identificables en el código de la página JSP. El objeto de presentación de entrada/salida en la tecnología escogida viene especificado por una etiqueta <*html:form*>, mientras que cada uno de los atributos de entrada/salida son especificados mediante las etiquetas relacionadas con los campos de control de un formulario HTML. La tabla 6.1 proporciona una revisión rápida del significado y funcionalidad de las etiquetas de Struts 1.X relacionadas con un formulario.

La Figura 6.12 ilustra la generación de un objeto de entrada/salida y sus atributos tanto simples como complejos. En el caso de <*html:form*>, puede ser necesario obtener el nombre del objeto de entrada de la información de mapeo petición-controlador de la base de datos de *ActionMappings* definida en el archivo de configuración. En el análisis de un formulario para identificar los atributos del objeto de presentación derivado, se pueden dar los siguientes casos:

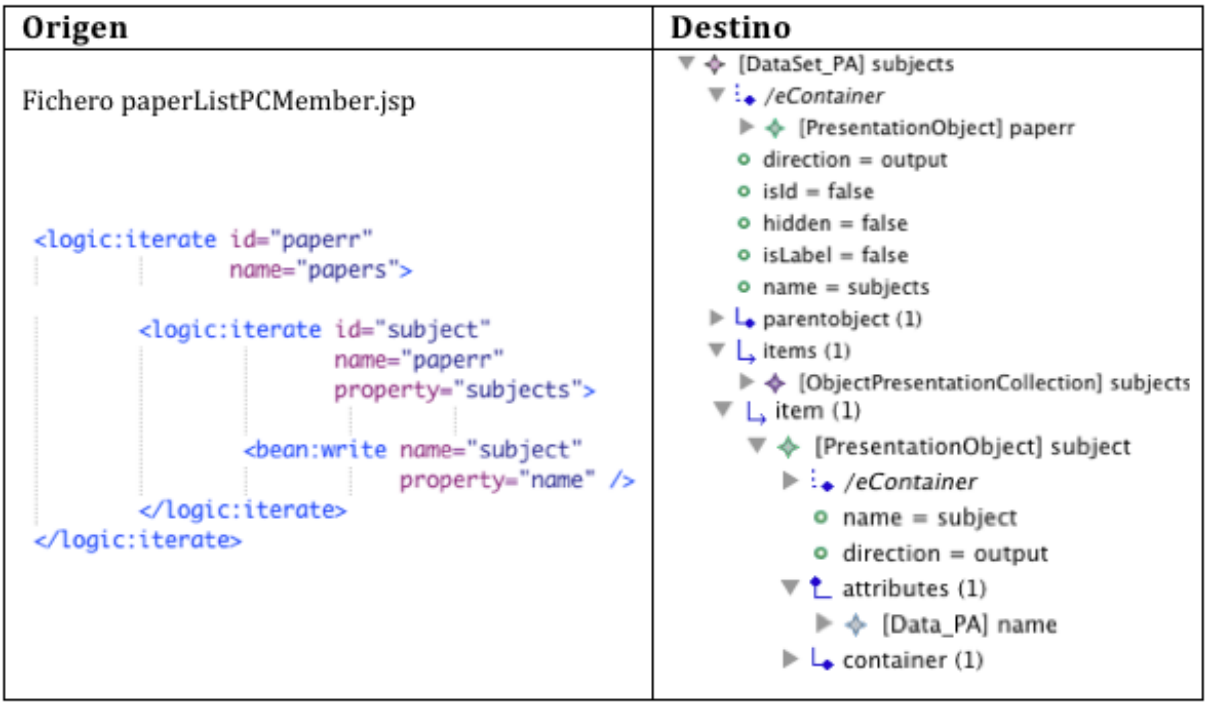


Figura 6.11: Generación de objeto de salida con atributo complejo

Etiqueta	Descripción
<html:text>	Permite especificar un campo de texto
<html:passwd>	Permite especificar un campo de texto protegido
<html:hidden>	Permite especificar un campo oculto
<html:textarea>	Permite especificar un campo área de texto
<html:radio>	Permite especificar un botón de tipo radio
<html:button>	Permite especificar un botón
<html:checkbox>	Permite especificar un control de tipo checkbox
<html:multibox>	Permite especificar un conjunto de controles checkbox
<html:select>	Permite especificar una lista de selección
<html:option>	Permite especificar una opción dentro de una lista de selección
<html:submit>	Pemite especificar un botón de envío de formulario

Tabla 6.1: Etiquetas de campos de control de Struts 1.X

1. Los atributos de salida relacionados con el objeto de entrada se identifican por etiquetas <*bean:write*> que no incluyen campo name y son transformados a entidades *Presentation Attributes* de tipo *DATA_PA* con direccionalidad de salida.

2. Los campos de control que permiten la entrada de un único valor y no proporcionan un rango de selección de valores son transformados a entidades *Presentation Attributes* de tipo *DATA_PA* con direccionalidad de entrada.

3. Los campos de control que permiten la entrada de uno o varios valores y proporcionan un rango de selección de valores son transformados a entidades *Presentation Attributes* de tipo *DATASet_PA* con direccionalidad de entrada, que están relacionadas con una Presentation Collection que se encarga de especificar el rango de valores permitido a partir de una colección de Presentation Objects. Este destino de generación se considera tanto para campos de tipo selección (*select*) como para conjuntos de campos *checkbox* (*multibox*).

El campo <*html:submit*>, sin embargo, supone un caso excepcional y su tratamiento será descrito posteriormente junto a las reglas de tratamiento de las peticiones.

Como ilustra la Figura 6.12, para generar el objeto de presentación de entrada *conferenceForm* es necesario localizar en el fichero de configuración de Struts el mapeo del action *ConferenceStatus* que aparece en el formulario. Este mapeo indica que el nombre del objeto que representa el formulario es *conferenceForm*, que será utilizado como el identificador del nuevo objeto de presentación de entrada. En lo referente a sus atributos, se generan dos atributos de tipo *Data*, *id* y *Submit*, a partir de los campos de valor único <*html:hidden*> y <*html:submit*>, mientras que se genera un atributo de tipo *DataSet* para el campo status que permite representar las opciones de estado disponibles mediante una colección de objetos de presentación.

6.3.2.3. Reglas para peticiones

Al igual que ha sucedido en la generación de los elementos anteriores, la generación de peticiones a partir de la Vista supone una generación parcial; ya que, por ejemplo, no se pueden categorizar convenientemente los tipos de peticiones con el simple análisis de la página JSP en la que está definida. Este tipo de información requiere de información contenida en el Controlador, que proporcione más información relacionada con el resultado de la petición, y de un procesamiento posterior, que descarte ambigüedades, redundancias e incoherencias.

Como ya se comentó anteriormente, existen dos grandes tipos de peticiones: las peticiones realizadas mediante enlaces y las realizadas mediante el envío de formularios. Estos son los casos que se consideran en esta fase de generación:

1. Una petición mediante enlace viene especificada mediante la etiqueta <*html.link*>. Se distinguen 3 casos distintos:

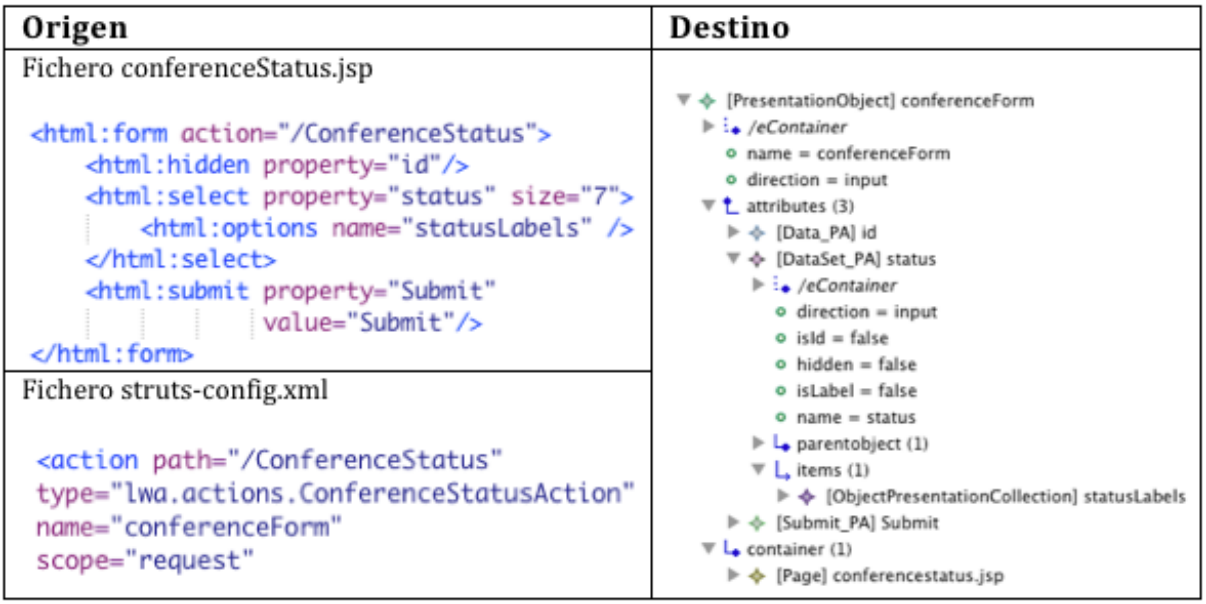


Figura 6.12: Generación de objeto de entrada/salida y sus atributos

a) No se indican parámetros de la petición. En principio, se categoriza como tipo de petición a nivel de página.

b) Se indican parámetros de la petición mediante los atributos *paramName*, *paramProperty* y *paramId*, que permiten indicar como parámetro de la petición la propiedad de un objeto de presentación. Por lo tanto, se categorizan como peticiones a nivel de objeto. En este caso, se generará una entidad Request que contendrá un parámetro del tipo *ObjectRequestParameter* cuyo nombre será *paramId* y que estará relacionado con el objeto de presentación *paramName* y su atributo *paramProperty*.

c) Se indican parámetros de la petición mediante la etiqueta <*html:param*>, que permite especificar pares nombre-valor.

2. Una petición mediante el envío de un formulario viene especificada mediante la etiqueta <*html:submit*>. En este caso, esa etiqueta define tanto una petición como un campo de datos de entrada del formulario contenedor. Por lo tanto, su procesamiento dará lugar tanto a una entidad *Request* como a un *Presentation Attribute* contenido en el *Presentation Object* definido por el formulario.

La figura 6.13 ilustra la generación de los elementos de destino para los casos expuestos. En el primer caso, se genera el objeto petición (*Request*) link.select que se corresponde con la etiqueta <*html:link*> de la página JSP de ejemplo. Como se puede apreciar, este ejemplo corresponde al caso 1.b, comentado anteriormente; por lo tanto, se indica que la petición generada es de tipo *object* y que contiene un parámetro de petición, *codeSubject*, cuyo valor proviene de un objeto (*ObjectRequestParameter*), en concreto del atributo *id* del objeto de presentación *subject*. En este ejemplo, el objeto de presentación

subject existe en el modelo destino antes de la generación de la petición relacionada; ya que, ese objeto ha sido previamente detectado en el análisis de la página. Sin embargo, puede ocurrir que no exista previamente dedido a que no existe una referencia previa o no existe referencia ninguna dentro de la página. En ambos casos, se debe generar previamente el objeto de presentación, resolviendo posibles redundancias posteriormente. Por su parte, en el segundo caso, petición mediante formulario, se genera la petición (*Request*) *submit.change* a partir del campo <*html:submit*> del formulario. En el caso de peticiones derivadas de formularios, siempre se considetan de tipo *object*. La petición generada tendrá como parámetro el atributo del objeto de presentación generado a partir de la misma etiqueta, en este caso el atributo *Submit*.

6.3.2.4. Reglas para colecciones

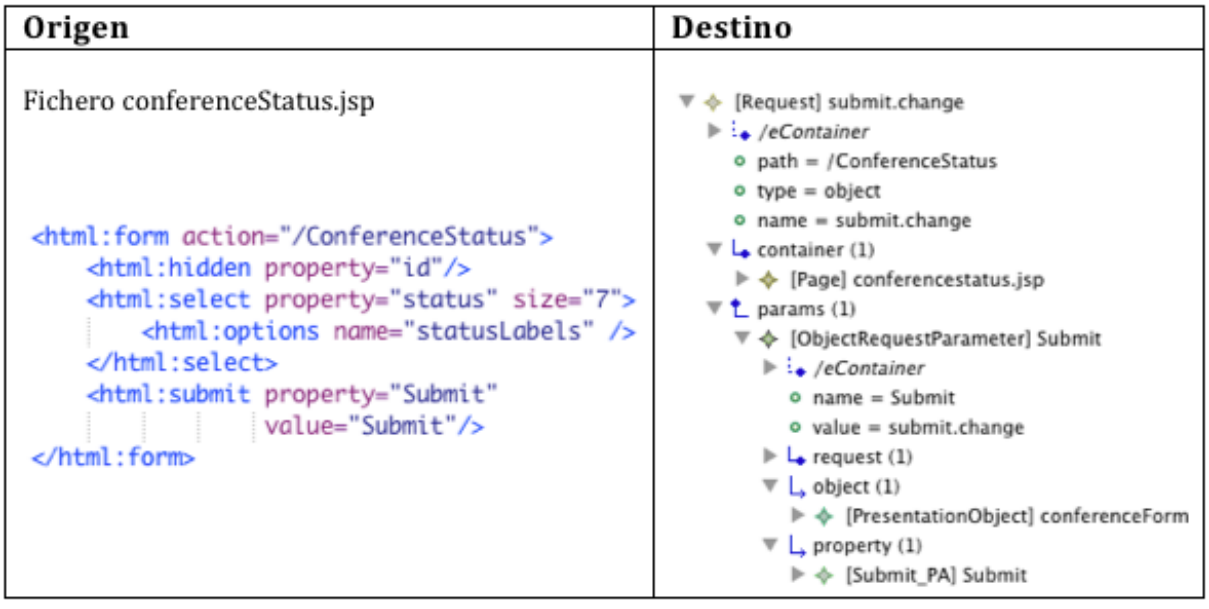
Las colecciones permiten especificar cualquier composición tanto de objetos como de peticiones, teniendo como regla fundamental que debe existir un objeto que juegue el rol de item de la colección. La generación de colecciones responde a dos patrones fundamentales:

1. Una colección de objetos de salida representada por una etiqueta <*logic:iterate*>. Este caso ya se ha ilustrado previamente en la regla de generación de un objeto de salida que contiene un atributo de tipo *DataSet*; sin embargo, se pueden dar otros casos que deben ser contemplados, como la generación de un listado de objetos de determinado tipo en una página. La figura 6.14 ilustra el resultado de aplicar la regla para la generación de la colección *subjects* de objetos de presentación de salida *subject*. Como se indica en el código de ejemplo de la página, para cada uno de estos objetos se muestra el valor de su atributo *name* y una petición asociada a cada uno. Por lo tanto, el objeto *subject*, item de la colección, debe estar compuesto de los atributos *name* e *id*. El primero se genera a partir de la etiqueta <*bean:write*> y el segundo a partir de la etiqueta <*html:link*>. Ambas etiquetas hacen referencia a propiedades de ese tipo de objetos y, por lo tanto, deben aparecer dentro del objeto de presentación *subject*. Además, la petición generada estará contenida dentro de la colección para indicar que es una petición relacionada con el objeto item de la colección. En el ejemplo, se genera la petición (*Request*) *link.select*.

2. Colecciones de objetos para poblar campos de selección en un formulario. En este caso, se representan mediante las etiquetas que permiten indicar opciones o alternativas dentro de los campos de control determinados. Este caso ya se ha tratado en un apartado anterior cuando se han analizado los posibles tipos de atributos de entrada correspondientes a un objeto de presentación de entrada/salida.

Muchos de los atributos que caracterizan una colección (filtro, orden de selección, etc.) se deben buscar en el código del controlador que gestiona la población de datos la página. Y, por lo tanto, no se detallan en este apartado.

(a) Petición de enlace



(b) Petición de formulario

Figura 6.13: Generación de peticiones

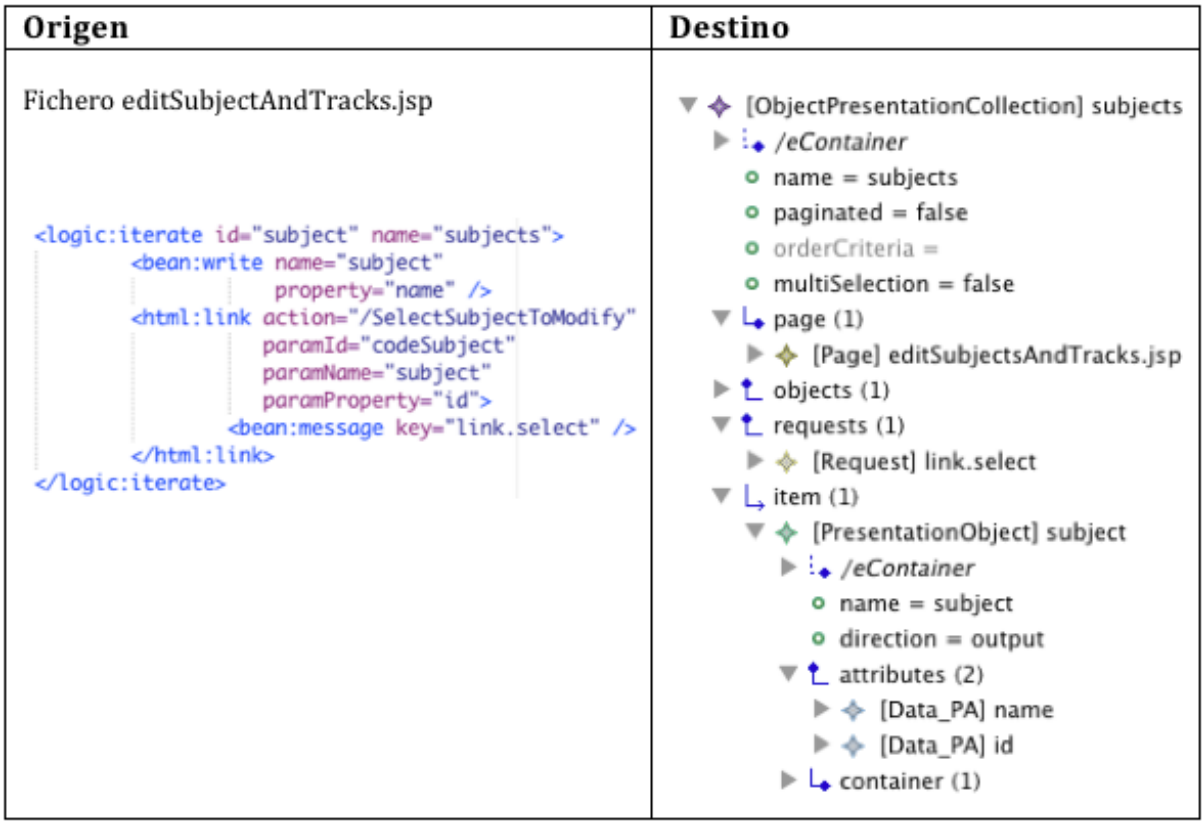


Figura 6.14: Generación de colección de salida

6.3.3. Reglas para el Controlador

El controlador es, sin duda alguna, el componente más complejo. Por un lado, al contrario de los otros dos, el controlador contiene la información relacionada con el comportamiento de la aplicación. Del análisis del controlador se puede extraer información estructural relevante, como por ejemplo, resolver la respuesta a las peticiones con lo que se obtiene el mapa de navegación del sistema heredado. Sin embargo, para poder extraer información relevante es necesario ser capaz de detectar patrones de comportamiento comunes, como pueden ser: mapeos entre los objetos del modelo y los objetos de presentación, invocación de operaciones, parametrización, generación de una respuesta, etc. Por otro lado, el lenguaje de especificación del controlador suele ser esencialmente más complejo y rico que el usado en el modelo o las vistas, complicando así su análisis. La riqueza del lenguaje conlleva la necesidad de definir más reglas para cada uno de los patrones que se quieren detectar. Considérese, por ejemplo, las opciones que hay para pasar un parámetro a una función que espera un dato de tipo entero en el lenguaje de programación Java: un literal, una variable, una posición de un array, el retorno de una llamada a función, etc. La naturaleza esencialmente dinámica del controlador y la riqueza de su lenguaje de especificación, por lo tanto, son dos factores que añaden un grado extra de complejidad y esfuerzo a la definición de las reglas de transformación para generar un modelo MIGRARIA MVC a partir de los modelos dependientes de la tecnología.

En el caso del componente controlador, se han definido reglas para:

- Generar flujos de control (*ControlFlow*).
  - Generar las relaciones con las peticiones que son atendidas por cada flujo de control.
  - Generar la respuesta generada por cada flujo de control.
- Generar las instancias de controlador.
  - Variables locales al flujo de control.
- Generar la secuencia de invocación de operaciones para atender la petición.
  - Generar el paso de parámetros en la invocación a operaciones.
  - Generar la gestión del retorno de la invocación de cada operación.
- Generar mapeos de datos
  - Entre objetos de presentación y otros objetos, normalmente, del modelo.
  - Entre parámetros de petición e instancias de controlador.

Adicionalmente, es interesante indicar que, en principio, podemos clasificar los controladores a analizar en dos grupos fundamentales, según el rol que jueguen en la atención a la petición:

1. Controladores que devuelven información. Estos controladores tienen como misión fundamental recuperar los datos necesarios del modelo para poblar la página de respuesta. Normalmente, su comportamiento es parametrizable en función de los parámetros contenidos en la petición, generando diferentes respuestas (datos) para diferentes parámetros. Están, habitualmente, relacionados con peticiones HTTP GET.
2. Controladores que realizan operaciones de modificación en el modelo. Estos controladores tienen como misión fundamental modificar los datos del modelo en función de los datos de entrada y las operaciones que invocan. Están relacionados con peticiones HTTP POST. Y, como respuesta, suelen generar una redirección a otro controlador que será del primer tipo (controladores que devuelven información), siguiendo el patrón Post-Redirect-Get[8].

Este apartado pretende realizar una exposición suficiente de las reglas más significativas utilizadas en el proceso de extracción de información para el componente Controlador, sin pretender realizar una exposición exhaustiva.

[8] http://www.theserverside.com/discussions/thread.tss?thread_id=20936

6.3.3.1. Controladores y flujos de control

Como se comentó en el capítulo anterior, un flujo de control representa la lógica relacionada con la atención de una petición determinada. Cada framework de desarrollo web proporciona un mecanismo propio para definir esta lógica. En el caso del framework Struts, la lógica se especifica en lenguaje JAVA y tiene como mecanismo base la jerarquía de clases *Action*[9]. De esta manera, para definir la lógica que atiende una petición en Struts, se debe implementar una clase JAVA derivada de alguna de las clases de la jerarquía *Action*.

En las primeras versiones de Struts, se recomendaba crear una clase *Action* para cada petición. Sin embargo, esta forma de proceder derivaba en muchos casos en la generación de un gran número de clases. Para solventar este problema se desarrolló una jerarquía de clases *Action* de tal manera que se definieron clases *Action* que son capaces, entre otras capacidades, de modularizar convenientemente la lógica para atender a diferentes peticiones, como por ejemplo, la clase *MappingDispatchAction* que permite ejecutar una lógica u otra según el valor del atributo *parameter* del *ActionMapping* relacionado. Alternativamente, fruto de la necesidad o del desconocimiento de esta extensión de las clases *Action*, un gran número de programadores han venido incluyendo la lógica relacionada con diferentes peticiones dentro del módulo *execute* de una sola clase *Action*, teniendo que codificar el control para relacionar la lógica que corresponde con una determinada petición. Este último escenario es muy común en el caso de un *Action* que muestra una página con un formulario para recoger datos y que, además, procesa los datos enviados por ese formulario, como recoge (Dudney & Lehr 2003). Este escenario es también típico en el uso de otros frameworks de aplicaciones web.

Estas circunstancias complican la generación de los elementos *ControlFlow* del modelo de Controlador; ya que, por una parte, hay que considerar un gran número de casos diferentes y, por otro lado, en algunos casos es necesario llegar a detectar a nivel de código estructuras de control que especifiquen mapeos petición-lógica ad-hoc. Con el fin de reducir la búsqueda e identificación de patrones, en el ámbito de este trabajo, se ha considerado un pequeño conjunto de casos que representan a las principales alternativas. Estos casos son:

- Una clase *Action* que atiende una única petición.
- Una clase *MappingDispatchAction* que atiende múltiples peticiones según el mecanismo de modularización definido por Struts.
- Una clase *Action* que atiende múltiples peticiones codificando el control último del mapeo petición-lógica.

El primer caso es suficiente para evaluar el proceso de generación correcta de elementos de flujo de control (*ControlFlow*). Por lo tanto, en el estado actual de la propuesta no se contemplan los dos últimos

[9] http://struts.apache.org/1.x/apidocs/org/apache/struts/actions/package-summary.html

casos señalados. El último caso contemplado conlleva una complejidad mayor; ya que, los flujos de control no están modularizados en funciones o clases, sino que se encuentran codificados dentro de un mismo módulo mediante el uso de estructuras de control condicionales. Por lo tanto, para generar todos los flujos de control especificados dentro de un mismo módulo es necesario, primero, identificar y procesar las estructuras de control que los definen. Aunque se han realizado algunas propuestas para poder dividir automáticamente una determinada clase *Action* del tercer caso en los flujos de control que la componen, estas propuestas presentan un estado demasiado inmaduro para incluirlas en esta memoria.

6.3.3.2. Reglas para generar flujos de control, invocación y respuesta

Un elemento *controlflow* se genera, por lo tanto, a partir de cada una de las clases del sistema heredado que implementen un controlador. En el caso de Struts, se han limitado las posibles opciones al escenario de una clase que herede directamente de la clase *org.apache.struts.action*. El nombre de este elemento será el mismo que el de la clase. Por lo tanto, para este primer paso sólo es necesario procesar las clases que se encuentren dentro del paquete de *lwa.Actions* del sistema heredado.

Como se ha comentado anteriormente, la invocación de un *Action* se puede deber a la atención de una petición o a la delegación del control por parte de otro *Action*. El primer caso, por un lado, supone detectar todas los elementos *Request* generados previamente en el paquete Vista y cuyo atributo *path* coincida con el del elemento *ActionMapping* del *Action* analizado. Por lo tanto, es necesario acceder también a la información del modelo de la configuración de Struts. El segundo, por otro lado, supone referenciar desde los *forwards* del elemento *controlflow* en generación a los elementos *ControlResponse* cuya referencia *controlflow* referencia al *Action* analizado. Esta referencia se establece como relación inversa de la referencia *controlflow* de un *ControlResponse* en el momento de generar elementos de este tipo.

Con respecto a la respuesta, se considera que un *controlflow* sólo tendrá una respuesta válida; por lo tanto, en el caso de que un *Action* defina más de un *forward* sólo se seleccionará el que defina el flujo normal de ejecución. Por simplicidad, se ha decidido identificar este *forward* por su nombre. Sólo se procesará el *forward* cuyo nombre sea *success*. La generación de elementos *response* se produce a partir de los elementos de tipo *forwardType* contenidos en el modelo de la configuración y, por lo tanto, se les asigna el mismo nombre. La distinción entre los tipos de elementos *response* también se realiza actualmente de manera sintáctica. Aquellos *forward* cuyo *path* termine en la cadena *".jsp"*, producirán elementos *PageResponse*. Mientras que los que terminen en *".do"*, producirán elementos *ControlResponse*. En el primer caso, es necesario resolver la referencia *page* del elemento que apuntará al elemento *Page* de la Vista cuyo *path* sea igual al del elemento *forwardType* origen. En el segundo caso, se debe resolver la referencia *control* que apuntará al elemento *controlflow* generado a partir de un *Action* cuyo *ActionMapping* tenga en su atributo *path* el mismo valor que el del elemento *forwardType* origen.

La figura 6.15 ilustra la generación del elemento *ControlFlow CreateSubjectsAction* de la clase del mismo nombre que extiende a la clase *Action*. Se trata de un *Action* que realiza operaciones de modificación en los datos del modelo. Su invocación sólo se produce a partir de la petición (*Request*) *submit.create* contenida dentro de la página *CreateSubjectsAndTracks.jsp*; ya que, el valor de su atributo *path* (*/SubmitCreateSubjects*) coincide con el del *ActionMapping* relacionado con el *Action* origen de la transformación. Y, como respuesta, delega el control al *Action CreateSubjectsAndTracks.do* mediante el *forward success*. Este patrón permite generar el elemento *ControlResponse success* que referencia al controlflow *CreateSubjectsAndTracks* desde su referencia *controlflow*.

6.3.3.3. Reglas para generar las instancias de controlador

Como se comentó en la especificación de la notación conceptual, los elementos *ControllerInstance* o instancias de controlador juegan un papel fundamental en la recuperación de la información relacionada con los mapeos presentación-modelo y con el paso de parámetros en la invocación de operaciones.

La generación de este elemento puede responder a múltiples y muy diferentes patrones; por lo tanto, por simplicidad, sólo se consideran como patrones a la declaración de variables locales al *Action* que da lugar al *ControlFlow* que los contendrá. Por lo tanto, toda declaración de variable local a un *Action* genera un elemento *ControllerInstance* contenido dentro del *ControlFlow* generado a partir de ese *Action*. A partir de este patrón general se han considerado siete especializaciones del mismo:

1. Declaración de variable de tipo primitivo de datos de Java.
2. Declaración de variable de tipo de datos de Java correspondientes a los tipos definidos en el enumerado *DataTypes* del Modelo.
3. Declaración de variable de tipo de datos del modelo de la aplicación heredada.
4. Declaración de array de variables de tipo primitivo de datos de Java.
5. Declaración de array de variables de tipo de datos de Java correspondientes a los tipos definidos en el enumerado *DataTypes* del Modelo.
6. Declaración de array de variables de tipo de datos del modelo de la aplicación heredada.
7. Declaración de colección Java de variables de tipo de datos del modelo de la aplicación heredada.

Con el fin de ilustrar la generación de este tipo de elementos, a continuación se procede a detallar los casos 3 y 7, que además de ser casos muy comunes son específicos de cada aplicación heredada a modernizar. Una instancia de controlador generada a partir de una variable local cuyo tipo es un objeto de datos presentará el valor *DataObject* en su atributo *type* y una referencia al *DataObject* concreto en el Modelo en su referencia *object*. Para detectar este patrón se comprueba que el tipo de datos de la variable es una declaración de clase (*ClassDeclaration* en el modelo Java) y que su identificador completamente

Figura 6.15: Generación de *ControlFlow*

cualificado se corresponde con uno de los *DataObject* del Modelo. Por otro lado, considerando el caso 7, se detecta que el tipo de la variable local es de tipo *ParameterizedType* que corresponde a un determinado tipo de colección Java. En este caso, el atributo *type* contiene el mismo valor que en el caso anterior, mientras que la referencia *object* se genera de la misma manera que antes pero tomando como base el tipo del elemento de la colección. Finalmente, al atributo *collection* se le asigna el tipo base de la colección detectada: *Hash*, *List*, *Set* o *Collection*.

La figura 6.16 ilustra la generación de los cuatro elementos *ControllerInstance* que se derivan a partir de las cuatro variables locales definidas dentro del *Action CreateSubjectsAction*. Estas cuatro variables permiten ilustrar los casos 1, 3 y 7 de la clasificación comentada anteriormente. En primer lugar, la variable de tipo primitivo *confId* produce la generación del elemento *ControllerInstance* homónimo y que tiene su mismo tipo (*Long*). Posteriormente, las variables *sub* y *conf* presentan tipos definidos como objetos de datos en el Modelo y, por lo tanto, generan elementos *ControllerInstance* cuyo tipo es *DataObject* y cuyas referencias *object* apuntan a *Subject* y *Conference*, respectivamente. Finalmente, la variable *subjects* consiste en una colección Java de tipo *Set* y genera, por tanto, un elemento *ControllerInstance* cuyo tipo es *DataObject*, el valor de su atributo *collection* es *Set* y su referencia *object* apunta al objeto de datos *Subject*.

6.3.3.4. Reglas para generar llamadas a operaciones

El procesamiento de una petición conlleva generalmente la invocación de una secuencia de operaciones por parte del flujo de control que atiende a esa petición. Desde el punto de vista del componente Controlador, la información más relevante a extraer consiste en la identificación de la secuencia de operaciones realizada y los objetos de presentación y de datos afectados.

Como se comentó en la definición de la notación conceptual, se han tenido en cuenta tres tipos diferentes de llamada a operación:

1. Llamada a operación genérica: *OperationCall*. Permite especificar cualquier tipo de llamada a operación, forme ésta parte del Modelo o no.

2. Llamada a operación de datos: *DataOperationCall*. Permite especificar llamadas a operaciones definidas en el Modelo: *DataOperation* y *DataOperationByRelation*.

3. Llamada a colección de datos: *CollectionOperationCall*. Permite especificar llamadas a operaciones comunes sobre colecciones de datos Java, como añadir o borrar elementos.

Evidentemente, cada uno de estos casos puede conllevar la detección de un elevado conjunto de patrones. En este trabajo hemos limitado el número de los mismos en base a las reglas de negocio definidas (véase Sección 7.2.4) y a la introducción de algunas restricciones para reducir la variabilidad. Con el fin de ilustrar la complejidad de la extracción de este tipo de patrones del modelo Java, se van a presentar lo casos fundamentales de generación de *DataOperationCall.*

Figura 6.16: Generación de *ControllerInstance*

Con respecto a una DataOperationCall que invoca una DataOperation, se deben localizar los objetos que se utilizan en el flujo de control y que implementan el patrón DAO. Una vez localizados estos objetos, se deben identificar las operaciones que se realizan sobre los mismos mediante la invocación de métodos definidos en su implementación de DAO. La información de qué objetos de datos existen en el sistema y qué implementaciones DAO existen para los mismos se obtuvo en la fase de generación del modelo del componente Modelo. Por lo tanto, en este caso, simplemente es necesario generar elementos *DataOperationCall* en el modelo de Controlador que contengan, por un lado, una referencia al elemento *DataOperation* concreto del modelo de datos y, por otro lado, una referencia al siguiente elemento *Operation* dentro de la secuencia. El nombre que se le asigna al elemento generado se construye a partir del nombre de la variable sobre la que se invoca el método y el nombre del método invocado.

Un componente fundamental de una *DataOperationCall* son sus parámetros. Para la generación de los *DataOperationCallParameter* se consulta el conjunto *arguments* del elemento *MethodInvocation* (modelo Java) asociado. Éste es un punto de gran variabilidad; ya que, un argumento en la invocación de un método puede adoptar formas muy diversas. Para ilustrar esta variabilidad, sirvan como ejemplo los diez tipos de argumentos presentes en el caso de estudio utilizado, que son: *InfixExpression*, *ArrayAccess*, *ClassInstanceCreation*, *NullLiteral*, *SingleVariableAccess*, *TypeLiteral*, *BooleanLiteral*, *MethodInvocation*, *FieldAccess*, *StringLiteral*. Por lo tanto, con el fin de simplificar la complejidad de las reglas sin afectar a la expresividad de la lógica del controlador, se ha decidido limitar los casos posibles a uno solo: todos los parámetros (*arguments*) de las llamadas a estas operaciones (*MethodInvocation*) deben ser variables locales (*SingleVariableAccess*). De esta manera, todos los elementos *arguments* que sean *SingleVariableAccess* darán lugar a nuevos elementos *DataOperationCallParameter*, cuyo nombre será el de la variable, su referencia *instance* apuntará al elemento *ControllerInstance* que represente esa variable y su referencia *parameter* apuntará al parámetro de la declaración de operación *DataOperation* que ocupe la misma posición en la lista de argumentos.

El otro componente fundamental de una DataOperationCall es su retorno. Para la generación de los OperationCallReturn, primero, se comprueba el contexto en el que se realiza la llamada para comprobar si se utiliza el valor de retorno de la llamada o no. Como ocurría en el caso de los parámetros, el contexto en el que está contenida la llamada es un punto de gran variabilidad. En el caso de estudio, se han detectado los siguientes once contextos: *arrayCreation*, *assignment*, *castExpression*, *classInstanceCreation*, *ifStatement*, *infixExpression*, *methodInvocation*, *parenthesizedExpression*, *returnStatement*, *variableDeclarationFragment* o *whileStatement*. En este trabajo, se han limitado los casos a cuatro: *assignment*, *variableDeclarationFragment*, *MethodInvocation* y *ClassInstanceCreation*. El elemento *assignment*, entre otros, contiene el atributo *leftHandSide* que se corresponde con la parte izquierda de la asignación. Este atributo es de tipo *Expression* del que existe una gran cantidad de subtipos. Algunas de las opciones más comunes son: *SingleVariableAccess*, *ArrayAccess* y *FieldAccess*. El primero indica que la asignación se realiza sobre una variable local. El segundo que se realiza sobre una posición de un array. Y el

tercero, que se realiza sobre un atributo de clase. Dar soporte a cada uno de estos tipos supone definir una regla distinta para la generación de la referencia instance del elemento *OperationCallReturn*, que deberá apuntar al *ControllerInstance* que represente el tipo concreto de elemento. Para obtener la referencia al elemento *return* del *DataOperation* relacionado, es necesario obtener éste del elemento *MethodDeclaration* invocado por el *MethodInvocation* (modelo Java).

Con respecto al caso de una *DataOperationCall* que invoca una *DataOperationByRelation*, se procesa de la misma manera y se distingue del caso anterior porque el elemento *MethodDeclaration* invocado por el *MethodInvocation* está relacionado con una *DataOperationByRelation* y no con una *DataOperation*.

La figura 6.17 ilustra la generación de las cinco llamadas a operaciones *OperationCall* que se derivan a partir de las cuatro invocaciones a método que aparecen dentro del *Action CreateSubjectsAction*. En concreto, por simplicidad, sólo se muestran los elementos *DataOperationCall* generados a partir de la primera sentencia de invocación a método y la tercera. Estos dos casos permiten ilustrar los dos casos fundamentales de uso de este elemento: la invocación a una *DataOperation* (*confDAO.get*) y la invocación a una *DataOperationByRelation* (*conf.getSubjects*). En el primer caso, se relaciona con la *DataOperation get* del *DataObject Conference*. Y, en el segundo caso, con la *DataOperationByRelation getSubjects* del *DataObject Conference*. Analizando el parámetro y el retorno de la primera, se ha generado un *DataOperationCallParameter* que relaciona a la *ControllerInstance confId* que se pasa como argumento en la llamada con el parámetro *id* de la *DataOperation* relacionada. Mientras que en el caso del retorno, se relaciona el retorno de la *DataOperation* relacionada con el *ControllerInstance conf* que almacena el resultado de la llamada. Este elemento *ControllerInstance conf* posteriormente se utiliza como parámetro implícito en la llamada a la *DataOperationByRelation getSubjects*, permitiendo ilustrar el paso de parámetros entre operaciones.

6.3.3.5. Reglas para generar mapeos entre datos

La correcta identificación de los mapeos entre datos dentro de un controlador es fundamental para poder entender: (1) qué relación existe entre los objetos de presentación y los objetos de datos; (2) cómo influyen los parámetros de petición en el comportamiento del controlador; y (3) qué misión cumple el controlador.

Del conjunto de los diferentes mapeos identificados durante la especificación de la notación, a continuación, la explicación se centra en los mapeos relacionados entre objetos de presentación y objetos de datos. Estos mapeos son, evidentemente, muy comunes en aplicaciones web intensivas de datos. Y, además, permiten ilustrar de manera significativa este tipo de reglas. En concreto, se van a detallar la generación de: los mapeos de entrada y salida entre un objeto de presentación de entrada y un objeto de datos; los mapeos de salida entre objetos de presentación de salida y objetos de datos; y los mapeos entre

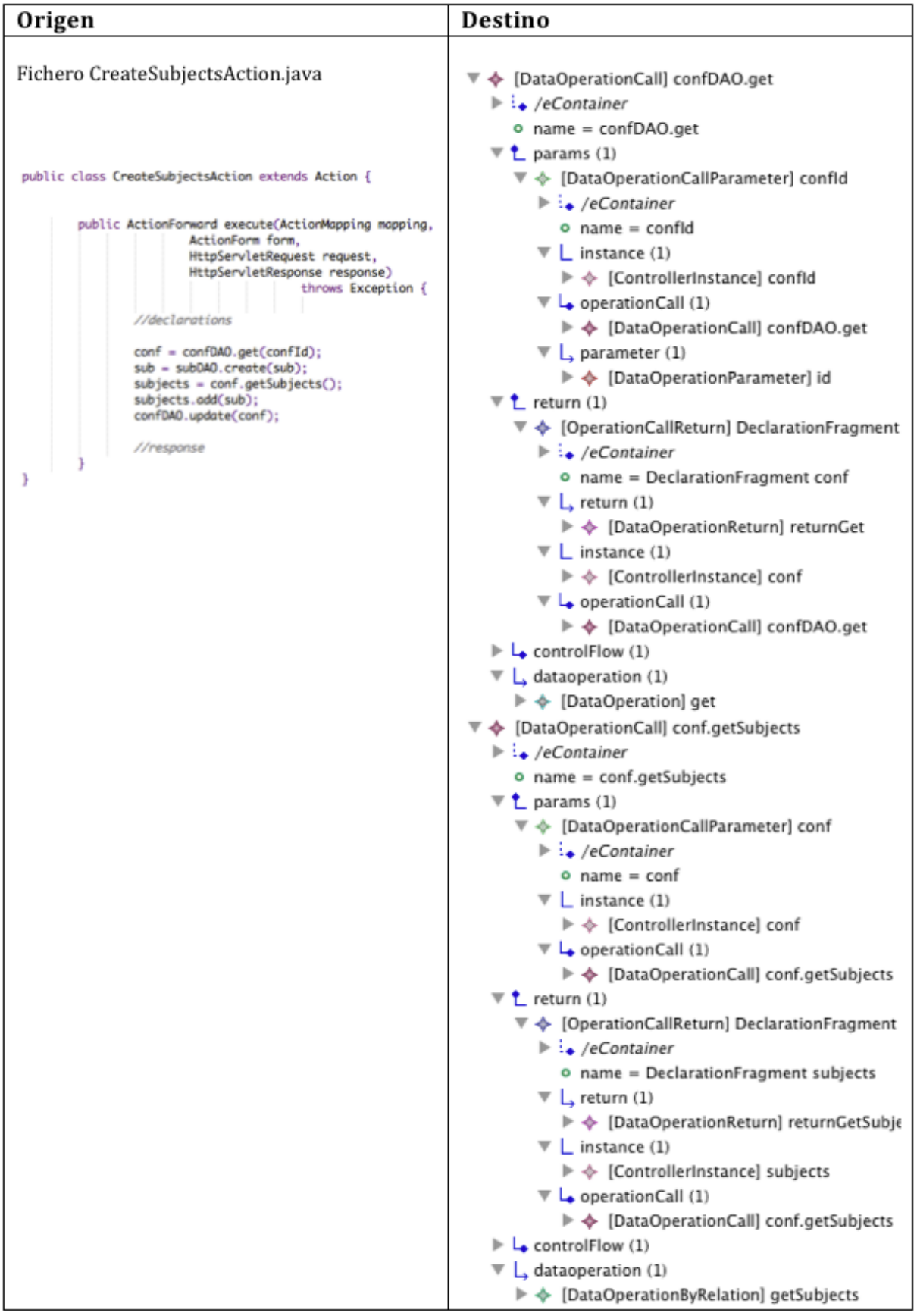


Figura 6.17: Generación de *DataOperationCall*

colecciones. Una vez más, se ha reducido el número posible de patrones a detectar al mínimo posible que sea suficientemente significativo para mantener un balance correcto entre esfuerzo y representatividad.

Con respecto a los mapeos de entrada entre un objeto de presentación de entrada y un objeto de datos, la detección no se realiza a nivel de objeto, sino de atributo. De esta manera, dentro un flujo de control, se localizan todas las invocaciones a métodos cuyo identificador comience por la cadena "get" realizadas sobre un objeto de la clase *ActionForm*, encargado de encapsular la entrada y salida de datos sobre un formulario en Struts. Cada una de estas invocaciones, genera un elemento *AttributeMapping* cuyo nombre se inicializa al nombre de la propiedad del formulario que ha sido accedida. Una vez detectado este mapeo, es necesario resolver las referencias a los atributos mapeados. Por un lado, se genera la referencia al atributo del objeto de presentación (*presentationAttribute*) a partir del atributo con el mismo nombre en el objeto de presentación de entrada sobre el que se realizó la petición. En el modelo MIGRARIA MVC, supone encontrar el parámetro *ObjectRequestParameter* de la petición que invoca al controlador que se está analizando. Este parámetro contiene la referencia *object* que apunta al objeto de presentación (formulario) sobre el que se ha realizado la petición. Por otro lado, es necesario también generar la referencia al atributo del objeto (*attribute*) destino de la entrada. En este punto, caben, al menos, dos opciones básicas: (1) el valor de la propiedad del formulario se almacena en una instancia de controlador no relacionada con un objeto de datos; (2) el valor de la propiedad se asigna a una instancia de controlador cuyo tipo es un objeto de datos del modelo mediante la invocación de un método que comienza con la cadena "set". En este último caso, la referencia *attribute* apuntará al atributo homónimo del objeto de datos que es del tipo de la instancia de controlador. Una vez generados los mapeos de atributo se genera un mapeo de objeto (*ObjectMapping*) para cada par (*PresentationObject*, *ControllerInstance*).

Con respecto a los mapeos de salida entre un objeto de presentación de entrada y un objeto de datos, la aproximación seguida coincide con la anterior presentando una variación en la detección del mapeo de atributo y de la localización de la referencia *presentationAttribute*. En un mapeo de salida, el objeto de presentación será el encargado de encapsular la información necesaria para generar la página de salida. Por lo tanto, dentro de un flujo de control, se localizan todas las invocaciones a métodos cuyo identificador comience por la cadena "set" realizadas sobre un objeto de la clase *ActionForm.* Cada una de estas invocaciones, genera un elemento *AttributeMapping* cuyo nombre se inicializa al nombre de la propiedad del formulario que ha sido accedida. En este caso, la referencia al atributo del objeto de presentación se debe realizar sobre el objeto de presentación de salida que estará contenido dentro de la página de respuesta. En el modelo MIGRARIA MVC, supone encontrar el objeto de presentación cuyo nombre coincida con el del tipo de datos de la instancia de controlador sobre la que se invocaron los métodos *set* dentro de la página apuntada desde *PageResponse*.

Finalmente, el proceso de generación de los mapeos de salida entre un objeto de presentación (o colección) y un objeto de datos presenta una complejidad apreciablemente menor; ya que, en este caso no se especifica el mapeo entre atributos. El mapeo entre atributos es implícito porque corresponde exac-

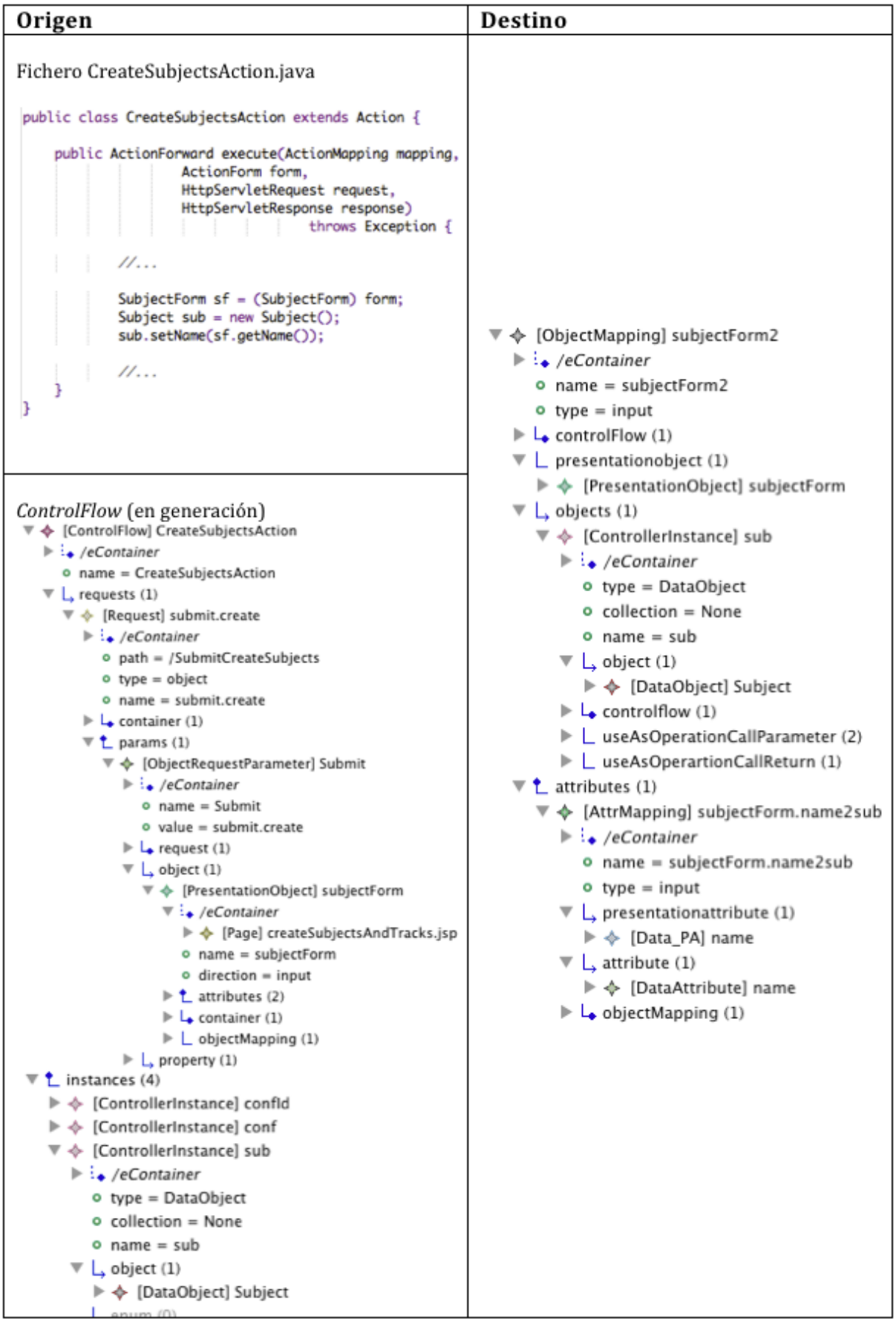


Figura 6.18: Generación de mapeo de entrada

tamente con los atributos del objeto de presentación; por lo tanto, no es necesario especificar este mapeo. La generación de estos mapeos se deriva de la presencia de invocaciones al método *setAttribute* sobre el objeto *Request* presente en el controlador. La declaración de este método presenta dos argumentos. El primer parámetro de la invocación, por un lado, contiene el nombre del objeto de presentación. A partir de este nombre, se puede localizar este objeto de presentación dentro de la página referenciada por el elemento *PageResponse* del flujo de control. El segundo parámetro, por otro lado, contiene la instancia de controlador que actúa como origen de los datos, cuyo tipo puede ser un objeto de datos o una colección. En el primer caso, se generará un elemento *ObjectMapping*. Mientras que, en el segundo, se generará un elemento *CollectionMapping*.

La Figura 6.18 ilustra la generación del elemento *ObjectMapping subjectForm2* que representa el mapeo de entrada entre el objeto de presentación de entrada *subjectForm* y el objeto de datos *Subject*. Esta generación se deriva de la generación del mapeo de atributos *(AttrMapping)* entre los atributos *name* de ambos objetos a partir de la última sentencia de código que aparece en la Figura. Para localizar, por un lado, el elemento *PresentationAttribute* referenciado desde el mapeo se ha tenido que consultar la petición (*Request*) *submit.create* responsable de la invocación de este flujo de control. Dentro de esa petición se puede localizar el parámetro *Submit* de tipo *ObjectRequestParameter* cuya referencia *object* apunta a *subjectForm* de la página *createSubjectsAndTracks.jsp*. Este objeto contiene un atributo (*Data_PA*) cuyo nombre es *name*. Por otro lado, para localizar el elemento *attribute* se ha tenido que consultar la instancia de controlador (*ControllerInstance*) *sub* sobre la que se realiza la invocación del método *set*. El tipo de datos de esta instancia es el objeto de datos *(DataObject) Subject*, que contiene un atributo (*DataAttribute*) cuyo nombre es *name*.

6.4. Resumen

Este capítulo presenta con detalle el proceso definido para la actividad de extracción de la información de la etapa de ingeniería inversa dentro del proceso de modernización definido en el proyecto MIGRARIA. Este proceso se ha diseñado para sacar el máximo partido de los métodos y herramientas de la IDM. En este sentido, define, por lo tanto, un proceso de ingeniería inversa dirigido por modelos. Este proceso consta, a su vez, de dos etapas fundamentales: la generación de los modelos dependientes de la tecnología y la generación del modelo MIGRARIA MVC. La primera etapa consiste en la generación de modelos a partir de los artefactos software de la aplicación heredada. Se ha implementado en base a la definición de transformaciones texto a modelo mediante el uso del proyecto MoDisco. Este proyecto ha sido convenientemente extendido para dar soporte a las características particulares de los artefactos software relacionados con el framework de aplicaciones web usado como referencia base, Struts 1. La segunda etapa, por su parte, consiste en la definición de análisis estáticos sobre los modelos generados a partir de los artefactos software. Estos análisis tienen como objetivo generar una representación de la

aplicación web heredada conforme a MIGRARIA MVC y se definen, por tanto, como transformaciones modelo a modelo mediante el lenguaje de transformación ATL.

El capítulo ilustra de manera conceptual las reglas de transformación definidas para la extracción de los conceptos a representar en el modelo MIGRARIA MVC. Su exposición está organizada por los tres paquetes de MIGRARIA MVC: Modelo, Vista y Controlador. Dada la gran expresividad del modelo JAVA en comparación con el resto de modelos de los artefactos software, ha sido necesario reducir los casos posibles a contemplar (patrones de búsqueda) para poder producir un conjunto suficientemente representativo de reglas relacionadas con el paquete Controlador dentro de unos márgenes adecuados de costes y tiempo. Esta reducción sólo pretende limitar el número de opciones disponibles de implementar un mismo concepto, sin reducir el número de conceptos.

Capítulo 7

Caso de estudio

7.1. Introducción

Con el objetivo de validar la propuesta presentada en este trabajo, se expone en este capítulo el desarrollo de un caso de estudio que permite ilustrar todos los detalles del proceso de modernización definido en el marco del proyecto MIGRARIA.

Los objetivos fundamentales de este caso de estudio son los siguientes:

1. Validar la notación conceptual propuesta, MIGRARIA MVC, para el tipo de aplicaciones heredadas que forman parte del dominio del proceso de modernización definido. En concreto, interesa validar que la expresividad de la notación definida es suficiente para representar todos los escenarios planteados en el caso de estudio.

2. Evaluar el proceso de generación de una representación conceptual a partir de la aplicación web heredada. En concreto, interesa evaluar, por un lado, la complejidad del proceso y, por otro lado, la bondad del resultado.

3. Ilustrar el alcance actual de la propuesta y guiar su evolución. En este sentido, se repasarán las limitaciones actuales de la propuesta que definen su ámbito de aplicación. Además, se proporcionará una ordenación por prioridad de esas limitaciones que permita guiar la evolución de la misma.

El capítulo comienza realizando una presentación suficiente del caso de estudio, su origen y su motivación. Posteriormente, se especifican los detalles relacionados con su implementación: pila tecnológica, reglas de negocio, etc. Una vez descrito con suficiente detalle el caso de estudio, se realizará una validación de la completitud de la notación. Por un lado, se revisará el grado de cobertura del caso de estudio. Y, por otro lado, se presentará una serie de ejemplos significativos que permitan completar los casos de ejemplo utilizados en esta memoria. Tras validar la notación, se procederá a evaluar el proceso de ingeniería inversa definido analizando, de nuevo, el grado de cobertura del caso de estudio por parte de los

modelos MIGRARIA MVC generados, así como su corrección. Finalmente, se realizará una exposición razonada de las limitaciones actuales de la propuesta.

7.2. Sistema de Revisión para Conferencias

El Sistema de Revisión para Conferencias, o Conference Review System (CRS)[1], constituye el caso de estudio principal de este trabajo. Este sistema tiene como objetivo principal servir, por un lado, de guía de definición y evolución del proceso de modernización y, por otro lado, permitir ilustrar el alcance de la propuesta en todo momento en base a un ejemplo vivo y disponible públicamente.

7.2.1. Motivación

A lo largo del desarrollo de este trabajo se han utilizado aplicaciones desarrolladas por terceros como casos de estudio para la definición de la propuesta. Sin embargo, este escenario, a priori ideal, presenta varias limitaciones fundamentales:

1. Dificultad de obtener el código completo de aplicaciones web heredadas dentro de la pila tecnológica contemplada en este trabajo.

2. Falta de un conjunto exhaustivo de escenarios de uso de la pila tecnológica contemplada en este trabajo.

3. Mezcla de tecnologías en la definición de la aplicación web heredada.

4. Limitaciones en las licencias de las aplicaciones que impiden poner a libre disposición el código para que la propuesta pueda ser validada por terceros.

Atendiendo a estas limitaciones, la mayoría de los casos de estudio contemplados se pueden organizar en dos grandes grupos:

1. Aplicaciones web heredadas de tamaño medio-pequeño que han sido desarrolladas por un número pequeño de desarrolladores y que suelen aplicar el mismo conjunto de patrones de codificación en toda la aplicación, limitando su utilidad en la validación.

2. Aplicaciones web heredadas de tamaño medio-grande que han seguido metodologías de desarrollo concretas pero que presentan un uso heterogéneo de tecnologías, que incrementan enormemente el proceso de extracción de información.

En cualquiera de estos dos grandes grupos, la limitación establecida por las licencias supone un inconveniente demasiado importante como para poder usarlos como un caso de estudio para un trabajo de

[1] http://www.eweb.unex.es/eweb/migraria/?Case_Studies:Conference_Review_System

investigación. Una posible solución a este problema podría consistir en utilizar aplicaciones cuyas licencias no prohiban la modificación ni la distribución. Sin embargo, dentro del dominio de aplicación de esta propuesta y dentro de la pila de tecnología seleccionada, no ha sido posible localizar ningún proyecto que se adecuase. Es necesario tener en cuenta que este proyecto se centra en aplicaciones heredadas que, en muchos casos, se caracterizan por el uso de tecnologías al final de su ciclo de vida (normalmente, obsoletas y abandonadas), como por ejemplo, el framework de aplicaciones web Struts 1[2]. Este hecho complica la existencia de proyectos libres en esta tecnología; ya que, los proyectos que se mantienen son aquellos que evolucionan constantemente manteniéndose atractivos para su comunidad.

Con el objetivo de disponer de un caso de estudio suficientemente pequeño para poder presentarlo en este trabajo y suficientemente grande para incluir un conjunto exhaustivo de escenarios de uso de la tecnología seleccionada, en este capítulo se presenta un caso de estudio basado en una aplicación web específicamente desarrollada para la definición y posterior validación de esta propuesta. Esta aplicación web heredada está basada en el caso de estudio *Conference Review System*[3] propuesto en el *First International Workshop on Web-Oriented Software Technology*[4] como caso de referencia para la evaluación de las diferentes propuesta IWDM presentadas. Por lo tanto, además de disponer de la descripción de requisitos del sistema, se dispone de las diferentes especificaciones de la aplicación web dada por cada una de las propuestas de IWDM presentadas en ese taller: UWE (Koch *et al.* 2001), WebML (Ceri *et al.* 2001), OOHDM (Schwabe & Rossi 2001), OO-H (Cachero *et al.* 2001), WSDM (De Troyer & Casteleyn 2001) y ADM-2 (Atzeni & Parente 2001), entre otras. Además, el desarrollo de un sistema propio conlleva como ventaja añadida la posibilidad de dotarle de una licencia que permite su libre uso, modificación y distribución; por lo tanto, se facilita la posibilidad de validación de la propuesta por parte de terceros.

7.2.2. Descripción base[5]

La misión fundamental del sistema consiste en gestionar el envío, la evaluación y la selección de artículos para una conferencia.

7.2.2.1. Actores

Los actores que participan en el sistema son los siguientes:

1. PC Chair (director del comité de programa) - es el responsable de crear la conferencia, determinar las líneas (tracks) y los temas (subjects) de la misma, establecer el comité de programa y, asesorado por los miembros del comité de programa, definir la lista final de artículos aceptados y rechazados. Se entiende que la conferencia debe tener un conjunto de líneas y opcionalmente un conjunto de

[2] http://struts.apache.org/struts1eol-press.html
[3] *http://users.dsic.upv.es/~west/iwwost01/IWWOSTContent.htm#case_study*
[4] *http://users.dsic.upv.es/~west/iwwost01/*
[5] Esta descripción es una traducción de la descripción original

temas. El PC Chair también es el responsable de definir los *deadlines* de la conferencia: envío, revisión y notificación.

2. PC Member (miembro del comité de programa) - es responsable de la evaluación de un conjunto de artículos que le han sido asignados. Un PC member puede delegar la revisión de un artículo en otra persona. Además, debe asesorar al PC Chair en la elaboración de la lista final de artículos aceptados y rechazados.

3. Reviewer (revisor) - es responsable de revisar un artículo.

4. Author (autor) - es responsable de enviar un artículo a la conferencia. Dado que los miembros del comité de programa y los revisores pueden ser también autores, deben tener diferentes identificadores para cada rol.

A excepción del director del comité de programa (PC Chair), todos los actores deben registrarse en el sistema para poder participar en los procesos.

Los miembros del comité de programa son pre-registrados por el director, pero deben confirmar su registro personalmente, que obligará a cambiar la contraseña y a proporcionar la información de afiliación y contacto. Los miembros del comité de programa pueden también indicar sus preferencias de líneas y temas de la conferencia, cuando se hayan definido.

Los revisores pueden ser pre-registrados por el director o por cualquier miembro del comité de programa. Al igual que el miembro del comité de programa, un revisor debe confirmar personalmente su registro. Los revisores no pueden indicar sus preferencias en cuanto a líneas y temas.

Los autores deben registrarse antes de enviar un artículo. Un autor puede pre-registrar a un coautor, en cuyo caso el coautor debe registrarse posteriormente también para poder tener acceso al sistema.

7.2.2.2. Funciones

El sistema debe soportar la siguiente funcionalidad.

Envío de artículos. Cualquier autor registrado puede enviar un artículo. El autor debe indicar la siguiente información: autores (todos deben registrarse), título, resumen, línea (track) de la conferencia, y un conjunto de temas (subjects) elegidos de la lista previamente definida por el director del comité de programa. El sistema, después de comprobar el registro de los autores, asigna un ID al nuevo artículo, y permite el envío del fichero que lo contiene. En cualquier momento, un autor puede comprobar los datos sobre los artículos que ha enviado. Hasta el cierre del plazo de envío, el autor puede sustituir el fichero enviado por uno nuevo o cambiar cualquier información sobre el artículo.

Asignación de artículos a miembros del comité. Una vez enviado un trabajo, el director del comité de programa puede consultar la lista de artículos y señalar potenciales conflictos de interés entre los

miembros del comité de programa y los artículos enviados. En particular, cualquier artículo cuyo autor es un miembro del comité de programa constituye automáticamente un conflicto de interés.

Una vez concluido el plazo de envío de trabajos, el director del comité de programa abre la lista de artículos enviados para que los miembros del comité de programa puedan realizar una lectura detallada de los mismos, quienes pueden indicar su interés ([0 - “sin interés”, 1 - “moderado”, 2 - “grande”]) y también sus conflictos de interés con los artículos. Una vez que se ha establecido un conflicto de interés entre un miembro del comité de programa y un artículo, el miembro del comité de programa no verá ninguna información sobre el mismo hasta la decisión final de aceptación o rechazo.

El director del comité de programa, basado en la información de preferencias de artículos, líneas y temas de cada miembro del comité de programa, asignará los artículos a los miembros del comité de programa para su revisión. Una vez se haya realizado la asignación, cada miembro del comité de programa recibirá un correo electrónico con la lista de artículos y una URL a la página donde puede acceder a los mismos.

Normalmente, el director del comité de programa asignará el mismo artículo a tres o más revisores.

El director del comité de programa puede también cambiar la línea (track) definida por un autor para un artículo concreto.

Asignación de artículos a revisores. Un miembro del comité de programa puede re-asignar un artículo a un revisor. En este caso, el revisor será el responsable de realizar la evaluación del artículo, pero el miembro del comité de programa será el responsable final de la evaluación del artículo durante el proceso de selección. Cada vez que un revisor realice una revisión, el miembro del comité de programa asociado debe ser notificado de este evento por el sistema.

Realizar una revisión. Un miembro del comité de programa, o un revisor, puede realizar una revisión a un artículo asignado a él. Cada revisión se compone de un conjunto de criterios de evaluación, cada uno de los cuales puede ser calificado en el rango [1-5]. Uno de esos criterios es una recomendación final, también en el rango [1-5], en el que 1 significa *Rechazar Fuerte*; 2 significa *Rechazar*; 3 significa *Indiferencia*; 4 significa *Aceptar*; y 5 significa *Aceptar Fuerte*. Además, debe contener dos campos de texto: uno para introducir los comentarios confidenciales al comité de programa, y otro para introducir los comentarios a los autores.

La revisión se realiza mediante un formulario web que contiene todos los criterios de evaluación. Adicionalmente, el miembro del comité de programa o el revisor pueden indicar si la revisión es definitiva (final) o no. Si la revisión no es definitiva, se puede modificar cualquier parte de la misma hasta el plazo máximo de revisión definido por el director del comité de programa.

Un miembro del comité de programa puede ver el resto de revisiones sobre los artículos que le han sido asignados, una vez que haya realizado su propia revisión. En cualquier momento puede conocer

cuántas revisiones se han realizado ya para cada artículo sobre el que tenga derechos de acceso, así como el estado de las mismas (definitivo o no).

El director del comité de programa tiene acceso total a todos los artículos y todas las revisiones.

Seleccionar artículos aceptados y rechazados. Una vez que el plazo de revisión ha terminado, el proceso de revisión se cierra. Todos los miembros del comité de programa tienen permiso para ver todos los artículos y sus revisiones correspondientes. El sistema es capaz de mostrar la lista de artículos ordenados por cualquiera de los criterios de evaluación (incluyendo la recomendación final), el autor, el título, la afiliación, el tema, etc.

Después de la deliberación, el director del comité de programa, considerando las recomendaciones de los miembros del comité de programa y los revisores, selecciona los artículos que serán aceptados y rechazados, marcándolos convenientemente mediante una interfaz proporcionado por el sistema. Durante el proceso de selección, el director y los miembros del comité de programa pueden visualizar el estado del proceso en cualquiera de sus fases mediante la revisión de la lista de los artículos marcados para aceptación, rechazo o todavía sin decisión. Se muestran también varias estadísticas, tales como los totales para cada categoría, los ratios de aceptación, número de artículos por línea, por estado, etc.

Una vez que el proceso de selección se marca como terminado por parte del director del comité de programa, el sistema envía un mensaje de notificación a los autores de los artículos, que incluye las partes apropiadas de las revisiones realizadas por los miembros del comité de programa y los revisores. Adicionalmente, debe ser posible consultar la lista de artículos aceptados y rechazados.

7.2.3. Desarrollo

7.2.3.1. Pila tecnológica

El trabajo presentado dentro de esta memoria se encuadra dentro de la plataforma tecnológica Java Enterprise Edition; por lo tanto, las tecnologías relacionadas serán la base de la pila tecnológica de este caso de estudio. Como primera consecuencia, el lenguaje de programación Java y el lenguaje de marcado XML constituirán la base para el resto de tecnologías.

Como se ha comentado en esta memoria, el elemento fundamental de la pila tecnológica de las aplicaciones web heredadas dentro del dominio de la propuesta es el framework de aplicaciones web Struts 1 [6]. Este framework, basado en el patrón MVC, ha sido el estándar de-facto para el desarrollo de aplicaciones web en Java. Se creó en el 2000 para mejorar la experiencia de desarrollo ofrecida por el uso directo de las Java Server Pages (JSP). Un gran número de empresas adoptaron Struts 1 como una plataforma estratégica y la mantuvieron incluso después de que apareciese el estándar Java Server Faces (JSF) para el desarrollo de aplicaciones web en Java EE. A principios de los 2000 el conocimiento de este framework era un requisito indispensable en la mayoría de las ofertas de trabajo relacionadas con

[6] http://struts.apache.org/release/1.3.x/index.html

el desarrollo web en Java. Incluso actualmente muchas aplicaciones web están desarrolladas con Struts 1. Muchos de los framework de aplicaciones web posteriores, como Spring MVC o WebWork, están basados en Struts 1. Este framework se encuentra actualmente en el fin de su ciclo de vida [7] y, por lo tanto, sin soporte. La última versión de este framework es la 1.3.10 que data de diciembre de 2008. Todas estas características lo convierten en el framework de aplicaciones web ideal para usarlo dentro de un proceso de modernización. La tecnología base de este framework que se ha considerado es la siguiente:

- Páginas generadas mediante la tecnología Java Server Pages con etiquetas propias del framework (Struts tag libs).
- Archivo de configuración del controlador frontal especificado mediante XML conforme al esquema definido para este tipo de archivos y la versión determinada del framework.
- Código JAVA para la implementación del código de control de la aplicación dentro de las clases y conceptos definidos por el framework.

Con respecto a la tecnología de acceso a los datos, Struts 1 es un framework enfocado en la especificación de capas de presentación web de sistemas de información y, como tal, no proporciona un soporte propio sino que recomienda el uso de un framework de persistencia de objetos externo. Éste es, por tanto, el escenario que hemos elegido para este trabajo, seleccionando como solución el framework Hibernate[8] en su versión 3.6, cuyo uso está muy extendido a nivel profesional. La tecnología base de este framework que se ha considerado es la siguiente:

- Las clases de los objetos de datos y las de acceso a objetos de datos (patrón DAO) se implementan en Java.
- Los mapeos objeto-relacional se definen en XML.

7.2.3.2. Metodología y planificación

El desarrollo de este sistema se ha realizado siguiendo una metodología iterativa, añadiendo nueva funcionalidad en cada una de las iteraciones. Se han planificado cuatro iteraciones para su desarrollo:

1. Iteración 1: puesta en marcha y sistema base. Principales tareas:
   a) Creación del proyecto base.
   b) Creación del entorno de desarrollo (y producción): servidor de aplicaciones JEE y servidor de base de datos.
   c) Diseño del esquema de base de datos.

[7] http://struts.apache.org/struts1eol-press.html
[8] http://www.hibernate.org/

d) Añadir funcionalidad necesaria para la gestión básica de los datos relacionadas con una conferencia.

2. Iteración 2: perfiles y funcionalidad de los miembros del comité de programa. Principales tareas:

   a) Gestión de la creación de los miembros de diferente perfil de la conferencia: miembros del comité de programa, revisores, autores y coautores.

   b) Funcionalidad relacionada con los flujos de trabajo del director del comité de programa (PC-Chair).

   c) Funcionalidad relacionada con los flujos de trabajo de los miembros del comité de programa.

   d) Añadir soporte de autenticación y autorización.

3. Iteración 3: autores y revisores. Principales tareas:

   a) Funcionalidad relacionada con los flujos de trabajo de los autores.

   b) Funcionalidad relacionada con los flujos de trabajo de los revisores.

4. Iteración 4: gestión de fases y notificaciones. Principales tareas:

   a) Añadir restricciones de temporalidad a la diferente funcionalidad en base a los plazos marcados por una conferencia para cada una de sus fases principales.

   b) Añadir soporte de notificaciones, necesarias en varios de los flujos de trabajo definidos.

Actualmente, el sistema se encuentra en la tercera iteración, presentando un estado suficientemente maduro para poder representar el papel de aplicación heredada dentro del proceso de modernización definido.

7.2.4. Reglas de negocio o convenios

En el desarrollo de este sistema, se han seguido reglas de negocio (convenios) bien extendidas en el desarrollo de aplicaciones web con Struts 1. Para ello, se ha consultado diferente bibliografía sobre el tema, como (Alur *et al.* 2001; Dudney & Lehr 2003). Y se ha analizado un conjunto de aplicaciones web heredadas desarrolladas mediante Struts 1, algunas disponibles públicamente[9] y otras, fruto de la colaboración con empresas y cuyo código no está disponible públicamente.

A continuación, se repasan las reglas de negocio fundamentales utilizadas en el desarrollo de este sistema.

[9]Lead App: http://www.dzone.com/links/crud_application_struts_1_hibernate_and_mysql.html

7.2.4.1. Modelo

Las reglas de negocio seguidas para la especificación del componente Modelo son las siguientes:

1. En este trabajo, hemos seguido el siguiente esquema de uso de hibernate:

   - Definir una clase como mapeo de cada tabla de la base de datos, por ejemplo, la clase *Conference* en el caso de estudio. Todas estas clases estarán dentro del paquete <*lwa*>.*model*.
   - Definir el mapeo clase-tabla en un archivo XML, que permite indicar el mapeo de atributos y el mapeo de relaciones, por ejemplo, el mapeo *Conference.hbm.xml* en el caso de estudio. Todos estos archivos estarán dentro de una carpeta *mappings*.
   - Definir una interfaz Service y una clase que lo implemente, por ejemplo, la interfaz *ConferenceDAO* y la clase *HibernateConferenceDAO* en el caso de estudio. Un service en hibernate permite definir las operaciones que se realizan sobre los objetos de datos. El concepto Service es una implementación del patrón Data Access Object. Todas estas clases estarán dentro del paquete <*lwa*>.*model.dao*.

2. Las clases DAO contienen, al menos, el siguiente conjunto de operaciones fundamentales:

   - *get(identificador)*.- Devuelve el objeto de datos con el identificador pasado por parámetro.
   - *getAll()*.- Devuelve todos los objetos de datos de un tipo determinado.
   - *getAll(objeto_datos_relacionado)*.- Devuelve todos los objetos de datos de un tipo determinado que estén relacionados con el objeto pasado por parámetro.
   - *create(objeto_datos)*.- Crea un nuevo objeto de datos para el tipo determinado a partir del objeto pasado por parámetro.
   - *remove(objeto_datos)*.- Elimina un objeto de datos para el tipo determinado a partir del objeto pasado por parámetro.
   - *update(objeto_datos)*.- Actualiza un objeto de datos para el tipo determinado a partir del objeto pasado por parámetro.

7.2.4.2. Vista

Las reglas de negocio seguidas para la especificación del componente Vista son las siguientes:

1. Uso de las librerías de etiquetas definidas en Struts 1.
2. No generar valores derivados en las JSP mediante el uso de *scriplets*.
3. No incluir código relacionado con la lógica de negocio en las JSP.
4. Las JSP no deben contener etiquetas de control sólo de presentación.

5. No se usa un mecanismo de gestión de plantillas.

6. Se enlazan acciones y no páginas. Buena práctica en Struts.

7. Usar ámbito de petición para formularios de una página que contienen todas las propiedades demandadas por el *Action*. Normalmente, no hay necesidad de mantener las entradas de formularios entre peticiones.

7.2.4.3. Controlador

Las reglas de negocio seguidas para la especificación del componente Controlador son las siguientes:

1. Un controlador sólo define un flujo de control, es decir, atiende sólo a una determinada petición. Este flujo termina siempre con un *forward* con identificador *success*. En su versión relajada, se permite definir más de un forward de respuesta, pero debe aparecer uno claramente identificado como el principal.

2. Un controlador puede definir un flujo de error, identificado por un *forward* con identificador *error*.

3. Un controlador puede seguir dos esquemas fundamentales de comportamiento, según el rol que juegue en la atención a la petición:

   a) Devolver información. Estos controladores tienen como misión fundamental recuperar los datos necesarios del modelo para poblar la página de respuesta. Por lo tanto, la respuesta de estos controladores debe ser una página, es decir, su *forward* debe indicar el nombre de una JSP. Y los objetos para poblar la página de respuesta son pasados al sistema de generación de la página de respuesta como atributos de la petición.

   b) Operar sobre el Modelo. Estos controladores tienen como misión fundamental modificar los datos del modelo en función de los datos de entrada y las operaciones que invocan. Están relacionados con peticiones HTTP POST. Y, como respuesta, suelen generar una redirección a otro controlador que será del primer tipo (recuperación de información), siguiendo el patrón Post-Redirect-Get[10].

4. La entrada de datos se puede realizar mediante un *ActionForm*, para controladores que operan sobre el Modelo, o mediante parámetros de la petición, para controladores que recuperan información.

7.2.5. Datos de complejidad del sistema

Este apartado presenta algunos datos cuantitativos que permiten valorar el tamaño y la complejidad del sistema heredado a modernizar. En primer lugar, se utilizan dos métricas genéricas de complejidad

[10] http://www.theserverside.com/discussions/thread.tss?thread_id=20936

	Modelo	Vista	Controlador	Total
Número de archivos	36	34	49	119
Número de líneas de código	1989	2781	3402	8172

Tabla 7.1: Número de archivos y de líneas de código de cada componente

Objeto de datos	Atributos	Relaciones	Operaciones
Coauthor	5	1	6
Conference	9	1	5
Group	3	0	6
Paper	5	7	9
Review	13	2	7
Subject	2	0	6
SummaryReviews	8	1	0
Track	3	1	6
User	8	8	11
Total	56	21	56

Tabla 7.2: Datos cuantitativos de los objetos de datos del sistema

como son el número de archivos y el número de líneas de código. Estas métricas permiten apreciar el tamaño del sistema. La Tabla 7.1 muestra el número de archivos y el número de líneas de código necesarias para la especificación de cada componente del patrón MVC. En el caso del componente Modelo, se han considerado las clases de definición de los objetos de datos, las clases de los DAO y los mapeos objeto-relacionales. Los dos primeros están formados exclusivamente por archivos de código Java, mientras que el último consiste en archivos XML. Para el componente Vista, se han considerado simplemente las *Java Server Pages*. Y, finalmente, para el componente Controlador, se han tenido en cuenta las clases definidas por la aplicación dentro de la jerarquía *Actions* y *ActionForms*.

Revisados los datos de manera general, se va a realizar una exposición más detallada de cada uno de los componentes, atendiendo a las características principales de los mismos de mayor interés para el proceso de modernización definido.

7.2.5.1. Modelo

En el caso del componente Modelo, las características más interesantes de cada uno de sus elementos principales (objetos de datos) son: el número de atributos, el número de relaciones y el número de operaciones. La Tabla 7.2 muestra los valores de cada uno de estos indicadores para cada uno de los objetos de datos del sistema heredado.

7.2.5.2. Vista

En el caso del componente Vista, las características más interesantes de cada uno de sus elementos principales (JSPs) son: el número de formularios, el número de peticiones realizadas a través de un formulario (*submit*), el número de peticiones realizadas mediante un enlace (*link*), el número de objetos

Java Server Page	forms	submit	link	POs	PCs
assignedPapersToReviewerList	0	0	7	1	4
assignPCmember	1	8	0	1	5
changeReviewer	1	1	0	1	2
conferencelist	0	0	0	0	1
conferencestatus	1	1	0	0	2
confirmUserRegistration	1	1	0	0	0
createconference	1	1	0	0	0
createPCmember	1	1	0	0	0
createSubjectsAndTracks	2	2	0	1	3
editconference	1	1	0	1	1
editPCmember	1	1	2	0	1
editSubjectsAndTracks	4	4	2	1	3
login	1	1	1	0	0
menu	0	0	16	0	0
navigation	0	0	8	0	0
paperCreate	1	1	0	0	2
paperDetails	0	0	0	1	3
paperEdit	2	3	2	0	3
paperList	0	0	3	1	1
paperListPCchair	1	1	1	0	3
paperListPCmember	1	0	2	0	3
papersAcceptRejectList	1	1	11	0	2
preferencesTrackSubjects	1	1	0	0	2
registerAuthor	1	1	0	0	0
reviewCreateEdit	1	2	0	1	6
Total	24	32	55	9	47

Tabla 7.3: Datos principales de las JSPs

de presentación que la componen y el número de colecciones de presentación. La Tabla 7.3 muestra los valores de cada uno de estos indicadores para cada una de las JSPs del sistema heredado. Se han eliminado nueve JSP auxiliares de la tabla porque no aportaban información útil para este caso de estudio.

7.2.5.3. Controlador

Finalmente, en el caso del componente Controlador, las características más interesantes de cada uno de los controladores definidos son: el número de respuestas de tipo página (*JSP*), el número de respuesta de tipo control (*Action*), el número de variables locales (*vars*), el número de llamadas a operaciones de datos (*DOpC*) y a colecciones (*COpC*), y el número de mapeos (*maps*). La Tabla 7.4 muestra los valores de cada uno de estos indicadores para cada uno de los *Actions* del sistema heredado. Por brevedad, sólo se incluyen aquellos *Actions* que son soportados por el proceso de ingeniería inversa definido.

7.2.5.4. Métricas adicionales

Tras su análisis y contextualización al proyecto de modernización, en principio, se ha descartado el uso de otras métricas de complejidad tradicionales principalmente porque o bien su aplicación carecía de

Controladores	JSP	Action	Vars	DOpC	COpC	Maps
AssignedPapersToReviewerListAction	1	0	5	4	3	3
CheckFirstLoginAction	0	5	1	3	0	3
ConfirmUserRegistration	1	3	3	3	0	1
CreateConferenceAction	1	1	1	1	0	1
CreateSubjectsAction	2	1	4	4	1	1
CreateSubjectsAndTracks	2	0	5	4	1	4
CreateTracksAction	2	1	3	3	0	1
DeleteSubjectAction	3	1	4	4	1	1
DeleteTrackAction	3	1	3	3	0	1
EditConferenceAction	2	1	2	2	0	1
EditSubjectAction	3	1	1	1	0	1
EditTrackAction	3	1	3	3	0	1
ListConferenceAction	0	1	1	1	0	1
ModifyAndDeleteSubjectsAndTracks	0	2	5	4	0	4
PaperCreateAction	0	1	5	7	0	1
PaperDeleteAction	0	1	2	2	0	1
PaperDetailsAction	1	0	4	3	0	4
PaperEditAction	0	1	5	7	0	1
PaperListAction	2	0	2	2	0	2
PaperListPCchairAction	2	0	8	8	2	2
PaperPopulateCreateAction	1	0	2	2	0	2
PaperPopulateEditAction	1	0	9	6	3	5
PapersAcceptRejectListAction	1	0	3	3	0	3
PopulateConfirmUserRegistration	1	0	1	2	0	1
PrefsTrackSubjectsAction	0	1	14	5	6	1
SelectSubjectToModify	2	0	7	5	0	6
SelectTrackToModify	2	0	7	6	0	7
SubmitPrefsTrackSubjectsAction	0	1	7	5	2	1
Total	40	20	117	103	19	58

Tabla 7.4: Datos principales de los controladores

sentido dentro del caso de estudio, o bien no aportaban información relevante para el proceso de modernización. Por un lado, dentro del primer grupo, podemos señalar, por ejemplo, el caso de la complejidad ciclomática, (McCabe 1976). No aplicable en nuestro caso de estudio, porque la capa de presentación web del caso de estudio está definida mediante clases *Action* que definen flujos de control bastante lineales. Por otro lado, dentro del segundo grupo, podemos señalar, por ejemplo, el caso del acoplamiento entre clases, (Harrison *et al.* 1998), cuyos valores no aportan demasiado al proceso de modernización.

En el mismo sentido, se han estudiado algunas de las métricas usadas en el contexto de aplicaciones Web, fundamentalmente a partir de la clasificación de (Calero *et al.* 2005), pero en la mayoría de los casos no nos proporcionan información directamente contrastable con la información obtenida por el proceso de modernización en su estado actual, que básicamente define un proceso de ingeniería inversa.

7.3. Alcance de la representación conceptual

Con el fin de ilustrar el alcance de la representación conceptual y proporcionar ejemplos para el desarrollo del siguiente capítulo, en esta sección se presenta con detalle la representación del subsistema de gestión del envío de artículos por parte de un autor. Este subsistema está formado por los siguientes elementos:

- Entidades de datos principales: *Paper*, *Subject*, *Track* y *Coauthor*.
- Páginas:
  - Listado de artículos enviados (*paperList.jsp*).
  - Enviar un nuevo artículo (*paperCreate.jsp*).
  - Editar un artículo enviado (*paperEdit.jsp*).
  - Detalles de un artículo enviado (*paperDetails.jsp*).
- Acciones:
  - Listar artículos de un autor (*PaperListAction*)
  - Mostrar los detalles de un artículo enviado (*PaperDetailsAction*)
  - Poblar página de envío de nuevo artículo (*PaperPopulateCreateAction*)
  - Procesar el envío de un nuevo artículo (*PaperCreateAction*)
  - Poblar página de edición de un artículo enviado (*PaperPopulateEditAction*)
  - Procesar edición de un artículo enviado (*PaperEditAction*)

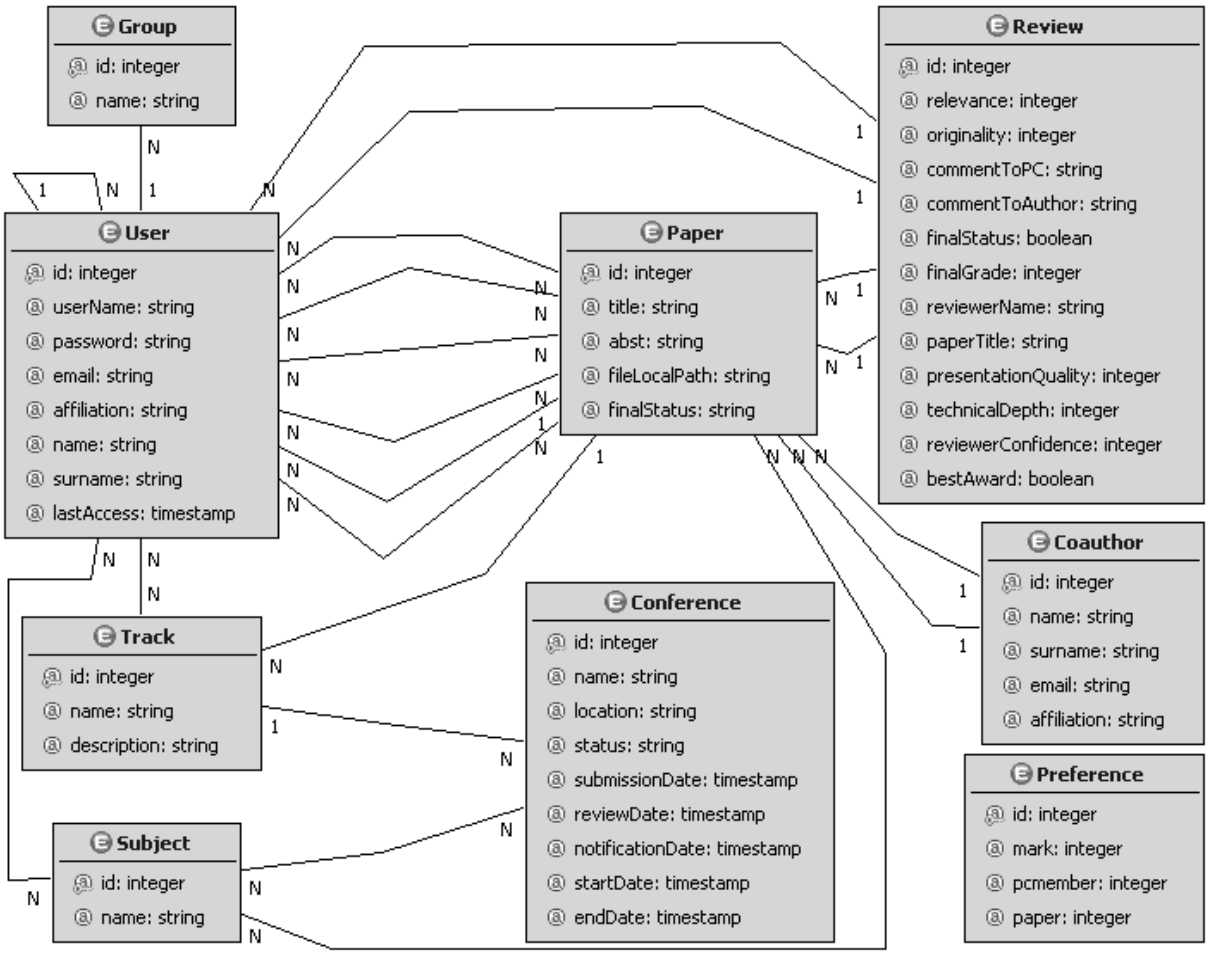


Figura 7.1: Modelo de datos del CRS

7.3.1. Modelo

Antes de detallar la especificación de las entidades de datos del ejemplo, es necesario identificar qué entidades de datos componen el sistema heredado. La Figura 7.1 presenta el modelo de datos, ilustrando las entidades existentes, sus atributos y relaciones. Como se puede observar, el sistema contiene nueve entidades de datos diferentes, siendo *Conference*, *User*, *Paper* y *Review* las entidades principales del sistema.

La Figura 7.2 muestra los objetos de datos generados a partir del sistema heredado para las entidades relacionadas con el subsistema de gestión de envío de artículos. Como se puede apreciar, la notación permite especificar de forma completa los atributos y las relaciones de cada una de las entidades involucradas. Por ejemplo, en el caso del objeto de datos *Paper*, se especifican:

- Cinco atributos de datos, que son: *id*, *title*, *abst*, *fileLocalPath* y *finalStatus*. El atributo *id* tiene el valor true para indicar que actúa como clave primaria, ilustrado en la Figura 7.3.

- Siete relaciones. Las relaciones con la entidad *User* (*assignedPcmembers*, *author* y *conflictPcmembers*), con la entidad *Subject* (*subjects*), con la entidad *track* (*track*) y con la entidad *CoAuthor* (*coauthors*).

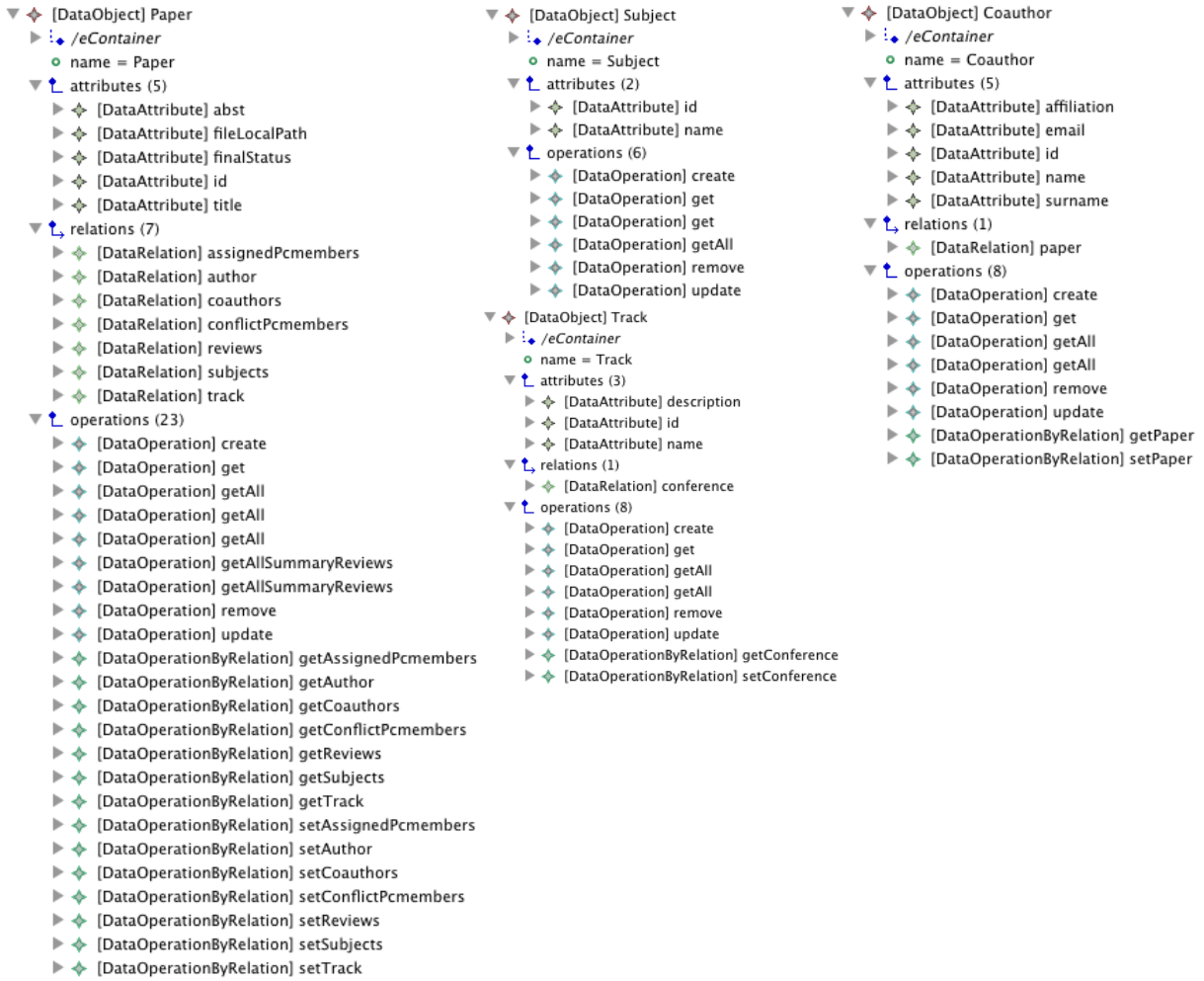


Figura 7.2: Objetos de datos del subsistema (modelo MVC)

Es importante observar que en el caso de relaciones muchos a muchos (N:N) sólo se especifica la relación en el objeto de datos que actúa como maestro de la relación. Por ejemplo, en la relación N:N entre *Paper* y *Subject*, sólo se genera la relación en el objeto de datos *Paper* (*subjects*) y no en el de *Subject*. Recuérdese que esta información puede extraerse de los ficheros de mapeo objeto-relacional del sistema de persistencia de objetos. La Figura 7.3 ilustra el nivel de detalle de la especificación de una relación, indicándose su tipo (*Manytomany*), el objeto de datos relacionado (*Subject*) y también las operaciones con las que está relacionada (*getSubjects* y *setSubjects*).

Por otro lado, según nuestra notación, debe especificarse un par de operaciones del tipo *DataOperationByRelation* por cada una de las relaciones que se indiquen en un objeto de datos, que permita recuperar datos relacionados (*get*) o relacionar datos (*set*). Por ejemplo, en el caso del objeto de datos *Paper*, se generan catorce operaciones de este tipo, como se ilustra en la Figura 7.2. La Figura 7.3 ilustra en detalle de la especificación de una operación de datos de este tipo, indicándose la relación de la que depende (*subjects*) y los detalles del retorno de la operación, la operación devuelve una lista (*List*) de objetos de datos (*type=DataObject)* del tipo de objeto de datos *Subject*.

Figura 7.3: Detalles objeto de datos Paper

```
public interface PaperDAO {
        public Paper create(Paper paper) throws Exception;
        public Paper update(Paper paper) throws Exception;
        public void remove(Paper paper) throws Exception;
        public Paper get(long id) throws Exception;
        public Collection<Paper> getAll(User user) throws Exception;
        public Collection<Paper> getAll(Track track) throws Exception;
        public Collection<Paper> getAll() throws Exception;

        public Collection<SummaryReviews> getAllSummaryReviews() throws Exception;
        public Collection<SummaryReviews> getAllSummaryReviews(String orderByColumn,
                                                               String orderType)
                throws Exception;
}
```

Figura 7.4: Código java de la interfaz PaperDAO

Finalmente, se deben especificar las operaciones de datos relacionadas con cada uno de los objetos de datos, y que se pueden extraer de los interfaces que definen el patrón DAO para cada uno de ellos. La Figura 7.4 muestra el código de la interfaz Java *PaperDAO* que define, como se puede ver, nueve operaciones distintas que permiten operar con el objeto de datos *Paper*. Estas operaciones son representadas con detalle, indicando la información relacionada con los parámetros y el retorno, en la especificación realizada del objeto *Paper* en el modelo MIGRARIA MVC. Como ilustra la Figura 7.3, para la operación create se especifica, por un lado, un parámetro denominado *paper* (*name*), cuyo tipo es el objeto de datos (*type=DataObject*) *Paper*, y por otro lado, el retorno de la misma como un objeto de datos *Paper*.

7.3.2. Vista

En el sistema heredado, un autor, al entrar en el sistema, es redirigido a la página que muestra un listado de los artículos enviados (*paperList.jsp*) y presenta un menú con dos opciones (peticiones): realizar un nuevo envío (*Create*) o revisar los envíos realizados (*List*). En el caso de que se haya enviado algún artículo previamente, el sistema muestra tres acciones relacionadas (peticiones) con cada ítem de la lista: acceder a los detalles del envío (enlace incluido en el campo título del artículo), editar ese envío

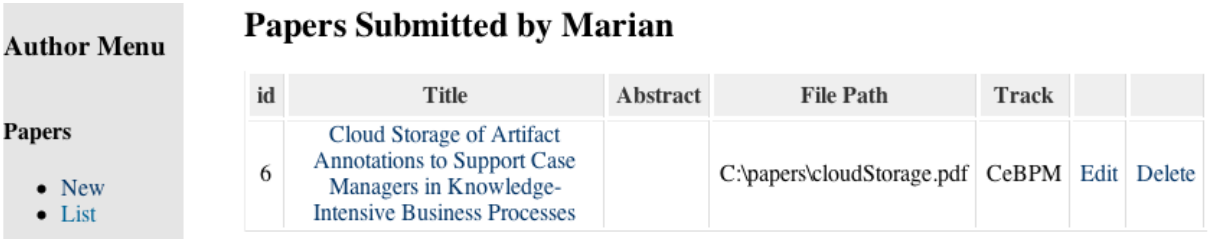


Figura 7.5: Página listado de artículos enviados

Figura 7.6: Página de envío de un nuevo artículo

(*Edit*) o borrarlo (*Delete*). Por lo tanto, el mapa de navegación de este subsistema se inicia en la página listado a partir de la cual se puede acceder al resto de páginas y, desde éstas, se acaba regresando de nuevo al listado. La Figura 7.5 muestra la composición de esta página, que está básicamente formada por el nombre del autor de los artículos (*Marian*) y por una colección de artículos. Por cada artículo se muestran los campos *id*, *title*, *abstract*, *file path* y *track*.

La página de envío de un nuevo artículo (*paperCreate*.jsp), Figura 7.6, por su parte, está formada por[11] un formulario que permite recoger los valores de los atributos del nuevo artículo a enviar. Los tres primeros atributos son campos de texto simple, mientras que los dos últimos son dos listas de selección: una de selección simple (*Track*) y otra de selección múltiple (*Subjects*). Además, el formulario incluye un botón para realizar la petición de envío de los datos al servidor (*new*).

La página de edición de un artículo ya enviado (*paperEdit.jsp*) contiene, Figura 7.7, por un lado, el mismo formulario que la de envío de nuevo artículo y, por otro, un listado y otro formulario que permite gestionar los coautores del artículo enviado. Para cada uno de los coautores se muestra el valor de sus atributos (*id*, *name*, *surname*, *email*, *affiliation*). Además, se puede editar (*Edit*) o borrar (*Delete*) cada uno de los coautores listados. El formulario permite especificar los valores de los atributos de un nuevo coautor y contiene un botón para realizar el envío de los datos del nuevo coautor al sistema (*new*).

[11] Se omiten los enlaces del menú por evitar redundancia

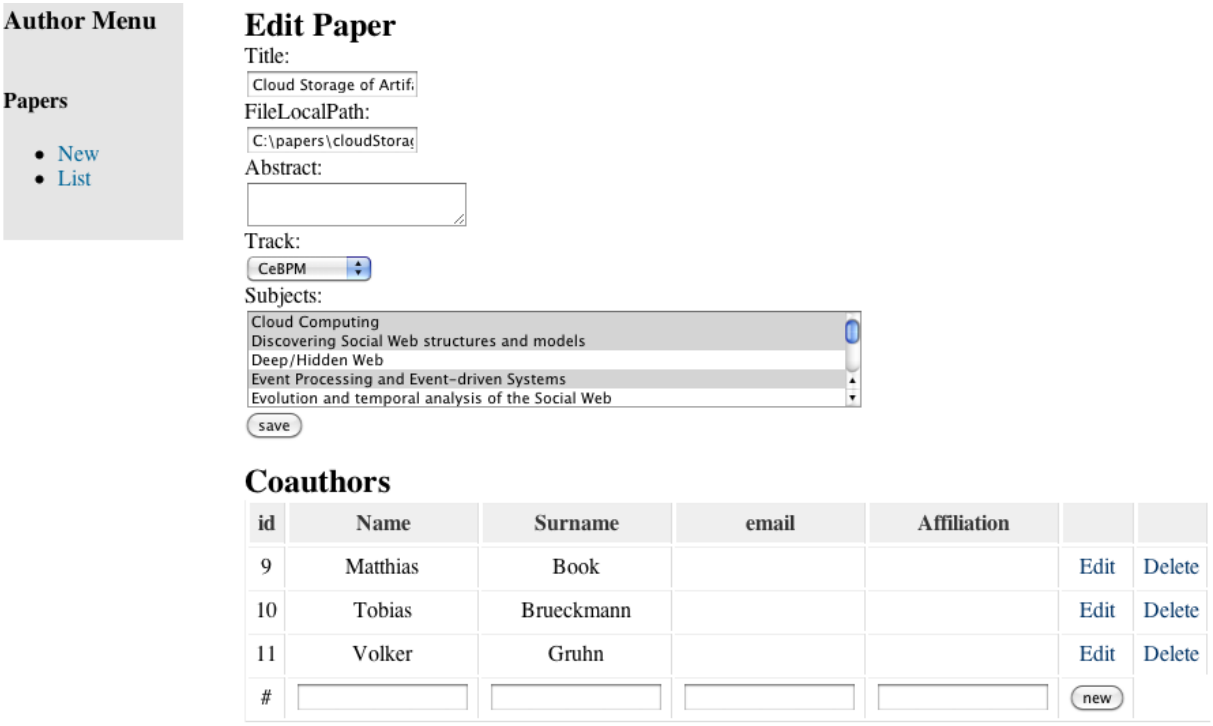


Figura 7.7: Página de edición de un artículo enviado

Finalmente, la página de detalles (*paperDetails.jsp*), Figura 7.8, muestra la información relacionada con un artículo enviado. En concreto, muestra los datos base del artículo (*id*, *title*, *abstract*, *track*, *subjects*), los datos de sus coautores (*id*, *name*, *surname*, *email*, *affiliation*) y los datos de sus revisiones (*reviewer name*, *comment to author*, *final grade*).

La Figura 7.9 muestra la especificación de la composición de las páginas descritas anteriormente en base a objetos de presentación, colecciones y peticiones. Como se puede comprobar, la notación es capaz de especificar todos los elementos identificados anteriormente en la revisión de la composición de las páginas.

Con el objetivo de ilustrar los detalles más significativos de la especificación de las vistas, la Figura 7.10 muestra algunos detalles de la página de listado de artículos enviados, mientras que la Figura 7.11 muestra los detalles de la página de envío de artículos.

En la imagen 7.10, se muestran los detalles del objeto de presentación *author* y de la colección de objetos de presentación *papers*. Por un lado, el objeto de presentación *author* está simplemente formado por un atributo de presentación de datos (*Data_PA*) que representa su atributo nombre (*name*). Por otro lado, la colección está compuesta por un solo objeto *paper* que actúa de ítem y de tres peticiones que se realizan a nivel de ítem: *bean.write[paper.title]*, *Delete* y *Edit*. El objeto de presentación de salida paper está compuesto de cinco atributos simples (*Data_PA*). La Figura 7.10 ilustra, también, los detalles de una de las peticiones contenidas dentro de la colección, la petición Edit. Como se puede apreciar, para esta petición se ha indicado como ruta (*path*) el relacionado con el *Action* que generará la página de edición (*PaperPopulateEdit*), que es una petición relacionada con un objeto de presentación *(type=object)* y

Author Menu

Papers

- New
- List

Paper 6 details

Title	Abstract	Track	Subjects
Cloud Storage of Artifact Annotations to Support Case Managers in Knowledge-Intensive Business Processes		CeBPM	Cloud Computing Discovering Social Web structures and models Event Processing and Event-driven Systems

Coauthors

id	Name	Surname	email	Affiliation		
9	Matthias	Book				
10	Tobias	Brueckmann				
11	Volker	Gruhn				

Reviews

Reviewer Name	Comment to Author	Final Grade
Karlr	okokok	1

Figura 7.8: Página listado de artículos enviados

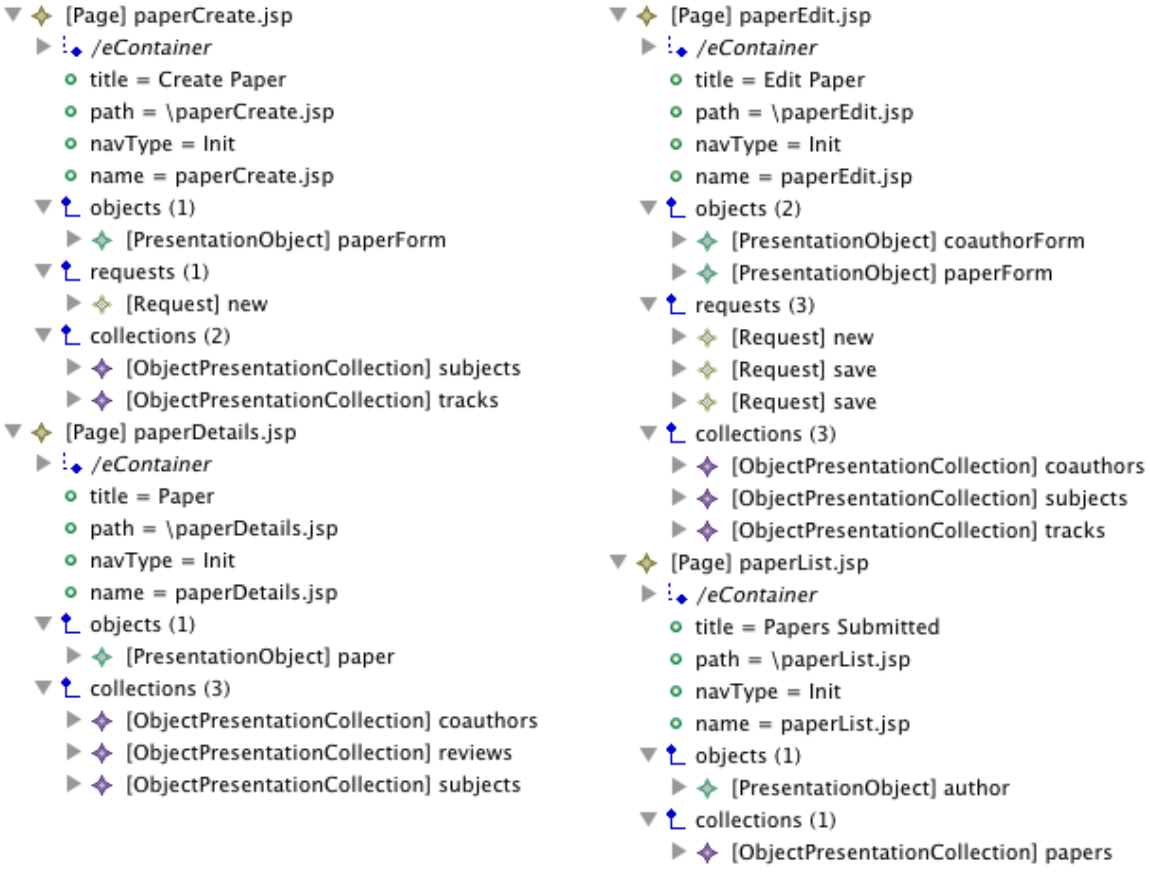


Figura 7.9: Modelos de las páginas de gestión de envío de artículos

Figura 7.10: Modelos de las páginas de gestión de envío de artículos

que contiene un parámetro de tipo *ObjectRequestParameter*, el parámetro *id* que se corresponde con el atributo (*property*) *id* del objeto de presentación (*object*) *paper* usado como ítem de la colección. Este parámetro será utilizado por el *Action* encargado de generar la página de edición para obtener el objeto de datos de tipo *Paper* determinado para poblar dicha página.

Y en la imagen 7.11, se muestran los detalles del objeto de presentación de entrada *paperForm*, de la colección *subjects* relacionada con el atributo de presentación *subjects* del objeto *paperForm* y de la petición *new*. Este objeto de presentación, que representa el formulario de la página, está compuesto de cuatro atributos simples (*Data_PA*), de dos atributos de conjunto de datos (*DataSet_PA*) y de un atributo de control (*Submit_PA*). Del conjunto de atributos simples se ha destacado id para ilustrar cómo aparece especificado como un campo oculto del formulario (*hidden=true*), mientras que del par de atributos de conjunto se ha destacado subjects para indicar su relación (*items*) con la colección de objetos de presentación subjects. Esta colección es la encargada de especificar el conjunto de datos que contendrá ese atributo (campo de selección en el formulario) y, por lo tanto, indica que está compuesta por un objeto con dos atributos simples (*Data_PA*) que son los que se utilizan finalmente en la población del atributo conjunto. Obsérvese, también, que la notación permite especificar qué atributo del objeto de la colección actuará como etiqueta. En este caso, uno de los atributos aparece identificado como etiqueta (*isLabel=true*), mientras que el otro no. Finalmente, la petición new de tipo *object* indica como ruta el *Action* encargado de operar con el modelo para generar un nuevo objeto de tipo *Paper* (*PaperCreate*). Esta petición contiene un parámetro de tipo objeto (*ObjectRequestParameter*) para indicar el objeto de presentación relacionado (*object*), en este caso es el objeto de presentación de entrada *paperForm* y su propiedad (*property*) op, que es de tipo *Submit_PA* señalando que se trata del envío del objeto de presentación completo como parámetro de la petición.

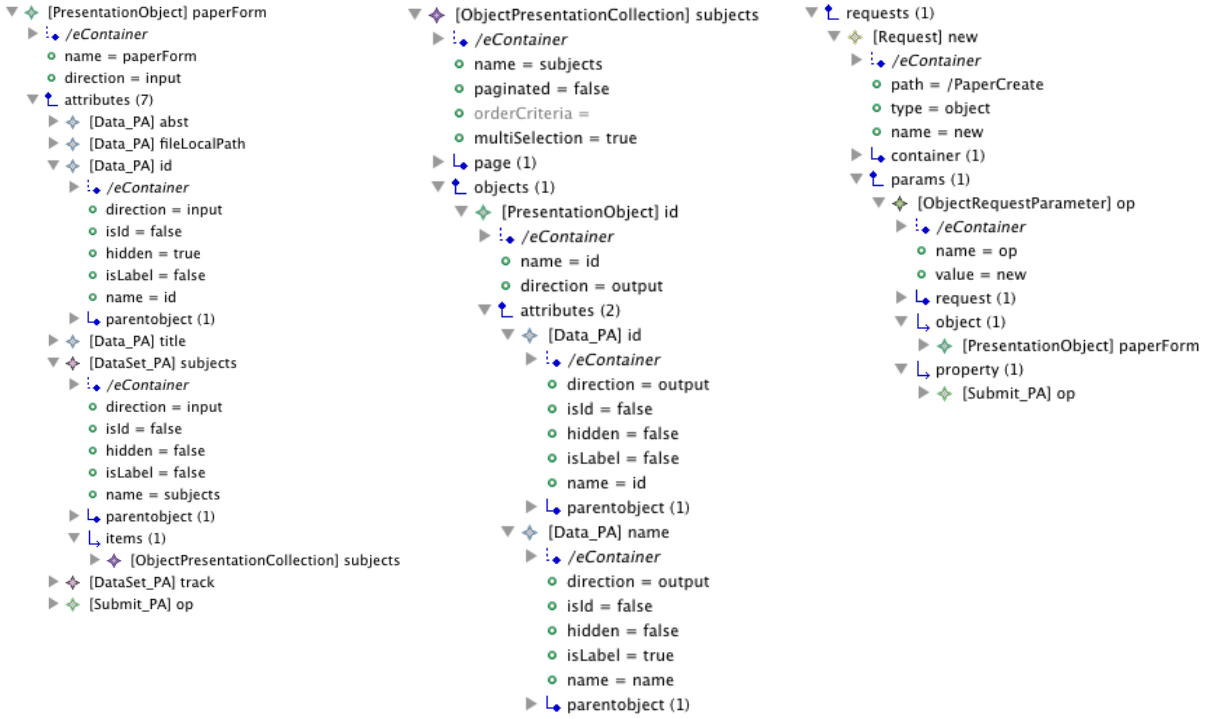


Figura 7.11: Modelos páginas de gestión de envío de artículos

7.3.3. Controlador

El subsistema de gestión del envío de artículos por parte de un autor está formado por siete controladores (*Action*) principales, relacionados anteriormente. A continuación se presenta y comenta una versión simplificada del código de alguno estos controladores para ejemplificar, posteriormente, su representación en la notación MIGRARIA_MVC.

En primer lugar, se muestra el código del controlador *PaperListAction*. Este controlador es el responsable de generar la página de listado de artículos enviados (*paperList.jsp*). Su funcionamiento, básicamente, consiste en obtener el objeto de datos del usuario actual (*user*) y todos los artículos relacionados con él dentro de una colección de datos (*papers*). Posteriormente, se encarga de indicar al sistema de gestión de plantillas JSP las relaciones entre los objetos de presentación y los objetos generados, en este caso las relaciones *author-user* y *papers-papers*.

```
1  //PaperListAction
2  public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
       request, HttpServletResponse response)throws Exception
3  {
4          //Obtener instancias DAO
5          HttpSession session = request.getSession();
6          UserAccesses userObj = (UserAccesses) session.getAttribute(Constants.
               SESSION_ATTRIBUTE_DATA_USER);
7          User user = userDAO.get(userObj.getId());
8          Collection<Paper> papers = paperDAO.getAll(user);
9          request.setAttribute("author",user);
10         request.setAttribute("papers",papers);
11         return mapping.findForward("success");
12 }
```

En segundo lugar, se muestra el código del controlador *PaperDetailsAction*. Este controlador es el responsable de generar la página de detalles de un artículo enviado (paperDetails.jsp). Su funciona-

miento, básicamente, consiste en recuperar el objeto Paper (paper) que se quiere presentar y sus objetos CoAuthor y Reviews relacionados. Para recuperar el objeto Paper determinado se utiliza el parámetro de la petición id. Esta petición se realiza desde la página de listado de artículos enviados a partir del título del artículo, como muestra la Figura 7.5. Finalmente, se indica cómo se puebla cada objeto y colección de presentación de la página a generar, en base a las relaciones paper-paper, coauthors-coauthors y reviews-reviews.

```
1  //PaperDetailsAction
2  public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
       request, HttpServletResponse response)throws Exception
3  {
4          //Obtener instancias DAO
5          Long pId = new Long(request.getParameter("id"));
6          Paper paper = paperDAO.get(pId);
7          Collection<Coauthor> coauthors = coauthorDAO.getAll(paper);
8          Collection<Review> reviews = reviewDAO.getAll(paper);
9          request.setAttribute("paper", paper);
10         request.setAttribute("coauthors", coauthors);
11         request.setAttribute("reviews", reviews);
12         return mapping.findForward("success");
13 }
```

Finalmente, se muestra el código del controlador *PopulatePaperCreateAction* y *PaperCreateAction* que son los relacionados con la página de envío de un nuevo artículo (*paperCreate.jsp*). El primero es el encargado de generar la página, mientras que el segundo es el encargado de atender la petición de envío de formulario generada desde esa página. El primer controlador sigue el mismo esquema que los presentados anteriormente para poblar los campos de selección (*Tracks* y *Subjects*) del formulario de la página de envío de un nuevo artículo, mostrados en la Figura7.6. Por su parte, el segundo controlador se encarga de obtener los datos de entrada a partir del formulario (*PaperForm*) y generar un nuevo objeto de datos *Paper*, a partir de esos datos, que será finalmente guardado por el sistema de persistencia (*paperDAO.create*). Es interesante resaltar la necesidad de realizar conversiones entre los datos recibidos y los datos requeridos por un objeto *Paper*, como son la conversión de la colección de campos *id* de *Subject* a la colección de objetos *Subject* con esos *id* (líneas 22-23).

```
1  //PaperPopulateCreateAction
2  public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
       request, HttpServletResponse response)throws Exception
3  {
4          //Obtener instancias DAO
5          Collection<Track> tracks = trackDAO.getAll();
6          Collection<Subject> subjects = subjectDAO.getAll();
7          request.setAttribute("tracks", tracks);
8          request.setAttribute("subjects", subjects);
9          return mapping.findForward("success");
10 }
11
12 //PaperCreateAction
13 public ActionForward execute(ActionMapping mapping, ActionForm form, HttpServletRequest
       request, HttpServletResponse response)throws Exception
14 {
15         //Obtener instancias DAO
16         HttpSession session = request.getSession();
17         UserAccesses userObj = (UserAccesses) session.getAttribute(Constants.
               SESSION_ATTRIBUTE_DATA_USER);
18         User user = userDAO.get(userObj.getId());
19         PaperForm pf = (PaperForm) form;
20         Paper paper = new Paper();
21         Track track = trackDAO.get(pf.getTrack());
```

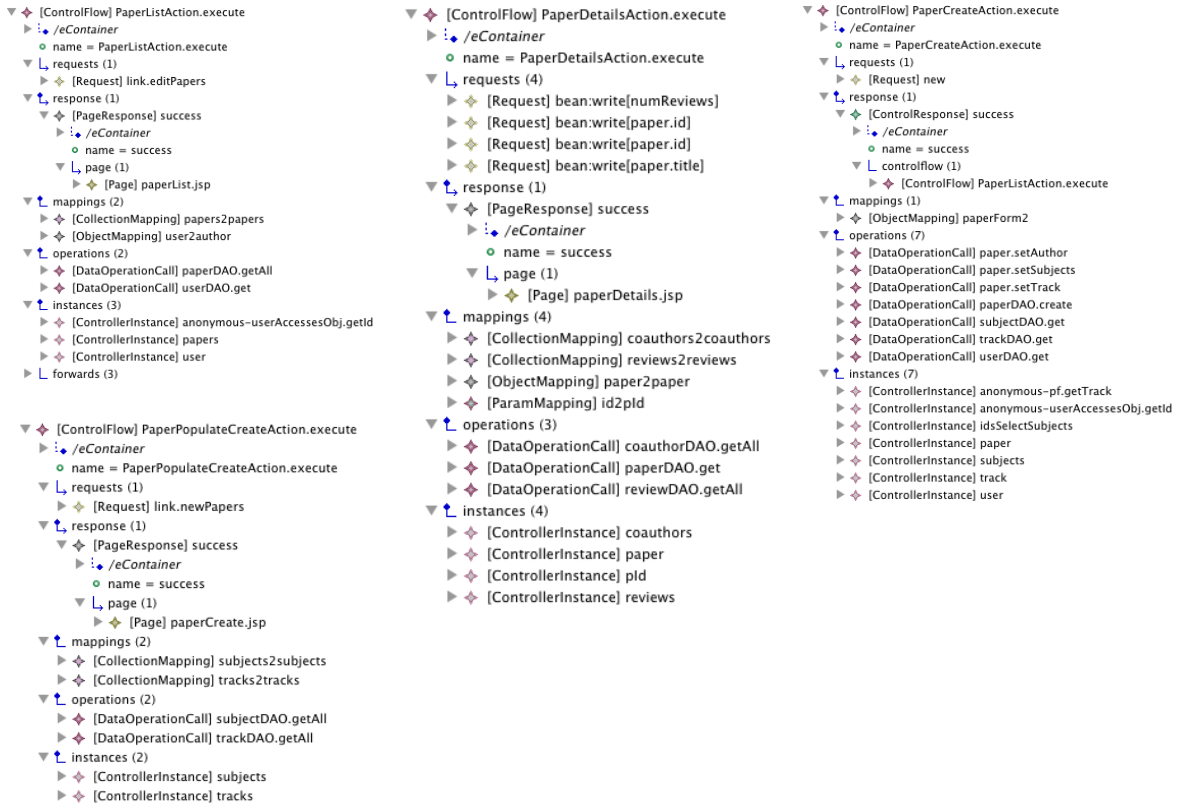


Figura 7.12: Modelos de controladores para la gestión de envío de artículos

```
22          Long idsSelectSubjects[] = convertStringArrayToLongArray(pf.getSubjects());
23          Set<Subject> subjects = new HashSet<Subject>(subjectDAO.get(idsSelectSubjects));
24          paper.setTitle(pf.getTitle());
25          paper.setAbst(pf.getAbst());
26          paper.setFileLocalPath(pf.getFileLocalPath());
27          paper.setTrack(track);
28          paper.setAuthor(user);
29          paper.setSubjects(subjects);
30          paperDAO.create(paper);
31          return mapping.findForward("success");
32  }
```

La Figura 7.12 muestra la especificación resumida de los controladores descritos anteriormente. Como se puede comprobar, la notación es capaz de especificar todos los elementos identificados anteriormente en la revisión de los controladores. También, se puede apreciar la especificación de la navegación mediante, fundamentalmente, los elementos *request* y *response*. De esta manera, en el caso del *ControlFlow PaperDetailsAction*, aparecen referenciadas cuatro peticiones (requests) distintas que permiten invocarlo, como por ejemplo, la última (*bean:write[paper.title]*) que se corresponde con la petición realizada desde la página de listado de artículos enviados a partir del título del artículo, como muestra la Figura 7.5. Con respecto a las respuestas, el *controlFlow PaperCreateAction* responde pasando el control (*ControlResponse*) al *ControlFlow PaperListAction*, mientras que los otros tres devuelven una página generada a partir de una *JSP* como respuesta (*PageResponse*), como por ejemplo, el *ControlFlow PaperListAcion* devuelve la página *paperList.jsp* (referencia a este elemento en el paquete de la vista).

La Figura 7.13 muestra los detalles más interesantes del controlador *PaperListAction*. En concreto, se muestran los mapeos de salida (*type=output*) entre los objetos y colleciones de datos y de presentación.

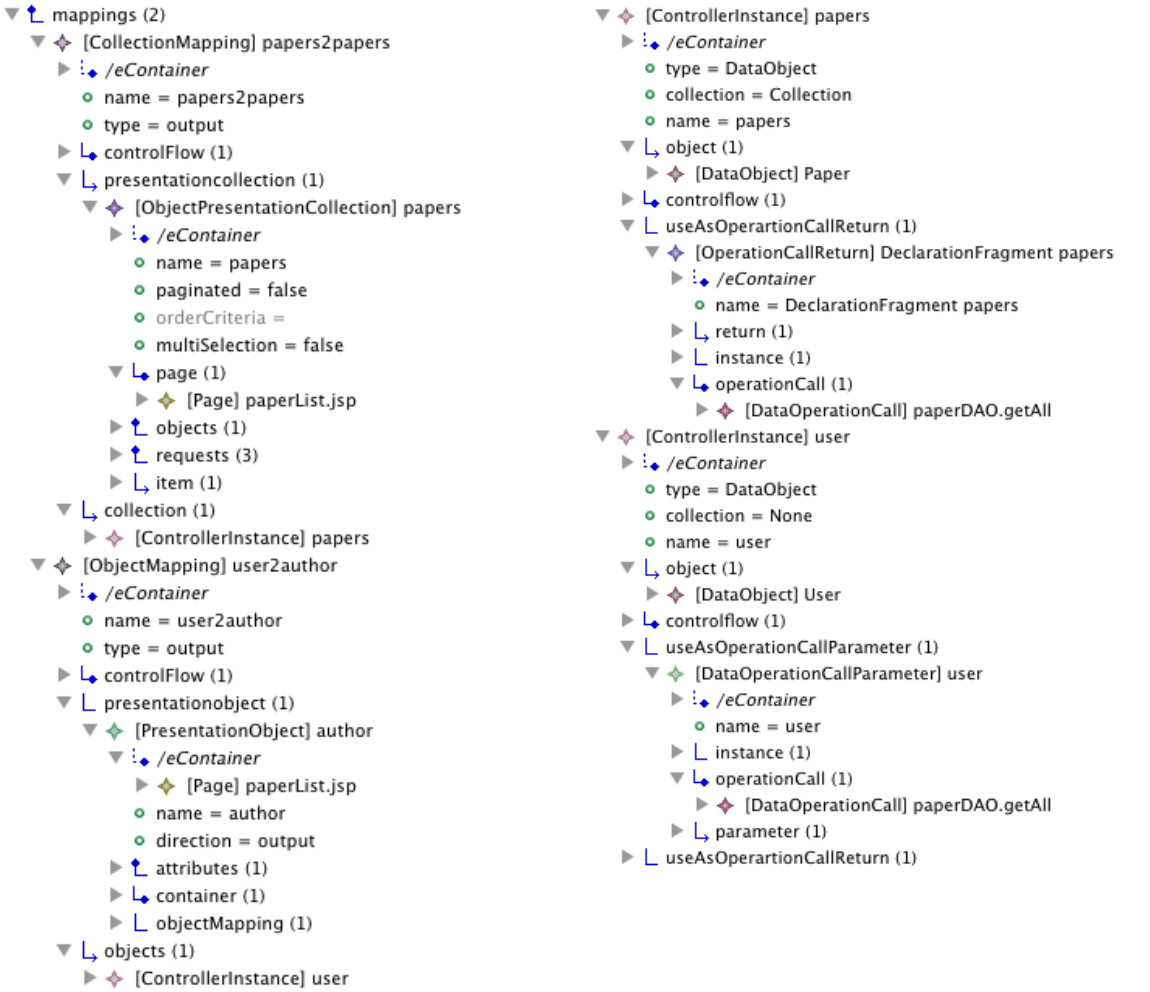


Figura 7.13: Modelo CRS PaperListAction

El mapeo de colección (*CollectionMapping*) *papers2papers* especifica un mapeo entre la colección de presentación de objetos (*papers*) contenida en la página (referencia *Page*) *paperList.jsp* y la instancia del controlador *papers*. Esta instancia está definida como una colección de objetos de datos de tipo *Paper* (referencia *object*). Se utiliza también como retorno de la llamada a operación *paperDAO.getAll*, de manera que almacena todos los objetos *Paper* devueltos por esa operación, que son los que se usarán para poblar la página de respuesta. El mapeo de objeto (*ObjectMapping*) *user2author* especifica un mapeo entre el objeto de presentación *author* contenido en la misma página y la instancia de controlador *user*. Esta instancia está definida como un objeto de datos del tipo (referencia *object*) *User*. Y se utiliza como parámetro de la llamada a operación de datos *paperDAO.getAll*, actuando como filtro del conjunto de datos a devolver.

La Figura 7.14 muestra los detalles más interesantes del controlador *PaperDetailsAction*. En este caso, se detalla el caso del mapeo de parámetros de petición (*ParamMapping*) *id2pid*, un mapeo de entrada (*type=input*) que se establece entre la instancia de controlador (referencia *instance*) *pid* y el parámetro (referencia *parameter*) *id* de la petición *bean:write[paper.title]* de la colección de objetos de presentación *papers* (referencia *container*) de la página *paperList.jsp*. La instancia de controlador *pid* es

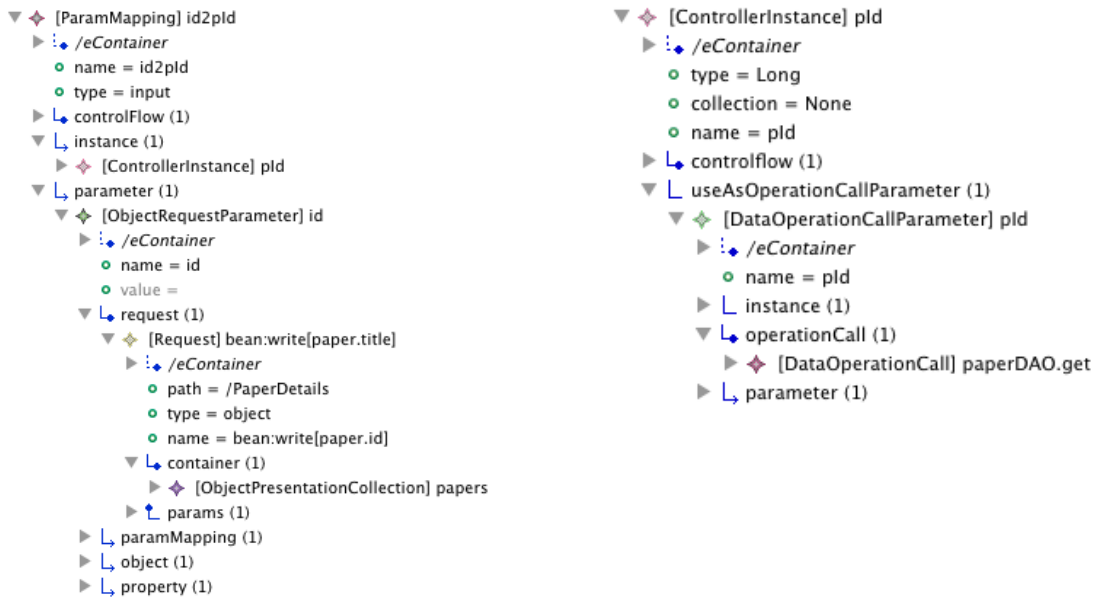


Figura 7.14: Modelo CRS PaperDetailsAction

de tipo *Long* y se utiliza como parámetro de la llamada a operación de datos *paperDAO.get* para indicar qué objeto *Paper* se quiere recuperar para poblar la página de respuesta.

La Figura 7.15 muestra los detalles más interesantes del controlador *PaperCreateAction*. En este caso, se ilustra la especificación del mapeo de entrada (*type=input*) entre el objeto de presentación de entrada (*direction=input*) *paperForm* contenido en la página (referencia *container*) *paperCreate.jsp* y varias instancias de controlador, entre otras *paper*. Este mapeo contiene, a su vez, mapeos de atributos (*AttrMapping*) que lo definen en una granularidad más fina. Por ejemplo, se definen tres mapeos de atributos entre el objeto de presentación de entrada y la instancia paper, que corresponden a los atributos *abst*, *fileLocalPath* y *title* de ambos objetos (líneas 24-26 del código). El resto de mapeos corresponde a la aparición de una invocación a un método get sobre el objeto paperForm (líneas 21 y 22 del código). La instancia *paper* es un objeto de datos de tipo *Paper* que se usa como parámetro de 4 llamadas a operaciones distintas: *paperDAO.create*, *paper.setTrack*, *paper.setAuthor* y *paper.setSubjects*. La primera referencia a una operación de datos (*DataOperation*), mientras que el resto referencian a operaciones de datos derivadas de relaciones (*DataOperationByRelation*), nótese la diferencia del objeto sobre el que se invoca el método.

7.3.4. Limitaciones

La notación definida posee suficiente expresividad para poder especificar las capas conceptuales de una aplicación web, como son datos, navegación y presentación, para el sistema presentado en el caso de estudio. Sin embargo, también presenta limitaciones que se corresponden con los temas que aún no se han abordado en profundidad en el estado actual de la notación. Las limitaciones fundamentales, por tanto, se pueden resumir en:

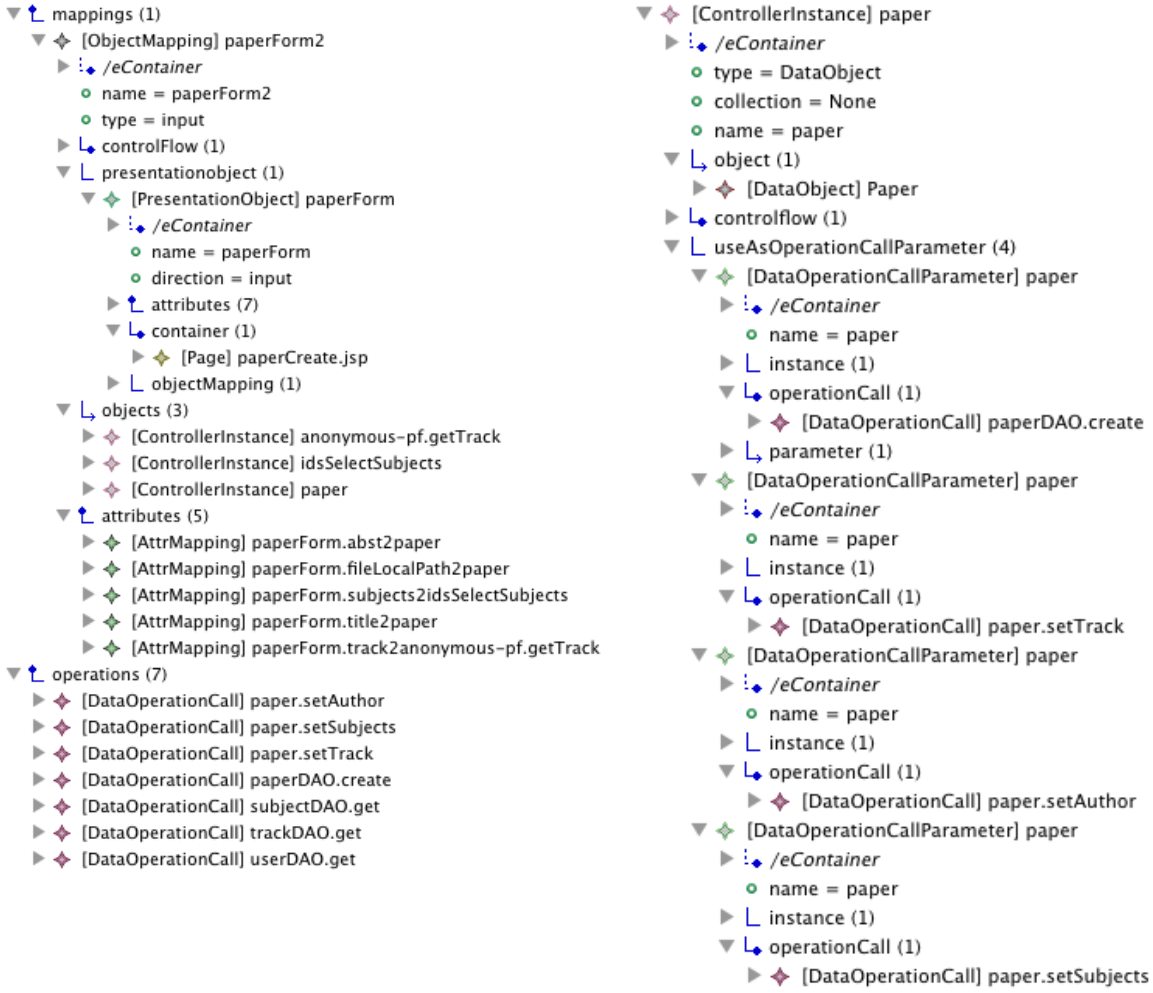


Figura 7.15: Modelo CRS PaperCreateAction

- Falta de soporte para el uso de la sesión dentro de un controlador. Aunque es posible especificar el uso que se hace de la sesión dentro de un controlador por medio de instancias de controlador y llamadas a operaciones, actualmente no se han definido conceptos concretos para especificar la misión de su uso en cada caso. Una posibilidad es completar los conceptos actuales con información que indique el uso de la sesión, pero todavía no se ha realizado un estudio completo para todos los casos de uso detectados.

- Falta de soporte para el paso de parámetros entre controladores cuando un controlador delega la respuesta en otro. En ocasiones es necesario pasar parámetros entre controladores, como por ejemplo, un controlador opera sobre un determinado objeto de datos y, posteriormente, delega la generación de la respuesta en un controlador que se encarga de presentar ese objeto determinado al usuario. Los patrones de programación utilizados se ven afectados por dos factores básicos. Primero, si el framework de aplicaciones web cuenta con una funcionalidad específica para ese tipo de casos. Segundo, si se produce o no una redirección.

- Falta de soporte para la especificación de la conversión de tipos. En los controladores que operan sobre el modelo y reciben datos de entrada, la conversión de tipos entre los datos recibidos y los atributos del objeto de datos a modificar es una de sus principales responsabilidades. Actualmente, la notación no permite almacenar información sobre estas conversiones de tipo en los mapeos definidos entre objetos de presentación y de datos.

- Falta de soporte para controladores que definan múltiples flujos de control válidos. En ocasiones, por ejemplo, se definen controladores que actúan de repartidores, delegando el control a otros controladores según los parámetros de la petición o el estado del sistema. Actualmente, la notación sólo da soporte a un flujo de control válido por controlador. La alternativa actual para soportar controladores con múltiples flujos de control es realizar un proceso previo que divida estos controladores en múltiples controladores, uno por cada flujo de control válido.

- Falta de soporte de autenticación y autorización. El uso de sistemas de autenticación y autorización es muy común en el desarrollo de aplicaciones web para definir áreas que sólo están accesibles a determinados usuarios o filtrar el tipo de operaciones que se pueden realizar sobre los objetos de datos. La notación actualmente no cuenta con soporte para especificar este tipo de áreas ni este tipo de filtrado.

7.4. Generación de una representación conceptual

En primer lugar, se analizan los resultados obtenidos del proceso de ingeniería inversa, con el fin de evaluar su grado de cobertura actual del caso de estudio. Posteriormente, con fines documentales, se relacionan algunas de las limitaciones actuales del proceso.

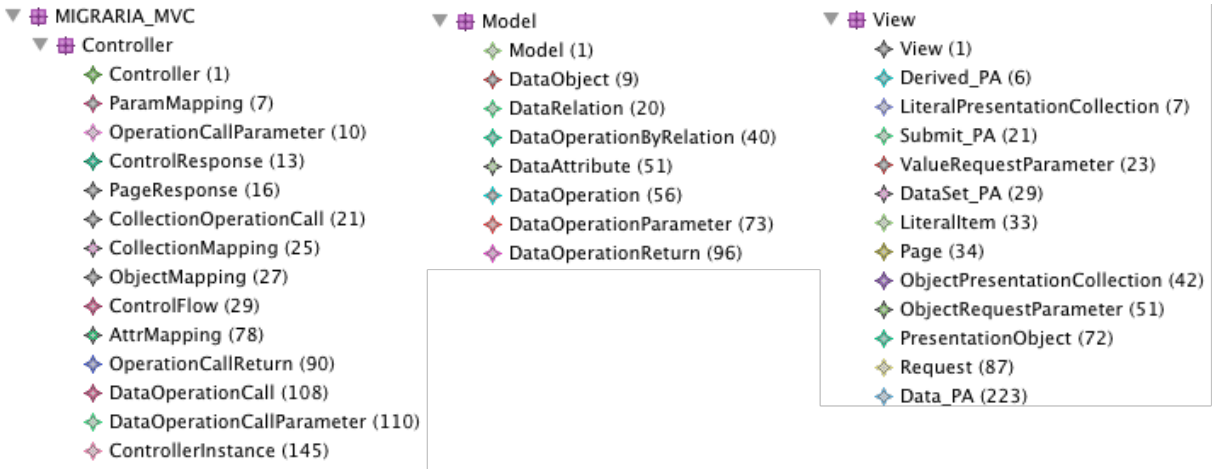


Figura 7.16: Número de instancias del modelo MIGRARIA MVC generado

7.4.1. Modelo MIGRARIA MVC

En esta sección se resumen y razonan los resultados obtenidos del proceso de ingeniería inversa en función de la descripción de las características de los elementos principales de los componentes del patrón MVC, que se realizó en el apartado 7.2.5. Estos datos reflejan los casos de generación perfectos o casi perfectos[12] en los que ha sido posible extraer toda la información necesaria para la especificación de un determinado elemento mediante la notación MIGRARIA MVC. Los casos de generación parcial o de no generación más significativos se mostrarán con mayor detalle para su razonamiento y discusión. La figura 7.16 muestra la cantidad total de instancias generadas para cada uno de los conceptos definidos en la notación MIGRARIA MVC que han sido utilizadas en la especificación del sistema heredado del caso de estudio. En total el proceso de ingeniería inversa definido genera 1655 instancias, siendo los conceptos más comunes, primero, los atributos de presentación simples (*Data_PA*) y después todos los relacionados con las llamadas a operaciones de datos (*ControllerInstance*, *DataOperationCallParameter*, *DataOperationCall*, *DataOperationCallReturn*). Con el fin de simplificar la exposición de la evaluación del proceso de ingeniería inversa, sólo se van a analizar en detalle la generación de las 697 instancias de los conceptos principales de la notación.

La Figura 7.17 resume los resultados obtenidos del proceso de ingeniería inversa en total (a la izquierda) y para cada uno de los componentes del patrón MVC (a la derecha). Como se puede apreciar, del total de 697 instancias principales a extraer, el proceso es capaz de extraer correctamente 650, dando lugar a un grado de cobertura de más del 93 %. Analizando los datos componente por componente, se puede apreciar que el componente Vista se genera completamente, mientras que en el caso de los componentes Modelo y Controlador no se produce una extracción total pero sí una extracción por encima del 90 % en ambos casos. En los siguientes subapartados, se exponen con detalle los resultados obtenidos para cada uno de los componentes, indicando las razones de los mismos.

[12] Carecen de alguna propiedad no relevante

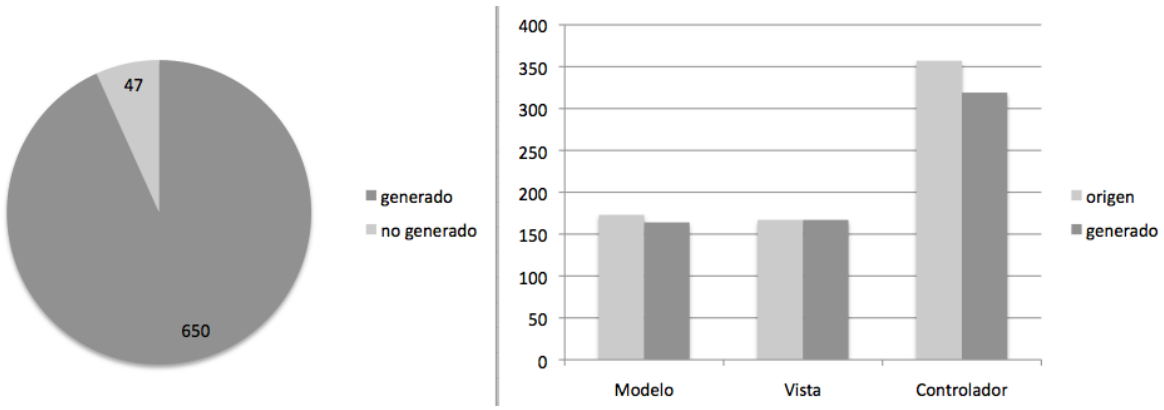


Figura 7.17: Resumen resultados proceso ingeniería inversa

Objeto de datos	Atributos	Relaciones	Operaciones
Coauthor	5	1	8*
Conference	9	1	7*
Group	3	0	6
Paper	5	7	23*
Review	13	2	11*
Subject	2	0	6
SummaryReviews	0*	0*	0
Track	3	1	8*
User	8	8	27*
Total	48	20	96

Tabla 7.5: Datos de los objetos de datos generados

7.4.1.1. Modelo

En el caso del componente Modelo, la Tabla 7.5 muestra la cantidad de elementos generados referentes a las características principales de los objetos de datos del sistema heredado. Se han destacado con un asterisco los valores que han cambiado con respecto a los tomados directamente del código fuente.

Como conclusión principal, se puede indicar que se genera el 100 % de los elementos detectables. La Figura 7.18 muestra un gráfico comparativo resumen de los datos obtenidos del código fuente y los obtenidos del modelo generado mediante el proceso de ingeniería inversa definido. El grado de cobertura del componente Modelo en el caso de estudio es, sin embargo, de aproximadamente del 95 %.

Como se puede observar, la gran mayoría de los cambios corresponden a un aumento del número de operaciones con respecto al valor tomado del código. Este aumento se debe a que el proceso de transformación genera dos operaciones (*get/set*) por cada relación en la que participe el objeto de datos como maestro de la misma. Esta situación se da, por ejemplo, en el objeto de datos *Conference* que ve aumentado en dos su número de operaciones, debido a la relación que mantiene con *Subjects*. Esta regla puede dar lugar a la generación de un gran número de operaciones adicionales en el caso de objetos de datos con un gran número de relaciones, como sucede por ejemplo en el caso del objeto de datos *Paper* para el que se generan 14 operaciones adicionales.

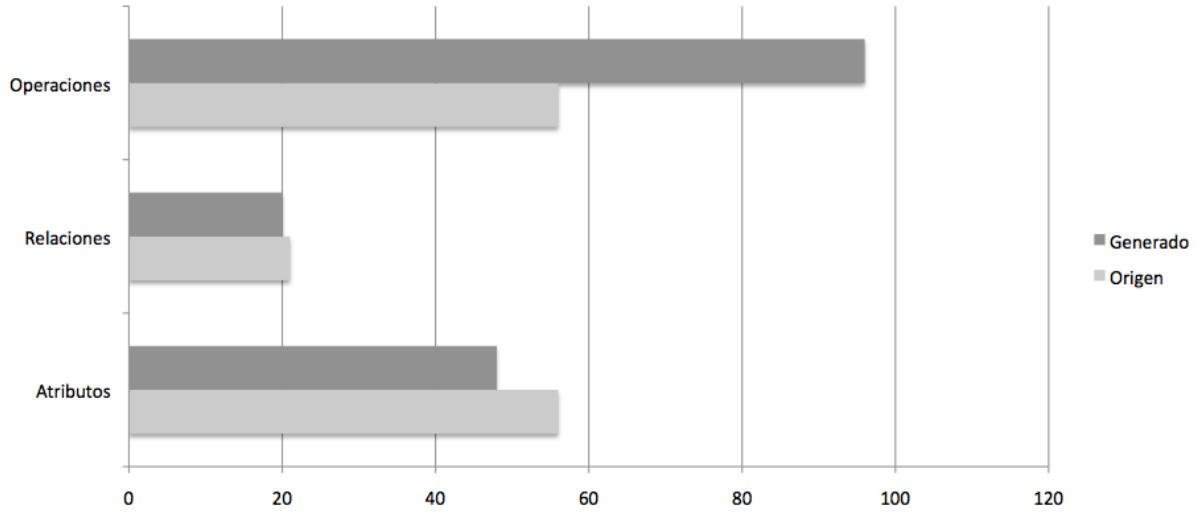


Figura 7.18: Gráfico comparativo Modelo origen y generado

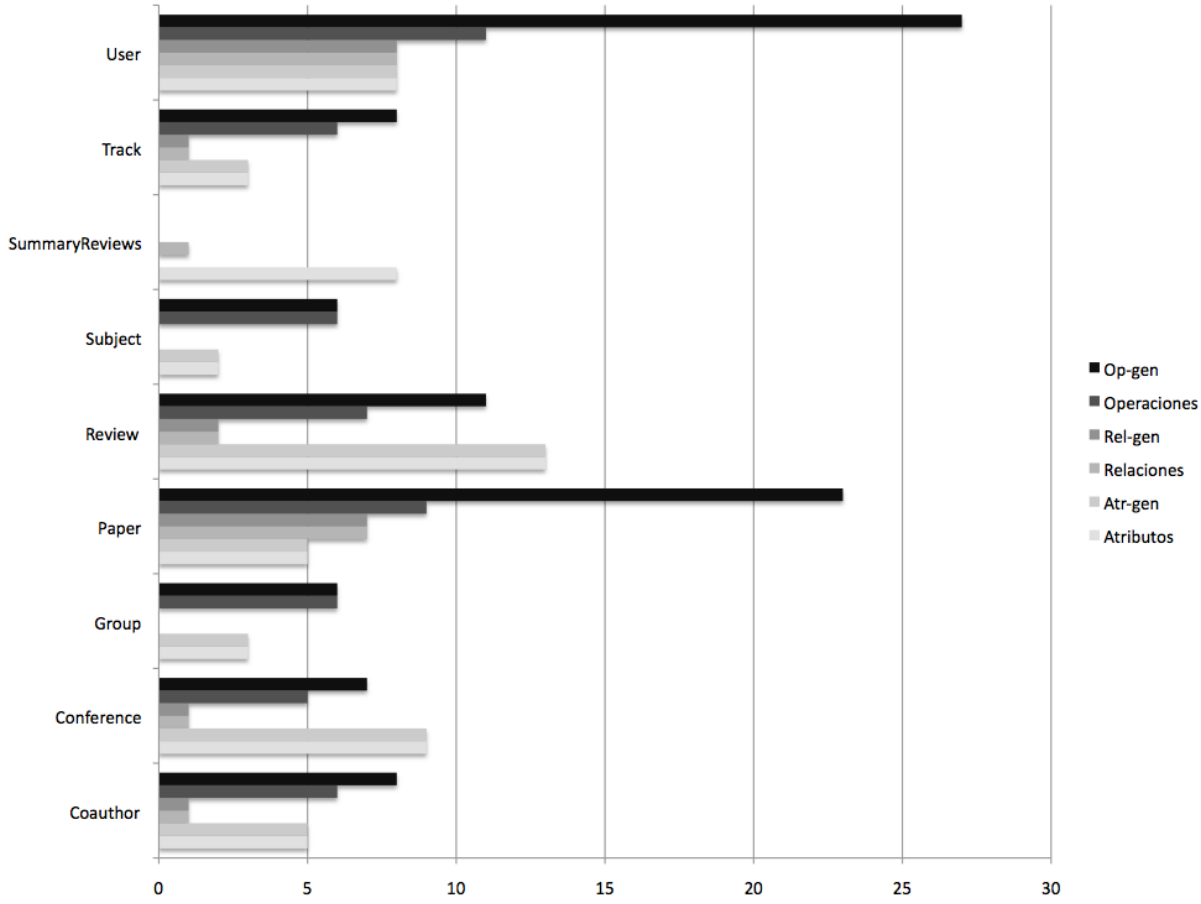


Figura 7.19: Gráfico comparativo Modelo origen y generado (detalle)

Finalmente, el objeto de datos *SummaryReviews* presenta un caso diferente; ya que, se trata de un objeto de datos que no tiene correspondencia con ninguna tabla del esquema de base de datos y, por lo tanto, no se especifica un mapeo objeto-relacional. Y tampoco contiene un objeto DAO que defina explícitamente operaciones sobre ese tipo de datos. Esta clase se ha definido para almacenar los resultados de una consulta que agrega valores de otros datos. En concreto, cada objeto de este tipo resume los valores de las revisiones realizadas a cada objeto *Paper* y permite generar el informe que revisarán los miembros del comité de programa una vez termine el período de revisiones. Este tipo de objetos de datos se representa en un modelo MIGRARIA MVC mediante instancias de *DerivedDataObject*. La extracción de este tipo de elementos no está soportada actualmente por el proceso.

La Figura 7.19 presenta la comparación para cada característica detallada por objeto de datos. En esta se puede apreciar, por un lado, que el objeto *SummaryReviews* es el que introduce las diferencias principales entre ambos modelos y, por otro lado, que el incremento en el número de operaciones se reparte entre los objetos de datos en función de su número de relaciones.

7.4.1.2. Vista

En el caso del componente Vista, la Tabla 7.6 muestra la cantidad de elementos generados referentes a las características principales de las JSPs del sistema heredado. En este caso, el identificador *forms* se refiere a objetos de presentación de entrada, el identificador *submit* se refiere a instancias de *ObjectRequestParameter*, el identificador *link* se refiere a peticiones generadas a partir de una etiqueta *html:link*, PO indica objeto de presentación y, finalmente, PC indica colección de presentación.

En conclusión, el proceso de ingeniería inversa es capaz de obtener el 100 % de los elementos detectables utilizados en el caso de estudio, según las reglas de negocio establecidas para la creación de *JSPs*. La Figura 7.20 ilustra estos resultados para cada una de las características contempladas en los elementos de las vistas. En este caso, sólo se muestran los datos resumen, ya que no tiene sentido presentar una relación detallada de cada una de las páginas al no haber diferencia entre la versión origen y la generada. El grado de cobertura del componente Vista en el caso de estudio se considera, por tanto, del 100 %.

7.4.1.3. Controlador

Finalmente, en el caso del componente Controlador, la Tabla 7.7 muestra la cantidad de elementos generados referentes a las características principales de los *Actions* del sistema heredado. Al igual que se realizó en el caso del Modelo, se marcan con un asterisco aquellos valores que no coinciden con lo esperado. En este caso, sólo aparecen marcados algunos valores en la columna *Action*, indicando que la generación de este elemento no está completa. En concreto, aunque se generan los elementos *ControlReponse* asociados, no se puede obtener el valor de su referencia *controlflow* que haría referencia al *ControlFlow* del *Action* sobre el que se delega la respuesta. La razón se debe a que actualmente el proceso no es capaz de procesar todas las *Action* definidas en el caso de estudio. Por lo tanto, no se

Java Server Page	forms	submit	link	POs	PCs
assignedPapersToReviewerList	0	0	7	1	4
assignPCmember	1	8	0	1	5
changeReviewer	1	1	0	1	2
conferencelist	0	0	0	0	1
conferencestatus	1	1	0	0	2
confirmUserRegistration	1	1	0	0	0
createconference	1	1	0	0	0
createPCmember	1	1	0	0	0
createSubjectsAndTracks	2	2	0	1	3
editconference	1	1	0	1	1
editPCmember	1	1	2	0	1
editSubjectsAndTracks	4	4	2	1	3
login	1	1	1	0	0
menu	0	0	16	0	0
navigation	0	0	8	0	0
paperCreate	1	1	0	0	2
paperDetails	0	0	0	1	3
paperEdit	2	3	2	0	3
paperList	0	0	3	1	1
paperListPCchair	1	1	1	0	3
paperListPCmember	1	0	2	0	3
papersAcceptRejectList	1	1	11	0	2
preferencesTrackSubjects	1	1	0	0	2
registerAuthor	1	1	0	0	0
reviewCreateEdit	1	2	0	1	6
Total	24	32	55	9	47

Tabla 7.6: Datos de las páginas generadas

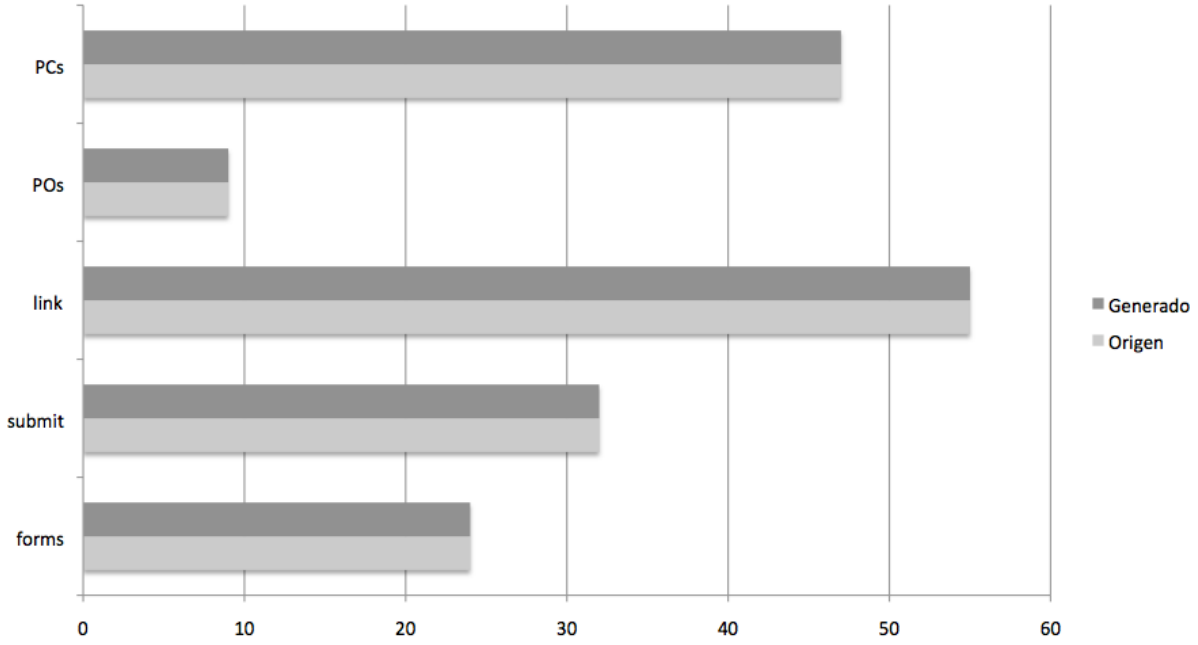


Figura 7.20: Gráfico comparativo Vista origen y generada

generan los *ControlFlow* a los que debería apuntar esta referencia y los elementos *ControlResponse* afectados se generan de manera incompleta.

Como ya se ha comentado, sólo se consideran los *Action* que son soportados por el proceso de ingeniería inversa. En concreto, se muestran 27 de los 49 *Actions* que tiene el sistema original. La generación de los 22 *Actions* restantes no cuenta todavía con soporte. Estos 22 *Actions* pertenecen a uno de estos tres casos:

1. Se han definido a partir de las clases *MappingDispatchAction* o *LookupDispatchAction* que permiten encapsular múltiples flujos de control, dando respuesta a diferentes peticiones. El proceso actual sólo tiene soporte para *Actions* que definen un sólo flujo de control principal. Por lo tanto, sólo se consideran aquellas que heredan directamente de la clase *Action*.

2. Utilizan como objeto de presentación de entrada un *DynaActionForm* en vez de un *ActionForm*. Este tipo de definición de formulario se realiza en el fichero de configuración y no mediante la definición de una clase que hereda de *ActionForm*. Por ahora, sólo se da soporte al caso *ActionForm*.

3. Reciben parámetros generados por otro *Action* que delega en ellos la generación de la respuesta al cliente. Actualmente, no se cuenta con soporte para este tipo de casos.

La Figura 7.21 muestra un gráfico comparativo resumen de los datos obtenidos del código fuente y los obtenidos del modelo generado mediante el proceso de ingeniería inversa definido. Como se puede observar, para el conjunto de *Actions* soportado, se genera correctamente prácticamente la totalidad de los elementos fundamentales de un *ControlFlow*, resultando en un grado de cobertura de aproximadamente el 100 %. La única excepción se da en los casos de *ControlFlow* con respuesta delegada en otro *ControlFlow* que no se ha generado debido a que el *Action* original se encuentra dentro de uno de los tres casos comentados anteriormente. Nótese también que el proceso de ingeniería inversa sólo extrae el flujo normal de ejecución; por lo tanto, para cada ControlFlow sólo genera el elemento asociado a la respuesta normal del controlador, ignorando respuestas de error o validación. Esa es la razón principal que hace que el número de elementos origen y generados no coincidan para las columnas *Action* y *JSP* en la Figura 7.21.

7.4.2. Limitaciones

En este apartado se relacionan las limitaciones más importantes de la versión actual del proceso de ingeniería inversa definido para la generación de un modelo MIGRARIA MVC a partir de una aplicación web heredada.

7.4.2.1. Modelo

En la generación del componente Modelo, la propuesta actual presenta las siguientes limitaciones:

Action	JSP	Action	Vars	DOpC	COpC	Maps
AssignedPapersToReviewerListAction	1	0	5	4	3	3
CheckFirstLoginAction	0	1*	1	3	0	0
ConfirmUserRegistration	0	1*	3	3	0	1
CreateConferenceAction	0	1	1	1	0	1
CreateSubjectsAction	0	1	4	4	1	1
CreateSubjectsAndTracks	1	0	5	4	1	4
CreateTracksAction	0	1	3	3	0	1
DeleteSubjectAction	0	1	4	4	1	1
DeleteTrackAction	0	1	3	3	0	1
EditConferenceAction	0	1*	2	2	0	1
EditSubjectAction	0	1	1	1	0	1
EditTrackAction	0	1	3	3	0	1
ListConferenceAction	1	0	1	1	0	1
ModifyAndDeleteSubjectsAndTracks	1	0	5	4	0	4
PaperCreateAction	0	1	5	7	0	1
PaperDeleteAction	0	1	2	2	0	1
PaperDetailsAction	1	0	4	3	0	4
PaperEditAction	0	1	5	7	0	1
PaperListAction	1	0	2	2	0	2
PaperListPCchairAction	1	0	8	8	4	2
PaperPopulateCreateAction	1	0	2	2	0	2
PaperPopulateEditAction	1	0	9	6	3	5
PapersAcceptRejectListAction	1	0	3	3	0	3
PopulateConfirmUserRegistration	1	0	1	2	0	1
PrefsTrackSubjectsAction	1	0	14	5	6	1
SelectSubjectToModify	1	0	7	5	0	6
SelectTrackToModify	1	0	7	6	0	7
SubmitPrefsTrackSubjectsAction	1	0	7	5	2	1
Total	14	11	117	103	19	58

Tabla 7.7: Datos de los controladores generados

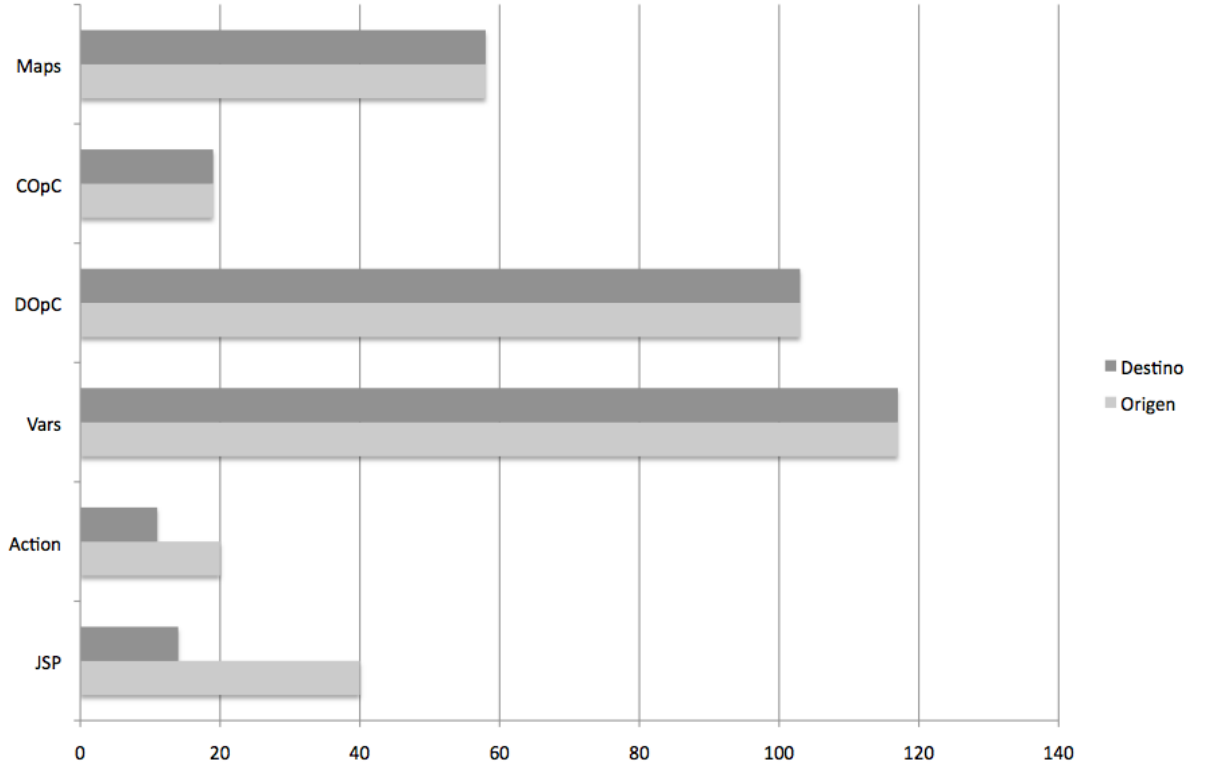


Figura 7.21: Gráfico comparativo Controlador origen y generado

1. No se han definido las reglas para la generación de los siguientes tipos de elementos: *DataEnum*, *DataEnumLiteral*, *DerivedDataObject*, *DerivedDataAttribute*.

2. La referencia *DataRelation.inverse* únicamente se establece cuando la relación es *manytomany* y se ha indicado explícitamente el atributo *inverse* en el archivo de mapeo de *Hibernate*.

3. Los valores de los atributos *DataRelation.maxcard* y *DataRelation.mincard* no se obtienen para ninguna de las relaciones porque en el archivo de mapeo objeto-relacional no se especifica de forma explícita.

4. Los valores del atributo *DataOperation.type* se obtienen correctamente cuando el identificador del método es igual al del *DataOperationType*. Actualmente, la regla sólo resuelve el tipo de una operación de datos mediante una aproximación sintáctica. Debido a que generalmente los métodos de tipo *GetFilter* no contienen en su identificador la cadena *getFilter* se ha creado una excepción para los métodos cuyo nombre comience por *getAll* y tengan más de un parámetro para considerarlo de tipo *GetFilter*. En el caso de los *DataOperationByRelation* debe revisarse el tipo asignado, siendo para el caso de las operaciones de tipo *get GetFilter* y para las de tipo *set Relate*.

5. Las referencias *DataOperationReturn.object* y *DataOperationParameter.object* sólo se establecen cuando la clase del objeto referenciado se corresponde con algún *DataObject*. Si no, esta relación no se establece.

6. Los valores de los atributos *DataOperationReturn.collection* y *DataOperationParameter.collection* siempre se establecen al valor *List* cuando el tipo de datos sea un tipo parametrizado de Java.

7.4.2.2. Vista

En la generación del componente Vista, la propuesta actual presenta las siguientes limitaciones:

1. Las reglas de transformación actuales sólo soportan el uso de las librerías de etiquetas JSP definidas por Struts 1.

2. La etiqueta *html:options* presenta un soporte parcial debido a la gran flexibilidad de uso de esta etiqueta. Se han identificado 8 casos diferentes de uso de esta etiqueta, que se relacionan a continuación. El soporte de las reglas actuales se limita a los dos primeros casos.

   a) se utilizan los atributos *collection*, *property* y *labelProperty*;

   b) se utiliza el atributo *name*;

   c) se utiliza el atributo *property*;

   d) se utilizan los atributos *name* y *property*;

   e) se utiliza la etiqueta *labelName* junto con el caso b, c o d;

   f) se utiliza la etiqueta *labelProperty* junto con el caso b, c o d;

   g) se utilizan las etiquetas *labelName* y *labelProperty* junto con el caso b, c o d.

7.4.2.3. Controlador

En la generación del componente Controlador, la propuesta actual presenta las siguientes limitaciones:

1. Las reglas de transformación actuales sólo reconocen los patrones definidos por las reglas de negocio (convenios) establecidos para el desarrollo de este caso de estudio.

2. Las reglas de transformación actuales no soportan el uso de *MappingDispatchAction* ni de *LookupDispatchAction*.

3. Las reglas de transformación actuales no soportan el uso de *ForwardAction*.

4. Las reglas de transformación actuales no soportan el uso de formularios dinámicos (*DynaActionForm*).

5. Las reglas de transformación actuales no soportan el paso de parámetros entre controladores.

6. Las reglas de transformación actuales no soportan el uso de una llamada a método como segundo parámetro del método *HttpRequest.setAttribute*.

7. Las reglas de transformación actuales no soportan el uso de constantes.

8. Todos los *ActionMappings* del fichero de configuración deben contener al menos un *forward* definido que representa el flujo normal de ejecución.

7.5. Resumen

Este capítulo presenta el caso de estudio principal que se ha desarrollado durante este trabajo de tesis para poder evaluar el estado de la propuesta y guiar su evolución. Después de valorar la posibilidad de utilizar un caso de estudio real, se decidió utilizar un caso de estudio de laboratorio fundamentalmente por la falta de un caso de estudio suficientemente representativo y adaptado a las necesidades del proyecto. Un caso de estudio de laboratorio nos permite, por ejemplo, conocer el sistema con mucho más detalle, pudiendo jugar el rol del desarrollador que participa en el proceso de modernización. Y también facilita el seguimiento de un conjunto de reglas de negocio (o convenios de programación) que pueda ser adaptado a la evolución de la propuesta. Adicionalmente, nos da la libertad de ponerle una licencia que permita su publicación y distribución para que otros grupos de investigación puedan utilizarlo.

El sistema desarrollado está basado en el caso de estudio *Conference Review System*[13] propuesto en el *First International Workshop on Web-Oriented Software Technology*[14] como caso de referencia para la evaluación de las diferentes propuesta IWDM presentadas. Se ha implementado en base a una pila tecnológica concreta cuya pieza fundamental es el framework de aplicaciones web Struts 1. En el capítulo se detalla también los convenios de programación seguidos, la planificación de desarrollo y la complejidad del sistema en base a los conceptos representados por el metamodelo MIGRARIA MVC.

En cuanto a la validación de la expresividad de la notación definida, se propone el modelado de un subsistema completo del sistema CRS, en concreto el subsistema de gestión del envío de artículos a una conferencia por parte de un autor. De esta manera, se ilustra el modelado de los conceptos del Modelo, de la Vista y del Controlador de este subsistema. Además, se especifican las principales limitaciones actuales de la notación.

En lo referente a la evaluación del proceso de generación del modelo MIGRARIA MVC, se exponen los resultados obtenidos de aplicar la cadena de transformación de modelos definida a todo el sistema CRS, indicando también las limitaciones actuales de dicho proceso. Los resultados obtenidos permiten afirmar que el proceso funciona correctamente dentro del conjunto de patrones que es capaz de reconocer. La definición concreta de su alcance y de sus limitaciones permite simplificar la valoración de su aplicación en otros casos de estudio; ya que, básicamente, consiste en identificar si la aplicación web

[13] *http://users.dsic.upv.es/~west/iwwost01/IWWOSTContent.htm#case_study*
[14] *http://users.dsic.upv.es/~west/iwwost01/*

heredada a procesar se encuentra dentro del alcance de la misma y, si se sale del mismo, qué limitaciones deben resolverse.

Capítulo 8

Estrategias de Generación RIA

8.1. Introducción

El objetivo principal del proceso de modernización definido por el proyecto MIGRARIA consiste en la generación de dos subsistemas diferentes. Por un lado, un cliente RIA que defina un nuevo modo de interacción adaptado a las tendencias tecnológicas del momento. Y, por otro lado, una capa de comunicación que permita establecer el nexo de unión entre el nuevo cliente RIA y el sistema de información heredado.

En este capítulo se exponen dos estrategias diferentes en la consecución de este objetivo. Primero, se presenta una estrategia para la generación de la especificación del sistema RIA final en una propuesta IWDM que cuente con soporte RIA. A partir de esta especificación, se define un proceso de enriquecimiento basado en la aplicación de patrones RIA mediante técnicas de composición de modelos. Posteriormente, se presenta una estrategia para la generación de la capa de conexión mediante la definición de una API[1] REST (Fielding 2000) a partir de la aplicación heredada.

Estos trabajos forman parte de las líneas de trabajo seguidas en la etapa de ingeniería directa del proceso marco definido por MIGRARIA. Su misión ha consistido fundamentalmente en evaluar posibles alternativas en la consecución del objetivo principal del proyecto y evaluar, por tanto, la viabilidad del propio proceso. Se presentan dentro de esta memoria con fines ilustrativos únicamente, de manera que se pueda obtener una visión del proceso de modernización en su totalidad. No se pretende, sin embargo, realizar una exposición detallada de los mismos.

Los cuatro apartados siguientes de este capítulo presentan los trabajos relacionados con la primera estrategia, en concreto se presenta: (1) la generación de un modelo de la aplicación heredada en WebML a partir del modelo MIGRARIA MVC; (2) la especificación de patrones de sincronización RIA en WebML; (3) la selección de patrones RIA en un escenario de modernización; y (4) la aplicación de los patrones RIA elegidos sobre el modelo WebML de la aplicación heredada mediante una propuesta orientada a

[1] Application Programming Interface

aspectos. Finalmente, por su parte, el último apartado se centra en la exposición del trabajo relacionado con la segunda estrategia: generación de una capa de conexión.

8.2. Proyección a una propuesta IWDM

La generación de los modelos conceptuales del sistema RIA a partir de los modelos conceptuales de la aplicación web heredada se puede realizar siguiendo diferentes estrategias. Básicamente, podemos dividir estas estrategias en dos grupos fundamentales según su punto de partida:

1. Las estrategias definidas como transformaciones de los modelos conceptuales propuestos en este trabajo.

2. Las estrategias que se basan en el uso de una propuesta de IWDM con capacidad para especificar características RIA.

Por un lado, el primer grupo supone, como requisito fundamental, disponer de un lenguaje de representación que permita la especificación de características propias de las RIAs. Sin embargo, el lenguaje definido en el ámbito de este trabajo no posee esa capacidad, porque ha sido diseñado con la misión de facilitar la extracción de información de una aplicación web heredada. Por otro lado, el segundo grupo ya cumple con el requisito fundamental comentado anteriormente y, por lo tanto, permite plantear la generación del nuevo sistema como un proceso de reestructuración de modelos dentro del mismo lenguaje. En este sentido, en el ámbito de este trabajo, se ha optado por definir un proceso de transformación de los modelos conceptuales obtenidos a una propuesta IWDM con soporte RIA para simplificar la generación del sistema final.

8.2.1. Proyección a WebML

Entre las principales propuestas IWDM con soporte RIA estudiadas previamente en el capítulo 2, se ha seleccionado WebML porque presenta un estado muy maduro y proporciona una herramienta de desarrollo de nivel profesional como es WebRatio[2]. Adicionalmente, esta propuesta está integrada también con RUX Tool[3] que permite dar soporte a sus extensiones RIA.

En los siguientes apartados, como se presentó en (Rodríguez-Echeverría *et al.* 2013a), se expone brevemente el proceso seguido para generar una especificación en WebML de la aplicación web heredada, partiendo de su representación en base al metamodelo MIGRARIA MVC. En primer lugar, se presenta el proceso seguido para el modelo de datos. Y, posteriormente, se presenta el proceso para el modelo de hipertexto.

[2]http://www.webratio.com/
[3]http://ruxtool.es/

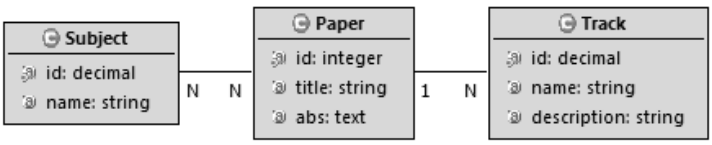


Figura 8.1: Modelo de Datos de WebML

8.2.1.1. Modelo de datos

Las reglas necesarias para transformar el modelo del componente Modelo de MIGRARIA MVC en el modelo de datos de WebML no presentan demasiada complejidad; ya que, el metamodelo del componente Modelo de MIGRARIA MVC puede considerarse definido, a efectos prácticos, como una extensión del modelo de datos de WebML. Básicamente, se puede considerar que la extensión realizada consiste en proporcionar los elementos necesarios para la especificación de operaciones sobre el modelo de datos.

En este sentido, las transformaciones definidas en ATL realizan una traducción directa de los objetos y atributos de datos, y de sus relaciones a sus versiones en WebML. La figura 8.1 presenta un extracto del modelo de datos de WebML generado para el caso de estudio (CRS). En concreto, aparecen representadas las tres entidades de datos resultantes de la transformación de un objeto de datos *Paper* y los objetos con los que está relacionado: *Subject* y *Tracks*.

8.2.1.2. Modelo de hipertexto

Con el objetivo de ilustrar el funcionamiento base del proceso y los resultados obtenidos, en este apartado se muestra la generación de las operaciones de creación y lectura de un nuevo objeto de datos.

La Figura 8.2 presenta los elementos generados para modelar la creación de un nuevo objeto de datos *Paper* (un nuevo envío de artículo) a partir del modelo MIGRARIA MVC de la Figura 7.3. El modelo resultado está compuesto fundamentalmente por los siguientes elementos:

- Una página, denominada *paperCreate*, como elemento contenedor principal, representando la página original de envío de artículos por parte de un autor.
- Una *Entry Unit*, denominada *paperForm*, representando el formulario HTML para la entrada de los datos referentes al envío.
- Dos *Selector Units*, denominadas *SelectorUnitSubjects* y *SelectorUnitTracks*, representando los controles de selección del formulario HTML.
- Una *Create Unit*, denominada *CreateUnitPaper*, representando la creación de un objeto de datos *Paper*.
- Una *Connect Unit*, denominada *ConnectUnitPaperSubjects*, representando la asociación entre el nuevo objeto de datos *Paper* y los objetos de datos *Subject* seleccionados.

- Y cuatro enlaces para conectar las unidades entre sí.

Considerando la Figura 7.12 del caso de estudio, en la que se mostraba la especificación de los controladores relacionados con la gestión de artículos, el controlador responsable de poblar los objetos que componen la vista de respuesta a la petición se transforma en las dos *Selector Unit* mostradas. Mientras que, por su parte, el controlador responsable de la atención de la petición de creación es modelado en base a las *Operation Unit* mostradas.

Una *Selector Unit* se deriva, por tanto, del siguiente patrón presente en el modelo MIGRARIA MVC: una colección de presentación de salida usada como un atributo de presentación de tipo *dataset* dentro de un objeto de presentación de entrada. En el ejemplo utilizado, la página *PaperCreate* contiene el objeto de presentación de entrada *paperForm* que presenta el atributo subjects de tipo *dataset subjects* que está relacionado con la colección de presentación *subjects*. Una vez identificada la *Selector Unit*, es necesario encontrar qué datos deben ser obtenidos por esta unidad. Esta información se puede extraer del mapeo definido en el controlador entre una colección determinada y la instancia que almacena los resultados obtenidos de la consulta en base de datos. Esta instancia contiene una referencia al objeto de datos del Modelo que define su tipo. Dado que existe una relación uno a uno entre objetos de datos y entidades, la entidad del *Selector Unit* es la representada por el objeto de datos correspondiente.

De la misma manera, un objeto de presentación de entrada es transformada en una *Entry Unit*. En el ejemplo, la unidad *EU_Paper* se genera a partir del objeto de presentación de entrada *paper* y se localiza dentro de la página generada anteriormente. Cada uno de los atributos de tipo *data* del objeto de presentación de entrada es transformado en un elemento *field* de la *Entry Unit*. Mientras que cada uno de los atributos de tipo *dataset* genera un elemento *selectionField* en la unidad. En el caso de que el atributo booleano *multiSelect* tenga un valor de verdadero en un atributo de tipo *dataset* el elemento a generar será un *multiSelectionField*.

Por su parte, las peticiones definidas sobre un objeto de presentación de entrada son transformadas en enlaces que conectan la *Entry Unit* con la correspondiente *Operation Unit*, mientras que el mapeo de parámetros asociado es generado del mapeo de atributos definido para el objeto de entrada en el controlador que maneja la petición. La Figura 8.2 presenta las propiedades de *EU_Paper* para mostrar sus atributos y las peticiones de transformación.

Adicionalmente, un *Transport Link* es generado entre una *Selector Unit* y un campo *(multi)selectionField* de una *Entry Unit*, cuando aparece el patrón *Selector Unit* comentado anteriormente en el controlador responsable de poblar la página que contiene el objeto de presentación de entrada. La Figura 8.2 presenta las propiedades del *Transport Link SelectorUnitSubjects-EntrynUnitPaper*, ilustrando cómo uno de los atributos del objeto de presentación (ítem de la colección) aparece marcado como *output* (*id*) y el otro como *label*.

Con respecto a las *Operation Unit,* su generación se produce a partir de la secuencia de llamadas a operación presente en un controlador del modelo MIGRARIA MVC. Una llamada a operación de

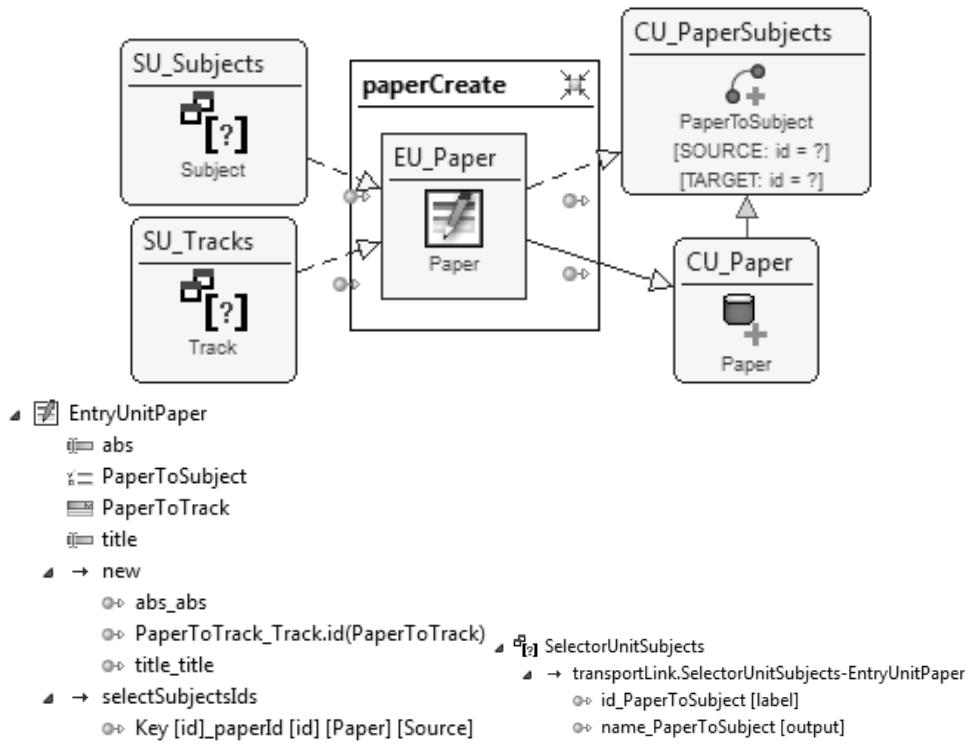


Figura 8.2: Operación de creación en WebML

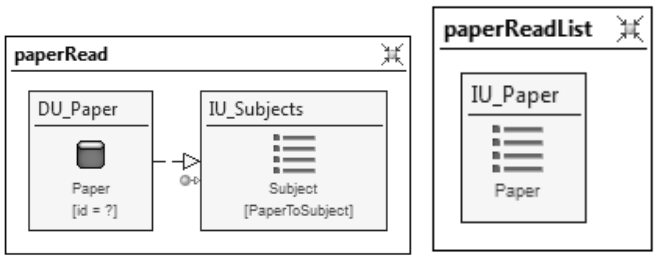


Figura 8.3: Operaciones de lectura en WebML

datos indica una referencia a una operación definida dentro de un objeto de datos concreto. El tipo de operación es usado para identificar la clase de *Operation Unit* a generar (*create*, *update* o *delete*). Y el objeto de datos indica además la entidad de la *Operation Unit*. En el ejemplo, la operación de datos *paperDAO.create* se transforma en la *Create Unit CU_Paper*. Y un enlace *OK* es generado para conectar esta operación con la siguiente. En este caso, se enlaza la *Connect Unit CU_PaperSubjects*. Una *Operation Unit Connect* se genera a partir de una llamada a operación que referencia a una operación de datos (OperationByRelation) que representa una relación n:m entre dos objetos de datos diferentes. En el ejemplo, la llamada a operación de datos *paperDAO.setSubjects* se convierte en la *Connect Unit CU_PaperSubjects*. El *Transport Link* relacionado surge de la instancia del controlador pasada como parámetro en la llamada a operación.

La Figura 8.3 presenta los elementos generados para modelar la presentación de los detalles de un objeto de datos *Paper* concreto (izquierda de la figura) y la lista de todos los objetos de datos de este tipo

(a la derecha de la figura) a partir del modelo MIGRARIA MVC de la Figura7.12. La estructura principal de este modelo se compone de los siguientes elementos:

- Dos elementos *Page*, denominadas *paperRead* y *paperReadList*, como contenedores principales, representando las páginas originales.
- Una *Data Unit*, denominada *DU_Paper*, indicando el objeto de datos *Paper* a presentar y sus atributos.
- Dos elementos *Index Unit*, denominadas *IU_Subjects* e *IU_Paper*, representando el conjunto de objetos de datos *Subject* asociados al objeto *paper*, por un lado, y la lista de objetos de datos *Paper*, por el otro.

En el modelo MIGRARIA MVC, una página de salida está relacionada estrechamente con un controlador responsable de poblar los objetos que la componen. En este sentido, estas páginas en su forma base sólo contendrán unidades de recuperación de datos pero no de entrada.

Una *Data Unit* se genera a partir de un objeto de presentación de salida y sus *display attributes* se derivan de los atributos de presentación que componen el objeto. Como siempre, para obtener la entidad de datos relacionada con la unidad es necesario identificar el objeto de datos en el modelo MIGRARIA MVC relacionado con el objeto de presentación mediante la localización del mapeo correspondiente en el controlador. En el ejemplo, la página contiene el objeto de presentación *paper* que se mapea al objeto de datos *Paper*, de manera que la *Data Unit* se define sobre la entidad *Paper*.

Una *Index Unit*, por su parte, se genera a partir de una colección de presentación de salida. En el primer caso, ilustrado por la Figura 8.3, esta colección se utiliza para poblar el atributo de tipo *dataset* (*subjects*) del objeto de presentación de salida (*paper*). El proceso para obtener la entidad de datos relacionada permanece inalterado pero, en este caso, se obtiene también la relación (*PaperToSubject*). Adicionalmente, un *Transport Link* es generado entre ambas unidades representando la relación entre el atributo de tipo *dataset* y la colección. En el segundo caso ilustrado, la colección permite presentar un conjunto de atributos del objeto de presentación que actúa como ítem de la colección.

Finalmente, simplemente dejar constancia de que se han definido las reglas de transformación necesarias para generar la especificación de las operaciones CRUD en WebML. Prácticamente, la propuesta da soporte al conjunto completo de las unidades del núcleo de WebML.

8.3. Definición de patrones RIA en WebML

El trabajo (Rodríguez-Echeverría *et al.* 2010) presenta una propuesta de definición de patrones de sincronización RIA en WebML. Estos patrones se definen genéricamente para poder ser reutilizados en el proceso de refinamiento de cualquier aplicación especificada mediante WebML. De manera que pueden

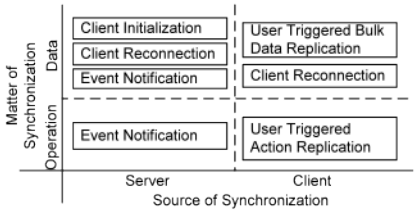


Figura 8.4: Escenarios de sincronización a partir de las dimensiones origen y sujeto

ser usados para refinar el modelo WebML generado mediante la propuesta presentada en el apartado anterior.

Estos patrones pueden ser almacenados en repositorios de modelos para facilitar su localización e instanciación en futuros procesos de modernización, con el fin de reducir los costes del mismo.

8.3.1. Patrones de Sincronización

Una de las principales característica de las RIA es la distribución de los datos y la lógica entre el servidor y el cliente minimizando las peticiones a realizar. Esta distribución conlleva la necesidad de definir mecanismos de sincronización entre cliente y servidor para asegurar la consistencia de datos entre ambas partes. Esta definición de procesos de sincronización suele definirse de forma explícita en los modelos de la aplicación, de manera que aparecen esparcidos y mezclados con la especificación de la funcionalidad principal de la aplicación.

Dado este escenario, en el contexto de este trabajo, se han clasificado los comportamientos de sincronización típicos de una RIA en función de dos dimensiones: la primera, el origen de la sincronización (cliente o servidor) y, la segunda, el sujeto de la sincronización (operación o datos). La figura 8.4 presenta esta clasificación.

Nótese que varios escenarios de sincronización pueden ser clasificados dentro de la misma categoría. Por ejemplo, cuando un cliente es inicializado o reconectado (modo desconectado), el servidor debe actualizar los datos de ese cliente para que pueda comenzar su ejecución. O, por ejemplo, cuando se producen cambios en los datos del servidor, todas las copias de los clientes deben ser actualizadas para mantener la consistencia. Este tipo de sincronizaciones en las RIA se implementan mediante notificaciones de eventos.

La notificación de eventos puede aparecer también en escenarios en los que se ha realizado una operación en el servidor que debe ser replicada en los clientes conectados. En este caso, el evento almacena los datos correspondientes a la operación ejecutada.

Los escenarios de sincronización originados desde el cliente son también muy comunes en las RIAs, especialmente en las que definen entornos colaborativos. Por ejemplo, si planteamos los casos anteriores desde la parte cliente, cuando un cliente se reconecta (modo desconectado), el cliente debe enviar al servidor todos los datos creados o modificados para sincronizarse con el servidor. También en aplicacio-

nes colaborativas, la necesidad de replicar en servidor una acción realizada en uno de los clientes es un escenario muy común.

8.3.2. Modelado de los Patrones de Sincronización

Una vez clasificados los escenarios de sincronización en RIAs, se pueden definir los patrones que modelan su comportamiento. En concreto, se ha definido un patrón por cada escenario de sincronización identificado. Este apartado ilustra simplemente dos de esos patrones:

1. Un patrón que modele la notificación de evento derivada de la necesidad de que el servidor envíe datos al cliente (*Event Notification* en la Figura 8.4).

2. Un patrón que modele el escenario de sincronización originado por un cliente que quiere mandar un conjunto de datos al servidor (*User Triggered Bulk Data Replication* en la Figura 8.4).

Teniendo en cuenta el origen de la sincronización, los patrones se pueden clasificar como sigue:

1. Sincronización iniciada en el servidor. Los patrones dentro de esta categoría tienen en común que el servidor es el responsable de iniciar el proceso de sincronización para mantener la consistencia de los datos con el cliente. Los patrones identificados dentro de esta categoría son:

   a) Sincronización en la inicialización del cliente. Este patrón recoge la transferencia de datos que se produce en la inicialización de la aplicación en el cliente. Este patrón debe recoger la creación de los datos locales para que la parte cliente pueda comenzar su ejecución con normalidad. Este patrón puede permitir modos de trabajo desconectados.

   b) Sincronización en la reconexión del cliente. Este patrón recoge la transferencia de datos necesaria para la sincronización de un cliente que ha estado trabajando en modo desconectado. Sigue un proceso similar al patrón anterior, pero no suele incluir la creación de nuevos tipos de datos locales.

   c) Notificación de evento de datos. Invirtiendo el flujo normal de comunicación, un cliente puede ser notificado por el servidor de la modificación de parte de los datos que tiene replicados localmente. En este caso, el proceso de actualización de los datos podría ser similar al usado en el patrón anterior. La principal diferencia estriba en el uso de un método de notificación de eventos. El modelado de este patrón en WebML se muestra en la Figura 8.5. Este patrón está basado en las entidades *Send Event* y *Receive Event* (Carughi *et al.* 2007). Este patrón también supone la creación del evento como una entidad de datos dentro del modelo de datos de la aplicación. Con el objetivo de poder aplicar este patrón en diferentes escenarios, se han usado *Linkable Elements* como entidades parametrizables. Estas unidades son instanciadas en el modelo de composición, explicado en el próximo apartado, de manera que se mezclan con unidades concretas del modelo de la aplicación.

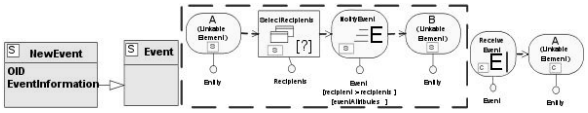


Figura 8.5: Patrón de Notificación de Evento de Datos

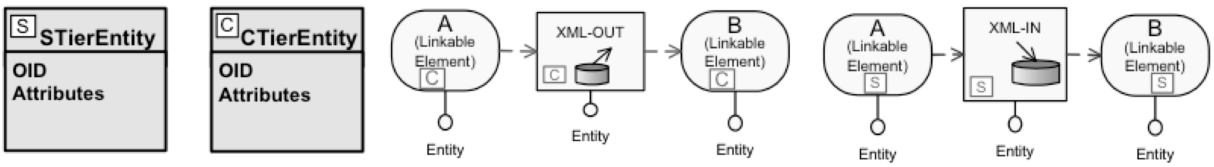


Figura 8.6: Replicación de una colección de datos iniciada por el cliente

2. Sincronización iniciada en el cliente. Los patrones dentro de esta categoría tienen en común que el cliente es el responsable de iniciar el proceso de sincronización para mantener la consistencia de los datos con el servidor. Los patrones identificados dentro de esta categoría son:

   a) Replicación instantánea. Este patrón recoge el escenario en el que los clientes trabajan con replicaciones de los datos del servidor y es necesario mantener en todo momento la consistencia, por lo tanto, cada modificación en cliente es notificada instantáneamente al servidor. Este patrón está muy relacionado con el Patrón de Caché Distribuida[4], que es muy común en sistemas colaborativos. Este patrón no es aplicable en modos de trabajo desconectado.

   b) Replicación de una colección de datos iniciada por el cliente (*User Triggered Bulk Data Replication*). Este patrón recoge el escenario en el que una colección de datos generada o cambiada en el cliente debe ser enviada al servidor para su sincronización. Es muy común de modos de trabajo desconectados, cuando el cliente reconecta necesita enviar al servidor todos los datos modificados. En estos escenarios es muy probable la aparición de conflictos entre los datos; por lo tanto, el servidor debe ser responsable de aplicar alguna política de resolución de conflictos para evitar inconsistencias de datos. Este patrón, no obstante, puede aparecer también en modos de trabajo conectados en los que los clientes envían periódicamente los datos creados o modificados. El modelado de este patrón en WebML se muestra en la Figura 8.6. Como se puede apreciar, el patrón es especificado en base a las entidades *XMLin* y *XMLout* para gestionar la serialización y envío de los mismos. Como se sugiere en las guías definidas en (Bozzon *et al.* 2006), esta sincronización normalmente requiere la duplicación de la entidad de datos en cliente y en servidor con distintos nombres.

8.4. Selección de patrones RIA

Este apartado presenta brevemente la propuesta definida en (Conejero *et al.* 2013). Esta propuesta presenta un proceso para asesorar en la selección de patrones RIA a la hora de satisfacer requisitos de

[4] http://www.javaranch.com/journal/2003/11/DistributedCachePattern_V2.html

RIA features / Requirements	Data design				Business logic design						Presentation design				Communication design				
					Tier			Computation time							Data granularity		Business logic communication		
	Volatile Client	Persistent Client	Volatile Server	Persistent Server	Client Side	Server Side	Mixed	Initially	Incrementally	With refresh	Widget/Look&Feel	Interaction Behaviour	Temporal Behaviour	Partial Refreshment	Tuple	DataSet	Client to Server	Server to Client	Inside each tier
	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18	F19

Tabla 8.1: Matriz de trazabilidad de requisitos

funcionalidad RIA que se quieren añadir al sistema modernizado. El proceso se basa en construir dos matrices de trazabilidad que permiten relacionar los requisitos con las características RIA y los patrones con las características RIA, respectivamente. Mientras que la primera debe desarrollarse en cada proceso de modernización, la segunda permanece constante, permitiendo su reutilización y disminuyendo los costes de modernización asociados. Esta segunda matriz ha sido desarrollada en el seno de este trabajo y contiene patrones extraídos de (Brambilla *et al.* 2009; Rodríguez-Echeverría *et al.* 2010; Vora 2009; Yahoo!-Patterns 2011). A partir de estas matrices se propone un algoritmo de operación y unas métricas que permiten sugerir los patrones RIA que se deben aplicar.

8.4.1. Mapeo entre requisitos y características RIA

El analista tiene la responsabilidad de decidir qué características RIA deben ser aplicadas para cada nuevo requisito. Por lo tanto, es necesario que posea un completo conocimiento de las características RIA utilizadas. Este apartado muestra cómo esta decisión puede ser soportada por una clase especial de matriz de trazabilidad que relaciona los requisitos de modernización con las características RIA. Esta matriz se denomina matriz de trazabilidad de requisitos (*requirements traceability matrix*, *rtm*) y su estructura se muestra en la Tabla 8.1. Las filas de esta matriz representan los requisitos, mientras que las columnas contienen las características RIA utilizadas. Para un requisito dado, las marcas indican las características RIA que deben ser consideradas en el desarrollo para satisfacerlo.

8.4.2. Mapeo entre patrones y características RIA

De forma similar a lo realizado en el caso de los requisitos, se puede obtener una matriz que muestre las características RIA cubiertas por cada patrón. Esta nueva matriz de trazabilidad recibe el nombre de matriz de trazabilidad de patrones (*patterns traceability matrix, ptm*), presentada en la tabla 8.2. En ésta las filas representan los patrones RIA caracterizados. Una celda con “o” indica que el patrón puede ser también aplicado en el caso de seleccionar opcionalmente la característica RIA de esa columna. Como ejemplo, se expone cómo se rellenan las filas para los patrones Notificación de Evento de Datos (*Data*

	Data design				Business logic design						Presentation design				Communication design				
					Tier			Computation time							Data granularity		Business logic communication		
RIA features / Patterns	Volatile Client	Persistent Client	Volatile Server	Persistent Server	Client Side	Server Side	Mixed	Initially	Incrementally	With refresh	Widget/Look&Feel	Interaction Behaviour	Temporal Behaviour	Partial Refreshment	Tuple	DataSet	Client to Server	Server to Client	Inside each tier
ClientReconnection (P1)		x		x			x	x	o							x		x	
Data EventNotification(P2)				x			x								x	o		x	
User Triggered Action Replication (P3)	o	x		x			x	x							x		x		
User Triggered Bulk DataReplication (P4)	o	x		x			x	x								x	x		
Partial Page Refresh (P5)	x				x					x	x	x		x					
Dynamic tooltips on page data (P6)	x			x	x			x						x		x		x	
Carousel (P7)	x			x	x			x			x	x		x		x		x	
Module Tabs (P8)	x				x			x			x	x		x					
Left Navigation Bar (P9)	x				x			x			x	x		x					
	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18	F19

Tabla 8.2: Matriz de trazabilidad de patrones

Event Notification) y Replicación de una Colección de Datos iniciada por el Cliente (*User Triggered Bulk Data Replication*) explicados previamente.

El patrón Notificación de Evento de Datos se caracteriza por la necesidad de crear una entidad en el servidor para la representación del evento, por lo que, la característica F4 debe ser activada. Este patrón presenta dividido su comportamiento entre el cliente y el servidor, por lo que, se debe activar también la característica F7. Con respecto a la comunicación, este patrón sugiere la comunicación con direccionalidad servidor-cliente, por lo que, la característica F18 debe ser activada. Y finalmente, referente a la granularidad de los datos, aunque normalmente se envía un sólo objeto de datos (F15) es posible que se pueda enviar una colección de datos (F16).

El patrón Replicación de una Colección de Datos iniciada por el Cliente (*User Triggered Bulk Data Replication*), por su parte, presenta funcionalidad tanto en servidor como en cliente (F7) así como datos (F2 y F4, opcionalmente F1). Además, los datos relacionados suelen enviarse en el momento de la inicialización del cliente (F8). Finalmente, el cliente mandará un conjunto de datos (F16) para su sincronización con los del servidor (F17).

El mismo razonamiento puede realizarse para el resto de los patrones considerados. Es importante volver a remarcar que esta matriz sólo debe ser creada una vez, contemplando todo el conjunto de patrones a utilizar. Posteriormente, puede ser reutilizada en todos los procesos de modernización que se desee.

Una vez que tanto los requisitos como los patrones se han caracterizado en base a las características RIA es el momento de mapear los requisitos con los patrones.

	P1	P2	P3	P4	P5	P6	P7	P8	P9
RQ1	4	4	6	6	1	4	4	2	2
RQ2	4	5	4	4	6	6	8	5	5
RQ3	4	3	3	4	3	7	7	4	4
RQ4	4	3	3	4	5	7	9	6	6
RQ5	3	4	3	3	4	4	6	3	3
RQ6	4	4	3	4	4	4	6	3	3
RQ7	1	0	2	2	5	4	6	6	6

Tabla 8.3: Matriz de correspondencia requisitos-patrones

8.4.3. Mapeo entre requisitos y patrones

Este proceso se basa en el mapeo entre las dos matrices de trazabilidad presentadas anteriormente. El proceso de mapeo consiste en realizar una serie de operaciones matemáticas que toman como entrada los valores obtenidos en las matrices. Este proceso es implementado mediante una sencilla herramienta (basada en hojas de cálculo) que realiza el mapeo entre las matrices y proporciona los patrones sugeridos a partir del análisis de los resultados obtenidos.

Como primer paso, se transforman las matrices de trazabilidad en matrices binarias, de manera que sean computables. Para ello se convierten los valores “x” o “o” en el valor “1” y a las celdas sin valor se les asigna el valor “0”.

Posteriormente, se realiza el producto matricial entre la primera matriz y la transpuesta de la segunda, obteniendo la matriz de correspondencia que muestra el número de características RIA compartidos entre requisitos y patrones. La Tabla 8.3 muestra un ejemplo de matriz de correspondencia. Nótese que por simplicidad no se introduce la exposición de los requisitos utilizados para el cálculo de esta matriz. El lector interesado puede encontrar una exposición más detallada en (Conejero *et al.* 2013).

A partir de estos valores, se puede inferir, por ejemplo, que los patrones P6 y P7 son buenos candidatos para la satisfacción del requisito RQ4, dado que son los que más características RIA comparten con este requisito. Sin embargo, estos valores pueden llevar a conclusiones erróneas porque pueden estar sesgados por la granularidad de los requisitos. En este sentido, requisitos con una granularidad más fina se comportarán mejor. Para minimizar el impacto de este problema, se normalizan los valores obtenidos mediante su relación con el número de características RIA requeridas por el requisito y patrón correspondiente. A estas relaciones se les denomina Grado de Realización del Requisito (Degree of requirement realization, DRR) y Grado de Realización del Patrón (Degree of pattern realization, DPR). El DRR para un requisito RQ_i con respecto al patrón P_j se define como:

$$Degree\,of\,requirement\,realization\,(RQ_i, P_j) = \frac{cm_{ij}}{\sum_{k=1}^{|F|} rtm_{ik}}$$

Por su parte, el DPR de un requisito RQ_i con respecto al patrón P_j se define como:

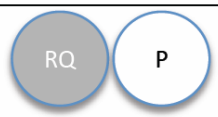
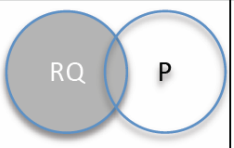
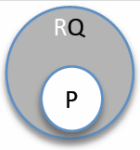
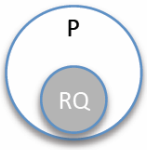
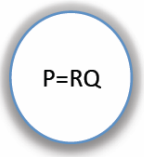
		DPR		
	Values	0	0 < DPR < 1	1
DRR	0	 $RQ \cap P = \emptyset$	*Unfeasible*	*Unfeasible*
	0 < DRR < 1	*Unfeasible*	 $RQ \cap P \neq \emptyset$	 $P \subset RQ$
	1	*Unfeasible*	 $RQ \subset P$	 $P = RQ$

Tabla 8.4: Combinaciones de DRR y DPR

$$Degree\,of\,pattern\,realization\,(RQ_i, P_j) = \frac{cm_{ij}}{\sum_{k=1}^{|F|} ptm_{jk}}$$

donde cm_{ij} representa la celda *[i,j]* de la matriz de correspondencia, $|F|$ representa el número de características RIA and rtm_{ik}y ptm_{jk} representan las celdas *[i,k]* y *[j,k]* de cada una de las matrices de trazabilidad.

Un valor de 1 en DRR indica que todas las características RIA demandadas por el requisito son soportadas por el patrón, aunque puede soportar más. Un valor cercano a 0 indica que el patrón solamente soporta un pequeño grupo de las características RIA demandadas por el requisito. Y, obviamente, un valor de 0 indica que requisito y patrón no comparten ninguna característica. De manera semejante, un valor de 1 en DPR denota que todas las características RIA soportadas por el patrón son necesarias para el requisito, aunque puede necesitar más. Un valor cercano a 0 en DPR señala que una cantidad no significativa de las características soportadas por el patrón son necesarias para el requisito. La tabla 8.4 muestra las diferentes combinaciones de los valores DRR y DPR que se pueden dar para un patrón y un requisito particulares. *RQ* representa el conjunto de características RIA demandadas por un requisito. Y *P* indica el conjunto de características RIA cubiertas por un patrón. Los patrones mejores candidatos serán, por tanto, aquellos que presenten valores más altos de DPR y DRR, siendo la mejor situación en la que coinciden ambos conjuntos (*P=RQ*).

8.5. Aplicación de los patrones RIA seleccionados

Para la aplicación de estos patrones de una forma no intrusiva en los modelos de la aplicación, se ha decidido seguir una aproximación basada en los principios del Modelado Orientado a Aspectos (Clarke &

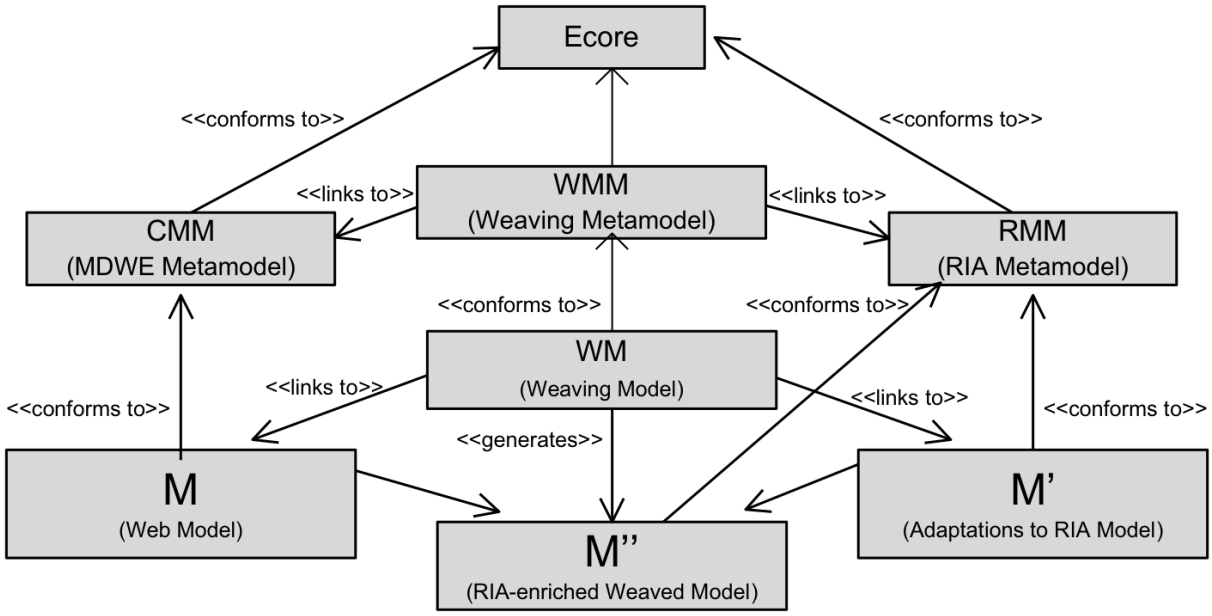


Figura 8.7: Esquema de la propuesta de composición (weaving) de modelos

Baniassad 2005). En este sentido, se han definido tres metamodelos distintos: Core Metamodel (CMM), RIA Metamodel (RMM) y Weaving Metamodel (WMM). CMM corresponde al metamodelo del lenguaje de especificación de la aplicación web heredada, por ejemplo, WebML. RMM consiste en una extensión del CMM para proporcionarle soporte de características RIA. Finalmente, WMM permite especificar las semánticas de composición entre los dos anteriores.

La Figura 8.7 presenta la arquitectura básica de esta propuesta. La aplicación web heredada es representada como el modelo M conforme a CMM. Las características RIA son recogidas en el modelo M' conforme a RMM. La composición de ambos modelos, usando el modelo de composición WM, da lugar al modelo M'' conforme a RMM. M'', por lo tanto, representa el modelo resultante del proceso de modernización a RIA de la aplicación web heredada.

A continuación se realiza una exposición de los metamodelos creados y del proceso de composición seguido.

8.5.1. Metamodelo RIA

Para este trabajo se ha usado como base el metamodelo de WebML definido por (Schauerhuber *et al.* 2006). Este metamodelo ha sido enriquecido con las características RIA necesarias para la especificación de los patrones presentados anteriormente. En concreto, como ilustra la Figura 8.8, se han realizado las siguientes extensiones:

- Se ha definido un nuevo atributo *tier* dentro de la metaclase *WebML:ModelElement*. Este atributo podrá tomar dos valores diferentes: *client* y *server*. Su misión es indicar si una determinada entidad del modelo se debe localizar en el cliente o en el servidor.

- Se han definido las metaclases *XMLin* y *XMLout* para poder usar entidades de este tipo.

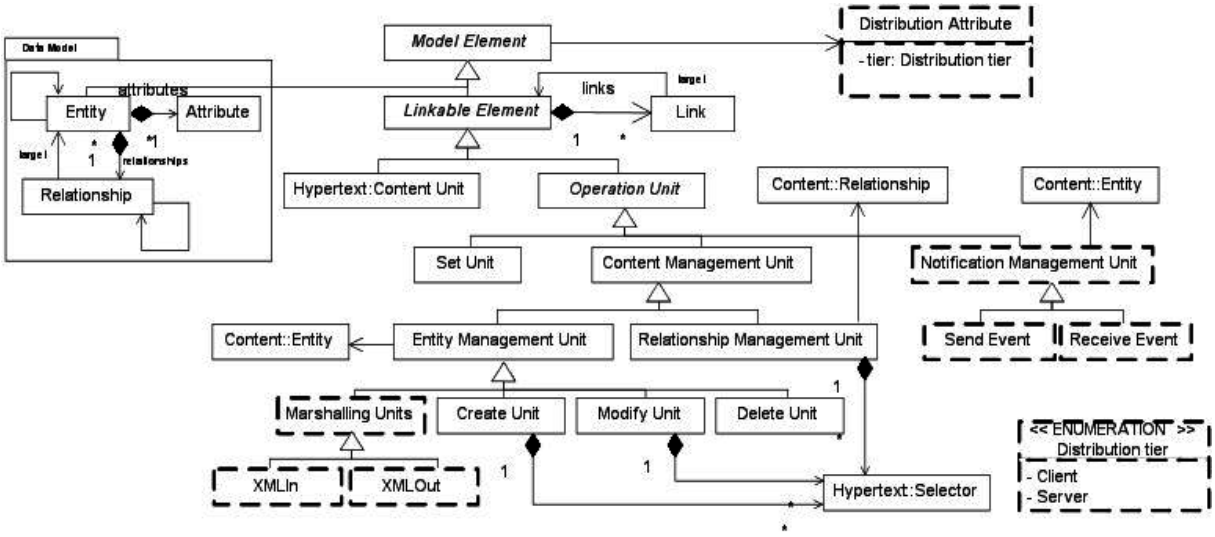


Figura 8.8: Metamodelo WebML extendido con características RIA

- Y, finalmente, se han definido las metaclases *Send Event* y *Receive Event* dentro de la jerarquía de operaciones.

8.5.2. Metamodelo de Weaving

El metamodelo de weaving se ha definido como una extensión del metamodelo genérico ATL Model Weaving (AMW) (Del Fabro *et al.* 2005; Didonet Del Fabro & Valduriez 2009) para adaptarlo al metamodelo de WebML. El proceso de composición consiste básicamente en la definición de modelos de weaving que son procesados mediante reglas ATL (Jouault & Kurtev 2006).

La Figura 8.9 presenta un extracto de este metamodelo que contiene las extensiones necesarias para componer entidades de datos y unidades de operación de WebML. Como se puede apreciar, estas extensiones consisten en:

- Se ha definido un nuevo tipo de modelo de weaving: *MigrationModel*. Este modelo contiene las referencias a los modelos implicados en la composición: el modelo web heredado (*leftModel*), el modelo RIA (*rightModel*) y el modelo resultado (*targetModel*). Además, está compuesto por una colección de enlaces de modernización (*MigrationLink*).

- Se ha extendido la metaclase *WLink* mediante la definición de tres subclases que corresponden con tres nuevos tipos de enlaces de modernización: *Merge*, *Enrich* y *Match*. Estos enlaces representan la semántica operacional de composición entre los elementos que unen.

    - La metaclase *Merge* indica una conexión de mezclado entre un elemento del modelo Web y un elemento del modelo RIA del mismo tipo. Concretamente, permite definir enlaces entre dos entidades de datos, dos unidades de operación o dos enlaces. En el caso de la mezcla de unidades de operación, puede ser necesario especificar información adicional para indicar la

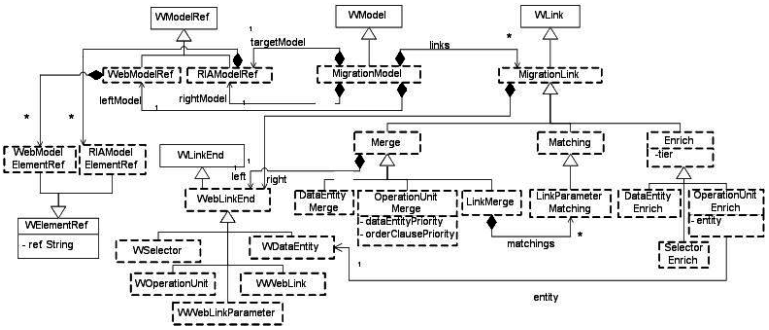


Figura 8.9: Extracto del WMM

prioridad de selección de sus diferentes componentes. Esta información es mantenida por los atributos *dataEntityPriority* y *orderClausePriority*. Estos atributos pueden indicar mediante sus valores *left* y *right* qué componente debe utilizarse en la composición de la unidad resultado.

- La metaclase *Enrich* permite especificar si el elemento de modelo generado se localizará en el cliente o en el servidor. En su forma más básica, este enlace presenta un único extremo que referencia a un elemento de cualquiera de los modelos fuente. Esta metaclase, sin embargo, ha sido extendida mediante la definición de tres nuevas subclases, específicas de cada tipo de elemento que puede ser enriquecido, i.e. entidades de datos, unidades de operación y selectores.
- La metaclase *Matching* permite definir mapeos entre elementos del modelo Web y elementos del modelo RIA. En concreto, la metaclase *LinkParameterMatching* permite especificar el mapeo de parámetros dentro un enlace resultado de la mezcla de dos enlaces originales.

8.5.3. El proceso de composición

El proceso de composición se ha implementado mediante la definición de reglas ATL. Con el objetivo de mantener lo más simple posible la definición de estas reglas, el proceso de composición se ha definido sobre las entidades de primer orden de WebML, i.e., entidades, unidades, páginas y áreas. En este sentido, siguiendo la descomposición en capas de WebML, las reglas ATL se han agrupado en dos conjuntos diferentes: uno para las entidades de datos, y otro para los elementos del modelo de hipertexto. A continuación, a modo de ejemplo ilustrativo, se proporciona una explicación conceptual del funcionamiento de las reglas de mezclado para entidades de datos y unidades de operación.

Para la composición del modelo de datos de WebML se han definido dos reglas principales:

1. Migración de entidades de datos que no se mezclan. Estas entidades aparecen en el modelo de weaving como instancias de *DataEntityEnrich*. Consiste en el grupo de entidades de cualquiera de los modelos de origen que pasan al modelo resultado prácticamente inalteradas. La aplicación de esta regla puede suponer dos tipos de modificación sobre la entidad origen: (1) actualización

del valor de su atributo de localización (cliente o servidor); y (2) resolver los extremos de sus relaciones con entidades de datos mezcladas (relaciones huérfanas).

2. Migración de entidades de datos a mezclar. Aparecen en el modelo de weaving como instancias de *DataEntityMerge*. En este caso, la aplicación de esta regla produce una nueva entidad de datos como resultado de la mezcla de las entidades de los modelos fuente relacionadas. Esta nueva entidad toma su nombre de la entidad origen del modelo Web heredado y se compone, por un lado, de la unión del conjunto de atributos contenidos en las entidades origen y, por otro lado, de todas las relaciones cuyo origen fuese alguna de esas dos entidades. En este caso, es también necesario resolver las relaciones huérfanas generadas, referenciando la entidad resultado de la mezcla en el modelo resultado. Finalmente, la posible redundancia en las relaciones debe ser eliminada en un proceso posterior de refinado.

Para la composición del modelo de hipertexto de WebML, por su parte, se han definido tres reglas principales:

1. Migración de unidades de operación que no se mezclan. Estas entidades aparecen en el modelo de weaving como instancias de *OperationUnitEnrich*. Consiste en el grupo de unidades de operación de cualquiera de los modelos de origen que pasan al modelo resultado prácticamente inalteradas. La aplicación de esta regla puede suponer tres tipos de modificación sobre la unidad origen: (1) actualización del valor de su atributo de localización (cliente o servidor); (2) resolver la entidad de datos relacionada en base a la referencia *entity*; y (3) eliminar los enlaces de salida que se vayan a mezclar (realizado posteriormente por la regla 3).

2. Migración de unidades de operación a mezclar. Aparecen en el modelo de weaving como instancias de *OperationUnitMerge*. En este caso, la aplicación de esta regla produce una nueva unidad de operación como resultado de la mezcla de las unidades de los modelos fuente relacionadas. Esta nueva entidad toma su nombre de la unidad origen del modelo Web heredado. A continuación, se realiza la descripción de cómo se mezclan los diferentes componentes de las unidades de operación:

   a) Si ambas están relacionadas con entidades de datos, el modelo de weaving especifica cuál de las dos se utilizará en el modelo resultado mediante el atributo *dataEntityPriority*.

   b) Si ambas unidades contienen selectores, todas sus condiciones son concatenadas en el selector resultante, indicando la localización (cliente o servidor) en la que serán procesados. En el modelo de weaving, la localización se especifica por medio de instancias *SelectorEnrich*.

   c) Si ambas presentan cláusulas de ordenación, los atributos de las mismas se concatenarán en la cláusula resultante, según la secuencia especificada en el atributo *orderClausePriority*.

d) Con respecto a la composición de los enlaces, en este caso, sólo se consideran los enlaces de salida de la unidad porque están contenidos dentro de las unidades a mezclar. En este sentido, sólo se selecciona el conjunto de enlaces de salida que no se mezclan porque tanto su unidad destino como sus parámetros no se modifican.

3. Migración de enlaces a mezclar. Aparecen en el modelo de weaving como instancias de *Link-Merge*. En este caso, es necesario resolver cuál será la unidad destino y el mapeo de parámetros a realizar. La unidad destino vendrá marcada por la referencia representada por el enlace *right*. Mientras que el mapeo de parámetros se realizará según lo indicado por la colección de instancias de *LinkParameterMatch*.

Como conclusión, es interesante resaltar que el modelo de weaving es la única parte de esta propuesta que depende de la aplicación concreta a modernizar. Mientras que las reglas definidas pueden ser reutilizadas en procesos de modernización de otras aplicaciones sin necesidad de modificación.

8.6. Generación de API REST

En esta segunda estrategia, el objetivo principal ha consistido en generar automáticamente una API REST de la aplicación Web heredada con el fin de proporcionar un punto de acceso alternativo al sistema heredado (Rodríguez-Echeverría *et al.* 2013b). La capa REST a generar debe satisfacer los siguientes requisitos:

1. Comportamiento adaptable según un fichero de configuración.
2. Asegurar soporte para los métodos PUT y DELETE del protocolo HTTP.
3. Soportar la conversión entre formatos de datos comunes de una API REST.
4. Delegar el manejo de la petición REST al controlador heredado (mapeo REST-controlador). El uso de un controlador heredado conlleva la necesidad de:

   a) Generar el contexto adecuado para la correcta ejecución del controlador heredado.

   b) Transformar la respuesta del controlador al formato de datos esperado.

En la especificación de esta capa REST, se pueden identificar dos partes claramente diferentes: (1) primero, una parte que depende del FAW usado para el desarrollo de la aplicación web heredada; y, (2) segundo, una parte específica de la aplicación heredada. La primera parte consiste en todas las extensiones y adaptaciones que se deben realizar el FAW para dotarle de las capacidades REST comentadas. Por otro lado, la segunda parte está relacionada con la especificación del comportamiento de la capa REST (especificación de la API) para una aplicación web determinada, que conlleva, al menos:

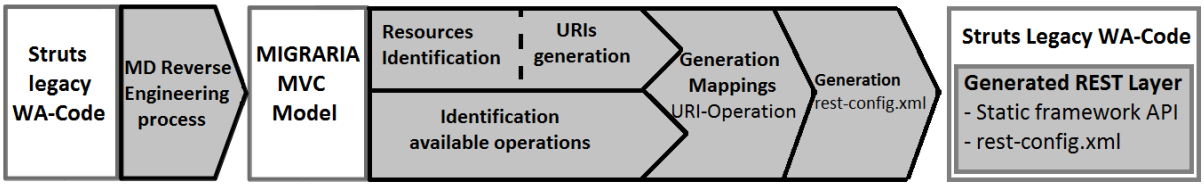


Figura 8.10: Esquema del proceso de generación de API REST

1. La definición de los mapeos entre peticiones REST y controladores heredados.

2. La conversión de los datos relacionados con la entrada y la salida de un controlador.

Obviamente, mientras que la primera parte de la capa REST permanece constante para diferentes aplicaciones web, la segunda parte (especificación de la API) debe adaptarse a la aplicación heredada a modernizar. El proceso seguido para especificar la parte dependiente de la aplicación (API) se ilustra en la Figura 8.10. Como se puede observar, el proceso toma como entrada el modelo MIGRARIA MVC de la aplicación heredada y genera su API REST. Este proceso está definido por las siguientes actividades:

1. Identificación de los recursos relacionados con el subsistema a modernizar.

2. Generación de las URI [5] REST para los recursos identificados.

3. Identificación de las operaciones disponibles sobre cada recurso según las vistas del subsistema seleccionado, sus peticiones y los controladores que las atienden.

4. Generación de los mapeos de petición para las operaciones disponibles en cada recurso para un formato de datos dado.

5. Delegación de la atención de la petición REST a los controladores heredados disponibles.

Los siguientes apartados proporcionan una descripción detallada de ambas partes.

8.6.1. Especificación de la capa REST dependiente del Framework

Con el fin de generar una capa REST que permita la especificación de APIs REST de la aplicación heredada, puede ser necesario extender convenientemente el FAW para dotarle del soporte necesario para la atención de peticiones REST. En este sentido, la aproximación seguida consiste, por un lado, en realizar una intervención manual del FAW y, por otro lado, en generar automáticamente (transformaciones modelo a texto) su configuración acorde a la información extraída de la aplicación web heredada y a la funcionalidad a incluir en el cliente RIA objetivo.

Con el objetivo de ilustrar cómo se realiza la instrumentación de un FAW concreto para dotarle de soporte para la especificación de APIs REST según nuestra propuesta, se ha desarrollado su implementación para el FAW Struts v1.3. Esta implementación se describe gráficamente en la Figura 8.11. Esta

[5] Uniform Resource Identifier

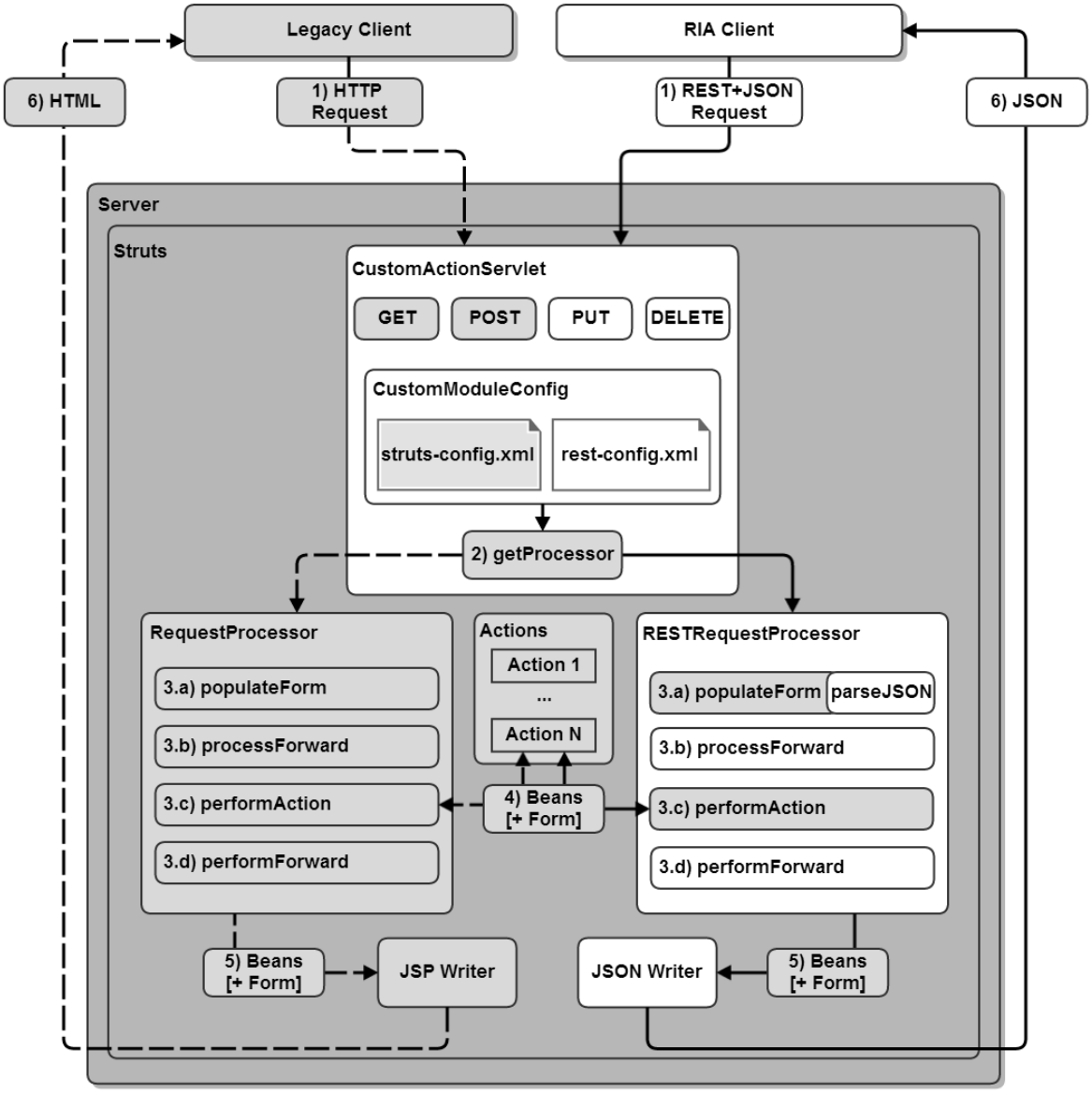


Figura 8.11: Extensión de Struts v1.3.X para soporte REST

figura compara el ciclo de vida de una petición en Struts (línea de puntos) con el ciclo de vida de una petición REST (línea sólida) en Struts extendido con soporte REST. Esta ilustración comparativa permite apreciar las modificaciones realizadas al FAW en este caso. Nótese que las cajas blancas indican elementos modificados o añadidos a Struts. Básicamente, las modificaciones introducidas consisten en la extensión de las clases *ActionServlet*, que implementa el patrón Front Controller [6] en Struts, y la clase *RequestProcessor*, que gestiona el ciclo de vida de una petición dentro del FAW.

Por un lado, se ha creado por tanto, la clase *CustomActionServlet* que posee las siguientes responsabilidades:

1. Añadir soporte a los métodos PUT y DELETE en las peticiones HTTP.
2. Cargar el fichero de configuración con la especificación de la API REST para una aplicación web heredada concreta.
3. Decidir qué *RequestProcessor* debe atender la petición.

Por otro lado, se ha creado la clase *RESTRequestProcesssor* que es responsable de:

1. Generar el contexto adecuado para la invocación del controlador heredado (paso 3.a en la figura).
2. Generar el objeto de respuesta acorde al formato de datos establecido (paso 3.b en la figura).
3. Delegar la atención de la petición al controlador heredado según el mapeo REST-controlador especificado en la configuración (paso 3.c en la figura).
4. Delegar la generación de la respuesta en el componente correspondiente según el formato de datos establecido (paso 3.d en la figura).

Además de las extensiones comentadas, se ha añadido la clase *JSONWriter* como un compositor de respuesta alternativa. En este caso, construye una respuesta válida en formato JSON a partir de los datos generados como respuesta por el controlador heredado invocado. Obviamente, la generación de respuestas en otros formatos de datos requerirían la implementación de diferentes clases *Writer*.

8.6.2. Especificación de la API REST dependiente de la aplicación

A continuación, se describen los pasos principales llevados a cabo en nuestra propuesta para extraer la información que permita configurar la implementación de la capa REST dependiente del framework. Esta información se obtiene del modelo MIGRARIA_MVC que representa la aplicación Web heredada.

[6] http://struts.apache.org/development/1.x/apidocs/org/apache/struts/action/ActionServlet.html

Vistas	Recursos potenciales	Recursos	I/O	Colección	Relaciones
paperCreate	paperForm	Paper	I	NO	1 Track : N Paper
	track	Track	O	YES	1 Paper : N Coauthor
paperEdit	paperForm	Paper	I	NO	1 Paper : N Review
	track	Track	O	YES	1 Author : N Paper
	coauthorForm	Coauthor	I	NO	1 Author : N Review
	coauthor	Coauthor	O	YES	N Author : M Track
paperDetails	paper	Paper	O	NO	
	coauthor	Coauthor	O	YES	
	review	Review	O	YES	
paperList	paper	Paper	O	YES	
	author	Author	O	NO	

Tabla 8.5: Recursos y relaciones REST

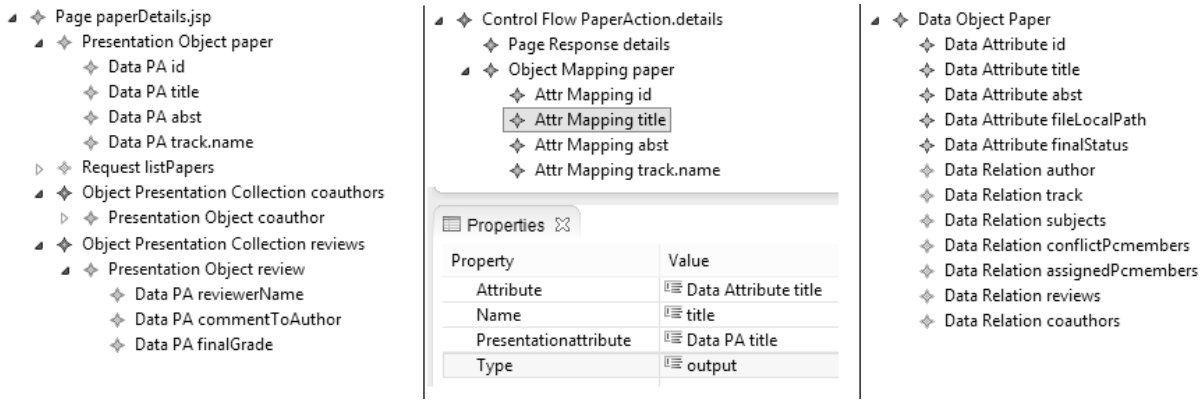


Figura 8.12: Identificación de recursos en un modelo MIGRARIA-MVC

8.6.2.1. Identificación de recursos

La especificación de una API REST implica rediseñar la interacción entre cliente y servidor desde un punto de vista centrado en los recursos. En este sentido, la identificación adecuada de los recursos involucrados y sus relaciones se convierte en un paso fundamental en el proceso de generación de la API REST. Con el fin de llevar a cabo esta actividad, el ingeniero de modernización debe analizar y consultar el modelo MIGRARIA_MVC para encontrar los recursos relacionados con las vistas que componen el subsistema a modernizar. Cada contenedor de la vista (página) se compone de uno o varios objetos de presentación, que definen una vista concreta sobre los objetos de datos correspondientes, definidos en el modelo de datos; por ejemplo, presentando sólo algunos de sus atributos. Estos objetos de presentación pueden representar datos de entrada o salida, que a su vez pueden presentarse individualmente u organizados en colecciones. Para poder identificar los objetos del modelo se requiere llevar a cabo un análisis del controlador, con el objetivo de identificar los mapeos concretos entre objetos de presentación y del modelo, dentro de una vista concreta.

La Figura 8.12 muestra el objeto de presentación *paper*, incluido en la vista paperDetails, y su mapeo con el objeto de datos *Paper*, definido en el controlador *paperAction.details*. Como resultado final del análisis del modelo MIGRARIA_MVC, el ingeniero debe obtener: i) los objetos de datos involucrados en

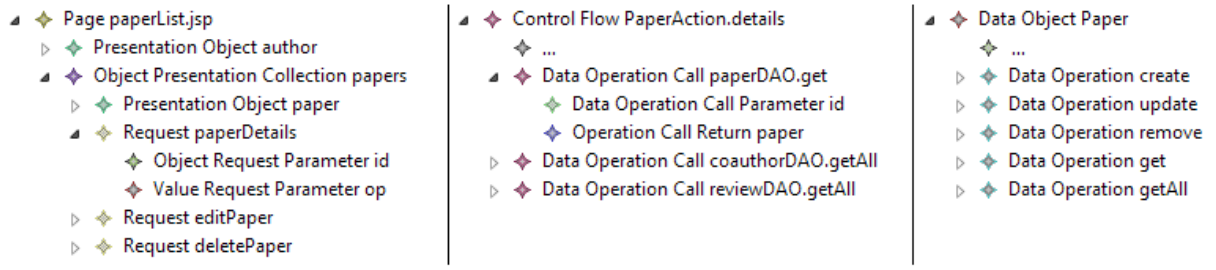


Figura 8.13: Identificación de operaciones en un modelo MIGRARIA-MVC

el subsistema a modernizar; ii) las relaciones entre ellos; y, además, iii) las diferentes maneras en las que son presentados en las vistas y consumidos por los controladores. Toda esta información se ha recopilado en la columna de recursos potenciales de la tabla 8.5, que muestra los resultados para el subsistema de envío de artículos en nuestro caso de estudio. Esta tabla contiene también las relaciones existentes entre los recursos encontrados.

8.6.2.2. Generación de URIs REST

Tomando como entrada la información referente a los recursos y sus relaciones obtenidos en el paso anterior, se generan todas las combinaciones factibles de URI REST, para cada recurso obtenido, mediante el proceso descrito a continuación. La columna URIs de la tabla 8.6 muestra un pequeño ejemplo de la colección generada para el ejemplo utilizado, que ilustra los bloques de construcción principales de nuestro proceso.

Los pasos de este proceso consisten en:

1. Generar las dos URIs base para cada recurso:

   a) /recurso

   b) /recurso/{id}

2. Generar las dos URIs base para cada relación 1-N:

   a) /recurso_1/{id}/recurso_N

   b) /recurso_1/{id}/recurso_N/{id}

3. Por cada relación N-M, identificar el recurso maestro y ejecutar el paso anterior, tomando este recurso como principal.

4. Completar las URIs, añadiendo a la cadena de consulta cualquier parámetro de control que aparezca en las peticiones originales.

8.6.2.3. Identificación de Operaciones Disponibles

Considerando el hecho de que el proceso de modernización toma como entrada una aplicación heredada, resulta obvio que el conjunto de operaciones por recurso que la API REST puede proporcionar está limitado por las operaciones disponibles en el sistema original. Por tanto, necesitamos identificar las operaciones definidas previamente sobre estos recursos a modernizar, para ser capaces de filtrar correctamente el conjunto de posibles peticiones REST que queremos proporcionar. De nuevo, esta información es obtenida por el ingeniero de modernización mediante el análisis y consulta del modelo MIGRARIA_MVC. Este análisis permite, en este caso, obtener todas las peticiones relacionadas con el conjunto de vistas que componen el sistema a modernizar. El modelo MIGRARIA_MVC permite la identificación del controlador que debe manejar una petición en particular, pero también al controlador responsable de la generación de una vista en particular. El componente Controlador del modelo MIGRARIA_MVC permite especificar la secuencia de llamadas a operaciones que ejecuta un controlador para manejar los objetos de datos en la aplicación heredada. Tomando como base el análisis de esta secuencia, se pueden identificar las operaciones llevadas a cabo sobre un recurso dentro de un controlador. Un análisis exhaustivo de los controladores que manejan el conjunto de peticiones relacionadas con la modernización permite obtener, como resultado, la lista de las operaciones disponibles para cada recurso.

Para ilustrar el proceso de identificación de operaciones, consideramos aquí una petición específica, contenida en una de las vistas a modernizar en el ejemplo utilizado, concretamente la petición *paperDetails* de la vista *paperList*. En base a esta petición, se identifica el controlador (*ControlFlow*) que la maneja, mediante las relaciones existentes entre los elementos *View.Request* y *Controller.ControlFlow*. En este caso, la petición *paperDetails* es manejada por el *ControlFlow paperAction.details*, que contiene la secuencia de llamadas a operaciones descrita en la Figura 8.13. En este caso, por ejemplo, la operación *get* es invocada sobre el objeto de datos *Paper*, pasando como parámetro el *id* del *paper* (parámetro de la petición) para recuperar y recibir un objeto *Paper* como resultado. Como se muestra en este extracto del modelo, todas las llamadas a operaciones están relacionadas con operaciones de datos concretas definidas en un objeto de datos concreto. En un modelo MIGRARIA_MVC, todas las operaciones de datos aparecen clasificadas de acuerdo a su función en una de las siguientes categorías: *create, update, remove, get, getAll* y *getFilter*. Dicha clasificación proporciona al ingeniero de modernización un vocabulario común para identificar de manera unívoca la lista de las operaciones de datos realizada por un controlador. Esta identificación es necesaria para emparejar las operaciones mencionadas con el método HTTP correcto que se usará en una petición REST. Dicho mapeo se muestra en la Tabla 8.6.

8.6.2.4. Generación de los mapeos a peticiones REST

Siguiendo una aproximación similar a la generación de las URIs REST, nuestro proceso genera todas las combinaciones posibles de URIs y métodos HTTP, produciendo cuatro mapeos de petición diferentes

URIs	Operaciones	Controladores	Peticiones REST
/paper	Create	paperAction.new	POST
	Read	paperListAction	GET
	Update		
	Delete		
/paper/{id}	Create		
	Read	paperAction.details	GET
	Update	paperAction.save	PUT
	Delete	paperAction.delete	DELETE
/paper/{id}/coauthor	Create	coauthorAction.new	POST
	Read	paperAction.details	GET
	Update		
	Delete		
/paper/{id}/coauthor/{id}	Create		
	Read		
	Update	coauthorAction.save	PUT
	Delete	coauthorAction.delete	DELETE

Tabla 8.6: Resumen del resultado del proceso de configuración REST

para cada URI REST. A continuación, dicha colección es filtrada con el conjunto de operaciones disponibles, por lo que sólo las peticiones REST que puedan emparejarse correctamente con un controlador heredado se mantienen, siendo el resto eliminadas. La Tabla 8.6 resume los resultados obtenidos para las URIs REST seleccionadas en nuestro ejemplo ilustrativo.

8.6.2.5. Delegación del procesado de peticiones REST a las Actions Heredadas de Struts

Esta actividad del proceso consta de los siguientes pasos:

1. La definición de los mapeos entre peticiones REST y los controladores existentes. Este mapeo deriva de la información obtenida previamente referente a la relación petición-controlador. Luego, a modo de ejemplo, y de acuerdo con la Tabla 8.6, el manejo de la petición HTTP GET para la ruta */paper/{id}* será delegada al controlador *paperAction.details*.

2. La conversión de los datos de entrada de la petición (parámetros o cuerpo del mensaje) al formato de datos esperado por el controlador. Dado que los controladores heredados son reutilizados en el proceso, se debe generar el contexto que esperan para asegurar una ejecución correcta de los mismos. Por tanto, podría necesitarse, por ejemplo, que un objeto concreto que representa la información enviada a través de un formulario HTML (esperado por el controlador) deba ser poblado con los datos obtenidos de la petición REST (por ejemplo en formato JSON). La generación de este contexto esperado por el controlador requiere conocer, por un lado, los parámetros de la petición original y, por el otro, los objetos de presentación relacionados con la petición original. Toda esta información puede obtenerse también con facilidad del modelo MIGRARIA_MVC de la aplicación heredada mediante el análisis de la vista que contiene la petición de origen. En nues-

```
{
  "paperForm" : {
    "id" : ""
    "title" : "My New Paper"
    "abst" : "Abstract..."
    "fileLocalPath" : "mypaper.pdf"
    "author" : "5"
    "track" : "3"
  }
}
```

```
public class PaperForm extends ActionForm {

    private Long id;
    private String title;
    private String abst;
    private String fileLocalPath;
    private long author;
    private long track;
```

Figura 8.14: Mapeo de JSON de entrada *form bean*

tro ejemplo, la petición *new* contenida en la vista *paperCreated* está relacionada con el objeto de presentación de entrada llamado *paperForm*. Por tanto, el objeto esperado por el controlador debe ser generado para manejar ese objeto. La Figura 8.14 presenta, en la parte izquierda, la estructura del mensaje JSON para la creación de un nuevo *paper*, y en la derecha, el objeto esperado por el controlador *paperCreate* que será creado e inicializado convenientemente a partir del mensaje JSON recibido.

3. La generación de la respuesta en un formato de datos particular (por ejemplo, en formato JSON) en vez de en una página HTML. Los clientes de la API REST normalmente esperan respuestas en un formato de datos particular (JSON, XML) para poder manejarlas de manera apropiada en la parte del cliente. Por ello, la respuesta generada por la aplicación heredada, en forma de página HTML, no es útil en este caso y debe ser reemplazada por la respuesta generada en el formato de datos acordado previamente. En los frameworks de desarrollo Web basados en el patrón MVC, los controladores suelen delegar a un componente distinto del framework la composición de la respuesta final, como por ejemplo, a un sistema de gestión de plantillas de presentación. De esta forma, la principal responsabilidad del controlador es generar los objetos de salida necesarios para poblar la plantilla seleccionada para construir la respuesta. Basándonos en esta aproximación, podemos conocer los objetos de presentación que un controlador en particular generará (y la estructura de los mismos), analizando únicamente el modelo MIGRARIA_MVC. Por ejemplo, el *ControlFlow* *paperAction.details* genera el objeto de presentación de salida *paper* para componer la respuesta por medio de la plantilla de la vista *paperDetails*. La estructura del objeto de presentación *paper*, definido dentro de la plantilla de la vista *paperDetails*, se muestra en la Figura 8.15 (parte izquierda). A partir de esta información, la respuesta final es generada en el formato de datos acordado, JSON en este caso.

8.6.2.6. Generación de la configuración de la capa REST: especificación de la API REST

Como se mencionó previamente, nuestra propuesta consiste en extender el framework de desarrollo Web utilizado como base, para incorporarle la funcionalidad REST y configurarlo automáticamente,

```
<h2>Paper <bean:write name="paper" property="id" /> details</h2>
<table id="paperDetailsTable">
    <tr>
        <th>Title</th>
        <th>Abstract</th>
        <th>Track</th>
    </tr>
    <tr>
        <td><bean:write name="paper" property="title" /></td>
        <td><bean:write name="paper" property="abst" /></td>
        <td><bean:write name="paper" property="track.name" /></td>
    </tr>
</table>
```

```
{
  ...
  "paper" : {
    "id" : "12"
    "title" : "Example Paper"
    "abst" : "Abstract..."
    "track.name" : "Example Track"
  }
  ...
}
```

Figura 8.15: Mapeo de objeto de salida a JSON

```
<action path="/paper"
    method="POST"
    dataFormat="JSON"
    type="migraria.casestudies.struts.CRS.actions.PaperAction"
    name="paperForm">
    <forward name="success" path="paper"/>
</action>
```

Figure 8.16: Fichero de configuración de la API REST

y así proporcionar una API al sistema heredado subyacente. Las secciones previas describen detalladamente ambos procesos. En este trabajo, la configuración (semi)automática se genera por medio de trasformaciones modelo a texto descritas en ATL. Estas transformaciones implementan el proceso descrito anteriormente y generan un archivo XML de salida. El esquema de dicho fichero se ha definido como una extensión del utilizado por el fichero de configuración de Struts v1.3 con el fin de simplificar la extensión del FAW implementada. La Figura 8.16 muestra un ejemplo representativo del mapeo de una petición REST, mostrando: (1) su ruta, (2) el método HTTP, (3) su formato de datos, (4) la clase del controlador heredado que debe manejar la petición (*type*), (5) una referencia a la definición de la estructura del objeto de entrada (*name*), y (6) una referencia a la definición estructural del objeto de respuesta (atributo *path* del elemento *forward*).

8.7. Resumen

Este capítulo presenta algunos de los trabajos realizados para las actividades de refinado RIA y de generación de código de las etapas de reestructuración e ingeniería directa dentro del proceso de modernización definido en el proyecto MIGRARIA. Estos trabajos se presentan con fines ilustrativos y, por tanto, no se profundiza en su exposición ni validación.

El primer trabajo presentado tiene como misión generar una representación en WebML de la aplicación web heredada. Con este fin, se ha definido una actividad de transformación modelo a modelo que toma como origen el modelo MIGRARIA MVC de la aplicación web heredada y produce como destino sus modelos correspondientes en WebML. En este capítulo se ilustra el proceso de transformación en base a la generación de los elementos WebML necesarios para representar las operaciones CRUD identificadas en el modelo MIGRARIA MVC origen. El objetivo último de esta transformación es poder

utilizar la herramienta WebRatio que, entre otras funcionalidades, nos permite generar el código de la aplicación final.

Posteriormente, se presentan varios trabajos relacionados con la definición de patrones RIA en WebML, la selección sistemática de los mismos basados en los requisitos RIA de la aplicación destino y la aplicación de los patrones seleccionados a la aplicación original mediante técnicas de weaving de modelos. Estos trabajos persiguen definir las actividades de modernización de la etapa de reestructuración utilizando una propuesta IWDM ya existente, como WebML.

Finalmente, el último trabajo presenta una propuesta de generación de una capa de servicios REST de la aplicación web heredada. Esta capa permite adaptar las necesidades del nuevo cliente RIA a las capacidades de la aplicación web heredada. Dentro de este trabajo, por un lado, se ha implementado una extensión del framework de aplicaciones web de referencia, Struts 1, para proporcionar soporte de servicios REST. Y, por otro lado, se ha definido un proceso que permite derivar la especificación de la capa de servicios REST a partir del modelo MIGRARIA MVC de la aplicación web heredada.

Parte III

Reflexión y Resultados

Capítulo 9

Conclusiones y Trabajos Futuros

9.1. Resumen de las principales contribuciones

A continuación, se relacionan las principales contribuciones de este trabajo de tesis:

1. Se ha definido un marco de referencia para la especificación de procesos de modernización de aplicaciones web heredadas a RIA. Este marco está compuesto por tres partes fundamentales: el proceso de ingeniería inversa, la reestructuración o refinado y el proceso de ingeniería directa. A partir de este marco, se ha especificado un **proceso concreto de modernización para el dominio de aplicaciones web heredadas desarrolladas mediante un framework de aplicaciones web basado en el patrón MVC**.

2. Dentro de las diferentes partes del proceso de modernización definido, el trabajo desarrollado se ha enfocado principalmente en la definición completa de un **proceso de ingeniería inversa de aplicaciones web heredadas mediante análisis estáticos de los artefactos software** que la componen. Este proceso se ha definido en base a técnicas y tecnologías de la Ingeniería Dirigida por Modelos para favorecer su sistematización, reusabilidad y mantenibilidad. La definición de este proceso ha conllevado las siguientes contribuciones:

   a) La **definición de una notación conceptual** (MIGRARIA MVC) para aplicaciones web heredadas dentro del dominio considerado y específicamente diseñada para procesos de ingeniería inversa. Esta notación conceptual permite independizar al proceso de modernización de la tecnología de implementación final. Esta notación se ha definido como un metamodelo conforme a e-core y usando las herramientas del proyecto Eclipse Modeling Framework.

   b) La **definición del proceso de ingeniería inversa como una cadena de transformaciones** que consta de dos actividades principales:

      1) La **generación de modelos dependientes de la tecnología** a partir de los diferentes artefactos software que componen la aplicación web heredada. Esta generación se ha

definido como transformaciones de texto a modelo y se han utilizado las herramientas proporcionadas por el proyecto MoDisco para este fin: los *discoverers*. Dentro de esta actividad, las contribuciones realizadas han sido:

- La definición de una extensión del metamodelo JSP proporcionado por MoDisco para incorporar las etiquetas definidas por las librerías de Struts 1 como elementos de primer nivel: metamodelo Struts-JSP.
- La definición de las reglas de transformación mediante ATL para refinar el modelo JSP generado por MoDisco a un modelo Struts-JSP.
- La definición de metamodelos para los archivos de configuración de Struts 1 y para los archivos de mapeo objeto-relacional de Hibernate 3.6.
- La implementación de dos discoverers nuevos para el proyecto MoDisco: uno, para la generación del modelo de un archivo de configuración de Struts 1, y otro para la generación de modelos a partir de los archivos de mapeo objeto-relacional de Hibernate 3.6. El primero de ellos ya ha sido contribuido al proyecto MoDisco. Para más información, revisar anexo B.

2) La **generación de un modelo conforme al metamodelo MIGRARIA MVC** a partir de los modelos dependientes de la tecnología mediante la definición de reglas de transformación modelo a modelo en ATL. Para más información, revisar anexo A. Las contribuciones realizadas han sido:

- La definición de patrones a localizar para la generación de los elementos del componente Modelo.
- La definición de patrones a localizar para la generación de los elementos del componente Vista.
- La definición de patrones a localizar para la generación de los elementos del componente Controlador.

3. Con el objetivo de evaluar y evolucionar la propuesta, se ha desarrollado un completo **caso de estudio** que ha incluido la implementación de una aplicación web heredada, como ejemplo de laboratorio, dentro de la pila tecnológica elegida y según las reglas de negocio definidas. Esta aplicación se ha desarrollado por la falta de aplicaciones web heredadas de acceso público que existe dentro de la pila tecnológica y el dominio elegido. Tanto el código fuente como una versión en producción de ejemplo de esta aplicación se ha puesto a disposición pública para que pueda ser utilizada como caso de estudio de otros trabajos.

4. Dentro del resto de partes del proceso de modernización, se han realizado trabajos siguiendo diferentes estrategias para valorar su viabilidad. Las contribuciones más destacadas son:

a) La definición de transformaciones modelo a modelo en ATL para **generar la especificación en WebML de la aplicación web heredada** a partir de su modelo MIGRARIA MVC. Los modelos WebML generados son compatibles con la herramienta WebRatio 5[1].

b) La definición de un **proceso sistemático de selección de patrones a partir de los requisitos RIA** definidos para el nuevo sistema.

c) La definición de un **proceso de composición mediante técnicas de weaving de modelos para poder aplicar patrones RIA a una aplicación web definida en WebML** sin alterar los modelos originales.

d) La definición de un **metamodelo de weaving** para especificar la composición de modelos WebML con modelos WebML4RIA que definan los patrones RIA a aplicar.

e) La **generación del código de una API REST para la conexión del cliente RIA a partir de la aplicación web heredada**. Las contribuciones de esta parte han sido:

1) La implementación de una extensión del framework de aplicaciones web Struts 1 para dar soporte a la definición de APIs REST usando el formato de intercambio de datos JSON[2].

2) La definición de transformaciones modelo a código mediante el proyecto JET[3] para la generación del archivo de configuración de esta capa de instrumentación a partir de la parte del modelo MIGRARIA MVC del subsistema de la aplicación web heredada seleccionado para modernizar.

En el siguiente apartado se aportan más detalles de algunas de las conclusiones obtenidas después de la realización de este trabajo.

9.2. Discusión

Este apartado discute algunas de las decisiones más relevantes tomadas en el desarrollo de este trabajo.

¿Sigue nuestra propuesta de arquitectura del sistema objetivo la estrategia "Sweeping it under the rug"? (Foote & Yoder 1997) definen "Sweeping it under the rug" como una de las alternativas para tratar con un sistema software de muy baja calidad, que los autores denominan literalmente "Big ball of mud" porque su mala implementación lo convierte en un "todo" muy difícil de comprender y de mejorar. En situaciones en las que es necesario proporcionar acceso a terceros o existe un límite de tiempo fuerte para realizar una extensión del sistema, los autores sugieren, como alternativa menos costosa, construir

[1] http://www.webratio.com/
[2] http://www.json.org/
[3] http://www.eclipse.org/modeling/m2t/?project=jet

una interfaz suficientemente completa al sistema de baja calidad que permita proporcionar ese acceso o realizar esa extensión reduciendo la complejidad y el acoplamiento, y favoreciendo el mantenimiento. Por lo tanto, "Sweeping it under the rug" propone que, si no se puede abordar la reconstrucción de un sistema de baja calidad en un tiempo determinado, al menos, se acordone ese sistema para simplificar tanto el desarrollo de la extensión como la reconstrucción evolutiva del sistema de baja calidad.

En lo referente a nuestra arquitectura del sistema objetivo, es necesario aclarar que esta propuesta viene condicionada por el objetivo de mantener el acceso al sistema a través de la capa Web definida. En definitiva, se trata de proporcionar dos formas de acceso e interacción diferentes al mismo sistema de información subyacente. Algunas compañías han decidido seguir una propuesta similar proporcionando un cliente RIA de acceso a sus sistemas y servicios, que convive con una interfaz web más tradicional, como los ejemplos aportados en la introducción de este trabajo. Esta propuesta de tener múltiples interfaces de acceso a un sistema de información es una solución actualmente muy extendida, sobre todo, con el despegue del acceso a internet desde dispositivos móviles.

Por otra parte, los casos que señalan los autores en (Foote & Yoder 1997), en nuestra opinión, no se corresponden de manera general con los casos comunes en nuestro dominio de aplicación; ya que, nuestro principal dominio de aplicación son sistemas empresariales que posean una interfaz web. Normalmente, la aparición de código de baja calidad es mucho más probable que se dé en la capa Web por sus especiales características de desarrollo (múltiples tecnologías, equipo multidisciplinar, presupuestos ajustados, tiempos de desarrollo muy cortos, etc.) que en el sistema subyacente que implementa la lógica de negocio. Por lo tanto, en nuestro caso, si se da *suciedad* en el sistema original, con mayor posibilidad ésta se encuentre justamente en el componente de la arquitectura original del que estamos generando una alternativa y, en este sentido, no estaríamos escondiendo esta *suciedad* sino proporcionando una alternativa. Por supuesto, es también posible que aparezca *suciedad* en el sistema de información subyacente. En esta situación, como medida de prevención el proceso de modernización debería incluir actividades que permitan evaluar la calidad del sistema subyacente, proporcionando métricas que permitan al ingeniero valorar adecuadamente la necesidad de llevar a cabo un proceso de reingeniería o de refactorización que lo subsane. Sin embargo, la definición de estas actividades quedan fuera del alcance de este trabajo.

¿En qué nivel de dominios de modernización, definidos por el modelo de herradura de ADM, se encuentra nuestra propuesta? El modelo de herradura de ADM define 3 niveles de dominio de modernización: tecnología, aplicación&datos y negocio. La propuesta aquí presentada se encuadraría dentro del segundo nivel (arquitectura de aplicación y datos) porque la modernización propuesta supone cambios sustanciales en la arquitectura original, más allá del plano tecnológico. Fundamentalmente, estos cambios están relacionados con:

- La estructura, composición y organización de la interfaz de usuario. La estructura multipágina de la capa de presentación de la aplicación original debe transformarse en una interfaz que siga el pa-

radigma single-page característico de clientes RIA. Es necesario definir un proceso que componga la nueva interfaz a partir de componentes RIA tomando como base los elementos de las páginas web originales.

- El modelo de interacción basado en el ciclo petición-respuesta debe ser transformado a un modelo basado en eventos. En un cliente RIA la navegación no se concibe como una secuencia de páginas hiperenlazadas, sino como una secuencia de transiciones de estados de la interfaz dirigida por eventos. Además, no todas las transiciones conllevan una petición al servidor. Una transición de estado de la interfaz de usuario se puede definir básicamente como la actualización de los componentes gráficos visibles actualmente o como la carga de nuevos componentes.

- El modelo de comunicación entre cliente y servidor también se ve afectado. Los clientes RIA interactúan frecuentemente con la lógica de negocio del servidor mediante un modelo orientado a servicios, que puede implicar cambios sustanciales en la interfaz de acceso a la lógica de negocio. Además, los clientes RIA suelen preponderar un modelo de comunicación asíncrona, sobre todo, para mantener una interfaz de usuario activo. Una de las características más interesantes de las RIAs es la posibilidad de que el usuario altere manual o automáticamente el modo de funcionamiento de la aplicación entre modo conectado y desconectado.

¿Por qué definir una nueva notación para representar aplicaciones web y no usar alguna de las propuestas en la comunidad IWDM? Las razones para especificar una nueva notación son múltiples. Básicamente, esta notación está concebida para ser usada en escenarios de modernización, mientras que las propuestas IWDM están fundamentalmente concebidas para la generación de aplicaciones web mediante ingeniería directa (generación de código). Estos escenarios de uso son muy diferentes e imponen decisiones de diseño diferentes para la notación. En el caso de escenarios de modernización, es necesario disponer de una notación mucho más cercana a la tecnología de implementación; ya que, por un lado, la información disponible está sujeta a las limitaciones de la propia tecnología y, por otro lado, dado el dominio de aplicaciones web heredadas en el que se encuadra este trabajo, posiblemente sea más probable contar con profesionales con gran experiencia en tecnologías Web pero sin experiencia en el uso de una propuesta IWDM determinada. Además, debe ser una notación flexible que permita representar sistemas parcialmente pero manteniendo gran parte de su semántica; ya que, la información recuperada por medio de ingeniería inversa puede no ser completa.

¿Por qué definir una nueva notación para representar aplicaciones web en vez de usar el estándar KDM[4] de la OMG? En los experimentos iniciales del trabajo se utilizó KDM como notación conceptual objetivo del proceso de ingeniería inversa. Sin embargo, en etapas posteriores nos dimos cuenta de que, por un lado, KDM presentaba una notación bastante limitada en niveles superiores al nivel de

[4] http://www.omg.org/spec/KDM/

código y, por otro lado, los modelos de KDM generados eran complejos y extensos introduciendo mayor complejidad a nuestro proceso de modernización que una notación más cercana al dominio de aplicación, como la que presentamos aquí.

¿Cuál es el grado de reutilización del proceso de ingeniería inversa propuesto? El proceso de ingeniería inversa se ha diseñado como una cadena de transformaciones en dos niveles distintos: el nivel dependiente y el independiente de las tecnologías de implementación.

Las actividades y herramientas definidas en el primer nivel tienen como objetivo obtener los modelos dependientes de las tecnologías de implementación. Por lo tanto, su reutilización está limitada a escenarios de modernización dentro de la misma pila tecnológica. Para abordar escenarios de modernización fuera de la misma, es necesario redefinir las actividades y artefactos de este nivel afectados. Básicamente, se pueden considerar dos casos:

1. Se modifica una de las tecnologías de implementación base. Por ejemplo, se utiliza un lenguaje de programación distinto a Java para definir los controladores.

2. Se modifica el framework de aplicación utilizado manteniendo intacta la pila tecnológica base. Por ejemplo, se utiliza el framework Spring MVC en vez de Struts 1.

En ambos casos, las modificaciones suponen sustituir (o añadir) la infraestructura necesaria (metamodelos y discoverer) para levantar modelos a partir de los ficheros fuente que han cambiado.

Las actividades y herramientas definidas en el segundo nivel tienen como objetivo obtener un modelo MIGRARIA MVC, independiente de la tecnología de implementación. Estas actividades toman como fuente los modelos obtenidos en el nivel anterior; por lo tanto, se ven afectadas por los posibles cambios que se puedan dar en el primer nivel. Fundamentalmente, es necesario adaptar las reglas de transformación a los nuevos patrones que se pretenden detectar. Por ejemplo, la información que se mantiene en los ficheros de configuración de distintos FAWs puede variar considerablemente; por lo tanto, es necesario adaptar las reglas de transformación en cada caso para poder recuperar la información necesaria para la generación del modelo MIGRARIA_MVC. A este nivel, la reutilización se ve comprometida, no sólo por cambios en las tecnologías de implementación, sino por cambios en las reglas de negocio o convenios de programación utilizados en el desarrollo de la aplicación web heredada. De esta manera, aunque la pila tecnológica se mantenga inalterada, es necesario extender o adaptar las reglas de transformación para identificar los patrones que se desprenden de los nuevos convenios utilizados.

En conclusión, tanto los cambios en la pila tecnológica como los cambios en los estilos o convenios de programación afectan negativamente a la reutilización del proceso de ingeniería inversa definido. En este sentido, una de las líneas de trabajo futuro intenta explorar posibles soluciones a este problema de variabilidad del origen y adaptación de las reglas de transformación. Desde un punto de vista industrial, sin embargo, es común disponer de un parque de aplicaciones web heredadas amplio dentro de la misma

pila tecnológica y en base a los mismos convenios de programación; por lo tanto, el esfuerzo necesario para adaptar el proceso, si fuese necesario, se ve justificado por el posible ahorro posterior al tener que procesar un número importante de sistemas heredados.

9.3. Conclusiones

A continuación se presentan las conclusiones fundamentales derivadas de este trabajo de tesis. En primer lugar, se revisan las ventajas e inconvenientes del uso de técnicas y herramientas propias de la IDM para la definición de procesos de modernización. En segundo lugar, se exponen las conclusiones derivadas directamente del desarrollo del trabajo.

9.3.1. Conclusiones del uso de IDM en la modernización

En este apartado, se utilizan las reflexiones presentadas por (Sánchez-Ramón *et al.* 2013) sobre las ventajas e inconvenientes de la definición de un proceso de modernización dirigido por modelos como marco para organizar las conclusiones derivadas de la experiencia del desarrollo de este trabajo.

9.3.1.1. Ventajas

1. **Productividad y rentabilidad**. Los autores señalan, por un lado, que uno de los beneficios bien conocidos que se obtiene al aplicar las técnicas IDM es el incremento de la productividad en el desarrollo, como consecuencia de la automatización que se introduce en la generación de los artefactos del sistema final. Y señalan, por otro lado, que el uso de IDM se hace más rentable cuanto mayor sea el número de artefactos cuya migración sea repetitiva siguiendo unas pautas bien definidas, y cuanto mayor sea el número de aplicaciones del mismo tipo a migrar. Si bien coincidimos con esta reflexión, creemos que también es necesario tener en cuenta la calidad de los artefactos generados para evaluar el esfuerzo que les supone a un equipo de desarrolladores entenderlos y utilizarlos. Este factor es más relevante cuanto mayor grado de intervención humana sea necesario en la adecuación o uso de los artefactos finales generados.

2. **Metamodelos, un formalismo unificador e integrador**. En este punto coincidimos con los autores con la idoneidad del uso de lenguajes de metamodelado como Ecore o MOF para representar el conocimiento extraído en un proceso de ingeniería inversa. Su mayor expresividad, la existencia de potentes lenguajes de transformación de modelos, la posibilidad de definir restricciones y la separación entre sintaxis abstracta y concreta los posicionan por encima de otras opciones, como XML. En el desarrollo de este trabajo, se ha definido un metamodelo Ecore que ha posibilitado la especificación del proceso de extracción de conocimiento como una cadena de transformaciones de modelos.

3. **Independencia de la tecnología origen y destino**. Coincidimos con los autores que este es un factor fundamental en la definición de un proceso de modernización. Y también hemos seguido la aproximación de definir una representación que desacople las tecnologías origen y destino del resto del proceso. En este trabajo se ha definido el metamodelo MIGRARIA MVC que especifica una notación conceptual para aplicaciones web desarrolladas con frameworks MVC independientemente de las tecnologías utilizadas en la implementación. La independencia se consigue con inyectores y transformaciones M2M que generan los modelos independientes del origen, y mediante transformaciones M2M y M2T a partir de los modelos independientes, para obtener el destino. Finalmente, coincidimos también en que buena parte del valor de una solución de modernización dirigida por modelos reside en la definición de estos metamodelos, que requiere gran capacidad de abstracción y buen conocimiento de las diferentes tecnologías. Tanto es así, que el estudio de frameworks de aplicaciones web conformes al patrón MVC, la definición del metamodelo MIGRARIA MVC y su validación han acaparado gran parte del esfuerzo invertido en esta tesis

4. **Extensibilidad**. Un proceso de modernización puede ser extendido con nuevas etapas o incluso ser modificado con etapas que reemplazan a las existentes. En una cadena de transformación, los modelos constituyen un punto de extensión válido para incorporar nuevas etapas. Para extender una cadena con una nueva etapa basta con incluir una cadena de transformación que tenga como entrada el modelo que actúa como punto de extensión y como resultado un modelo que pueda ser incorporado al proceso de modernización en cualquier punto. Esta característica es fundamental para poder adaptar el proceso de modernización a diferentes escenarios de aplicación. Por ejemplo, en este trabajo se plantea como extensión en la etapa de reestructuración la transformación de los modelos MIGRARIA MVC a modelos WebML para poder incorporar las herramientas de esta aproximación en el proceso de modernización.

5. **Reutilización**. Los fragmentos de las cadenas de transformación pueden reutilizarse definiendo metamodelos que desacoplen unas partes de otras, y definiendo las transformaciones hacia y desde esos metamodelos. Este mecanismo se puede utilizar para reutilizar metamodelos y transformaciones de terceros, mediante transformaciones M2M que conviertan nuestros modelos en modelos conformes al metamodelo de terceros. En este punto, aunque coincidimos con el razonamiento aportado por los autores, sólo lo hemos experimentado por ahora para lograr la independencia entre origen y destino. Si bien, dentro de las líneas de trabajo futuro está aprovechar cadenas de transformación definidas desde WebML, como por ejemplo para su transformación a RUX Method (Linaje *et al.* 2007).

6. **Interoperabilidad con aplicaciones de terceros**. Los modelos constituyen un punto de extensión que facilita la integración con aplicaciones de terceros, aunque sean cerradas y no estén implementadas con un enfoque MDE. Una aplicación de terceros que requiera una entrada serializada y

genere una salida serializada puede ser integrada en una cadena de transformación. Desde nuestro punto de vista, ésta es una ventaja fundamental porque permite enriquecer el proceso de modernización con herramientas no consideradas a priori pero que aportan gran valor. Nuestra experiencia en este sentido se limita al uso de WebRatio que es la herramienta de desarrollo definida para WebML.

7. **Fácil obtención de modelos a partir de artefactos software**. Un proceso de modernización requiere la construcción de analizadores sintácticos (parsers) que analicen los artefactos de la aplicación existente. Estamos totalmente de acuerdo que la existencia de DSLs para la inyección de modelos, como Gra2MoL (Cánovas Izquierdo & García-Molina 2008) y Schemol (Cánovas Izquierdo *et al.* 2011), y de frameworks como MoDisco (Bruneliere *et al.* 2010) facilita considerablemente el trabajo ya que evita la creación de tales analizadores manualmente. Como ya se ha señalado anteriormente, en el desarrollo de este trabajo hemos usado MoDisco principalmente por los inyectores (discoverers) que ya tenía definidos. Adicionalmente, hemos desarrollado algunos inyectores nuevos para artefactos no contemplados por MoDisco. El proceso de desarrollo de estos inyectores se ha visto claramente simplificado por la existencia de MoDisco.

8. **Integración de espacios tecnológicos**. Un espacio tecnológico es un contexto de trabajo que tiene asociado un conjunto de conceptos, métodos, técnicas y herramientas (Kurtev *et al.* 2002) y que se construye en torno a un par de conceptos básicos que se relacionan a través de una relación instancia-de. En la modernización de software frecuentemente están involucrados diferentes espacios tecnológicos que es necesario integrar. Los inyectores de modelos permiten transformar artefactos de cualquier espacio tecnológico en modelos, aplicar transformaciones M2M y entonces generar artefactos del mismo o diferente espacio tecnológico. En este punto coincidimos plenamente con la descripción y el razonamiento desarrollados por los autores.

9. **Decoración de modelos y trazabilidad**. Desde el enfoque IDM, es posible definir un metamodelo para los aspectos comunes y separar en diferentes metamodelos los aspectos variables. De este modo, un modelo puede ser anotado o decorado con otros modelos para añadirle cierta información variable. Durante el proceso de reingeniería se aplican alteraciones al sistema original que frecuentemente suelen omitir detalles del sistema original. Además, se pierde la relación (traza) entre los elementos creados en un modelo a partir de los elementos de otros. En ocasiones es necesario mantener esta información de trazabilidad para utilizarla en posteriores transformaciones. El mecanismo de decoración puede ser empleado para mantener la trazabilidad del proceso de modernización por medio de una solución no intrusiva, que son los modelos de trazas. En el desarrollo de este trabajo, aunque no se presenta en esta memoria, se ha definido un modelo de traza muy sencillo entre los elementos del modelo MIGRARIA MVC y los elementos originales

presentes en los modelos dependientes de la tecnología. Este modelo se puede generar a la vez que el modelo de MIGRARIA MVC.

9.3.1.2. Inconvenientes

1. **Insuficiencia de detalle de KDM**. KDM (Knowledge Discovery Metamodel) es el metamodelo central de ADM el cual está destinado a representar los artefactos software de una aplicación de forma independiente a las plataformas usadas, estableciendo varios niveles de abstracción. Fue ideado principalmente para soportar la interoperabilidad entre herramientas de modernización. En este punto, como ya se ha señalado anteriormente, coincidimos con los autores en la carencia de expresividad suficiente en niveles superiores al de código, como por ejemplo el de interfaces de usuario. Y añadimos como inconveniente, además, la complejidad y extensión de los modelos generados, que complican procesos de transformación posteriores.

2. **Dificultad para trabajar con modelos de gran escala**. En el escenario de un proceso de modernización, con frecuencia se requiere manipular modelos de gran escala. Dado que XMI mostró importantes limitaciones para trabajar con modelos de gran escala, han aparecido herramientas basadas en el uso de bases de datos relacionales como CDO[5]. Pero como la naturaleza del modelo relacional se ha demostrado inadecuada para tratar el problema (Barmpis & Kolovos 2012), han surgido nuevas propuestas de repositorios de modelos, como Morsa (Espinazo PagÃ¡n *et al.* 2011) que emplean bases de datos basadas en documentos y otros prototipos basados en grafos (Barmpis & Kolovos 2012). Aunque coincidimos con los autores en señalar la relevancia de este inconveniente, dentro del desarrollo de este trabajo no hemos llegado a manejar modelos de esa escala y, por lo tanto, no podemos derivar ninguna conclusión en este sentido de nuestra experiencia.

3. **Carencia de inyectores para lenguajes de programación**. Existen repositorios[6] que ofrecen numerosos metamodelos para lenguajes de programación ampliamente extendidos como PL/1 y Cobol, pero no existen inyectores disponibles de manera abierta y gratuita para estos lenguajes, por lo que suele ser necesario implementarlos. En este punto, coincidimos totalmente con los autores en que la existencia de un repositorio de inyectores para estos lenguajes facilitaría el uso de IDM en los procesos de modernización.

4. **Carencia de repositorios de transformaciones directas**. A menudo es interesante realizar transformaciones entre representaciones para poder aprovechar ciertas ventajas asociadas a la representación destino. Por ejemplo, para aprovechar los razonadores existentes para ontologías se requieren conversiones entre los formalismos implicados, en este caso entre modelos Ecore y ontologías OWL. Estas transformaciones son en gran medida correspondencias directas que no implican de-

[5] http://www.eclipse.org/cdo/
[6] http://www.emn.fr/z-info/atlanmod/index.php/Zoos

cisiones de diseño por parte del usuario. Volvemos a coincidir plenamente con los autores en que sería muy beneficioso disponer de un repositorio con estas transformaciones para metamodelos estándar.

5. **Inexistencia de soluciones predefinidas para ingeniería inversa**. Durante el análisis estático de código, se realizan ciertas tareas previas como la eliminación de código muerto o la identificación de bloques básicos. Dichas tareas son relativamente generales y pueden ser aplicadas a un gran número de proyectos de modernización. En este sentido, estamos de acuerdo con los autores que sería de gran valor disponer de soluciones predefinidas en forma de transformaciones que se apoyen en metamodelos estándar. La existencia de los tres tipos de repositorios identificados en estos tres últimos inconvenientes para metamodelos estándar, como por ejemplo KDM, aportarían un valor específico muy importante al estándar animando su utilización real en procesos de modernización.

6. **Carencia de estructuras de datos complejas en lenguajes de transformación**. Frecuentemente cuando se realiza reingeniería del software es necesario tratar con algoritmos relativamente complejos que requieren de estructuras de datos complejas. Se necesita de un lenguaje de transformación de modelos capaz de definir estructuras de datos complejas y/o que proporcione tipos de datos complejos. Además, estos lenguajes deberían ser capaces de expresar algoritmos de cualquier complejidad. No todos los lenguajes de transformación son, por tanto, apropiados para la reingeniería. En este punto, aunque entendemos el planteamiento de los autores, no podemos derivar nada de nuestra experiencia porque sólo hemos usado un lenguaje de transformación en el desarrollo de este trabajo. Si bien es verdad que nos hemos encontrado con problemas de modularidad y de expresividad en el desarrollo de las transformaciones, no podemos concluir con seguridad que se trate de una limitación del lenguaje.

9.3.2. Conclusiones del desarrollo

En cuanto al alcance del trabajo desarrollado, la definición e implementación del proceso de ingeniería inversa mediante la especificación de una cadena de transformaciones se ha acabado convirtiendo en el trabajo fundamental de esta tesis. En primer lugar, conseguir una notación suficientemente expresiva de los conceptos relacionados con un FAW basado en el patrón MVC ha supuesto un esfuerzo importante de varios meses de trabajo. A partir de los trabajos presentados en (Rodríguez-Echeverría *et al.* 2012c;a;b), se ha realizado una redefinición completa de la notación siguiendo un proceso iterativo hasta obtener una versión que fuese capaz de cumplir con los objetivos marcados por el caso de estudio. En paralelo, se ha tenido que abordar la extensión del proyecto MoDisco para poder obtener los modelos de todos los artefactos software implicados en la implementación de la aplicación web heredada del caso de estudio. Posteriormente, una vez obtenida una versión suficientemente expresiva y resueltos todos los problemas de generación de los modelos dependientes de la tecnología, se ha procedido a definir el

proceso de análisis mediante reglas de transformación. La definición de las reglas de generación de los componentes Modelo y Vista se pudieron abordar de forma satisfactoria dentro de las estimaciones de tiempo previstas. Sin embargo, en el caso del componente Controlador, las estimaciones de tiempo se han visto superadas ampliamente, debido a la complejidad del lenguaje a analizar en este caso con respecto a los otros dos componentes y a la naturaleza dinámica de un controlador frente a la estática de un objeto de datos o una plantilla de página. En este sentido, la gran diversidad de patrones a analizar para cada uno de los conceptos a extraer ha supuesto la necesidad de limitar estrictamente el número de casos a contemplar para la definición de controladores. Estas limitaciones permiten definir el camino a seguir en la extensión del proceso de ingeniería inversa. Todas estas tareas junto con la definición, implementación y evaluación del caso de estudio han impedido desarrollar en profundidad los trabajos de generación de aplicaciones RIA. Es por ello que en esta memoria estos trabajos se presentan de forma simplemente ilustrativa para exponer líneas de trabajo que se encuentran actualmente en desarrollo.

En lo referente al enfoque seguido en este trabajo, desde el principio, todo el trabajo realizado dentro de esta tesis ha presentado un marcado enfoque práctico, demandado por los propios objetivos del mismo. Este planteamiento ha supuesto, por una parte, una experiencia muy enriquecedora porque ha requerido no sólo el estudio sino el conocimiento en profundidad de herramientas y tecnologías múltiples y heterogéneas, como por ejemplo el framework de aplicaciones web Struts 1 o el lenguaje de transformación de modelos ATL, por citar dos ejemplos opuestos en el espectro de tecnologías utilizadas dentro del trabajo. Sin embargo, por otra parte, ha supuesto también un esfuerzo muy importante; ya que, cada paso dado ha conllevado, primero, conocer la tecnología a utilizar de manera práctica y, después, realizar extensiones para proporcionar la funcionalidad no disponible. La gestión de este esfuerzo ha sido posible gracias a la organización del trabajo en equipo puesta en marcha en las últimas fases de este trabajo. Desde el punto de vista de los resultados, la principal bondad de este enfoque práctico, en cualquier caso, consiste en la posibilidad de contrastar los resultados obtenidos en todo momento para poder mejorar la propuesta de manera constante siguiendo una metodología de desarrollo iterativa.

9.4. Trabajo futuro

El trabajo desarrollado dentro de esta tesis, como ya se ha puesto de manifiesto, forma parte de un proyecto mayor; por lo tanto, parte de las líneas de trabajo futuro pasan por los objetivos definidos dentro del mismo. En este sentido, este apartado se organiza en dos partes. La primera comenta las líneas de trabajo futuro para evolucionar la propuesta presentada. Mientras que la segunda muestra las líneas de trabajo futuro para extender la propuesta en línea con los objetivos planteados en el proyecto.

9.4.1. Evolución de la propuesta

Las líneas de trabajo prioritarias en la evolución de la propuesta se centran en la mejora del proceso de ingeniería inversa definido. En este sentido, se planea:

- El desarrollo de más casos de estudio que permitan la identificación de un mayor número de patrones que enriquezca el proceso de extracción de conocimiento.
- La incorporación en la notación de conceptos relacionados con el uso de la sesión, la autenticación y la autorización.
- La definición de una sintaxis concreta para la notación definida, MIGRARIA MVC, de manera que disminuya tanto el esfuerzo de su uso como su curva de aprendizaje. La notación debe resultar natural para un desarrollador de aplicaciones web mediante frameworks basados en el patrón MVC.
- El desarrollo de una herramienta que integre todas las actividades relacionadas con el proceso de ingeniería inversa de la aplicación web heredada. Esta herramienta debe incorporar también funcionalidades para simplificar las tareas del ingeniero de modernización.
- La exploración de posibles técnicas y herramientas para simplificar la adaptación (y evolución) de las reglas de transformación a escenarios en los que varían las tecnologías de implementación o los convenios de programación utilizados.

9.4.2. Extensión de la propuesta

Las líneas de trabajo prioritarias en la extensión de la propuesta se centran en el desarrollo de trabajos relacionados con las partes no abordadas por esta tesis. En este sentido, se planea:

- La definición de análisis dinámicos de la aplicación web heredada desde la parte cliente. El conocimiento obtenido debe permitir completar el obtenido mediante el proceso definido en este trabajo.
- La definición de procesos de minería de datos que permitan obtener información real de comportamiento de la aplicación web heredada mediante los análisis de los registros de ejecución. El objetivo consiste en poder utilizar el conocimiento extraído para guiar actividades posteriores del proceso de modernización, como por ejemplo la selección de patrones RIA, la composición concreta de la interfaz del cliente RIA a generar o la identificación de los datos que se pueden almacenar en el cliente.
- El desarrollo de trabajos de conexión con herramientas que permitan la generación final de código RIA, como por ejemplo RUX Tool, (Linaje *et al.* 2009), y OOH4RIA (Meliá *et al.* 2008).

Capítulo 10

Publicaciones

Este capítulo presenta las publicaciones realizadas a lo largo del desarrollo de esta tesis.

10.1. Revistas indexadas

1. Conejero Manzano, José Mª, ROBERTO RODRÍGUEZ ECHEVERRÍA, Fernando Sánchez Figueroa, Marino Linaje Trigueros, Juan Carlos Preciado Rodríguez, and Pedro José Clemente Martín. 2013. "Re-engineering legacy Web applications into RIAs by aligning modernization requirements, patterns and RIA features." Journal of Systems and Software. JCR: 1,140 (publicación pendiente)

2. Conejero Manzano, José Mª, Pedro José Clemente Martín, ROBERTO RODRÍGUEZ ECHEVERRÍA, Juan María Hernández Núñez, and Fernando Sánchez Figueroa. 2013. "A Model-Driven Approach for Reusing Tests in Smart Home Systems." Personal and Ubiquitous Computing, 15, 317-327. JCR: 1,554

3. Linaje, Marino, Adolfo Lozano-Tello, Miguel A. Perez-Toledano, Juan Carlos Preciado, ROBERTO RODRIGUEZ-ECHEVERRIA, and Fernando Sanchez-Figueroa. 2011. Providing RIA user interfaces with accessibility properties. Journal of Symbolic Computation 46 (2) (February): 207-217. JCR: 0,853

10.2. Congresos

1. RODRÍGUEZ-ECHEVERRÍA, ROBERTO, Fernando Macías, Víctor M. Pavón, José María Conejero, Pedro J. Clemente, and Fernando Sánchez-Figueroa. 2013. "Generating a Conceptual Representation of a Legacy Web Application." In Proceedings of 14th International Conference on Web Information System Engineering. Nanjing, China. (publicación pendiente)

2. RODRÍGUEZ-ECHEVERRÍA, ROBERTO, Fernando Macías, Víctor M. Pavón, José María Conejero, and Fernando Sánchez-Figueroa. 2013. "Generating a REST Service Layer from a Legacy

System." In Proceedings of 22nd International Conference on Information System Development. Seville, Spain. (publicación pendiente)

3. RODRÍGUEZ-ECHEVERRÍA, ROBERTO, Fernando Macías, Víctor M. Pavón, José María Conejero, and Fernando Sánchez-Figueroa. 2013. "Model-Driven Generation of a REST API from a Legacy Web Application." Current Trends in Web Engineering (publicación pendiente).

4. Fernando Macías, Víctor M. Pavón, ROBERTO RODRIGUEZ-ECHEVERRIA, and Fernando Sánchez-Figueroa. 2013. "Generación dirigida por modelos de una API REST para una aplicación Web heredada." In Proceedings of XVIII Jornadas De Ingeniería Del Software y Bases De Datos, 255-261. Madrid, Spain.

5. RODRÍGUEZ-ECHEVERRÍA, ROBERTO, José María Conejero, Pedro J. Clemente, María D. Villalobos, and Fernando Sánchez-Figueroa. 2012. "Extracting Navigational Models from Struts-based Web Applications." In International Conference on Web Engineering, ed. Marco Brambilla, Takehiro Tokuda, and Robert Tolksdorf, 7387:419–426. Berlin: Springer Berlin Heidelberg.

6. RODRÍGUEZ-ECHEVERRÍA, ROBERTO, José María Conejero, Pedro J Clemente, María D. Villalobos, and Fernando Sánchez-Figueroa. 2012. "Generation of WebML Hypertext Models from Legacy Web Applications." In 14th IEEE International Symposium on Web Systems Evolution, 91–95. Riva del Garda (Italy): IEEE Computer Society.

7. RODRÍGUEZ-ECHEVERRÍA, ROBERTO, José María Conejero, Pedro J Clemente, Juan Carlos Preciado, and Fernando Sánchez-Figueroa. 2012. "Modernization of Legacy Web Applications into Rich Internet Applications." Current Trends in Web Engineering 7059/2012 (Lecture Notes in Computer Science): 236–250.

8. RODRÍGUEZ-ECHEVERRÍA, ROBERTO, José María Conejero, Pedro J Clemente, Juan Carlos Preciado, and Fernando Sánchez-Figueroa. 2012. Modernization of Legacy Web Applications into Rich Internet Applications. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics, ed. Andreas Harth and Nora Koch, 236-250. Paphos, Cyprus: Springer Verlag.

9. RODRÍGUEZ-ECHEVERRÍA, ROBERTO, José María Conejero, Marino Linaje, Juan Carlos Preciado, and Fernando Sanchez-Figueroa. 2010. "Re-engineering Legacy Web Applications into Rich Internet Applications." In 10th International Conference on Web Engineering, ed. Boualem Benatallah, Fabio Casati, Gerti Kappel, and Gustavo Rossi, 6189:189–203. Vienna: Springer Berlin Heidelberg.

10. Linaje, Marino, Adolfo Lozano-Tello, Juan Carlos Preciado, Fernando Sanchez-Figueroa, and ROBERTO RODRIGUEZ-ECHEVERRIA. 2009. Obtaining accessible Rich Internet Applications by

combining RUX-Method and SAW. In 5th International Workshop on Automated Specification and Verification of Web Systems. Castle of Hagenberg, Austria.

11. Linaje, Marino, Juan Carlos Preciado, Rober Morales-Chaparro, ROBERTO RODRIGUEZ-ECHEVERRIA, and Fernando Sanchez-Figueroa. 2009. Automatic Generation of RIAs Using RUX-Tool and Webratio. In International Conference on Web Engineering, ed. Martin Gaedke, Michael Grossniklaus, and Oscar Díaz, 5648:501-504. Berlin, Heidelberg: Springer Berlin Heidelberg, June.

12. Lozano-Tello, Adolfo, Alvaro E Prieto, and ROBERTO RODRIGUEZ-ECHEVERRIA. 2007. "Model Integration on MDA Based on Ontology Merging Method." In New Perspectives on Systems and Information Technology Vol 1 Actas De 2a Conferencia Iberica De Sistemas y Tecnologias De Informacao, 197–210. Edicoes Univ Fernando Pessoa.

13. RODRIGUEZ-ECHEVERRIA, ROBERTO, Fernando Sanchez-Figueroa, José María Conejero, and Javier Pedrero. 2005. Modeling Reusable Web Business Processes from an Aspect Oriented Perspective. IEEE Latin America Transactions 3 (1): 98-103.

14. RODRIGUEZ-ECHEVERRIA, ROBERTO, and Fernando Sanchez-Figueroa. 2004. "Modelando Procesos De Negocio Web Desde Una Perspectiva Orientada a Aspectos." In X Jornadas De Ingeniería Del Software y Bases De Datos. Málaga, Spain.

15. RODRIGUEZ-ECHEVERRIA, ROBERTO, and Fernando Sanchez-Figueroa. 2003. "Modelado Orientado a Aspectos De Patrones Hipermedia." In VIII Jornadas De Ingeniería Del Software y Bases De Datos. Alicante, Spain.

16. Conejero, José María, Pedro J Clemente, ROBERTO RODRIGUEZ-ECHEVERRIA, Juan Hernández, and Fernando Sanchez-Figueroa. 2011. "A Model-driven Approach for Reusing Tests in Smart Home Systems." Personal and Ubiquitous Computing: 1–11–11.

17. González, Julia, Mercedes Macias, ROBERTO RODRIGUEZ-ECHEVERRIA, and Fernando Sanchez-Figueroa. 2003. "Accessibility Metrics of Web Pages for Blind End-users." In 3rd International Conference on Web Engineering, 2722/2003:374–383. Oviedo, Spain: Springer Berlin Heidelberg.

Bibliografía

Alur, Deepak, Malks, Dan, & Crupi, John. 2001. *Core J2EE Patterns: Best Practices and Design Strategies*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

Antoniol, G., Di Penta, M., & Zazzara, M. 2004. Understanding web applications through dynamic analysis. *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.*, 120–129.

Antoniol, Giuliano, Canfora, Gerardo, Casazza, G., & Lucia, A. De. 2000. Web Site Reengineering using RMM. *Pages 9—-16 of: Proc. of the International Workshop on Web Site Evolution*.

Atzeni, Paolo, & Parente, Alessio. 2001. Specification of Web applications with ADM-2. *In: First International Workshop on Web-Oriented Software Technology, IWWOST*.

Banker, RD Rajiv D, Datar, SM Srikant M, Kemerer, Chris F, & Zweig, Dani. 1993. Software complexity and maintenance costs. *Commun. ACM*, **36**(11), 81–94.

Barmpis, Konstantinos, & Kolovos, Dimitrios S. 2012. Comparative analysis of data persistence technologies for large-scale models. *Pages 33–38 of: Proceedings of the 2012 Extreme Modeling Workshop on - XM '12*. New York, New York, USA: ACM Press.

Benedusi, P, Cimitile, A, & De Carlini, U. 1992. Reverse engineering processes, design document production, and structure charts. *Journal of Systems and Software*, **19**(3), 225–245.

Bergey, John, Smith, D, Weiderman, N, & Woods, S. 1999. Options Analysis for Reengineering (OAR): Issues and Conceptual Approach.

Bergey, John, O'Brien, L, & Smith, D. 2001. *Options Analysis for Reengineering (OAR): A Method for Mining Legacy Assets*. Tech. rept. June. Carnegie Mellon University.

Bermudez, Fco Javier, & García-Molina, Jesús. 2012. Un framework basado en modelos para la modernización de datos. *In: JISBD 2012*.

Bezivin, J, Barbero, M, & Jouault, F. 2007. On the Applicability Scope of Model Driven Engineering. *Pages 3–7 of: Model-Based Methodologies for Pervasive and Embedded Software, 2007. MOMPES '07. Fourth International Workshop on*.

Boldyreff, C., & Kewish, R. 2001. Reverse engineering to achieve maintainable WWW sites. *Proceedings Eighth Working Conference on Reverse Engineering*, 249–257.

Bouillon, L., & Vanderdonckt, J. 2002. Retargeting of Web Pages to Other Computing Platforms with VAQUISTA. *Page 339 of: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*. Washington, DC, USA: IEEE Computer Society.

Bozzon, Alessandro, Comai, Sara, Fraternali, Piero, & Carughi, Giovanni Toffetti G.T. 2006. Conceptual modeling and code generation for rich internet applications. *International Conference On Web Engineering; Vol. 263*, 360.

Bozzon, Alessandro, Comai, Sara, Fraternali, Piero, & Carughi, Giovanni Toffetti. 2009. Towards Web 2.0 Applications: A Conceptual Model for Rich Internet Applications. *Pages 75–95 of: Handbook of Research on Web 2.0, 3.0, and X.0*. IGI Global.

Brambilla, Marco, Fraternali, Piero, & Molteni, Emanuele. 2009. A Tool for Model-Driven Design of Rich Internet Applications Based on AJAX. *Pages 96–118 of:* Murugesan, San (ed), *Handbook of Research on Web 2.0, 3.0, and X.0*. IGI Global.

Brambilla, Marco, Cabot, Jordi, & Wimmer, Manuel. 2012. *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers.

Bruneliere, Hugo, Cabot, Jordi, & Jouault, Frédéric. 2010. MoDisco : A Generic And Extensible Framework For Model Driven Reverse Engineering. *Pages 1–2 of: IEEE/ACM International Conference on Automated Software Engineering*.

Cachero, Cristina, Gómez, Jaime, Párraga, Antonio, & Pastor, Oscar. 2001. Conference review system: A case of study. *In: First International Workshop on Web-Oriented Software Technology, IWWOST*.

Cachero Castro, Cristina. 2003. *OO-H: Una Extensión para el Modelado y Generación Automática de Interfaces Hipermediales*. Ph.D. thesis, Universidad de Alicante.

Calero, Coral, Ruiz, J, & Piattini, Mario. 2005. Classifying web metrics using the web quality model. *Online Information Review*, **29**(3), 227 – 248.

Cánovas Izquierdo, Javier Luis, & García-Molina, Jesús. 2008. Gra2MoL: Una Herramienta para la Extracción de Modelos en Modernization de Software. *In: XIII Jornadas de Ingeniería del Software y Bases de Datos*.

Cánovas Izquierdo, Javier Luis, & García Molina, Jesús. 2012. Extracting models from source code in software modernization. *Software & Systems Modeling*, 1–22.

Cánovas Izquierdo, Javier Luis, Díaz, Oscar, Puente, Gorka, & García-Molina, Jesús. 2011. ScheMoL: Un lenguaje especifico del dominio para extraer modelos de bases de datos relacionales. *In: XVI Jornadas en Ingeniería del Software y Bases de Datos (JISBD)*.

Carughi, GT Giovanni, Comai, Sara, Bozzon, Alessandro, & Fraternali, Piero. 2007. Modeling distributed events in data-intensive rich internet applications. *Pages 593–602 of: Web Information Systems Engineering*. Lecture Notes in Computer Science, vol. 4831. Berlin, Heidelberg: Springer Berlin Heidelberg.

Ceri, Stefano, Fraternali, Piero, & Bongio, Aldo. 2000. Web Modeling Language (WebML): a modeling language for designing Web sites. *Computer Networks*, **33**(1), 137–157.

Ceri, Stefano, Fraternali, Piero, Matera, Maristella, & Maurino, Andrea. 2001. Designing multi-role, collaborative Web sites with WebML: a conference management system case study. *In: First International Workshop on Web-Oriented Software Technology, IWWOST*.

Chikofsky, EJ, & Cross, JH. 1990. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, **7**(1), 13–17.

Clarke, S.A., & Baniassad, E.A. 2005. *Aspect-Oriented Analysis And Design: The Theme Approach*. Object Technology Series. Addison-Wesley Professional.

Comai, Sara, & Carughi, G.T. 2007. A behavioral model for rich internet applications. *Pages 364–369 of: Proceedings of the 7th International Conference on Web engineering*. Springer-Verlag.

Conallen, J. 1999. *Building Web Applications with UML*. Addison-Wesley.

Conejero, José M., Rodríguez-Echeverría, Roberto, Sánchez-Figueroa, Fernando, Linaje, Marino, Preciado, Juan C., & Clemente, Pedro J. 2013. Re-engineering legacy Web applications into RIAs by aligning modernization requirements, patterns and RIA features. *Journal of Systems and Software*, **pending**(May).

Conejero, José María, Clemente, Pedro J, Rodriguez-Echeverria, Roberto, Hernández, Juan, & Sanchez-Figueroa, Fernando. 2011. A model-driven approach for reusing tests in smart home systems. *Personal and Ubiquitous Computing*, **15**(4), 317–327.

De Troyer, Olga, & Casteleyn, Sven. 2001. The conference review system with WSDM. *In: First International Workshop on Web-Oriented Software Technology, IWWOST*, vol. 1.

Del Fabro, M. D., Bézivin, J., Jouault, F., Breton, E., & Gueltas, G. 2005. AMW: A Generic Model Weaver. *Proc. of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles*.

Di Lucca, Giuseppe Antonio, Di Penta, Massimiliano, Antoniol, Giuliano, & Casazza, G. 2001. An approach for reverse engineering of web-based applications. *Pages 231–240 of: Proceedings Eighth Working Conference on Reverse Engineering*, vol. 16. IEEE Comput. Soc.

Di Lucca, Giuseppe Antonio, Fasolino, Anna Rita, & Tramontana, Porfirio. 2002. Towards a Better Comprehensibility of Web Applications : Lessons Learned from Reverse Engineering Experiments. *Page 33 of: Fourth International Workshop on Web Site Evolution (WSE'02)*.

Di Lucca, Giuseppe Antonio, Fasolino, Anna Rita, & Tramontana, Porfirio. 2004. Reverse engineering Web applications: the WARE approach. *Journal of Software Maintenance and Evolution: Research and Practice*, **16**(12), 71–101.

Di Lucca, Giuseppe Antonio, a.R. Fasolino, & Tramontana, P. 2005. Recovering Interaction Design Patterns in Web Applications. *Ninth European Conference on Software Maintenance and Reengineering*, 366–374.

Di Lucca, Giuseppe Antonio, Distante, Damiano, & Bernardi, Mario Luca. 2006. Recovering Conceptual Models from Web Applications. *Pages 113–120 of:* Pierce, Rob, & Stanrney, John (eds), *SIGDOC '06 Proceedings of the 24th annual ACM international conference on Design of communication*. ACM Press.

Didonet Del Fabro, Marcos, & Valduriez, Patrick. 2009. Towards the efficient development of model transformations using model weaving and matching transformations. *Software and Systems Modeling*, **8**(3), 305–324.

Dolog, Peter, & Stage, Jan. 2007. Designing Interaction Spaces for Rich Internet Applications with UML. *Pages 358 – 363 of: LNCS. International Conference on Web Engineering*. Lecture Notes in Computer Science, vol. 4607. Berlin, Heidelberg: Springer Berlin Heidelberg.

Dolog, Peter, & Stage, Jan. 2009. *ADRIA: A Method for Abstract Design of Rich Internet Applications for the Web 2.0*. Tech. rept. Aalborg University, Aalborg, Denmark.

Dorfman, Merlin, & Thayer, Richard H. 2002. *Software Engineering (Vol. 1 & Vol. 2)*. IEEE Computer Society Press.

Draheim, Dirk, Fehr, Elfriede, & Weber, Gerald. 2003. JSPick-a server pages design recovery tool. *Pages 230–236 of: Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* IEEE Comput. Soc.

Draheim, Dirk, Lutteroth, Christof, & Weber, Gerald. 2005. A Source Code Independent Reverse Engineering Tool for Dynamic Web Sites. *Pages 168–177 of: CSMR 2005 Proceedings of the 9th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society.

Dudney, B., & Lehr, J. 2003. *Jakarta pitfalls: time saving solutions for Struts, Ant, JUnit, and Cactus*. Java open source library. Wiley.

Erlikh, L. 2000. Leveraging legacy system dollars for e-business. *IT professional*, 17–23.

Espinazo PagÃ¡n, Javier, SÃ¡nchez Cuadrado, JesÃºss, & GarcÃa Molina, JesÃºs. 2011. Morsa: A Scalable Approach for Persisting and Accessing Large Models. *Pages 77–92 of:* Whittle, Jon, Clark, Tony, & KÃŒhne, Thomas (eds), *Model Driven Engineering Languages and Systems*. Lecture Notes in Computer Science, vol. 6981. Springer Berlin Heidelberg.

Favre, JM. 2004. Foundations of Model (Driven) (Reverse) Engineering : Models. *Pages 1–31 of: Seminar on Language Engineering for Model-Driven. Dagstuhl Seminar Proceedings.*

Fialho, Andre T.S., & Schwabe, Daniel. 2007. Enriching Hypermedia Application Interfaces. *Pages 188–193 of:* Baresi, Luciano, Fraternali, Piero, & Houben, Geert-Jan (eds), *International Conference on Web Engineering*. Lecture Notes in Computer Science, vol. 4607. Springer Berlin Heidelberg.

Fielding, Roy Thomas. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, University of California, Irvine.

Fleurey, Franck, Breton, Erwan, & Baudry, Benoit. 2007. Model-driven engineering for software migration in a large industrial context. *Pages 482–497 of: 10th International Conference On Model Driven Engineering Languages And Systems*.

Foote, Brian, & Yoder, Joseph. 1997. Big Ball of Mud. *In: Fourth Conference on Patterns, Languages of Programs*.

Fraternali, Piero, Comai, Sara, Bozzon, Alessandro, & Carughi, Giovanni Toffetti. 2010. Engineering rich internet applications with a model-driven approach. *ACM Transactions on the Web*, **4**(2), 1–47.

Gall, Harald, & Weidl, Johannes. 1999. Object-Model Driven Abstraction-to-Code Mapping. *Pages 1–14 of: In Proceedings of the 2nd Workshop on Object-Oriented Reengineering (WOOR 1999)*. IEEE Computer Society.

Garrigós, Irene, Meliá, Santiago, & Casteleyn, Sven. 2009. Personalizing the Interface in Rich Internet Applications. *In: Proc. 10th International Conferenece on Web Information Systems Engineering*.

Harrison, Rachel, Counsell, Steve, & Nithi, R. 1998. Coupling metrics for object-oriented design. *Pages 150–157 of: Software Metrics Symposium, 1998. Metrics 1998. Proceedings. Fifth International*. IEEE.

Hassan, A.E., & Holt, R.C. 2001. Towards a better understanding of Web applications. *Proceedings 3rd International Workshop on Web Site Evolution. WSE 2001*, 112–116.

Jacobson, Ivar. 1992. *Object-Oriented Software Engineering. A Use-Case Driven Approach*. Addison-Wesley.

Jouault, Frédéric, & Kurtev, Ivan. 2006. Transforming models with ATL. *Pages 128–138 of: Proceedings of the 2005 international conference on Satellite Events at the MoDELS*. MoDELS'05. Berlin, Heidelberg: Springer-Verlag.

Katsimpa, T., Panagis, Y., Sakkopoulos, E., Tzimas, G., & Tsakalidis, a. 2006. Application modeling using reverse engineering techniques. *Proceedings of the 2006 ACM symposium on Applied computing - SAC '06*, 1250.

Kazman, R, O'Brien, L, & Verhoef, C. 2003. *Architecture Reconstruction Guidelines*. Tech. rept. November. Carnegie Mellon University.

Kazman, Rick, & Carrière, SJ. 1997. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, **6**(2), 107–138.

Koch, Nora, Kraus, Andreas, & Hennicker, Rolf. 2001. The authoring process of the uml-based web engineering approach. *In: First International Workshop on Web-Oriented Software Technology, IWWOST*.

Koch, Nora, Pigerl, Matthias, Zhang, Gefei, & Morozova, Tatiana. 2009. Patterns for the Model-Based Development of RIAs. *Pages 283–291 of:* Gaedke, Martin, Grossniklaus, Michael, & Díaz, Oscar (eds), *ICWE '9 Proceedings of the 9th International Conference on Web Engineering*. Lecture Notes in Computer Science, vol. 5648. Berlin, Heidelberg: Springer Berlin Heidelberg.

Kraus, Andreas, Knapp, Alexander, & Koch, Nora. 2007. Model-driven generation of web applications in UWE. *Pages 23–38 of: MDWE2007-3rd International Workshop on Model-Driven Web Engineering, July*.

Kurtev, Ivan, Bézivin, Jean, & Aksit, M. 2002. Technological Spaces : an Initial Appraisal. *Pages 1–6 of: CoopIS, DOA'2002 Federated Conferences, Industrial track*.

Lehman, MM, Ramil, JF, Wernick, P.D., Perry, D.E., & Turski, W.M. 1997. Metrics and laws of software evolution-the nineties view. *Pages 20–32 of: Proceedings Fourth International Software Metrics Symposium*. IEEE Comput. Soc.

Lima, Fernanda, & Schwabe, Daniel. 2003. Application modeling for the semantic web. *Pages 93–102 of: Web Congress, 2003. Proceedings. First Latin American*. IEEE.

Linaje, Marino, Preciado, Juan Carlos, & Sanchez-Figueroa, Fernando. 2007. Engineering Rich Internet Application User Interfaces over Legacy Web Models. *IEEE Internet Computing*, **11**(6), 53–59.

Linaje, Marino, Preciado, Juan Carlos, Morales-Chaparro, Rober, Rodriguez-Echeverria, Roberto, & Sanchez-Figueroa, Fernando. 2009. Automatic Generation of RIAs Using RUX-Tool and Webratio. *Pages 501–504 of:* Gaedke, Martin, Grossniklaus, Michael, & Díaz, Oscar (eds), *International Conference on Web Engineering*. Lecture Notes in Computer Science, vol. 5648. Berlin, Heidelberg: Springer Berlin Heidelberg.

Machado, Leonardo, Filho, Orlando, & Ribeiro, João. 2009. UWE-R: an extension to a web engineering methodology for rich internet applications. *WSEAS Transactions on Information Science and Applications*, **6**(4), 9.

Mann, William C, & Thompson, Sandra A. 1988. Rhetorical structure theory: Toward a functional theory of text organization. *Text*, **8**(3), 243–281.

McCabe, Thomas J. 1976. A complexity measure. *Software Engineering, IEEE Transactions on*, 308–320.

Meliá, Santiago, Gómez, Jaime, Pérez, Sandy, & Díaz, Oscar. 2008. A Model-Driven Development for GWT-Based Rich Internet Applications with OOH4RIA. *In Proceedings of the Eighth International Conference on Web Engineering*, July, 13–23.

Meliá, Santiago, Gómez, Jaime, Pérez, Sandy, & Díaz, Oscar. 2010. Architectural and Technological Variability in Rich Internet Applications. *IEEE Internet Computing*, **14**(3), 24–32.

Paganelli, L., & Paterno, F. 2002. Automatic reconstruction of the underlying interaction design of web applications. *Pages 439–445 of: Proceedings of the 14th international conference on Software engineering and knowledge engineering*. SEKE '02. New York, NY, USA: ACM.

Pastor, Oscar, Fons, Joan, Pelechano, Vicente, & Abrahão, Silvia. 2006. Conceptual modelling of web applications: the OOWS approach. *Pages 277–302 of: Web Engineering*. Springer.

Patel, Reshma, Coenen, Frans, Martin, Russell, & Archer, Lawson. 2007. *Reverse Engineering of Web Applications: A Technical Review*. Tech. rept. July 2007. University of Liverpool Department of Computer Science, Liverpool.

Pérez, Sandy, Díaz, Oscar, Meliá, Santiago, & Gómez, Jaime. 2008. Facing Interaction-Rich RIAs: The Orchestration Model. *2008 Eighth International Conference on Web Engineering*, July, 24–37.

Perez-Castillo, R., de Guzman, I. G-R, Piattini, Mario, & Ebert, Christof. 2011. Reengineering Technologies. *IEEE Software*, **28**(6), 13–17.

Preciado, Juan Carlos, Linaje, Marino, Sanchez, Fernando, & Comai, Sara. 2005. Necessity of methodologies to model Rich Internet Applications. *Seventh IEEE International Symposium on Web Site Evolution*.

Reus, Thijs, Geers, Hans, & Deursen, Arie Van. 2006. Harvesting software systems for MDA-based reengineering. *Pages 1–13 of: European Conference on Model Driven Architecture: Foundations and Applications*.

Ricca, Filippo, & Tonella, Paolo. 2000. Web site analysis: structure and evolution. *Proceedings International Conference on Software Maintenance ICSM-94*, 76–86.

Ricca, Filippo, & Tonella, Paolo. 2001. Understanding and Restructuring Web Sites with ReWeb. *Multimedia, IEEE*, 40–51.

Rodríguez-Echeverría, Roberto, Conejero, José María, Linaje, Marino, Preciado, Juan Carlos, & Sánchez-Figueroa, Fernando. 2010. Re-engineering legacy Web applications into Rich Internet Applications. *Pages 189–203 of: 10th International Conference on Web Engineering*.

Rodríguez-Echeverría, Roberto, Conejero, José María, Clemente, Pedro J., Villalobos, María D., & Sánchez-Figueroa, Fernando. 2012a. Extracting navigational models from struts-based web applications. *Pages 419–426 of:* Brambilla, Marco, Tokuda, Takehiro, & Tolksdorf, Robert (eds), *International Conference on Web Engineering*. Lecture Notes in Computer Science, vol. 7387. Berlin: Springer Berlin Heidelberg.

Rodríguez-Echeverría, Roberto, Conejero, José María, Clemente, Pedro J, Villalobos, María D., & Sánchez-Figueroa, Fernando. 2012b. Generation of WebML Hypertext Models from Legacy Web Applications. *Pages 91–95 of: 14th IEEE International Symposium on Web Systems Evolution*. Riva del Garda (Italy): IEEE Computer Society.

Rodríguez-Echeverría, Roberto, Conejero, JoséM., Clemente, PedroJ., Pavón, VíctorM., & Sánchez-Figueroa, Fernando. 2012c. Model Driven Extraction of the Navigational Concern of Legacy Web Applications. *Pages 56–70 of:* Grossniklaus, Michael, & Wimmer, Manuel (eds), *Current Trends in Web Engineering*. Lecture Notes in Computer Science, vol. 7703. Springer Berlin Heidelberg.

Rodríguez-Echeverría, Roberto, Conejero, José María, Clemente, Pedro J, Preciado, Juan Carlos, & Sánchez-Figueroa, Fernando. 2012d. Modernization of legacy web applications into rich internet applications. *Current Trends in Web Engineering*, **7059/2012**(Lecture Notes in Computer Science), 236–250.

Rodríguez-Echeverría, Roberto, Macías, Fernando, Pavón, Víctor M., Conejero, José María, Clemente, Pedro J., & Sánchez-Figueroa, Fernando. 2013a. Generating a Conceptual Representation of a Legacy Web Application. *Page pending of: 14th International Conference on Web Information System Engineering - LNCS 8181*. Namjing: Springer Berlin / Heidelberg.

Rodríguez-Echeverría, Roberto, Macías, Fernando, Pavón, Víctor M., Conejero, José María, & Sánchez-Figueroa, Fernando. 2013b. Generating a REST Service Layer from a Legacy System. *Page pending of: 22nd International Conference on Information Systems Development (ISD2013)*.

Rossi, Gustavo, Urbieta, M, Schwabe, Daniel, Ginzburg, J, Argentina, U., & Aires, B. 2007. Designing the interface of rich internet applications. *Pages 144–153 of: Web Conference, 2007. LA-WEB 2007. Latin American*.

Rossi, Gustavo, Urbieta, Matias, Ginzburg, Jeronimo, Distante, Damiano, & Garrido, Alejandra. 2008a. Refactoring to Rich Internet Applications. A Model-Driven Approach. *2008 Eighth International Conference on Web Engineering*, July, 1–12.

Rossi, Gustavo, Pastor, Oscar, Schwabe, Daniel, & Olsina, Luis. 2008b. *Web Engineering: Modelling and Implementing Web Applications (Human-Computer Interaction Series)*. 1 edn. Springer Berlin Heidelberg.

Sánchez Ramón, Óscar, Sánchez Cuadrado, Jesús, & García Molina, Jesús. 2010. Model-driven reverse engineering of legacy graphical user interfaces. *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, 147.

Sánchez-Ramón, Óscar, Bermudez, Fco Javier, & García-Molina, Jesús. 2013. Una valoración de la Modernización de Software Dirigida por Modelos. *Pages 288–315 of:* Moreno, Ana, & Vara, Juan M (eds), *XVIII Jornadas en Ingeniería del Software y Bases de Datos (JISBD)*.

Schauerhuber, Andrea, Wimmer, Manuel, & Kapsammer, Elisabeth. 2006. Bridging existing Web modeling languages to model-driven engineering. *Page 5 of: Workshop proceedings of the sixth international conference on Web engineering - ICWE '06*. New York, New York, USA: ACM Press.

Schwabe, Daniel, & Rossi, Gustavo. 2001. A Conference Review System with OOHDM. *In: First International Workshop on Web-Oriented Software Technology, IWWOST*.

Schwabe, Daniel, Rossi, Gustavo, & Barbosa, Simone DJ. 1996. Systematic hypermedia application design with OOHDM. *Pages 116–128 of: Proceedings of the the seventh ACM conference on Hypertext*. ACM.

Shan, T.C., & Hua, W.W. 2006. Taxonomy of java web application frameworks. *2006 IEEE International Conference on e-Business Engineering (ICEBE'06)*, 378–385.

Tilley, Scott R, & Huang, Shihong. 2001. Evaluating the reverse engineering capabilities of Web tools for understanding site content and structure: a case study. *Pages 514–523 of: Proceedings International Conference on Software Engineering*. IEEE Comput. Soc.

Tilley, Scott R, & Smith, Dennis B. 1995. *Perspectives on legacy system reengineering*. Tech. rept. Carnegie Mellon University.

Toffetti Carughi, Giovanni, Comai, Sara, Preciado, Juan Carlos, Linaje, Marino, & Toffetti, Giovanni. 2011. State-of-the-art and trends in the systematic development of rich internet applications. *Journal of Web Engineering (JWE)*, **10**(1), 070–086.

Ulrich, William. 2010. *Modernization Standards Roadmap*. Pages 46–64.

Valverde, Francisco. 2010. *OOWS 2.0 Un Método de Ingeniería Web Para la Producción de Aplicaciones Web 2.0*. Ph.D. thesis, Universidad Politécnica de Valencia.

Valverde, Francisco, & Pastor, Oscar. 2009. Facing the Technological Challenges of Web 2.0: A RIA Model-Driven Engineering Approach. *Web Information Systems Engineering-WISE 2009*, 131–144.

Vanderdonckt, J., Bouillon, L., & Souchon, N. 2001. Flexible reverse engineering of web pages with VAQUISTA. *Proceedings Eighth Working Conference on Reverse Engineering*, 241–248.

Vanderdonckt, Jean, Limbourg, Quentin, Michotte, Benjamin, Bouillon, Laurent, Trevisan, Daniela, & Florins, Murielle. 2004. UsiXML: a user interface description language for specifying multimodal user interfaces. *In: Proceedings of W3C Workshop on Multimodal Interaction WMI*, vol. 2004.

Vora, Pawan. 2009. *Web Application Design Patterns*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Wright, Jevon M., & Dietrich, J. 2008. Survey of existing languages to model interactive web applications. *Pages 113–123 of: Proceedings of the fifth on Asia-Pacific conference on conceptual modelling-Volume 79*. Australian Computer Society, Inc.

Wright, Jevon M., & Dietrich, JB. 2010. The Development of a Modelling Language for Rich Internet Applications. *Pages 2–5 of: New Zealand Comp. Sci. Research Students Conference*.

Yahoo!-Patterns. 2011 (March). *http://developer.yahoo.com/ypatterns/*.

Anexos

Anexo A

Reglas de transformación en ATL

A.1. Reglas para el Modelo

A.1.1. Reglas para objetos y atributos

```
1  rule DataObject {
2   from
3    class : mmHibernateMappings!ClassType
4   to
5    dataObject : mmMIGRARIAMVC!DataObject (
6     name <- class.name.toString().split('[.]').last(),
7     attributes <- class.id,
8     attributes <- dataObject.attributes.append(class.property),
9     relations <- class.manyToOne,
10    relations <- dataObject.relations.append(class.set->collect(iterator | iterator.
          manyToMany)),
11    relations <- dataObject.relations.append(class.set->collect(iterator | iterator.
          oneToMany)),
12    operations <- if not class.getAssociateJavaDAOInterfaceDeclaration().
          oclIsUndefined() then
13          class.getAssociateJavaDAOInterfaceDeclaration().bodyDeclarations->select(
                bodyDec | bodyDec.oclIsTypeOf(mmJava!MethodDeclaration))
14          ->collect(iterator | thisModule.methodDeclarationDAOInterface2DataOperation(
                iterator))
15           else
16            OclUndefined
17           endif
18   )
19 }
20
21 rule id2DataAttribute {
22  from
```

```
23    id : mmHibernateMappings!IdType
24   to
25    dataAttribute : mmMIGRARIAMVC!DataAttribute (
26     name <- id.name,
27     type <- if(not id.type.oclIsUndefined()) then
28         thisModule.convertString2DataType(id.type.toString())
29       else
30         let hibernateClassType :  mmHibernateMappings!ClassType = id.
               refImmediateComposite() in
31          hibernateClassType.getJavaClassDeclaration().getTypeAttribute(id.name)
32       endif,
33     isId <- true
34     ,getMethodDeclaration <-  id.getJavaClassDeclaration().
           getAllMethodsClassDeclarations()
35             ->any(var | var.name.substring(4, var.name.size()).toLower()=
                   dataAttribute.name.toString().toLower() and var.name.toString().
                   startsWith('get'))
36     ,setMethodDeclaration <-  id.getJavaClassDeclaration().
           getAllMethodsClassDeclarations()
37             ->any(var | var.name.substring(4, var.name.size()).toLower()=
                   dataAttribute.name.toString().toLower() and var.name.toString().
                   startsWith('set'))
38    )
39  }
40
41  rule property2DataAttribute {
42   from
43    property : mmHibernateMappings!PropertyType
44   to
45    dataAttribute : mmMIGRARIAMVC!DataAttribute (
46     name <- property.name,
47     type <- if(not property.type.oclIsUndefined()) then
48         thisModule.convertString2DataType(property.type.toString())
49       else
50         let hibernateClassType :  mmHibernateMappings!ClassType = property.
               refImmediateComposite() in
51          hibernateClassType.getJavaClassDeclaration().getTypeAttribute(property.name)
52       endif,
53     isId <- false
54     ,getMethodDeclaration <-  property.getJavaClassDeclaration().
           getAllMethodsClassDeclarations()
55             ->any(var | var.name.substring(4, var.name.size()).toLower()=
                   dataAttribute.name.toString().toLower() and var.name.toString().
                   startsWith('get'))
```

```
56      ,setMethodDeclaration <- property.getJavaClassDeclaration().
            getAllMethodsClassDeclarations()
57              ->any(var | var.name.substring(4, var.name.size()).toLower()=
                    dataAttribute.name.toString().toLower() and var.name.toString().
                    startsWith('set'))
58    )
59  }
```

A.1.2. Reglas para relaciones

```
1   rule manyToOne2DataRelation {
2    from
3     manyToOne : mmHibernateMappings!ManyToOneType
4    to
5     dataRelation : mmMIGRARIAMVC!DataRelation (
6       name <- manyToOne.name,
7       type <- #Manytoone,
8       target <- mmHibernateMappings!ClassType.allInstances()->any(var | var.name.
            toString()=manyToOne.class.toString())
9       ,dataOperationByRelation <- thisModule.dataOperationByRelation(manyToOne.
            getJavaClassDeclaration().getAllMethodsClassDeclarations()
10          ->any(var | var.name.substring(4, var.name.size()).toLower()= manyToOne.name.
                toString().toLower() and var.name.toString().startsWith('get')),
                manyToOne.refImmediateComposite())
11      ,dataOperationByRelation <- thisModule.dataOperationByRelation(manyToOne.
            getJavaClassDeclaration().getAllMethodsClassDeclarations()
12          ->any(var | var.name.substring(4, var.name.size()).toLower()= manyToOne.name.
                toString().toLower() and var.name.toString().startsWith('set')),
                manyToOne.refImmediateComposite())
13    )
14  }
15
16  rule manyToMany2DataRelation {
17   from
18    manyToMany : mmHibernateMappings!ManyToManyType
19   to
20    dataRelation : mmMIGRARIAMVC!DataRelation (
21      name <- manyToMany.refImmediateComposite().name,
22      type <- #Manytomany,
23      target <- mmHibernateMappings!ClassType.allInstances()->any(var | var.name.
            toString()=manyToMany.class.toString()),
24      inverse <- let inverseRelation : mmHibernateMappings!SetType =
            mmHibernateMappings!SetType.allInstances()
25                      ->any(var | var.name<>manyToMany.refImmediateComposite().name
                            and
```

```
26                              var.table=manyToMany.refImmediateComposite().table) in
27          if inverseRelation.oclIsUndefined() then
28           OclUndefined
29          else
30           inverseRelation.manyToMany
31          endif
32       ,dataOperationByRelation <- thisModule.dataOperationByRelation(manyToMany.
             getJavaClassDeclaration().getAllMethodsClassDeclarations()
33          ->any(var | var.name.substring(4, var.name.size()).toLower()= manyToMany.
                refImmediateComposite().name.toString().toLower() and var.name.toString()
                .startsWith('get')), manyToMany.refImmediateComposite().
                refImmediateComposite())
34       ,dataOperationByRelation <- thisModule.dataOperationByRelation(manyToMany.
             getJavaClassDeclaration().getAllMethodsClassDeclarations()
35          ->any(var | var.name.substring(4, var.name.size()).toLower()= manyToMany.
                refImmediateComposite().name.toString().toLower() and var.name.toString()
                .startsWith('set')), manyToMany.refImmediateComposite().
                refImmediateComposite())
36     )
37  }
38
39  rule oneToMany2DataRelation {
40   from
41    oneToMany : mmHibernateMappings!OneToManyType
42   to
43    dataRelation : mmMIGRARIAMVC!DataRelation (
44      name <- oneToMany.refImmediateComposite().name,
45      type <- #Onetomany,
46      target <- mmHibernateMappings!ClassType.allInstances()->any(var | var.name.
             toString()=oneToMany.class.toString())
47       ,dataOperationByRelation <- thisModule.dataOperationByRelation(oneToMany.
              getJavaClassDeclaration().getAllMethodsClassDeclarations()
48          ->any(var | var.name.substring(4, var.name.size()).toLower()= oneToMany.
                refImmediateComposite().name.toString().toLower() and var.name.toString()
                .startsWith('get')), oneToMany.refImmediateComposite().
                refImmediateComposite())
49       ,dataOperationByRelation <- thisModule.dataOperationByRelation(oneToMany.
             getJavaClassDeclaration().getAllMethodsClassDeclarations()
50          ->any(var | var.name.substring(4, var.name.size()).toLower()= oneToMany.
                refImmediateComposite().name.toString().toLower() and var.name.toString()
                .startsWith('set')), oneToMany.refImmediateComposite().
                refImmediateComposite())
51     )
52  }
```

A.1.3. Reglas para operaciones

```
1  lazy rule dataOperationByRelation {
2   from
3     methodDeclaration : mmJava!MethodDeclaration
4     ,class : mmHibernateMappings!ClassType
5   to
6    dataOperationByRelaction : mmMIGRARIAMVC!DataOperationByRelation (
7     methodDeclaration <- methodDeclaration
8     ,name <- methodDeclaration.name
9     ,type <- #GetAll
10    ,return <- if methodDeclaration.returnType.type.oclIsTypeOf(mmJava!
          PrimitiveTypeVoid) then
11        thisModule.dataOperationVoidReturn(methodDeclaration.returnType)
12         else
13        thisModule.dataOperationReturn(methodDeclaration.returnType)
14         endif
15    ,parameters <- methodDeclaration.parameters->collect(iterator | thisModule.
          dataOperationParameter(iterator))
16    ,dataObject <- class
17   )
18
19
20 }
21
22 lazy rule methodDeclarationDAOInterface2DataOperation {
23  from
24   methodDeclarationDAOInterface : mmJava!MethodDeclaration
25  to
26   dataOperation : mmMIGRARIAMVC!DataOperation (
27    name <- methodDeclarationDAOInterface.name,
28    type <- methodDeclarationDAOInterface.methodDeclarationToDataOperationTypes(),
29    return <- if methodDeclarationDAOInterface.returnType.type.oclIsTypeOf(mmJava!
          PrimitiveTypeVoid) then
30        thisModule.dataOperationVoidReturn(methodDeclarationDAOInterface.returnType)
31         else
32          thisModule.dataOperationReturn(methodDeclarationDAOInterface.returnType)
33         endif,
34    parameters <- methodDeclarationDAOInterface.parameters->collect(iterator |
          thisModule.dataOperationParameter(iterator)),
35    methodDeclaration <- methodDeclarationDAOInterface
36   )
37 }
38
```

```
39 lazy rule dataOperationVoidReturn {
40  from
41   returnTypeAccess : mmJava!TypeAccess
42  to
43   dataOperationReturn : mmMIGRARIAMVC!DataOperationReturn (
44    name <- 'return'+returnTypeAccess.refImmediateComposite().name.toString().
          setFirtLetterToCapital(),
45    type <- #Void,
46    collection <- #None
47
48   )
49 }
50
51 lazy rule dataOperationReturn {
52  from
53   returnTypeAccess : mmJava!TypeAccess
54  to
55   dataOperationReturn : mmMIGRARIAMVC!DataOperationReturn (
56    name <- 'return'+returnTypeAccess.refImmediateComposite().name.toString().
          setFirtLetterToCapital(),
57    type <- thisModule.convertString2DataType(returnTypeAccess.type.name),
58    collection <- returnTypeAccess.type.getDataCollectionType(),
59    object <-  if (returnTypeAccess.type.getDataCollectionType()= #None) then
60         mmHibernateMappings!ClassType.allInstances()->any(var | var.name.toString()=
             returnTypeAccess.type.getFullName())
61        else
62         mmHibernateMappings!ClassType.allInstances()->any(var | var.name.toString()=
             returnTypeAccess.type.typeArguments.first().type.getFullName())
63        endif
64   )
65 }
66
67 lazy rule dataOperationParameter {
68  from
69   parameter : mmJava!SingleVariableDeclaration
70  to
71   dataOperationReturn : mmMIGRARIAMVC!DataOperationParameter (
72    name <- parameter.name,
73    type <- thisModule.convertString2DataType(parameter.type.type.name),
74    collection <- parameter.type.type.getDataCollectionType(),
75    object <-  if (parameter.type.type.getDataCollectionType()= #None) then
76         if (thisModule.convertString2DataType(parameter.type.type.name)=#DataObject)
             then
```

```
77          mmHibernateMappings!ClassType.allInstances()->any(var | var.name.toString()=
                parameter.type.type.getFullName())
78         else
79          OclUndefined
80         endif
81        else
82         mmHibernateMappings!ClassType.allInstances()->any(var | var.name.toString()=
              parameter.type.type.typeArguments.first().type.getFullName())
83        endif
84    )
85 }
```

A.2. Reglas para la Vista

A.2.1. Reglas para páginas

```
1  rule model2View {
2   from
3    model : mmStrutsJSP!Model
4   to
5    view : mmMIGRARIAMVC!View (
6     pages <- model.pages
7    )
8  }
9
10
11 rule pageJSP2page {
12  from
13   sPage : mmStrutsJSP!Page
14  to
15   tPage : mmMIGRARIAMVC!Page (
16    name <- sPage.name,
17    title <- sPage.getPageTitle(),
18    path <- sPage.originalFilePath.toString().split(thisModule.rootWebDirectory).last()
19   )
20 }
```

A.2.2. Reglas para objetos y atributos de salida

```
1  unique lazy rule presentationObject {
2   from
3    beanNameAndContainer : TupleType(beanName : String, objectContainer : mmMIGRARIAMVC!
         ObjectRequestContainer)
4   to
5    presentationObject : mmMIGRARIAMVC!PresentationObject (
6     name <- beanNameAndContainer.beanName,
7     container <- beanNameAndContainer.objectContainer,
8     direction <- #output
9    )
```

```
10 }
11
12 rule PresentationAttribute {
13  from
14   beanWrite : mmStrutsJSP!JSPActionStrutsBeanWriter (beanWrite.isPresentationAttribute())
15  to
16   presentationAttribute : mmMIGRARIAMVC!Data_PA (
17    name <- beanWrite.getAttributeValue('property'),
18    direction <- #output,
19    hidden <- false,
20    isId <- false, -- TO_DO esta propiedad no se puede establecer a un valor correcto con la
          información actual
21    isLabel <- false,
22    parentobject <- thisModule.presentationObject(Tuple{beanName=beanWrite.getAttributeValue('
          name'), objectContainer=beanWrite.getBeanContainer()})
23   )
24 }
```

A.2.3. Reglas para objetos y atributos de entrada/salida

```
1  rule htmlForm2PresentationObject {
2   from
3    htmlForm : mmStrutsJSP!JSPActionStrutsHTMLForm
4   to
5    presentationObject : mmMIGRARIAMVC!PresentationObject (
6     name <- htmlForm.formBean,
7     direction <- #input,
8     attributes <- htmlForm.formTag,
9     container <- htmlForm.getFormBeanContainer()
10   )
11 }
12
13
14 rule htmlFormTag2PresentationAttribute {
15  from
16   htmlFormTag : mmStrutsJSP!JSPActionStrutsHTMLFormTag (not (htmlFormTag.name='html:submit' or
          htmlFormTag.name='html:select'))
17  to
18   presentationAttribute : mmMIGRARIAMVC!Data_PA (
19    name <- htmlFormTag.property,
20    direction <- #input,
21    hidden <- if (htmlFormTag.name='html:hidden') then true else false endif
22   )
23 }
24
25
26 rule htmlSelectFormTag2DataSet_PA {
27  from
28   htmlFormTag : mmStrutsJSP!JSPActionStrutsHTMLFormTag (htmlFormTag.name='html:select')
29  to
30   presentationAttribute : mmMIGRARIAMVC!DataSet_PA (
```

```
31      name <- htmlFormTag.property,
32      direction <- #input,
33      hidden <- false,
34      items <- htmlFormTag.children->select(iterator | iterator.oclIsTypeOf(mmStrutsJSP!
            JSPActionStrutsHTMLFormSelectOption))->select(opt | opt.name='html:options')
35    )
36  }
37
38  rule htmlOption2literalItem {
39   from
40    htmlOption : mmStrutsJSP!JSPActionStrutsHTMLFormSelectOption (htmlOption.name='html:option')
41   to
42    literalItem : mmMIGRARIAMVC!LiteralItem (
43     name <- htmlOption.children.first().name,
44     value <- htmlOption.getAttributeValue('value'),
45     parentcollection <- thisModule.htmlSelectWithHtmlOptionChildren2LiteralPresentCollection(
           htmlOption.SelectParent)
46    )
47  }
48
49  unique lazy rule htmlSelectWithHtmlOptionChildren2LiteralPresentCollection {
50   from
51    htmlSelect : mmStrutsJSP!JSPActionStrutsHTMLFormTag
52   to
53    literalPresentationCollection : mmMIGRARIAMVC!LiteralPresentationCollection (
54    name <- htmlSelect.property,
55    page <- htmlSelect.getPageContainer()
56    )
57  }
58
59  rule htmlOptionsUseCollection2ObjectPresentationCollection {
60   from
61    htmlOptions : mmStrutsJSP!JSPActionStrutsHTMLFormSelectOption (htmlOptions.name='html:
          options' and
62                        not htmlOptions.getAttributeValue('collection').oclIsUndefined())
63   to
64    objectPresentationCollection : mmMIGRARIAMVC!ObjectPresentationCollection (
65     name <- htmlOptions.getAttributeValue('collection'),
66     page <- htmlOptions.getPageContainer(),
67     objects <- thisModule.presentationObject(Tuple{beanName=htmlOptions.getAttributeValue('
           property'), objectContainer=htmlOptions}),
68     multiSelection <- if htmlOptions.SelectParent.getAttributeValue('multiple').oclIsUndefined
           () then
69          false
70           else
71          if htmlOptions.SelectParent.getAttributeValue('multiple')='true' then
72           true
73          else
74           false
75          endif
76           endif
```

```
77    ),
78    valueAttribute : mmMIGRARIAMVC!Data_PA (
79     name <- htmlOptions.getAttributeValue('property'),
80     direction <- #output,
81     parentobject <- thisModule.presentationObject(Tuple{beanName=htmlOptions.getAttributeValue
           ('property'), objectContainer=htmlOptions})
82    ),
83    labelAttribute : mmMIGRARIAMVC!Data_PA ( --- TO_DO atributo opcional no siemrpe existe
84     name <- htmlOptions.getAttributeValue('labelProperty'),
85     direction <- #output,
86     isLabel <- true,
87     parentobject <-  thisModule.presentationObject(Tuple{beanName=htmlOptions.getAttributeValue
           ('property'), objectContainer=htmlOptions})
88    )
89   }
90
91  rule htmlOptionsNotUseCollection2ObjectPresentationCollection {
92   from
93    htmlOptions : mmStrutsJSP!JSPActionStrutsHTMLFormSelectOption (htmlOptions.name='html:
          options' and
94                        htmlOptions.getAttributeValue('collection').oclIsUndefined())
95   to
96    objectPresentationCollection : mmMIGRARIAMVC!ObjectPresentationCollection (
97     name <- htmlOptions.getAttributeValue('name'),
98     page <- htmlOptions.getPageContainer()
99    )
100 }
```

A.2.4. Reglas para peticiones

```
1   rule htmlSubmit2PresentationAttributeAndInputRequest {
2    from
3     htmlFormTag : mmStrutsJSP!JSPActionStrutsHTMLFormTag (htmlFormTag.name='html:submit')
4    to
5     presentationAttribute : mmMIGRARIAMVC!Submit_PA (
6      name <- htmlFormTag.property,
7      direction <- #input ,
8      parentobject <- htmlFormTag.formParent
9     ),
10    inputRequest : mmMIGRARIAMVC!Request (
11     name <- htmlFormTag.getInputRequestName(),
12     path <- htmlFormTag.formParent.action,
13     container <- if htmlFormTag.formParent.logicIterateParent.oclIsUndefined() then
14          htmlFormTag.getPageContainer()
15          else
16           htmlFormTag.formParent.logicIterateParent
17         endif,
18     type <- #object,
19     params <- objectRequestParameter
20
21    ),
```

```
22    objectRequestParameter : mmMIGRARIAMVC!ObjectRequestParameter (
23     name <- htmlFormTag.property,
24     value <- htmlFormTag.getInputRequestName(),
25     object <- htmlFormTag.formParent,
26     property <- htmlFormTag
27    )
28  }
29
30  rule htmlLinkActionWithoutParamNamePropertyAndId {
31   from
32    htmlLinkAction : mmStrutsJSP!JSPActionStrutsHTMLLinkAction (let childrenNames : Sequence(
          String) = htmlLinkAction.children->collect(iterator | iterator.name.toLower()) in
33                    not (childrenNames.includes('paramname') and childrenNames.includes('
                          paramproperty') and childrenNames.includes('paramid')))
34   to
35    request : mmMIGRARIAMVC!Request (
36      name <- htmlLinkAction.getHtmlLinkNameRequest(),
37      path <- htmlLinkAction.action,
38      type <- #page,
39      container <- if htmlLinkAction.logicIterateParent.oclIsUndefined() then
40           htmlLinkAction.getPageContainer()
41           else
42            htmlLinkAction.logicIterateParent
43           endif
44    )
45  }
46
47  rule htmlLinkActionWithParamNamePropertyAndId {
48   from
49    htmlLinkAction : mmStrutsJSP!JSPActionStrutsHTMLLinkAction (let childrenNames : Sequence(
          String) = htmlLinkAction.children->collect(iterator | iterator.name.toLower()) in
50                    (childrenNames.includes('paramname') and childrenNames.includes('
                          paramproperty') and childrenNames.includes('paramid')))
51   to
52    request : mmMIGRARIAMVC!Request (
53      name <- htmlLinkAction.getHtmlLinkNameRequest(),
54      path <- htmlLinkAction.action,
55      type <- #object,
56      container <- if htmlLinkAction.logicIterateParent.oclIsUndefined() then
57           htmlLinkAction.getPageContainer()
58           else
59            htmlLinkAction.logicIterateParent
60           endif,
61      params <- objectRequestParameter
62    )
63    ,
64    objectRequestParameter : mmMIGRARIAMVC!ObjectRequestParameter (
65     name <- htmlLinkAction.paramId,
66     property <- presentationAttribute,
67     object <- thisModule.presentationObject(Tuple{beanName=htmlLinkAction.paramName,
           objectContainer=  if htmlLinkAction.logicIterateParent.oclIsUndefined() then
```

```
68                                  htmlLinkAction.getPageContainer()
69                                  else
70                                   htmlLinkAction.logicIterateParent
71                                  endif})
72
73    ),
74    presentationAttribute : mmMIGRARIAMVC!Data_PA (
75     direction <- #output,
76     hidden <- true,
77     name <- htmlLinkAction.paramProperty,
78     parentobject <- thisModule.presentationObject(Tuple{beanName=htmlLinkAction.paramName,
          objectContainer=  if htmlLinkAction.logicIterateParent.oclIsUndefined() then
79                                  htmlLinkAction.getPageContainer()
80                                  else
81                                   htmlLinkAction.logicIterateParent
82                                  endif})
83    )
84  }
85
86
87  rule htmlParam2valueRequestParameter {
88   from
89    htmlParam : mmStrutsJSP!JSPActionStrutsHTML (htmlParam.name='html:param')
90   to
91    valueRequestParameter : mmMIGRARIAMVC!ValueRequestParameter (
92     name <- htmlParam.getAttributeValue('name'),
93     value <- htmlParam.getAttributeValue('value'),
94     request <- htmlParam.parent
95    )
96  }
97
98
99  rule htmlLinkPage {
100  from
101   htmlLinkPage : mmStrutsJSP!JSPActionStrutsHTMLLinkPage
102  to
103   request : mmMIGRARIAMVC!Request (
104     name <- htmlLinkPage.getHtmlLinkNameRequest(),
105     path <- htmlLinkPage.page,
106     type <- #page,
107     container <- if htmlLinkPage.logicIterateParent.oclIsUndefined() then
108          htmlLinkPage.getPageContainer()
109          else
110           htmlLinkPage.logicIterateParent
111          endif
112   )
113 }
```

A.2.5. Reglas para colecciones

```
1
```

```
2  rule objectPresentationObjectAndPresentationCollectionCase2a {
3   from
4    logicIterate : mmStrutsJSP!JSPActionStrutsLogicIterate (logicIterate.getAttributeValue('
         property').oclIsUndefined())
5   to
6    objectPresentationCollection : mmMIGRARIAMVC!ObjectPresentationCollection (
7     name <- logicIterate.getAttributeValue('name'),
8     item <- thisModule.presentationObject(Tuple{beanName=logicIterate.getAttributeValue('id'),
          objectContainer=logicIterate}),
9     page <- logicIterate.getPageContainer(),
10    requests <- logicIterate.htmlLinks
11   )
12 }
13
14 rule presentationObjectAndPresentationCollectionCase2b {
15  from
16   logicIterate : mmStrutsJSP!JSPActionStrutsLogicIterate (not logicIterate.getAttributeValue('
         property').oclIsUndefined())
17  to
18   presentationCollection : mmMIGRARIAMVC!ObjectPresentationCollection (
19    name <- logicIterate.getAttributeValue('property'),
20    item <- thisModule.presentationObject(Tuple{beanName=logicIterate.getAttributeValue('id'),
          objectContainer=logicIterate}),
21    page <- logicIterate.getPageContainer(),
22    requests <- logicIterate.htmlLinks
23   ),
24   presentationAttributeParentObjet : mmMIGRARIAMVC!DataSet_PA (
25    name <-  logicIterate.getAttributeValue('property'),
26    direction <- #output,
27    items <- presentationCollection,
28    parentobject <- thisModule.presentationObject(Tuple{beanName=logicIterate.getAttributeValue
          ('name'),
29                objectContainer=
30                 if logicIterate.isNested() then
31                  logicIterate.getFatherLogicIterate()
32                 else
33                  logicIterate.getPageContainer()
34                 endif})
35   )
36 }
37
38 rule beanSize2Derived_PA {
39  from
40   beanSize : mmStrutsJSP!JSPActionStrutsBean (beanSize.name='bean:size')
41  to
42   derived_PA : mmMIGRARIAMVC!Derived_PA (
43    name <- beanSize.getAttributeValue('id'),
44    expression <- 'size('+beanSize.getAttributeValue('name')+'.'
45          +beanSize.getAttributeValue('property')+')',
46    direction <- #output,
```

```
47     parentobject <- thisModule.presentationObject(Tuple{beanName=beanSize.getAttributeValue('
           name'),
48                     objectContainer=
49                     if beanSize.getHierarchyOfCompositesElements()->any(var | var.oclIsTypeOf(
                           mmStrutsJSP!JSPActionStrutsLogicIterate)).oclIsUndefined() then
50                      beanSize.getPageContainer()
51                     else
52                      beanSize.getHierarchyOfCompositesElements()->any(var | var.oclIsTypeOf(
                            mmStrutsJSP!JSPActionStrutsLogicIterate))
53                     endif})
54    )
55 }
```

A.3. Reglas para el Controlador

A.3.1. Reglas para flujos de control y respuesta

```
1  rule actionsPackage2controller {
2   from
3    actionsPackage : mmJava!Package (actionsPackage.getFullPackagePath()=thisModule.
         fullPackageActionsClasses)
4   to
5    controller : mmMIGRARIAMVC!Controller (
6     controlflows <- actionsPackage.ownedElements->select(element | element.
          oclIsTypeOf(mmJava!ClassDeclaration))
7                 ->select(classDeclaration | classDeclaration.isActionClass())
8                 ->collect(actionClass | actionClass.getClassDeclarationControlFlow())
9    )
10  do {
11   mmMIGRARIAMVC!MVC.allInstancesFrom('OUT').first().refSetValue('controller',
         controller);
12  }
13 }
14
15 lazy rule forward2response {
16  from
17   forward : mmStrutsConfig!ForwardType
18  to
19   response: mmMIGRARIAMVC!Response (
20    name <- forward.name
21   )
22
23 }
24
25 lazy rule forwardPage2PageResponse extends forward2response {
26  from
```

```
27    forward: mmStrutsConfig!ForwardType (forward.path.endsWith('.jsp'))
28   to
29    response: mmMIGRARIAMVC!PageResponse (
30     page <- forward.getPage()
31    )
32  }
33
34  lazy rule forwardAction2ControlResponse extends forward2response {
35   from
36    forward: mmStrutsConfig!ForwardType (forward.path.endsWith('.do'))
37   to
38    response: mmMIGRARIAMVC!ControlResponse (
39     controlflow <- thisModule.getAllActionsClassDeclaration()->
40          any(actionClass | actionClass.getFullName()=thisModule.getStrutsConfigAction(
                forward.path).type).getClassDeclarationControlFlow()
41    )
42  }
```

A.3.2. Reglas para las instancias de controlador

```
1   rule variableDeclarationFragment2controllerInstance {
2    from
3     variableDeclarationFragment : mmJava!VariableDeclarationFragment (not
          variableDeclarationFragment.getContainerControlFlow().oclIsUndefined() and
4                       not variableDeclarationFragment.getCIAttributes().oclIsUndefined
                            ())
5    using {
6     controllerInstanceAttributes :  TupleType(type : mmMIGRARIAMVC!DataTypes, object :
           mmMIGRARIAMVC!DataObject, collection : mmMIGRARIAMVC!CollectionsTypes) =
          variableDeclarationFragment.getCIAttributes();
7    }
8    to
9     controllerInstance : mmMIGRARIAMVC!ControllerInstance (
10     name <- variableDeclarationFragment.name
11     ,controlflow <- variableDeclarationFragment.getContainerControlFlow()
12     ,type <- controllerInstanceAttributes.type
13     ,object <- controllerInstanceAttributes.object
14     ,collection <- controllerInstanceAttributes.collection
15    )
16  }
17
18  unique lazy rule anonymousControllerInstance {
19   from
20    methodInvocation : mmJava!MethodInvocation
21   to
```

```
22    controllerInstance : mmMIGRARIAMVC!ControllerInstance (
23     name <- 'anonymous-'+methodInvocation.getName()
24     ,controlflow <- methodInvocation.getContainerControlFlow()
25    )
26  }
```

A.3.3. Reglas para llamadas a operaciones

```
1   rule methodInvocationDataOperation2DataOperationCall {
2    from
3     methodInvocationDO : mmJava!MethodInvocation (not methodInvocationDO.
          getContainerControlFlow().oclIsUndefined() and
4                   methodInvocationDO.isDataOperationInvocation() and
5                  not methodInvocationDO.isDataOperationByRelationInvocation())
6    using {
7     dataOperation : mmMIGRARIAMVC!DataOperation = methodInvocationDO.getDataOperation
          ();
8    }
9    to
10    dataOperationCall : mmMIGRARIAMVC!DataOperationCall (
11     name <- methodInvocationDO.getName()
12     ,params <- methodInvocationDO.arguments->collect(argument | thisModule.
           dataOperationCallParameter(argument, dataOperation.parameters.at(
           methodInvocationDO.arguments.indexOf(argument))))
13     ,return <- if methodInvocationDO.refImmediateComposite().oclIsTypeOf(mmJava!
           ExpressionStatement) then
14          OclUndefined
15           else
16             thisModule.operationCallReturn(methodInvocationDO)
17          endif
18     ,controlFlow <- methodInvocationDO.getContainerControlFlow()
19     ,dataoperation <- dataOperation
20    )
21  }
22
23  rule methodInvocationDataOperationByRelation2DataOperationCall {
24   from
25    methodInvocationDO : mmJava!MethodInvocation (not methodInvocationDO.
          getContainerControlFlow().oclIsUndefined() and
26                  methodInvocationDO.isDataOperationInvocation() and
27                  methodInvocationDO.isDataOperationByRelationInvocation())
28
29   using {
30    dataOperation : mmMIGRARIAMVC!DataOperation = methodInvocationDO.getDataOperation
          ();
```

```
31   }
32   to
33    dataOperationCall : mmMIGRARIAMVC!DataOperationCall (
34     name <- methodInvocationDO.getName()
35     ,params <- methodInvocationDO.arguments->collect(argument | thisModule.
         dataOperationCallParameter(argument, dataOperation.parameters.at(
         methodInvocationDO.arguments.indexOf(argument))))
36     ,params <- dataOperationCall.params.append(contextParam)
37     ,return <- if methodInvocationDO.refImmediateComposite().oclIsTypeOf(mmJava!
         ExpressionStatement) then
38          OclUndefined
39           else
40             thisModule.operationCallReturn(methodInvocationDO)
41           endif
42     ,controlFlow <- methodInvocationDO.getContainerControlFlow()
43     ,dataoperation <- dataOperation
44    )
45    ,contextParam : mmMIGRARIAMVC!DataOperationCallParameter (
46     name <- methodInvocationDO.expression.getName()
47     ,instance <- methodInvocationDO.expression.getInstanceAccess()
48    )
49  }
50
51  rule addMethodInvocationOverCollection2CollectionOperationCall {
52   from
53    methodInvocationOverCollection2CollectionOperationCall : mmJava!MethodInvocation (
        not methodInvocationOverCollection2CollectionOperationCall.
        getContainerControlFlow().oclIsUndefined() and
54                          not methodInvocationOverCollection2CollectionOperationCall.
                              isDataOperationInvocation() and
55                          not methodInvocationOverCollection2CollectionOperationCall.
                              isDataOperationByRelationInvocation() and
56                           methodInvocationOverCollection2CollectionOperationCall.
                               method.abstractTypeDeclaration.name='Collection' or
57                           methodInvocationOverCollection2CollectionOperationCall.
                               method.abstractTypeDeclaration.name='Set')
58   to
59    collectionOperationCall : mmMIGRARIAMVC!CollectionOperationCall (
60     controlFlow <- methodInvocationOverCollection2CollectionOperationCall.
         getContainerControlFlow()
61     ,instance <- methodInvocationOverCollection2CollectionOperationCall.expression.
         getInstanceAccess()
62     ,name <- methodInvocationOverCollection2CollectionOperationCall.method.name
```

```
63      ,params <- methodInvocationOverCollection2CollectionOperationCall.arguments->
            collect(argument | thisModule.operationCallParameter(argument))
64      ,return <- if methodInvocationOverCollection2CollectionOperationCall.
            refImmediateComposite().oclIsTypeOf(mmJava!ExpressionStatement) then
65          OclUndefined
66           else
67             thisModule.operationCallReturn(
                   methodInvocationOverCollection2CollectionOperationCall)
68           endif
69      ,type <- if methodInvocationOverCollection2CollectionOperationCall.method.name='
            add' then
70          #add
71          else
72           if methodInvocationOverCollection2CollectionOperationCall.method.name.
                startsWith('remove') then
73           #clear
74          else
75           OclUndefined
76          endif
77          endif
78    )
79  }
```

A.3.3.1. Reglas para los parámetros de llamadas a operaciones

```
1   lazy rule dataOperationCallParameter {
2    from
3     methodArgument : mmJava!Expression,
4     dataOperationParameter : mmMIGRARIAMVC!DataOperationParameter
5    to
6     dataOperationCallParameter : mmMIGRARIAMVC!DataOperationCallParameter (
7      name <- methodArgument.getName()
8      ,instance <- methodArgument.getInstanceAccess()
9      ,parameter <- dataOperationParameter
10    )
11  }
12
13  lazy rule operationCallParameter {
14   from
15    methodArgument : mmJava!Expression
16   to
17    operationCallParameter : mmMIGRARIAMVC!OperationCallParameter (
18     name <- methodArgument.getName()
19     ,instance <- if methodArgument.oclIsTypeOf(mmJava!MethodInvocation) then
20          if methodArgument.method.isGetMethod() then
```

```
21          thisModule.anonymousControllerInstance(methodArgument.refImmediateComposite
                ())
22         else
23          methodArgument.getInstanceAccess()
24         endif
25          else
26           methodArgument.getInstanceAccess()
27          endif
28   )
29 }
```

A.3.3.2. Reglas para los retornos de llamadas a operaciones

```
1  lazy abstract rule operationCallReturn {
2   from
3    methodInvocation : mmJava!MethodInvocation
4   to
5    operationCallReturn : mmMIGRARIAMVC!OperationCallReturn (
6     return <- methodInvocation.getDataOpReturn()
7
8    )
9  }
10
11 lazy rule operationCallReturnVariableDeclarationFragment extends operationCallReturn
       {
12  from
13   methodInvocation : mmJava!MethodInvocation (methodInvocation.refImmediateComposite
         ().oclIsTypeOf(mmJava!VariableDeclarationFragment))
14  using {
15   instanceAccess : mmJava!VariableDeclarationFragment = methodInvocation.
         refImmediateComposite();
16  }
17  to
18   operationCallReturn : mmMIGRARIAMVC!OperationCallReturn (
19     name <- 'DeclarationFragment '+instanceAccess.getName()
20    ,instance <- if instanceAccess.isConvertToControllerInstance() then
          instanceAccess else OclUndefined endif
21    )
22 }
23
24 lazy rule operationCallReturnAssignment extends operationCallReturn {
25  from
26   methodInvocation : mmJava!MethodInvocation (methodInvocation.refImmediateComposite
         ().oclIsTypeOf(mmJava!Assignment))
27  using {
```

```
28    instanceAccess : mmJava!VariableDeclarationFragment = methodInvocation.
          refImmediateComposite().leftHandSide;
29   }
30   to
31     operationCallReturn : mmMIGRARIAMVC!OperationCallReturn (
32     name <- 'Assignment '+instanceAccess.getName()
33     ,instance <- instanceAccess.getInstanceAccess()
34    )
35  }
36
37  lazy rule operationCallReturnMethodInvocation extends operationCallReturn {
38   from
39    methodInvocation : mmJava!MethodInvocation (methodInvocation.refImmediateComposite
          ().oclIsTypeOf(mmJava!MethodInvocation))
40   using {
41    containerMethodInvocation : mmJava!MethodInvocation = methodInvocation.
          refImmediateComposite();
42   }
43   to
44    operationCallReturn : mmMIGRARIAMVC!OperationCallReturn (
45     name <- 'MethodInvocation '+containerMethodInvocation.method.name
46     ,instance <- thisModule.anonymousControllerInstance(containerMethodInvocation)
47    )
48  }
49
50  lazy rule operationCallReturnClassInstanceCreation extends operationCallReturn {
51   from
52    methodInvocation : mmJava!MethodInvocation (methodInvocation.refImmediateComposite
          ().oclIsTypeOf(mmJava!ClassInstanceCreation))
53   using {
54    containerMethodInvocation : mmJava!MethodInvocation = methodInvocation.
          refImmediateComposite();
55   }
56   to
57    operationCallReturn : mmMIGRARIAMVC!OperationCallReturn (
58     name <- 'ClassInstanceCreation '+containerMethodInvocation.method.name
59     ,instance <- methodInvocation.refImmediateComposite().refImmediateComposite().
           getInstanceAccess()
60    )
61  }
62
63  lazy rule operationCallReturnNoContemplado extends operationCallReturn {
64   from
```

```
65    methodInvocation : mmJava!MethodInvocation (not methodInvocation.
          refImmediateComposite().oclIsTypeOf(mmJava!VariableDeclarationFragment) and
66                not methodInvocation.refImmediateComposite().oclIsTypeOf(mmJava!
                      Assignment) and
67                not methodInvocation.refImmediateComposite().oclIsTypeOf(mmJava!
                      MethodInvocation) and
68                not methodInvocation.refImmediateComposite().oclIsTypeOf(mmJava!
                      ClassInstanceCreation))
69   to
70    operationCallReturn : mmMIGRARIAMVC!OperationCallReturn (
71     name <- 'Incompleto'.debug('error 4; Tipo OperationCallReturn no contemplado. '+
           methodInvocation.refImmediateComposite().oclType().toString())
72    )
73  }
```

A.3.4. Reglas para mapeos entre datos

```
1  rule actionFormGetMethodInvocation2inputAttrMapping {
2   from
3    actionFormGetMethodInvocation : mmJava!MethodInvocation (
         actionFormGetMethodInvocation.method.isGetMethod() and
4                 actionFormGetMethodInvocation.method.refImmediateComposite().
                      extendsStrutsForm() and
5                 not actionFormGetMethodInvocation.getContainerControlFlow().
                      oclIsUndefined())
6    using {
7    presentationObjectName : String = actionFormGetMethodInvocation.
         getContainerControlFlow().getStrutsConfigAction().name;
8    presentationAttributeName : String = actionFormGetMethodInvocation.method.
         getAccessAttributeName();
9    presentationObject : mmMIGRARIAMVC!PresentationObject = let request :
         mmMIGRARIAMVC!Request = actionFormGetMethodInvocation.getContainerControlFlow
         ().getRequest() in
10                if request.oclIsUndefined() then
11                 OclUndefined.debug('No hay request de entrada al controlFlow '+
                       actionFormGetMethodInvocation.getContainerControlFlow().name)
12                else
13                 request.getPOSendInObjectRequestParameter(presentationObjectName)
14                endif;
15   controllerInstance : mmMIGRARIAMVC!ControllerInstance = let ci : mmMIGRARIAMVC!
         ControllerInstance = actionFormGetMethodInvocation.refImmediateComposite().
         getSetControllerInstanceInInputMapping() in
16                if ci.oclIsUndefined() then
17                 thisModule.anonymousControllerInstance(
                       actionFormGetMethodInvocation)
```

```
18                  else
19                   ci
20                  endif;
21  }
22  to
23   inputAttrMapping : mmMIGRARIAMVC!AttrMapping (
24    name <- presentationObjectName+'.'+presentationAttributeName+'2'+
          controllerInstance.name
25    ,type <- #input
26    ,presentationattribute <- presentationObject.getAttribute(
          presentationAttributeName)
27    ,attribute <- let attribute : mmMIGRARIAMVC!MVCAttribute =
          actionFormGetMethodInvocation.refImmediateComposite().
          getSetAttributeInInputMapping() in
28          if attribute.oclIsUndefined() then
29           controllerInstance
30          else
31           attribute
32          endif
33    ,objectMapping <- thisModule.inputObjectMappings(Tuple{ presentationObj :
          mmMIGRARIAMVC!PresentationObject = presentationObject,
34                  controlFlow : mmJava!MethodDeclaration =
                        actionFormGetMethodInvocation.getContainerControlFlow()})
35   )
36   do {
37    if (controllerInstance.oclIsTypeOf(mmJava!VariableDeclarationFragment)) {
38     inputAttrMapping.objectMapping.objects<-inputAttrMapping.objectMapping.objects.
          append(thisModule.resolveTemp(controllerInstance, 'controllerInstance'));
39    } else {
40     inputAttrMapping.objectMapping.objects<-inputAttrMapping.objectMapping.objects.
          append(controllerInstance);
41    }
42   }
43 }
44
45
46 unique lazy rule inputObjectMappings {
47  from
48   inputObjectMapping : TupleType( presentationObj : mmMIGRARIAMVC!PresentationObject
        , controlFlow : mmJava!MethodDeclaration)
49  to
50   objectMapping : mmMIGRARIAMVC!ObjectMapping (
51    name <- inputObjectMapping.presentationObj.name+'2'
52    ,presentationobject <- inputObjectMapping.presentationObj
```

```
53      ,type <- #input
54      ,controlFlow <- inputObjectMapping.controlFlow
55    )
56  }
57
58  rule actionFormSetMethodInvocation2outputAttrMapping {
59   from
60    actionFormSetMethodInvocation : mmJava!MethodInvocation (
          actionFormSetMethodInvocation.method.isSetMethod() and
61                    actionFormSetMethodInvocation.method.refImmediateComposite().
                          extendsStrutsForm() and
62                    not actionFormSetMethodInvocation.getContainerControlFlow().
                          oclIsUndefined())
63   using {
64    presentationObjectName : String = actionFormSetMethodInvocation.
          getContainerControlFlow().getStrutsConfigAction().name;
65    presentationAttributeName : String = actionFormSetMethodInvocation.method.
          getAccessAttributeName();
66    response : mmStrutsConfig!ForwardType = actionFormSetMethodInvocation.
          getContainerControlFlow().getResponse();
67    presentationObject : mmMIGRARIAMVC!PresentationObject = response.getPage().
          getPresentationObject(presentationObjectName);
68    controllerInstance : mmMIGRARIAMVC!ControllerInstance =
          actionFormSetMethodInvocation.arguments.first().
          getGetControllerInstanceInOutputMapping();--actionFormSetMethodInvocation.
          arguments.first().getSetControllerInstanceInInputMapping();
69   }
70   to
71    outputAttrMapping : mmMIGRARIAMVC!AttrMapping (
72     name <- controllerInstance.name.toString()+'2'+presentationObjectName+'.'+
            presentationAttributeName
73      ,type <- #output
74      ,presentationattribute <- presentationObject.getAttribute(
            presentationAttributeName)
75      ,attribute <- actionFormSetMethodInvocation.arguments.first().
            getGetAttributeInOutputMapping()
76      ,objectMapping <- thisModule.outPutObjectMappings(presentationObject,
            actionFormSetMethodInvocation.getContainerControlFlow())
77    )
78   do {
79    if (controllerInstance.oclIsTypeOf(mmJava!VariableDeclarationFragment)) {
80     outputAttrMapping.objectMapping.objects<-outputAttrMapping.objectMapping.objects.
            append(thisModule.resolveTemp(controllerInstance, 'controllerInstance'));
81    } else {
```

```
82      outputAttrMapping.objectMapping.objects<-outputAttrMapping.objectMapping.objects.
            append(controllerInstance);
83    }
84   }
85  }
86
87  unique lazy rule outPutObjectMappings {
88   from
89    presentationObject : mmMIGRARIAMVC!PresentationObject,
90
91    controlFlow : mmJava!MethodDeclaration
92   to
93    objectMapping : mmMIGRARIAMVC!ObjectMapping (
94     name <- presentationObject.name
95     ,presentationobject <- presentationObject
96     ,type <- #output
97     ,controlFlow <- controlFlow
98    )
99  }
100
101 abstract rule requestSetAttribute2outputMapping {
102  from
103   requestSetAttribute : mmJava!MethodInvocation (requestSetAttribute.method.name='
        setAttribute' and
104                 requestSetAttribute.method.abstractTypeDeclaration.name='
                        ServletRequest' and
105                 not requestSetAttribute.getContainerControlFlow().oclIsUndefined())
106    using {
107   object : mmMIGRARIAMVC!ControllerInstance = requestSetAttribute.arguments.at(2).
        getInstanceAccess();
108   presentationName : String = requestSetAttribute.arguments.at(1).escapedValue.
        deleteQuotationMarks();
109   response : mmStrutsConfig!ForwardType = requestSetAttribute.
        getContainerControlFlow().getResponse();
110  }
111  to
112   outputMapping : mmMIGRARIAMVC!Mapping (
113    name <- object.name+'2'+presentationName
114    ,type <- #output
115    ,controlFlow <- requestSetAttribute.getContainerControlFlow()
116   )
117 }
118
119 rule paramMapping {
```

```
120  from
121   requestGetParameter : mmJava!MethodInvocation (requestGetParameter.method.name='
          getParameter' and
122                   requestGetParameter.method.abstractTypeDeclaration.name='
                          ServletRequest' and
123                   not requestGetParameter.getContainerControlFlow().oclIsUndefined())
124  using {
125   controllerInstance : mmMIGRARIAMVC!ControllerInstance = requestGetParameter.
          getSetControllerInstanceInInputParamMapping();
126   request : mmMIGRARIAMVC!Request = requestGetParameter.getContainerControlFlow().
          getRequest();
127   parameterRequestName : String = requestGetParameter.arguments.first().escapedValue
          .deleteQuotationMarks();
128  }
129  to
130   paramMapping : mmMIGRARIAMVC!ParamMapping (
131    name <- parameterRequestName+'2'+controllerInstance.name
132    ,controlFlow <- requestGetParameter.getContainerControlFlow()
133    ,instance <- controllerInstance
134    ,parameter <- request.getRequestParameter(parameterRequestName)
135    ,type <- #input
136   )
137 }
```

Anexo B

Entorno de Modernización

B.1. Necesidad

La complejidad intrínseca de un proceso de modernización supone la necesidad de contar con una herramienta que asista a los ingenieros encargados de llevarla a cabo. Entre otras funcionalidades esta herramienta debe ser capaz de:

1. Generar modelos a partir de los artefactos software de la aplicación web heredada.
2. Definir y gestionar la ejecución de la cadena de transformaciones definida para un determinado proceso de modernización.
3. Simplificar la toma de decisiones del ingeniero aportando la información precisa y la visualización adecuada de la misma. Algunos escenarios de uso serían:
   a) Proporcionar una vista de todos los elementos relacionados con la generación de una determinada vista.
   b) Proporcionar una vista de todos los elementos relacionados con la atención de una determinada petición.
   c) Proporcionar una vista con todos los elementos dentro del subsistema seleccionado para modernizar.
4. Medir la cobertura y precisión de la representación conceptual obtenida a partir del sistema heredado.

Siguiendo la estrategia definida en el desarrollo de esta tesis, para la construcción de esta herramienta se pretende hacer uso de otras herramientas ya existentes que aporten un valor directo al proceso de modernización, como por ejemplo el proyecto MoDisco.

B.2. MoDisco Discoverers

Al día de hoy, los esfuerzos realizados en la construcción del entorno de modernización se han centrado en el desarrollo de inyectores de código a modelo para poder procesar todos los tipos de artefactos software presentes en las aplicaciones web heredadas consideradas. Para la construcción de estos inyectores se ha decidido extender el proyecto MoDisco mediante la creación de dos nuevos discoverers: uno que permita procesar archivos de configuración de Struts 1 y otro que procese archivos de mapeo objeto-relacional del framework de persistencia Hibernate.

B.2.1. Procedimiento de construcción de un discoverer

Para desarrollar un discoverer los pasos a seguir son los siguientes:

1. Creación de un proyecto de tipo *MoDisco Project* en Eclipse
2. Una vez creado el proyecto creamos dentro de este un elemento de tipo *MoDisco Discoverer*. Se mostrará una pantalla de configuración en la que se deben indicar diferentes propiedades como son:

   a) *Source folder*: directorio donde se generará el código del discoverer.

   b) *Package*: paquete de Java al que pertenecerá el código

   c) *Class name*: nombre de la Java que implementa el discoverer

   d) *Input type*: tipo de entrada soportada por el discoverer se puede indicar si serán archivos simples, carpetas o proyectos.

   e) *Discovery parameters declaration*: se podrán definir parámetros de entrada, salida o entrada/salida que soporte el discoverer.

   f) Indicar si el discoverer manejará como salida modelos,

   g) Proporcionar un nombre al discoverer.

Los pasos anteriores generan una clase Java que extiende a la clase *AbstractModelDiscoverer*. Esta nueva clase debe implementar los siguientes métodos:

- El método *basicDiscovererElement* contiene el código de ejecución principal del discoverer.
- El método *isApplicableTo* para comprobar la adecuación de la entrada.
- El método *initializeResourceFactory* inicializa el objeto *ResourceXML* correspondiente a la versión del archivo que se desea analizar. Permite que el discoverer soporte diferentes versiones de los archivos de entrada, evitando tener que generar un discoverer para cada versión.

B.2.2. Discoverer StrutsConfig

El discoverer *StrutsConfig* se ha construido siguiendo la estructura de los discoverers para WebApp de MoDisco, que permiten analizar los archivos de despliegue de una aplicación web JEE. Estos discoverers aprovechan la funcionalidad de EMF que permite la persistencia y carga de modelos en formato XML cuando el metamodelo se ha generado a partir de un esquema de XML. Sin embargo, ha sido necesario definir un proceso ligeramente diferente, porque los modelos obtenidos no eran soportados directamente por el lenguaje de transformación ATL debido a que no incorporaban información de su metamodelo.

Finalmente, el proceso principal definido por este discoverer cuenta con los siguientes pasos:

1. Cargar el modelo mediante un objeto de la clase *XMLResource*.
2. Obtener el objeto raíz del esquema (*StrutsConfig*).
3. Crear un objeto de la clase *XMIResource* a partir del objeto raíz.
4. Guardar el modelo mediante el método save de este objeto.

Adicionalmente, se ha implementado el método *initializeResourceFactory* para proporcionar soporte a todas las versiones del archivo de configuración de Struts 1.

Este discoverer ha sido contribuido al proyecto MoDisco[1].

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=393523