UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

PLESG:
Propuesta de Línea de Ensamblado de Software de Gestión

TESINA
Máster en Ingeniería del Software, Métodos Formales y Sistemas de Información

Autor:
David Vicent Ferrer

Diciembre 2013

Dirigida por:
Óscar Pastor López

Índice general

Tabla de Anexos

Tabla de Figuras

Capítulo 1 - Introducción

En este capítulo se establecen las bases fundamentales para el desarrollo del presente trabajo final de master. Primero presentamos la motivación, una hipótesis de partida y el objetivo a cumplir. Finalmente se presentará la metodología utilizada para el desarrollo de esta tesis de máster junto con el desarrollo temporal de la investigación y la organización de este documento.

1.1 Motivación

Durante los últimos quince a veinte años, la tecnología asociada al desarrollo del software ha evolucionado de forma espectacular, hasta el punto que hoy en día los desarrolladores son capaces de construir aplicaciones cuya concepción y desarrollo sería muy difícil, si no imposible, hasta hace poco tiempo. Sin embargo, el esfuerzo requerido para generar una simple aplicación basada en formularios es actualmente mucho mayor, principalmente debido a la complejidad creciente no sólo de los requerimientos funcionales, sino también de la infraestructura necesaria y de las tecnologías asociadas a la aplicación en cuestión [1. R. Hopkins]. Esto ha llevado, en consecuencia, a un aumento proporcional de la complejidad del proceso de desarrollo y de su organización.

La industria ha intentado en repetidas ocasiones la confrontación entre los costes crecientes y la complejidad. El desarrollo basado en componentes, los frameworks, el uso de librerías estándares de código, la generación automática de código y las metodologías ágiles de desarrollo fueron todos introducidos con la idea de que contribuirían significativamente a la optimización del desarrollo del software. El grado de éxito de cada iniciativa es ciertamente variable (y discutible), pero en cualquier caso ninguna ha probado ser decisiva en este campo. Es más, muchas de las tecnologías más prometedoras, tales como herramientas CASE, lenguajes de 4ª generación (4GL) o el desarrollo basado en modelos han fallado estrepitosamente en proporcionar las expectativas generadas.

Esto puede ser entendido en el contexto de estas herramientas, consideradas como solución para lo que habitualmente se conoce como “complejidad accidental”. Este concepto fue introducido en la disertación “No Silver Bullet” [2. F. Brooks], donde se distingue entre la complejidad accidental, que surge puramente de desajustes en la elección particular de herramientas y los métodos aplicados en la solución, y la complejidad esencial, que se refiere a aquellas situaciones donde todas las soluciones razonables a un problema dado deben ser complicadas (y posiblemente confusas) porque las soluciones “simples” no resolverían el problema de forma satisfactoria. Veamos un ejemplo para entender mejor el problema subyacente.

Un ejemplo de complejidad esencial

Las aplicaciones desarrolladas bajo la tecnología ASP.Net MVC son ejemplos típicos de aplicaciones Web de última generación. Están basadas en estándares de la industria y principios de arquitectura sólidos y bien conocidos.

Como se describe en la siguiente figura, aplicaciones como NerdDinner[1] están construidas bajo una arquitectura distribuida basada en capas. Cada capa es responsable de un conjunto diferente de actividades: desde el acceso a datos, la lógica de dominio y/o las reglas de negocio, hasta la lógica de presentación de la aplicación Web ejecutada en el navegador del usuario.

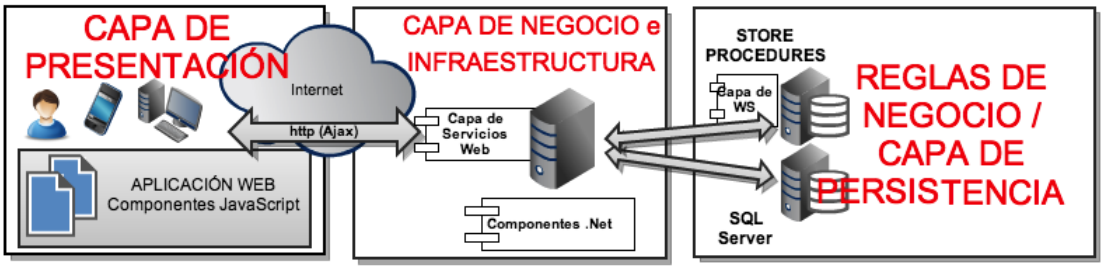


Figura 1.1: ejemplo de arquitectura distribuida basada en capas

Estas capas están altamente desacopladas. Funcionan independientemente y se comunican a través de interfaces bien definidas. La capa de presentación es responsable de la entrada y salida de datos en el cliente, mientras que la capa de negocio encapsula las reglas de negocio claramente definidas. Las capas son soportadas por el middleware que es responsable de los servicios de infraestructura (seguridad, disponibilidad, interoperabilidad y escalabilidad). Para cada capa encontramos un conjunto diferente de lenguajes de programación, tecnologías de componentes, plataformas y herramientas distintas que son necesarias.

En el caso de la aplicación NerdDinner, se hace uso de varios lenguajes de programación (XML, HTML5, JQuery, C# o Visual Basic .Net), una tecnología de bases de datos, dos protocolos de comunicación distintos, múltiples versiones posibles para el software del cliente (navegadores y/o sistemas operativos) y un amplio abanico de componentes, todos dependiendo del entorno asociado. Además se sigue una arquitectura estricta de tipo MVC (Model-View-Controller), lo cual facilita el desarrollo y sobre todo el mantenimiento de la aplicación.

Por tanto puede concluirse que la arquitectura de aplicación global es compleja por el diseño. Es así para poder ser implementada y garantizar operaciones seguras y la mantenibilidad del sistema. Sin una arquitectura así, una aplicación de esta escala y alcance sería simplemente irrealizable.

[1] Ver: http://www.nerddinner.com

En tanto que la complejidad accidental puede ser minimizada utilizando buenas arquitecturas, diseños e implementaciones, la complejidad esencial es inherente al dominio del problema y por tanto inevitable. Cualquier solución efectiva orientada a la complejidad accidental tiene por tanto un efecto limitado sobre el total del esfuerzo del desarrollo del proyecto. Esto explica por qué el uso de frameworks, programación basada en componentes y similares tienen un impacto limitado sobre la naturaleza del desarrollo software. Esto es así porque la mayoría de las soluciones orientadas a contrarrestar la complejidad, como herramientas y metodologías, realmente lo que están luchando es contra la complejidad accidental. Su fracaso subyace en que no fueron diseñadas o no son adecuadas para abordar los aspectos relacionados con la complejidad esencial.

Los famosos argumentos esgrimidos por Brooks se centran en la idea que no existe ninguna varita mágica, método o herramienta infalible que permita incrementar significativamente la productividad del programador. Sin embargo, en la opinión de Brooks esta limitación a la productividad del programador únicamente aplica a la complejidad accidental. Él sostiene que mejoras relativas a la complejidad esencial podrían ofrecer ganancias de productividad significativas, si no enormes. En este caso tampoco existe ninguna varita mágica, por lo que se concluye que no existen soluciones milagrosas en la reducción del aumento de la complejidad y por tanto sufrir altos costes de desarrollo son hechos que deben asumirse. Como veremos más adelante esto no tiene porqué ser así. Brooks sí argumentó que un cambio fundamental en el proceso de desarrollo, acompañado de ciertas innovaciones asociadas, podría atacar la complejidad esencial y por tanto llevarnos a mejoras significativas en la productividad.

En los últimos años ha aparecido una nueva aproximación al desarrollo del software denominada MDD (Model-Driven Development o Desarrollo Dirigido por Modelos) o MDSD (Model-Driven Software Development) que se considera un nuevo paradigma dentro del campo de la ingeniería del software. MDD eleva el nivel de abstracción de los lenguajes tradicionales mediante el empleo de modelos, permitiendo de ese modo utilizar conceptos cercanos a los dominios de los problemas. Además existen estudios [3. Middleware Company] donde se ha demostrado como el paradigma MDD puede incrementar la productividad hasta un 37%.

Sin embargo, MDD no es el único paradigma que clama un aumento en la productividad. Existen fuentes [4. Hibbs, 2007] que argumentan cómo el desarrollo de aplicaciones basadas en frameworks de desarrollo ágil, en este caso con Ruby on Rails (RoR), puede conseguir multiplicar x5 (o x10) la productividad frente a desarrollos más tradicionales (por ejemplo, basados en Java) sin hacer sacrificios en la calidad de la aplicación.

Por otra parte, las iniciativas más recientes, lideradas por fabricantes como IBM o Microsoft, han introducido el concepto de “Software Factory” (SF). Esto vendría a ser similar a una línea de ensamblado de software en la que se aplican los principios y técnicas propios de los procesos de manufactura pero en este caso al desarrollo de software. Como en la

producción tradicional en fábricas, la parte de ingeniería reside en la recogida de requerimientos del sistema y en la creación de los componentes/artefactos. Microsoft y Siemens Research han establecido, por medio de casos de estudio, que la productividad del desarrollo de software siguiendo este concepto podría llegar a ser multiplicada x10 (punto 2.1.3: un ejemplo de viabilidad).

Llegados a este punto, aún nos queda por realizar un pequeño análisis sobre la evolución de la complejidad del software y la influencia que ejerce sobre éste el factor humano.

Si durante los últimos quince a veinte años, la tecnología asociada al desarrollo de software ha evolucionado de forma espectacular, hasta el punto que hoy en día los desarrolladores son capaces de construir aplicaciones cuya concepción y desarrollo sería muy difícil, si no imposible, hasta hace bien poco tiempo, la pregunta obligada es:

¿por qué el desarrollo de software, incluso contando con estos avances, se torna cada vez más complicado y no más sencillo?

Parece existir una cierta contradicción entre las herramientas disponibles y tecnología por un lado y en el otro, la creciente complejidad del proceso de producción de software, incluso para aplicaciones relativamente menores. Y aunque tanto software como hardware han experimentado un crecimiento potencial enorme, la complejidad total de los requerimientos funcionales, infraestructura y plataformas tecnológicas ha aumentado de forma mucho más rápida.

Hace veinte años una aplicación informática consistía típicamente en un conjunto de procesos relativamente lineales o transacciones, a menudo con *operativa batch*. Su uso estaba restringido a un número limitado de usuarios, un par de docenas a lo sumo. Muchas de estas aplicaciones estaban intrínsecamente ligadas a una determinada plataforma tecnológica y carecían de una interfaz de usuario rica.

Hoy día las aplicaciones necesitan operar en entornos complejos y heterogéneos, a la vez que proporcionan una arquitectura por capas, soportada por una compleja infraestructura que garantice entre otros aspectos; usabilidad, tiempos de respuesta, escalabilidad, etc. Adicionalmente la industria ya no consiste únicamente en el desarrollo y despliegue de nuevos sistemas software. Cada vez más entornos consisten, en gran medida, en sistemas y aplicaciones “legacy”. Esto implica que cualquier nueva arquitectura software debe considerar coexistir con el software ya existente. Esta circunstancia ha llevado a la aparición de aplicaciones compuestas, que crean sinergias aprovechando un amplio conjunto de fuentes de información diversas dentro de la organización [5. M. Fowler]. Los componentes pueden ser funciones seleccionadas de otras aplicaciones, o sistemas enteros que han sido empaquetados como funciones de negocio, módulos o Web Services. Este escenario va más allá que el concepto reciente de arquitecturas orientadas a servicios (SOA). En la

práctica las aplicaciones compuestas hacen uso de cualquier tipo de tecnología o arquitectura disponible [6. Amberpoint]. Y aunque muchas de las aplicaciones no necesiten objetivamente soportar miles de usuarios concurrentes, la influencia de la "mentalidad Web" es tal que las prácticas del desarrollo moderno para Internet acaban siendo usadas también en el resto de proyectos de desarrollo.

Llegamos al aspecto más limitante de todo el proceso, el factor humano. El increíble volumen de tecnologías que necesitan ser integradas en el transcurso de un proyecto medio supone una pesada losa sobre los recursos habitualmente escasos. Si antes un programador podía crear una aplicación trabajando con una única herramienta, hoy se requieren de cinco a diez lenguajes y herramientas distintos para construir una aplicación de complejidad media. En este contexto los Ingenieros de Software tienen la ardua tarea de decidir cual de estos es el mejor, atendiendo a criterios como satisfacción de los requerimientos, velocidad de desarrollo, complejidad, escalabilidad, mantenimiento, etc. Por ejemplo, un ingeniero de software podría decidir entre J2EE y ASP.Net. Si elige J2EE, entonces ahora debe decidir entre Strust o Spring, Hibernate o EJB, etc. En la mayoría de los casos la decisión estará basada en la propia experiencia con el uso del framework y/o del lenguaje de programación, pero a veces el dilema crece si se analizan nuevas tecnologías donde no existe experiencia previa, por ejemplo:

- Model Driven Architecture (MDA) o Model Driven Development (MDD)
- Unified Modeling Language (UML) o Domain Specific Language (DSL)
- Modelado Textual o Gráfico

Por todo lo anterior, existen multitud de debates en Internet con criterios similares a la hora de seleccionar un determinado framework/paradigma de desarrollo. Cada elección tiene sus propios pros y contras, y si bien una combinación de opciones podría incluso ser más difícil de evaluar, este proceso podría verse incluso agravado por las incompatibilidades existentes entre algunas opciones debido a sus características inherentes, por ejemplo, la elección del paradigma MDA elimina la opción del Modelado Textual.

Además de la diversidad de paradigmas, herramientas y lenguajes de programación, las plataformas subyacentes y las tecnologías que dan soporte a las mismas son mucho más diversas, y especialmente en entornos profesionales de programación. Y para añadir aún más dificultad, el desarrollador debe lidiar con los aspectos de red, conectividad, seguridad, disponibilidad de ancho de banda y escalabilidad, además de entender el sistema operativo y el software del servidor sobre el que se instala. Sin contar con que es cada vez más difícil encontrar desarrolladores cualificados que puedan manejar y construir aplicaciones asociadas a las tecnologías más emergentes, así como responsables de proyecto que puedan tomar decisiones acertadas acerca de su equipo y proyectos.

Ese incremento de la complejidad tecnológica unido al factor humano nos ha llevado a un incremento en la complejidad del proceso de desarrollo y de

la organización del mismo, y aunque ese aumento en la complejidad del desarrollo del software es inevitable e incluso necesario, pues las actuales aplicaciones serían inalcanzables sino se contara con la base de los conceptos y prácticas avanzadas sobre los que la arquitectura software ha establecido sus pilares, resulta importante señalar el aumentado considerable del nivel mínimo de cualificación intelectual requerido para trabajar en cualquier nuevo proceso de desarrollo de software actual.

En este contexto, la principal motivación de este proyecto surge por la necesidad de dar una solución al aumento de la complejidad del proceso de desarrollo de software y de su organización dentro de las pequeñas y medianas empresas donde muchas veces no se entienden los sobrecostes producidos por dicha complejidad. La idea es simplificar los actuales procesos de desarrollo en dichas organizaciones con el objetivo final de abaratar los costes totales. Para ello intentaremos reducir la complejidad esencial diseñando un nuevo proceso, basado en los nuevos paradigmas y metodologías de desarrollo del software arriba mencionados.

1.2 Hipótesis

Entre las muchas herramientas y tecnologías que han sido desarrolladas a través de los años, algunas de ellas han pretendido precisamente capturar el proceso completo de generación de software. Las herramientas CASE, los lenguajes de cuarta generación (4GL) y el desarrollo basado en modelos profetizaron un cambio radical en la naturaleza del desarrollo de software. La mayoría de ellas han fracasado en el cumplimiento de sus objetivos por diversas razones, entre ellas:

- Ofrecen soluciones extremadamente dependientes de una determinada tecnología de fabricante.

- Las soluciones no eran mantenibles o incluso usables sin la presencia de las propias herramientas. Las aplicaciones no podían ser mantenidas por otros medios.

- Las herramientas no eran usables por los expertos del dominio, generalmente personas no expertas en informática.

- Las soluciones generadas no abordaban los requerimientos no funcionales de manera tan efectiva como las soluciones actuales basadas en capas.

Quizás sea posible reducir la complejidad sin enfocarse únicamente en componentes o partes individuales del proceso de desarrollo u ofreciendo soluciones cerradas y dependientes del fabricante. Pero si limitar la complejidad de ciertas partes del sistema parece no surtir el efecto deseado, ¿existe alguna alternativa?.

Sí, quizás sea posible hacer esto en el marco de un proceso que genere soluciones acorde con los estándares, independiente del fabricante, fácilmente mantenibles, que otorguen suficiente margen para la parametrización e intente reducir la complejidad del sistema y del proceso para desarrollarlo en su globalidad. Recientemente han surgido nuevos intentos de atacar el problema, tal y como lo planteó Brooks hace aproximadamente veinte años atrás, mencionados en el punto anterior:

- por una parte, algunos fabricantes como IBM o Microsoft están convencidos de que actualmente es posible alcanzar estos requisitos y han empezado a introducir estas ideas en su abanico de productos en lo que se conoce como Software Factory. El objetivo es abstraer y extraer los puntos en común del proceso de desarrollo y automatizar su producción en un proceso, modelado bajo el concepto subyacente de una línea de ensamblado.

- por otro lado, cuando se utiliza la aproximación de desarrollo MDD, una de las principales ventajas es que son los expertos en un dominio concreto de conocimiento los que pueden realizar programas (o una parte importante de estos) a través de una herramienta de modelado. En este sentido, existen metodologías que siguen esta aproximación. Podemos destacar [7], pues a partir de ésta se han extraido parte de las ideas sobre las que se basa la propuesta planteada.

- sin embargo, las actuales propuestas orientadas al desarrollo de aplicaciones Web mediante el uso de frameworks de desarrollo ágil que están basadas en MDD también ofrecen soluciones productivas y de alta calidad.

A partir de estas ideas se plantea la siguiente hipótesis de partida:

Es posible que expertos en un dominio de conocimiento específico sean capaces de crear y cambiar satisfactoriamente sus aplicaciones software

Sin embargo, es muy razonable que necesiten ayuda por parte de personal técnico, típicamente de un equipo de desarrollo, para integrar los artefactos generados (a partir de los modelos especificados) con el resto del código fuente de la aplicación, además de compilar cada cambio, realizar pruebas, realizar modificaciones en la/s base/s de dato/s, generar informes de rendimiento o, por ejemplo, volver a desplegar.

1.3 Objetivo

De la hipótesis citada anteriormente se deriva el objetivo general de esta tesis de master:

Crear un proceso de desarrollo o "línea de montaje" centrada en la producción acelerada de aplicaciones de gestión Web para frameworks[2] de desarrollo ágil basados en MDD

En concreto, se pretende acelerar el proceso de desarrollo de dichas aplicaciones mediante la adopción de una *Software Factory* centrada en el uso de una "cadena de herramientas" (tool chain), las cuales abarcan una serie de tecnologías que van desde la creación y uso de Lenguajes Específicos de Dominio o *Domain Specific Languages* (DSL), el desarrollo dirigido por modelos, la captura de metadatos del sistema, la generación de código y otras formas de automatización.

Dentro de este proyecto, aparece la necesidad de implementar, por una parte, un DSL que permita a los analistas funcionales especificar los distintos modelos del Análisis, y por otra parte, la creación de distintos traductores, los cuales, partiendo de los modelos definidos anteriormente, generarán automáticamente una serie de artefactos que podrán extenderse a conveniencia y que deberán integrarse en la aplicación Web (carrocería).

El uso de un lenguaje específico de dominio es en este caso interesante, pues se conseguirían dos sub-objetivos principales:

- Es más adecuado para especificar modelos y procesos de negocio frente a lenguajes de programación de propósito general.

- Permite el uso del lenguaje por personal no informático (expertos del dominio o analistas funcionales) lo cual facilita la implicación directa de estos perfiles en la creación de la aplicación. *Este es el objetivo que, una vez cumplido, validaría la hipótesis de partida***.**

El objetivo general nos lleva a construir un prototipo que permita acelerar el proceso de desarrollo del software, elevando el nivel de abstracción y convirtiendo así el modelo en la implementación del sistema, facilitando la tarea de crear o modificar una aplicación por parte de los analistas funcionales de pequeñas y/o medianas empresas mediante el uso de un lenguaje específico de dominio que les permita cierta independencia con los expertos informáticos.

En el siguiente apartado se presentará la metodología utilizada para el desarrollo de esta tesis de máster, junto con el desarrollo temporal de la investigación, exponiendo los principales hitos en orden cronológico. Finalmente, en el último apartado, se especificará la organización de este documento.

[2] Frameworks open source tales como Grails, Django, RoR, etc., están basados todos ellos en MDD.

1.4 Metodología

El presente trabajo tiene una orientación de carácter investigador y se ha desarrollado en colaboración con miembros del Departamento de Sistemas Informáticos y Computación (DSIC) de la Universidad Politécnica de Valencia (UPV) para el máster en Ingeniería del Software, Métodos Formales y Sistemas de Información (ISMFSI).

Este trabajo de investigación ha sido desarrollado en varias etapas, utilizando un enfoque iterativo e incremental. En la figura 1.2 se muestra el marco metodológico de la investigación con referencias temporales.

Tras finalizar mis estudios como Ingeniero Informático en la Facultad de Informática de la UPV (Septiembre 2004), tuve la suerte de trabajar primero en CARE Technologies® (periodo 2004 – 2006) y posteriormente en Sogeti® (periodo 2006 – 2012). En el primer caso pude experimentar en primera persona la puesta en práctica de *Model Driven Arquitecture* (MDA) [8]. En el segundo caso, concretamente durante el periodo 2010/2012, fue cuando pude adquirir un amplio conocimiento en el desarrollo tradicional de aplicaciones Web basadas en frameworks de desarrollo ágil, junto con el uso avanzado de librerías JavaScript para la capa de presentación. Un aspecto importante que pude observar en ambas compañías fue que dejaban fuera de su alcance a las pequeñas (y medianas) empresas.

El punto inicial de la investigación surge al comenzar mis estudios en el máster de ISMFSI en 2010. Ese año, en la asignatura Ingeniería del Lenguaje Natural (ILN), donde se utilizó Python como lenguaje de programación de las diferentes actividades, se hizo hincapié en cómo puede contribuir el procesamiento del lenguaje natural (PLN) al desarrollo de software y al modelado de sistemas de información.

A partir de los conocimientos adquiridos durante el primer curso académico 2010/2011 del máster y de la propia experiencia profesional me aventuré a desarrollar lo que sería una primera versión de PLESG; formado por una gramática basada en un parser[3] descendente recursivo y un generador de código basado en Python. Dicho parser, a partir de un simple modelo textual de entrada (diagrama de clases), generaba un *Abstract Syntax Tree* (AST) el cual era recorrido por el generador de código con el fin de generar diferentes artefactos textuales que se integraban dentro de un proceso de desarrollo de aplicaciones Web basadas en Django. La idea era elevar el nivel de abstracción mediante la especificación formal de requisitos. Sin embargo, este primer prototipo fue descartado por varios motivos; primero, debido a que el propio parser dejó de tener soporte, segundo, porque el proceso de construcción de la gramática era bastante complejo, y por último, no se disponía de un Entorno de Desarrollo Integrado (IDE, de sus siglas en ingles) que facilitara aspectos tan deseables como la validación de modelos/sintaxis, un asistente de contenido, colorido sintáctico, etc.

[3] Parser LEPL: http://www.acooke.org/lepl/

En el siguiente año académico (2011/2012) cursé, entre otras, las asignaturas Introducción a MDA (IMD), Gestión de Modelos (GMO) e Ingeniería de Sistemas de Información (ISI).

Es importante señalar la adquisición de nuevos conocimientos gracias a la investigación de diferentes artículos científicos, tesis y trabajos prácticos realizados en cada una de dichas asignaturas. Por una parte, en ILN analicé como trabajo final de dicha asignatura el artículo [9], donde se pone en práctica el PLN para la obtención de modelos conceptuales. En IMD analicé un interesante artículo [10], donde se destaca la importancia del tratamiento de los Requisitos No Funcionales (RNFs ó NFRs) dentro de cualquier proceso de producción de software fiable y eficiente. Para GMO se analizó la tesis de máster [11], donde pude investigar otra herramienta basada en MDA (Moskitt). Por último, pude experimentar en ISI, gracias a la realización de un ejercicio práctico, la importancia de la participación de los expertos del dominio en el desarrollo de los modelos conceptuales. En todos ellos obtuve la calificación de 10.

Los conocimientos adquiridos en dichas asignaturas, junto con la experiencia del primer proyecto PLESG, sirvieron para marcar el comienzo del presente trabajo de investigación, que consistió en profundizar en los problemas detectados bajo el punto de vista del desarrollo de software dirigido hacia las pequeñas y medianas empresas. A partir del estado del arte realizado y de las conclusiones obtenidas se inició el desarrollo de esta tesina, estableciendo de forma precisa el problema a resolver y los objetivos e hipótesis de partida.

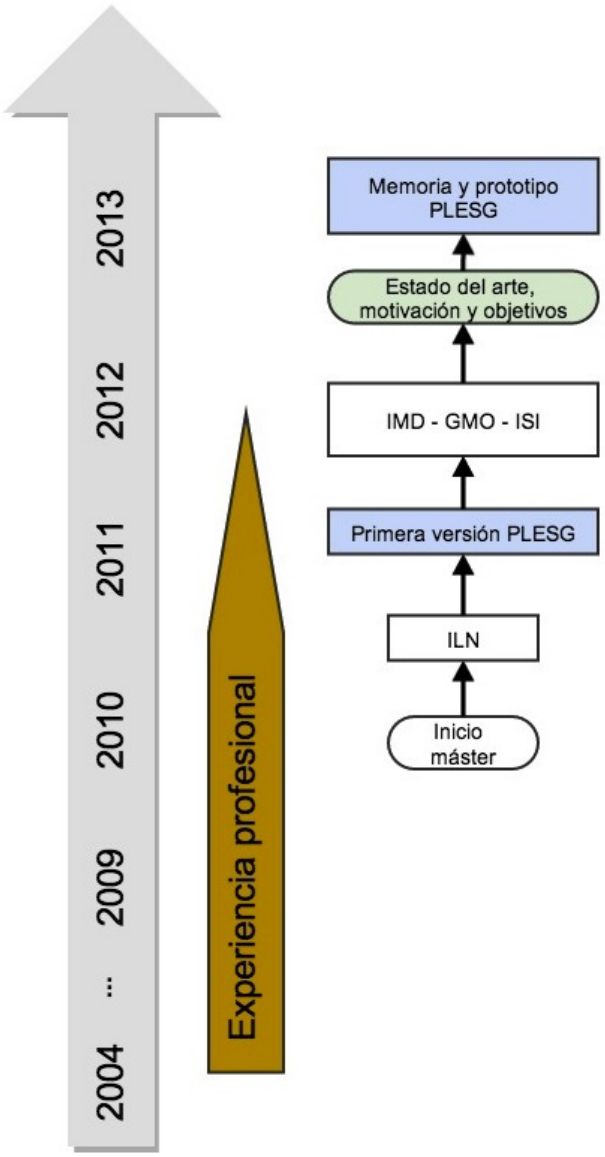


Figura 1.2: Metodología utilizada en la investigación

1.5 Organización de este documento

Esta Tesis de máster, que aborda todas las fases de un trabajo de investigación tecnológica, está estructurada en varios capítulos. A continuación se describe la organización principal del presente documento con el objetivo de facilitar su lectura.

- Planteamiento del problema

  Capítulo 1: se centra en asentar la motivación, justificación y objetivos de la Tesis. También refleja la metodología general utilizada, su desarrollo temporal y la presente organización estructural.

- Diseño de la propuesta basada en un marco teórico

  Capítulo 2: se aborda el marco teórico donde se explican los beneficios que aporta una arquitectura basada en capas, así como los conceptos básicos sobre MDE y de las Software Factories, necesarios para una mejor comprensión de la propuesta planteada. Así pues, tras analizadar las bases teóricas se presenta el diseño de la Propuesta de Línea de Ensamblado de Software de Gestión (PLESG).

  Capítulo 3: se analiza el contexto tecnológico y de infraestructura requerida para albergar la nueva propuesta, analizando las diferentes herramientas que den un soporte total/parcial al proceso definido.

- Aplicación del proceso PLESG

  Capítulo 4: análisis detallado de cada una de las fases definidas en el proceso PLESG mediante la adopción de un caso de estudio concreto.

- Implementación del prototipo

  Capítulo 5: una vez analizado el proceso y establecido el contexto tecnológico nos centraremos en los pasos requeridos para la construcción de un prototipo inicial.

- Validación y conclusiones

  Capítulo 6: validación del proceso PLESG mediante un ejemplo real aplicado.

  Capítulo 7: conclusiones, trabajo futuro e incidencias detectadas.

Finalmente, tras los anexos, que incluye diversa información que complementa el trabajo realizado, se incorporan las referencias bibliográficas y los acrónimos utilizados a lo largo de todo el trabajo.

Capítulo 2 - Propuesta de Línea de Ensamblado de Software de Gestión

El presente capítulo se divide en dos subapartados principales: marco teórico y solución adoptada.

Dentro del marco teórico veremos las mejoras que aporta una arquitectura basada en cuatro niveles, las ventajas de la elección de una iniciativa MDE (Model Driven Engineering o Ingeniería Dirigida por Modelos), en concreto MDD (Model Driven Development o Desarrollo Dirigido por Modelos), que define una aproximación para el desarrollo de software dirigido por modelos, y en último lugar presentamos el concepto de Software Factory, junto con un estudio de viabilidad.

Dicho marco teórico, junto con los objetivos y el análisis de los problemas anteriormente mencionados han motivado la solución adoptada, presentada al final de este capítulo.

2.1 Marco teórico

En el presente apartado se presenta la base teórica sobre la que se asienta la propuesta de este proyecto.

2.1.1 Arquitectura basada en capas

Pasar de la manera tradicional de trabajar sólo con datos, modelos y un metamodelo fijo a una arquitectura conceptual de cuatro niveles (incorporando un meta-metamodelo global y múltiples metamodelos) permite que las soluciones investigadas y las herramientas desarrolladas sean más potentes y generales. En la siguiente figura podemos observar los cuatro niveles conceptuales.

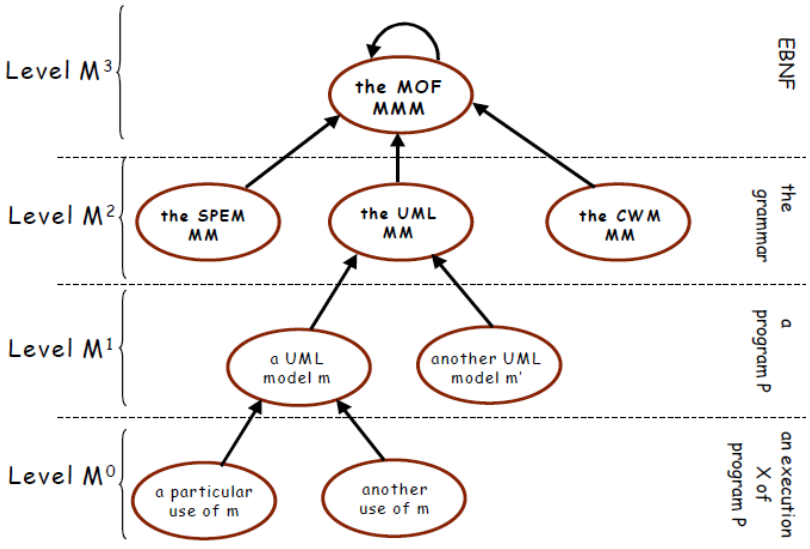


Figura 2.1: Arquitectura conceptual de cuatro niveles

El nivel M3 o **meta-metamodelo** parece una idea complicada, pero es más simple de lo que parece: esta capa consiste en un modelo compacto que sirve de cimentación para el conjunto de la arquitectura. Es el nivel más abstracto, donde se encuentra Ecore, que permite definir el lenguaje para la siguiente capa, que es el metamodelo. El meta-metamodelo es común al MetaObject Facility (MOF) y a UML, lo que asegura que compartan una base común, y asegura, por ejemplo, que los tipos de datos manejados por el lenguaje de modelado y el servicio MOF sean totalmente compatibles.

El nivel M2 o **metamodelo,** es una instancia (es decir, una expresión particular) del meta-metamodelo. Esta es la capa donde se definen los conceptos que sirven para especificar los modelos que van a crear los especialistas del dominio de la aplicación. En otras palabras, el metamodelo sirve para describir los elementos que van a componer nuestros diagramas/modelos. Es aquí donde se definen los componentes del lenguaje: Clase, Atributo, TipoDato...

De acuerdo a esta arquitectura, cuando los analistas crean el objeto Empleado en un diagrama, están creando una instancia del metaobjeto Clase; es decir, están usando Clase como si fuera el molde para producir una expresión concreta (con características propias) llamada Empleado. De igual forma, si al objeto Empleado le añadimos el atributo categoría, en realidad estamos instanciando la metaclase Atributo.

El nivel M1 o **modelo** es una instancia del metamodelo, y es lo que los expertos del dominio van a crear. Cuando creamos un modelo estamos utilizando un lenguaje (DSL) para describir el área que estamos analizando o el sistema que estamos diseñando. Cuando creamos los objetos Empleado, Departamento y Categoría estamos creando objetos que pertenecen a esta capa, y que servirán a su vez de moldes para los datos que vamos a introducir, manipular, almacenar y procesar en nuestras aplicaciones.

El nivel M0 u **objetos** del usuario, modela el sistema real. Sus elementos son los datos y objetos que describen el área o dominio a los que está dedicada la aplicación. Por ejemplo, los nombres y características de las personas almacenados en el objeto Empleado son elementos de esta capa.

En la siguiente figura veremos un ejemplo claro entre la relación de los distintos elementos de cada uno de los niveles descritos anteriormente.

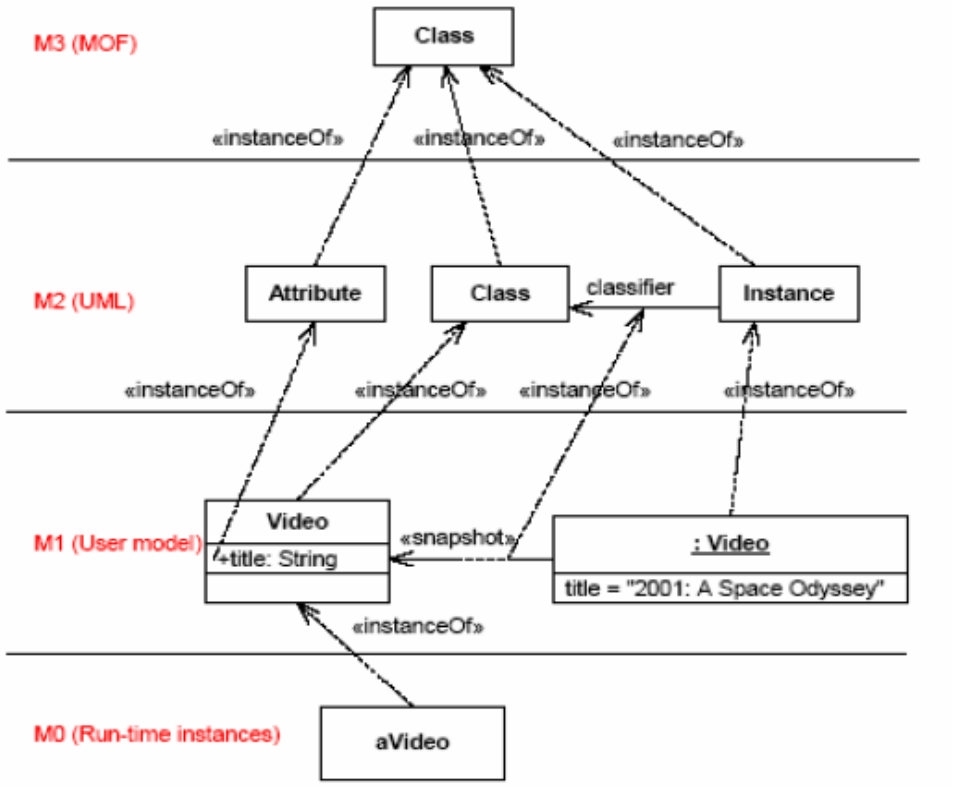


Figura 2.2: correspondencia entre niveles y sus elementos

2.1.2 Model Driven Development

El Desarrollo Dirigido por Modelos (MDD) se caracteriza por utilizar los modelos como principal artefacto para guiar el proceso de desarrollo de software, y la aplicación de pasos (semi-) automáticos para la transformación de dichos modelos. Esto implica la generación (semi) automática de los sistemas a partir de sus modelos, garantizando la fiabilidad entre los modelos y los sistemas producidos, puesto que todos viven en el mismo mundo.

MDD tiene muchos peligros [12, 1], muchos de los cuales provienen de no entender su verdadero valor y de cómo utilizarlo, por lo que los empresarios podrían dudar en adoptarlo en su empresa. Sin embargo el desarrollo de software dirigido por modelos simplifica el proceso de diseño de las aplicaciones a través del uso de modelos y de patrones de diseño que se repiten en el dominio de una aplicación. Veamos algunas razones por las cuales conviene adoptar este paradigma:

- El MDD introduce gran flexibilidad en las empresas

Flexibilidad significa la capacidad de responder al cambio y adaptarse, tanto en el aspecto técnico como en el comercial. Si seleccionas un lenguaje y escribes una solución a mano, puedes tener un tipo de flexibilidad a corto plazo (relacionada con el poder del lenguaje elegido), pero agregas mucha rigidez a largo plazo. Así que debes tener en cuenta el tiempo que deseas que dure tu inversión en el desarrollo, antes de definir qué camino tomar.

- Existen estupendas herramientas que aceptan MDD

Posiblemente la discusión aquí sería decidir bajo qué términos/licencias deseamos que operen dichas herramientas.

- Hay plataformas MDD que soportan el control de versiones

Esto permite administrar totalmente los proyectos software creados

- El equipo de requerimientos puede centrarse en el negocio

Esto les permite centrarse en el qué y no en el cómo

- Puedes vender la solución, no el paradigma

Al ofrecer aplicaciones desarrolladas bajo este paradigma no es necesario explicar qué es MDD al cliente final; basta decirle que la solución será extremadamente flexible en el largo plazo, y que conservará el conocimiento del negocio, representado en los modelos que se apliquen para su uso posterior.

- Ya no debe preocuparle la evolución tecnológica

Si tienes un negocio, probablemente querrás utilizar la mejor tecnología en cada momento, pero puedes perder tu enfoque en el negocio si necesitas preocuparte por cualquier nuevo detalle tecnológico.

2.1.3 Software Factory

Las Factorías de Software (Software Factories - SF) [13] son una iniciativa MDE para elevar el nivel de abstracción y aumentar así el grado de automatización del desarrollo de software cuando dicho software tiene unas características, funcionalidades y arquitectura comunes. Su objetivo primordial es evitar las causas conocidas que producen los problemas tradicionales en el desarrollo de software.

Hay que apreciar que el concepto SF no se refiere a una fábrica en la que se desarrolla software al igual que se haría en las fábricas de coches. Se refiere a una aproximación que eleva el nivel de abstracción para así mejorar el desarrollo de software, que tiene su origen en Microsoft, pero cuyas ideas son independientes de las tecnologías que se utilicen.

Así pues una SF promete ser, al menos como concepto, menos caro, más rápido y fiable como enfoque al desarrollo de aplicaciones. Las Software Factories tienen el potencial de cambiar significativamente las prácticas habituales del desarrollo software, al reducir los costes de la construcción de elementos reutilizables, tales como, patrones, lenguajes, frameworks y herramientas para dominios específicos, y aplicarlos para acelerar el ensamblado de las aplicaciones en estos dominios. Esto es posible debido al incremento significativo del nivel actual en la automatización del desarrollo de aplicaciones, consistiendo básicamente en una serie de tecnologías que van desde el uso de lenguajes específicos de domino (DSL), el desarrollo dirigido por modelos (MDD), la captura de metadatos del sistema, las

transformaciones del modelo especificado, la generación de código y otras formas de automatización.

Por descontado debe cuestionarse si la metodología es de veras efectiva y si el esfuerzo requerido para desarrollar la infraestructura que de soporte a una Software Factory merece la pena.

Un ejemplo de viabilidad (relación coste-beneficio)

Bien conocido es el esfuerzo realizado por Microsoft y Siemens Research para cuantificar el ahorro, o Retorno de la Inversión (ROI), que se obtendría por la adopción de una Software Factory [14].

En este caso concreto, los autores afirman que fueron capaces de incrementar la productividad del proceso de desarrollo en un factor de 10. Por supuesto, la inversión inicial en la creación de la línea de ensamblado de la Software Factory fue mucho mayor que el coste de una sola aplicación. En la siguiente figura se observa tal diferencia.

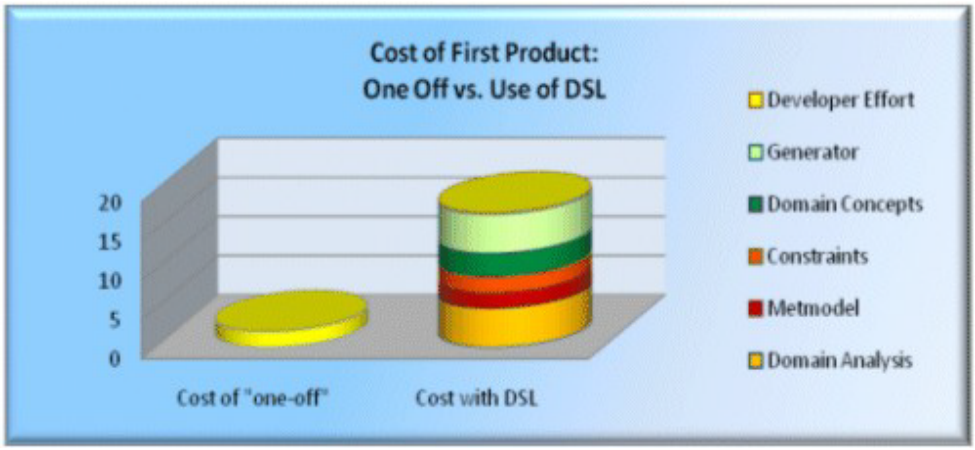


Figura 2.3: relación coste 1º aplicación a mano vs. Software Factory

Sin embargo, la productividad se incrementó de forma dramática y causó una tasa de crecimiento del ROI una vez se añadió la automatización de más componentes de la aplicación a la Software Factory. En el plazo habitual de desarrollo de un proyecto medio se podría mostrar un ROI de más del 300 por cien con una inversión mínima.

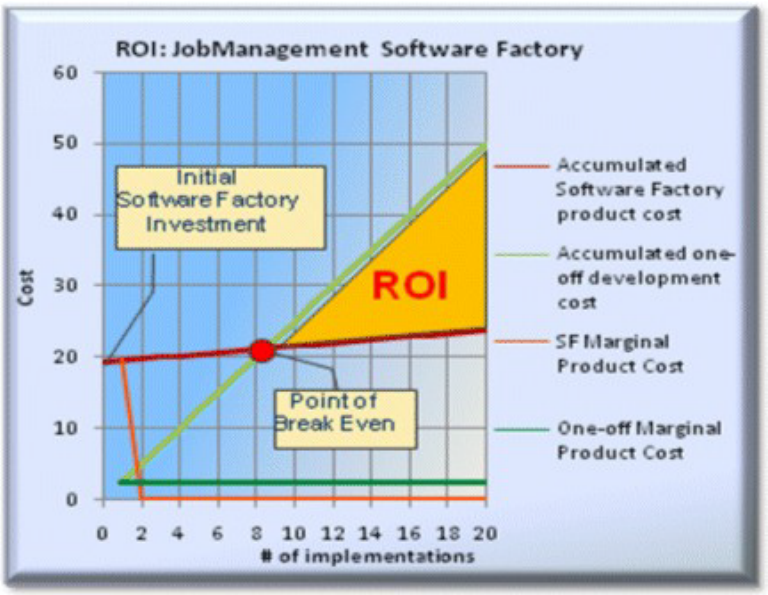


Figura 2.4: Retorno de la Inversión (ROI)

Debe hacerse notar que los incrementos más significativos se dieron como resultado del uso del Microsoft Domain-Specific-Language Toolkit y del Microsoft Guidance and Automation Toolkit (GAT). Estos módulos son específicos de Microsoft pero lo suficientemente genéricos como para establecer que con un conjunto de herramientas similares para la construcción de una "línea de ensamblado" de una Software Factory podría proporcionar ahorros comparables a los mencionados.

2.2 Solución adoptada

Es posible desarrollar aplicaciones utilizando el paradigma MDE mediante el empleo de diferentes iniciativas que, aunque parecidas, tienen diferencias a nivel conceptual y tecnológico. Por ejemplo, MDD se basa en la separación entre la funcionalidad del sistema a desarrollar y la implementación de dicho sistema para una plataforma concreta. Por tanto, busca separar claramente el análisis y el diseño de la implementación. Para conseguirlo se utilizan diferentes modelos software. Por otra parte, Model Driven Architecture (MDA), que es el método más popular de MDD, se basa en el uso de estándares industriales definidos por el Object Management Group (OMG[4]), entre ellos UML como lenguaje de modelado. Otras iniciativas, como las Software Factories son más productivas debido al beneficio de una mayor integración con la plataforma para la cual se quiera desarrollar el software pero generalmente no emplean estándares.

El problema es que las herramientas existentes para trabajar con modelos están enfocadas a trabajar con tecnologías muy diversas, lo que generalmente conlleva a tener que elegir entre trabajar con estándares o trabajar con herramientas orientadas a una determinada plataforma. Idealmente, habrá que llegar a una solución de compromiso que favorezca el

[4] Object Management Group: http://www.omg.org

uso de estándares pero que permita emplear, al mismo tiempo, tecnologías y herramientas orientadas a una determinada plataforma.

El beneficio de solventar el problema de cara a cumplir el objetivo de este trabajo sería que se podrían utilizar la mayoría de tecnologías y herramientas disponibles para trabajar con modelos, estén o no basadas en estándares. De ese modo, se lograría una independencia tecnológica para el resto de actividades de este trabajo.

Atendiendo a estos criterios, hemos encontrado en openArchitectureWare el ecosistema perfecto (ver punto 3.4), pues proporciona algunos de los elementos clave para la creación de una línea de ensamblado de software que se ajusta a nuestros objetivos y soporta el paradigma MDD.

Antes de pasar directamente a mostrar la propuesta planteada, es conveniente tener en cuenta algunas consideraciones previas.

2.2.1 Consideraciones previas

Una vez hemos despejado las dudas sobre las ventajas que aporta MDD dentro del proceso de desarrollo del software, cabe analizar su aplicación para alcanzar con éxito nuestros objetivos. Podemos afirmar que no existen recetas mágicas para aplicar con éxito el paradigma MDD dentro del proceso de desarrollo software de una empresa, ya que la mejor estrategia depende de muchos factores, principalmente:

- El tipo de aplicaciones a desarrollar
  Cuando se desarrollan aplicaciones en el ámbito empresarial, es muy importante pensar en los requisitos necesarios y en el tipo de negocio para el que se destinará la aplicación, ya que hay aplicaciones en las que prima la disponibilidad, en otras la seguridad y en otras, por ejemplo, la escalabilidad. En cualquier caso, en prácticamente todas las aplicaciones de cierta envergadura hay aspectos comunes a tener en cuenta, los cuales son independientes de la aproximación o metodología de desarrollo empleada:
    - Comunicación con la base de datos
    - Autenticación y autorización de usuarios
    - Configuración de las aplicaciones desarrolladas
    - Utilización de sistemas de registro de eventos o logging
    - Gestión de errores
    - Mecanismos de caché y control de sesión
    - Generación de informes

- El nivel de conocimientos del equipo de desarrollo

- El grado de cambio a acometer en el proceso de desarrollo del software

Dentro del desarrollo de aplicaciones de gestión empresarial, que es justo lo que proponemos en este documento, nos hemos centrado en el desarrollo

de aplicaciones de gestión Web basadas en frameworks de desarrollo ágil (Grails, Django, Ruby on Rails, etc.) basados todos ellos en MDE, los cuales proporcionan una solución integrada a muchos de los aspectos comunes a tener en cuenta.

Por otra parte, no estamos interesados en abordar una estrategia MDD "completa" que elimine por completo a los expertos informáticos del proceso de desarrollo del software. Entendemos que éstos deben formar parte en la solución pero creemos necesario minimizar su impacto, invitando a los responsables de las empresas en la participación directa del proceso.

Teniendo en cuenta estos factores, podemos establecer una serie de pautas o guías que nos ayuden a la hora de definir nuestra estrategia:

- uso de lenguajes de modelado; UML o DSL
- uso de herramientas open source o comerciales, donde hay que escoger entre el pago por uso y/o compra de una licencia.
- generación de código con sentido común, definiendo claramente qué partes son susceptibles a ser generadas.

Respecto al uso de lenguajes de modelado, algunas fuentes son más proclives a usar el paradigma MDD frente a una aproximación MDA [12, 2]. Esto es debido a que MDD promueve el uso de DSLs frente a UML, el cual puede llegar a ser demasiado complicado para personas no expertas en la materia. En este contexto, un DSL permite proporcionar al usuario un lenguaje (de modelado) perfectamente adaptado a la semántica del dominio. De hecho la idea no es nueva, por ejemplo SQL es un DSL y UsiXML, para definir interfaces gráficas, otro. Lo bueno es que ahora hay herramientas open source (textuales y/o gráficas) que facilitan mucho la creación de DSLs.

Con respecto a la generación de código, y al tratarse de un prototipo inicial, hemos aplicado la regla pareto para MDD [15]:

aprox. 20% of the modeling effort suffices to generate 80% of the application code

Esto implica que podemos mejorar y mucho nuestra productividad y conseguir la mayoría de los beneficios del modelado sin tener que dibujar modelos totalmente completos. Es decir, con un 20% de esfuerzo (sólo la definición de los aspectos estáticos del sistema) podemos generar automáticamente un 80% de todas las operaciones que tiene que ofrecer la aplicación. Por ejemplo, una gran parte de las funcionalidades de cualquier aplicación de gestión se puede deducir de:

- Los servicios CRUD de cada clase del dominio
- Generación de las interfaces de la lógica de negocio
- Gestión de usuarios y roles
- Generación de los componentes visuales: listados, formularios…
- Generación de informes a partir de los listados

Es muy probable que para muchas empresas estas funcionalidades mencionadas sean más que suficiente, o como mínimo les pueden ser de gran utilidad para empezar a experimentar los beneficios de MDD.

2.2.2 Proceso PLESG

Una Software Factory, en el entorno propuesto y según las consideraciones anteriores, supondría adoptar el proceso representado en la figura 2.5, donde podemos ver reflejadas varias fases enumeradas que constituyen la línea de ensamblado propiamente dicha.

Adicionalmente se ha representado tanto la gramática (punto A), que derivará el DSL que permitirá a los analistas funcionales especificar los distintos modelos del análisis, como los traductores (punto B), los cuales, partiendo de los modelos definidos anteriormente, generarán automáticamente una serie de artefactos que podrán extenderse a conveniencia y que deberán integrarse en la aplicación Web (carrocería).

En el capítulo 4 veremos la aplicación del proceso definido mediante un caso de estudio concreto.

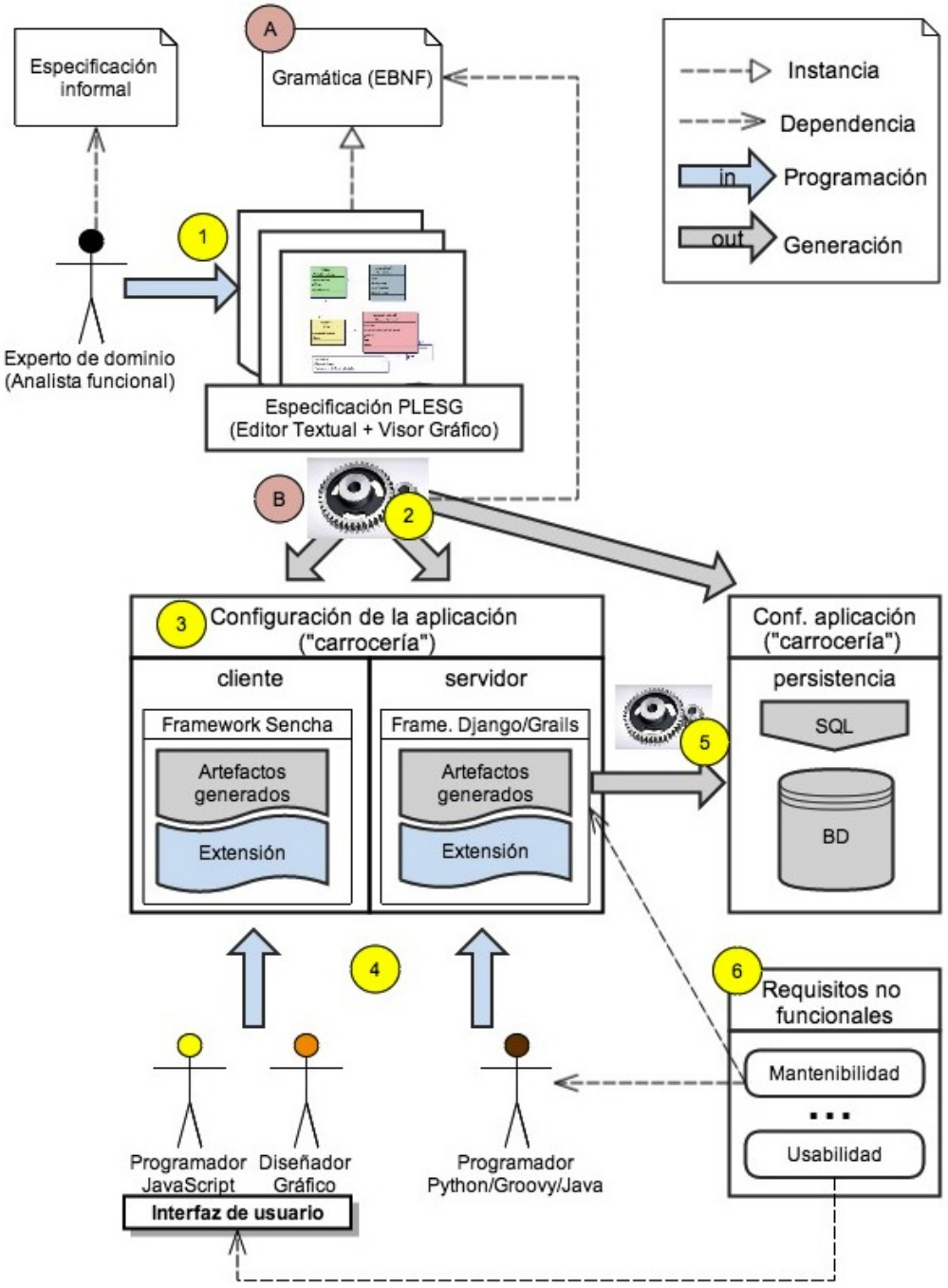


Figura 2.5: Propuesta de Línea de Ensamblado de Software de Gestión.

A continuación describimos brevemente las diferentes fases enumeradas de la figura anterior.

1. Especificación PLESG

Mediante un editor textual se construyen los distintos modelos del Análisis (objetual, dinámico, funcional y de presentación), obteniendo una visión completa del sistema. Estos modelos son creados por los expertos del dominio y/o analistas funcionales:

- Modelo de objetos: representado por medio del Diagrama de Clases, que es un modelo textual donde las clases son las

unidades básicas de modelado, y en las que se incluyen sus atributos y servicios. También se pueden representar las distintas asociaciones y herencia entre clases, y se incluye la representación de qué agentes pueden activar los servicios que proporciona cada una de las clases (CRUD, específicos y/o de navegación entre clases) así como qué atributos pueden observar en cada una de ellas.

- Modelo dinámico: utilizado para especificar las vidas válidas de los objetos de cada una de las clases y la interacción interobjetual. Para describir las vidas válidas de los objetos, utilizamos un Diagrama de Transición de Estados (uno para cada clase). Por otra parte, de ser necesaria la interacción entre los objetos, ésta se documentaría textualmente pero se implementaría en el punto tres, que forma parte de la extensión de la aplicación generada.

- Modelo funcional: se especificaría de forma textual el efecto de cada evento sobre los atributos. El valor de cada atributo se modificará dependiendo de la acción ocurrida, de los argumentos del evento y del estado actual del objeto. Si bien se generarán automáticamente las interfaces, su lógica debe ser desarrollada como parte de la extensión de la aplicación (punto tres).

- Modelo de presentación: permite de forma básica la modificación y el refinamiento de la interfaz, o lo que es lo mismo, el diseño de los listados (grids) y formularios, ambos parametrizados en función del agente. Es importante señalar que los listados deben ser especificados en función del dispositivo; desktop (por defecto) y/o móvil.

Una vez hemos especificado los distintos modelos del análisis, podemos ir a la siguiente fase.

2. Generación de los artefactos y metadatos del sistema

Invocación de los traductores para generar los diferentes artefactos textuales. Adicionalmente, se deben de generar unos artefactos (sentencias SQL) que son los metadatos de la aplicación y definen el comportamiento dinámico de ésta en tiempo de ejecución.

3. Configuración del proyecto de desarrollo

Todos los artefactos generados se integran perfectamente en lo que podemos llamar “carrocería” de la aplicación Web. Por otra parte, esta estructura deja abierta la posibilidad de incluir código no generado y/u otros componentes específicos o plugins, que pueden ser integrados fácilmente gracias a la filosofía de los frameworks de desarrollo ágil.

4. Extensión del modelo funcional:

a. implementación de las reglas de negocio específicas, interacción interobjetual, etc.
b. modificación de los estilos e imágenes

Toda la lógica de negocio debe ser desarrollada como parte de la extensión de la aplicación. Los aspectos gráficos de las aplicaciones, como imágenes y estilos, son parametrizados mediante el uso de ficheros CSS y plantillas HTML.

5. Generación del modelo de persistencia

En este punto, a partir del modelo de datos especificado para cada framework y gracias a ciertos scripts que ofrecen es posible generar automáticamente la base de datos. Adicionalmente también es necesario incluir los metadatos del sistema generados en el punto dos.

6. Estudio de los requisitos no funcionales

a. configuración de los componentes/plugins necesarios que den soporte a estos requisitos

Algunos de los requisitos no funcionales como usabilidad, mantenibilidad, seguridad, concurrencia, disponibilidad, escalabilidad, etc. deben ser tomados en cuenta a la hora de desarrollar este tipo de aplicaciones y en algunos casos a la hora de desplegar el aplicativo generado en la arquitectura elegida (servidores, sistemas operativos, etc.).

Es importante mencionar que esta fase no puede situarse en un punto concreto dentro del proceso PLESG, pues cada "requisito no funcional" tiene su dominio de actuación, según podemos ver en la siguiente tabla.

Requisito No Funcional	Dominio	Recae
usabilidad	PLESG	Analizada en el punto 4.7.3. Recae sobre el diseñador gráfico y el arquitecto software. Se ha tenido en cuenta la experiencia profesional. Es posible mejorarla mediante la modificación de las hojas de estilos, o haciendo cambios en los artefactos textuales (generados y no generados).
mantenibilidad	Framework utilizado y programadores	La facilidad del mantenimiento viene dado por el framework en uso, los cuales siguen el patrón de diseño MVC. Adicionalmente los programadores deben seguir criterios profesionales.
concurrencia	Software y Hardware	Uso de BBDD sólidas, bien balanceadas (en caso de requerirse), servidor de aplicaciones robusto, así como de servidores potentes y bien configurados.
disponibilidad		
escalabilidad		
seguridad	Plugins/ componentes	Es posible delegar dicho requisito en terceras aplicaciones/componentes.

Respecto a los plugins/componentes, estos deben estar basados en prácticas estándar, relevantes al framework en uso y su naturaleza debe ser desacoplada de la implementación interna de dicho framework. Siempre que sea posible deben ser creados de forma funcional y no intrusiva, que garantice el desacoplamiento y facilite el testing. Por ejemplo, en el punto 4.7.1 se muestra la existencia/uso de componentes relativos a la seguridad en el acceso a la aplicación. Otro tipo de

componentes puede ser el uso de protocolos SSL, firewalls, etc. con el fin de garantizar la integridad y seguridad en el acceso a la información.

2.2.3 Conclusión

Podemos concluir que la adopción de esta propuesta de línea de ensamblado de software de gestión supondría las siguientes ventajas:

- Implicación directa de los expertos del dominio y reducción de la dependencia de los programadores más experimentados.

- Creación de aplicaciones basadas en frameworks de desarrollo ágil que serán fácilmente mantenibles (y teóricamente menos costosas) incluso fuera del ámbito de la Software Factory.

- Las aplicaciones son generadas de acuerdo a las prácticas de diseño más aprobadas, por ejemplo, mediante el uso de patrones de diseño como Model View Controller (MVC), Interfaces, Singleton, etc.

- Ahorro en costes de desarrollo y mantenimiento del software.

La aplicación, tal y como es producida por este proceso, es un aplicativo muy similar tanto en arquitectura como en implementación a las aplicaciones existentes hoy en día. Por tanto ofrece las mismas ventajas de escalabilidad, seguridad y mantenibilidad en el caso de que se tomara una eventual decisión de no continuar en el futuro con la Software Factory, si bien, mientras la aplicación esté sujeta a la línea de ensamblado, la mayor parte de la misma será generada automáticamente**,** basándose en los modelos creados**/**mantenidos por los expertos de domino y/o analistas funcionales.

Antes de ver la aplicación de PLESG mediante un caso de estudio (capítulo 4) vamos a hacer un estudio relativo al contexto tecnológico en el siguiente capítulo.

Capítulo 3 - Contexto Tecnológico

En este capítulo se presentan las tecnologías y estándares existentes que han servido de base para poder realizar el prototipo.

Primero se presenta Eclipse, que es un entorno de desarrollo de código abierto multiplataforma sobre el cuál se ha desarrollado el proyecto. A continuación se comenta Eclipse Modeling Project (EMP) y en tercer lugar se exponen algunas herramientas de generación de artefactos que siguen los principios MDE (Model Driven Engineering o Ingeniería Dirigida por Modelos). Una vez analizado el estado del arte, se establecen las conclusiones sobre qué herramientas utilizar finalmente para construir el prototipo PLESG y cuya implementación puede verse en el capítulo 5.

Los puntos 3.1 y 3.2.1 han sido extraídos de [11, capítulo 2.2 y 2.3 respectivamente] donde ya se ha realizado un buen análisis de la herramienta que queremos tratar. Éstos puntos han sido incluidos con el objetivo de dar una visión de continuidad y uniformidad al presente trabajo.

3.1 Eclipse

Eclipse es una herramienta de código libre que proporciona un completo entorno de trabajo con el que podemos desarrollar fácilmente nuestros proyectos, así como disponer de una ayuda en línea muy completa. Se basa en el concepto de plug-ins, que son los encargados de darle funcionalidad a Eclipse. Cada uno de ellos le aporta algo nuevo, de modo que el conjunto de todos ellos forma una potente herramienta, que nos sirve para desarrollar nuestro propósito.

El lenguaje de programación principal que maneja Eclipse es Java. Posee una función de ayuda al programador, llamada *Intelligence*, que va auto completando el código a medida que se va escribiendo, mostrando las posibles alternativas.

El trabajo de Eclipse se basa en un proyecto central, el cual incluye un framework genérico para la integración de las herramientas, y un entorno de desarrollo Java construido utilizando el primero. Otros proyectos extienden el framework central para soportar clases específicas de herramientas y entornos de desarrollo.

El software producido por Eclipse está disponible bajo la Common Public License (CPL), la cual básicamente dice que puedes usar, modificar, y distribuir el software de manera gratuita, o incluirlo como parte de un producto propietario. CPL es una Open Source Initiative (OSI), aprobada y reconocida por la Free Software Foundation (FSF) como una licencia de software libre. Cualquier software que contribuya a Eclipse debe hacerlo bajo la licencia Common Public License (CPL).

Eclipse.org es un consorcio de compañías las cuales tienen un acuerdo para proporcionar el soporte esencial al proyecto Eclipse en términos de tiempo, experiencia, tecnología y conocimiento.

El trabajo de desarrollo en Eclipse se divide en tres proyectos principales:

- el proyecto Eclipse,
- el proyecto de Herramientas y
- el proyecto de Tecnología.

El proyecto Eclipse es el que contiene los componentes centrales necesarios que conforman una única unidad conocida como Eclipse SDK (Software Development Kit). Los componentes de los otros dos proyectos son usados para propósitos específicos y son generalmente independientes.

El proyecto Eclipse soporta el desarrollo de una plataforma, o framework, para la implementación de entornos de desarrollo integrados (Integrated Development Environments, IDEs). El framework Eclipse esta implementado utilizando Java, pero es usado para implementar herramientas de desarrollo para otros lenguajes (por ejemplo C++ y XML, entre otros).

El proyecto Eclipse (Eclipse SDK) se divide a su vez en tres subproyectos:

- La Plataforma (Platform) es el componente central de Eclipse, y es a menudo considerado como el propio Eclipse. Se utiliza para definir frameworks y los servicios requeridos para soportar la conexión e integración de las herramientas.

- En segundo lugar, las Herramientas de Desarrollo Java (Java Development Tools, JDT) proporcionan un completo entorno de desarrollo Java. Pueden ser utilizadas para desarrollar programas Java para Eclipse o para otras plataformas.

- Por último, el Entorno de Desarrollo de Plug-ins (Plug-in Development Environment, PDE) proporciona vistas y editores para facilitar la creación de plug-ins para Eclipse. Además proporciona soporte para actividades propias del desarrollo de un plug-in, como el registro de extensiones.

En conjunto, estos tres subproyectos proporcionan todo lo necesario para extender el framework y desarrollar herramientas basadas en Eclipse (ver figura 3.1 para más detalles).

El proyecto de herramientas define y coordina la integración de diferentes conjuntos o categorías de herramientas basadas en la plataforma. Por ejemplo, uno de los subproyectos más importantes del proyecto Eclipse es el EMP (Eclipse Modeling Project), analizado en el punto 3.2, que engloba todos los temas relacionados con MDE dentro de la plataforma Eclipse.

Por último, el proyecto de Tecnología proporciona una oportunidad a los investigadores y educadores para formar parte de la continua evolución de Eclipse.

3.1.1 Descripción de la plataforma

La Plataforma Eclipse es un framework para construir entornos de desarrollo integrados (IDEs, por sus siglas en ingles). Ha sido descrita como "una IDE para cualquier cosa, y para nada en particular". Su cometido es definir una estructura básica para un IDE. Las herramientas específicas extienden el framework, conectándose a él para definir un IDE particular de manera colectiva.

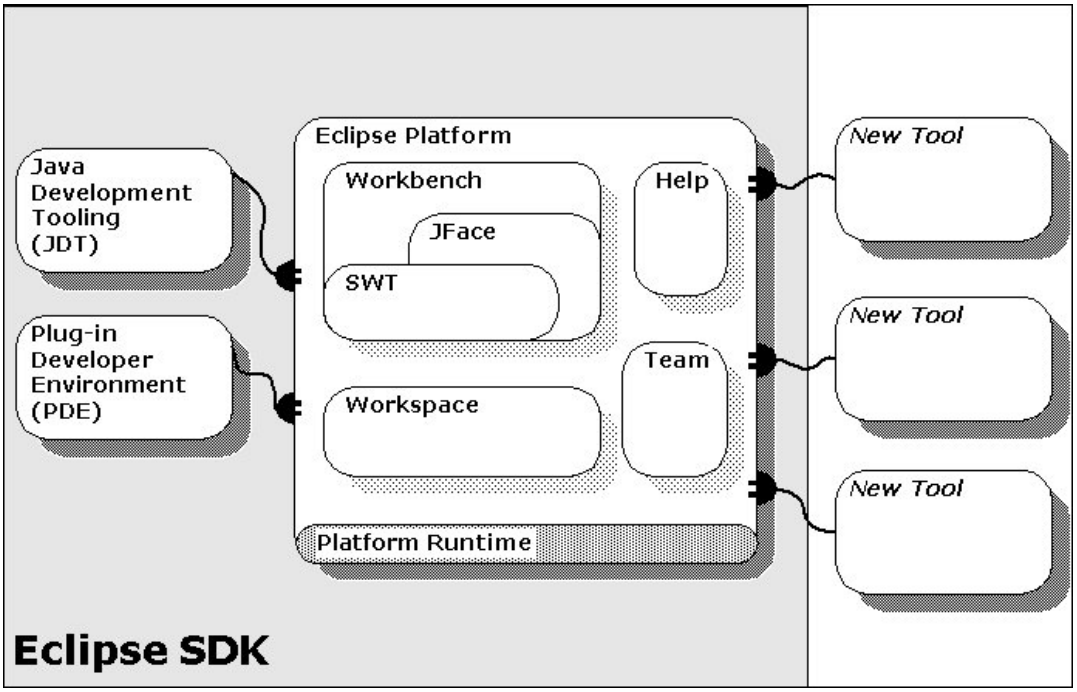


Figura 3.1: Arquitectura de la Plataforma Eclipse

Veamos una descripción más detallada de los componentes principales de la Plataforma Eclipse.

3.1.1.1 La arquitectura de plug-ins

Una unidad funcional básica, o componente, en Eclipse se conoce como plug-in. La misma Plataforma Eclipse, y las herramientas que la extienden, están compuestas por plug-ins.

Desde una perspectiva de paquetes, un plug-in incluye todo lo que se necesita para que funcione un componente, como código Java, imágenes, texto traducido, etc. También incluye un archivo de manifiesto, llamado plugin.xml, que declara las interconexiones con otros plug-ins. En este archivo se declaran, entre otras cosas, lo siguiente:

- Required plug-ins: sus dependencias con otros plug-ins.

- Exported plug-ins: la visibilidad de sus clases públicas con otros plug-ins.
- Extension points: declaración de la funcionalidad que se pone a la disposición de otros plug-ins.
- Extensions: implementación de puntos de extensión de otros plug-ins.

En el arranque, la Plataforma Eclipse (específicamente Platform Runtime) descubre todos los plug-ins disponibles y asocia las extensiones con sus correspondientes puntos de extensión. Un plug-in, sin embargo, solamente se activa cuando su código necesita ejecutarse, evitando una lenta secuencia de arranque.

Cuando se activa un plug-in, es asignado a su propia clase de carga, la cual provee la visibilidad declarada en su archivo de manifiesto.

3.1.1.2 Recursos del espacio de trabajo (Workspace)

Las herramientas integradas en Eclipse trabajan con archivos y carpetas ordinarios, pero utilizan una API de alto nivel basada en recursos (resources), proyectos (projects), y un espacio de trabajo (workspace).

Un recurso es la representación que utiliza Eclipse para los archivos y carpetas, proporcionando funcionalidades adicionales (registro de controladores de cambios en el recurso, marcadores, mantenimiento de historiales, etc.).

Un proyecto es un tipo especial de recurso que se asocia a una carpeta del usuario en el sistema de ficheros. Las subcarpetas del proyecto son las mismas que las de la carpeta física, pero los proyectos son carpetas de alto nivel en un contenedor virtual de proyectos, llamado espacio de trabajo.

3.1.1.3 El Framework UI (Workbench component)

El framework UI (User Interface) de Eclipse consiste en dos conjuntos de herramientas de propósito general, SWT y JFace, y el Escritorio de Eclipse (workbench UI).

SWT (Standard Widget Toolkit) es un conjunto de librerías gráficas independientes del sistema operativo utilizado.

JFace es un conjunto de herramientas de alto nivel, implementado usando SWT. Proporciona clases para soportar tareas comunes relativas a interfaces de usuario, como manejo de registros de imágenes y fuentes, diálogos, asistentes, monitores de progreso, etc. Una parte importante de JFace son las clases utilizadas, como visores para listas, árboles y tablas, que proporcionan un mayor nivel de conexión con los datos de SWT (por ejemplo, mecanismos de población a partir de un modelo de datos y sincronización con este). Otra parte importante de JFace es su framework para acciones, usado para añadir comandos a menús y barras de herramientas.

Por último, el Escritorio de Eclipse (workbench) es la ventana principal que el usuario ve cuando arranca Eclipse. Esta implementado mediante SWT y JFace. Como interfaz principal de usuario, se considera a menudo que es la propia Plataforma.

3.1.1.4 Soporte para el trabajo en grupo (Team component)

La Plataforma Eclipse permite asignar una versión a un proyecto en el workspace y gestionar sus versiones mediante un repositorio de trabajo en grupo. Pueden coexistir multitud de repositorios de trabajo en grupo en la Plataforma, no obstante la Plataforma Eclipse incluye, por defecto, soporte para repositorios CVS accedidos mediante protocolos pserver o ssh.

3.1.1.5 Ayuda (Help component)

Eclipse Platform Help incluye herramientas que permiten definir y contribuir a la documentación de uno o más libros en línea.

3.2 Eclipse Modeling Project

Uno de los subproyectos más importantes del proyecto Eclipse es el Eclipse Modeling Project (EMP), que engloba todos los temas relacionados con MDE dentro de la plataforma Eclipse. Tiene una serie de características como que todos los frameworks, herramientas o implementaciones de estándares dentro de EMP han de estar bajo la Eclipse Public Licence (EPL)[5] o que, pese a no estar ligado de ninguna manera al grupo OMG, implementa varios de sus estándares.

El EMP se subdivide en varios proyectos de menor alcance, siendo algunos de los más importantes los siguientes:

- Eclipse Modeling Framework
- Graphical Modeling Project
- Textual Modeling Framework

3.2.1 Eclipse Modeling Framework

El núcleo central del EMP es el Eclipse Modeling Framework (EMF). EMF es básicamente “un framework de modelado y generación de código para la construcción de herramientas y otras aplicaciones basadas en un modelo de datos estructurado”[6]. EMF ayuda además a generar código Java a partir de estos modelos de una manera fácil, correcta, personalizable y eficiente. EMF utiliza XMI como una manera canónica de definir y persistir los modelos. Además proporciona un editor gráfico para definir modelos.

[5] La EPL hace que todas las herramientas sean Open Source y de libre distribución

[6] http://www.eclipse.org/modeling

Cuando se ha especificado un MetaModelo EMF, el generador EMF puede crear una serie de clases Java que permiten crear instancias del MetaModelo y manipular sus elementos. Una vez generado el código se pueden editar las clases generadas para añadir nuevos métodos y variables. Además, se proporcionan mecanismos para la notificación de cambios en el modelo, serialización por defecto en XMI y XML Schema, un entorno de trabajo para la validación de modelos, y una API reflexiva eficiente para manipular objetos EMF de manera genérica. Y lo más importante, EMF proporciona las bases para la interoperabilidad con otras herramientas y aplicaciones basadas en EMF.

EMF consiste en dos entornos de trabajo fundamentales: el core y EMF.Edit. El entorno core proporciona la generación de código básica y el soporte en tiempo de ejecución para la creación de las clases Java del modelo. El EMF.Edit extiende y se construye sobre el core, añadiendo soporte para la generación de las clases que permiten utilizar los comandos para editar los modelos, además de activar los visores y editores gráficos básicos para un modelo.

Ecore

EMF comenzó como una implementación de la especificación MOF de OMG. En realidad EMF se puede considerar una eficiente implementación en Java de un subconjunto del core de la API de MOF (Essential MOF). Sin embargo, para evitar cualquier confusión, el MetaMetaModelo en EMF se llama Ecore. Por tanto, Ecore permite definir los vocabularios locales de dominio o MetaModelos, que permiten la creación o modificación de modelos en distintos contextos, es decir, Ecore es un vocabulario diseñado para permitir la definición de cualquier tipo de MetaModelos. Para ello proporciona elementos útiles para describir conceptos y relaciones entre ellos. En definitiva, Ecore es un subconjunto de MOF, el cual esta basado en el diagrama de clases de UML.

En la siguiente figura se muestran los elementos principales del metamodelo de Ecore.

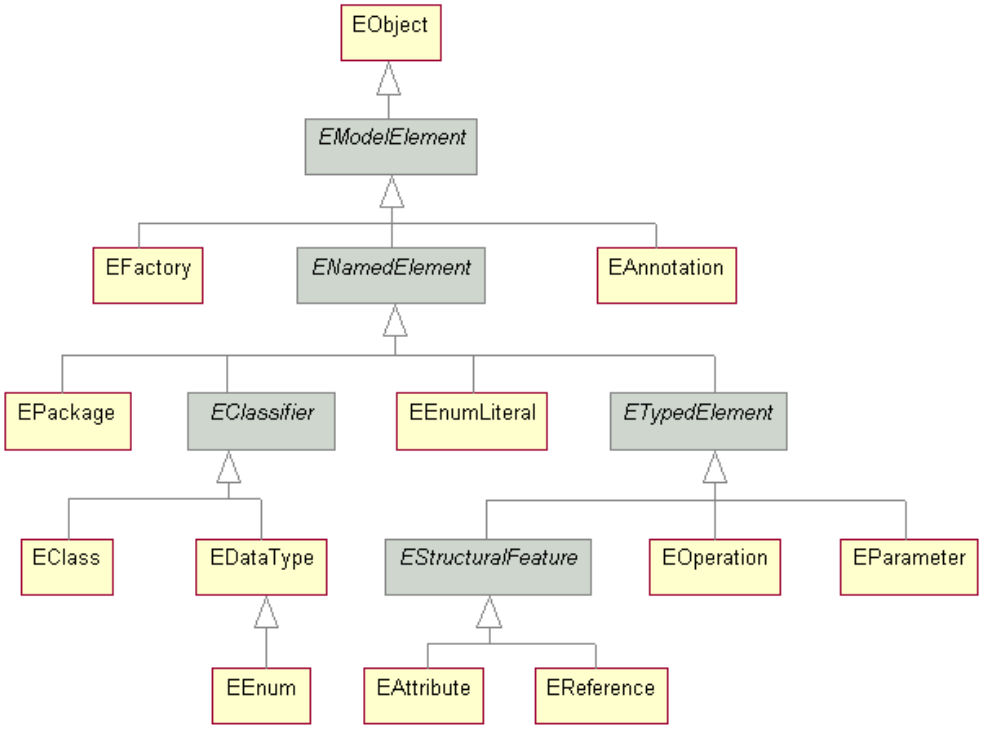


Figura 3.2: Jerarquía de clases Ecore

El elemento más importante es EClass, que modela el concepto de clase, con una semántica similar al elemento Clase de UML. EClass es el mecanismo principal para describir conceptos mediante Ecore. Una EClass está compuesta por un conjunto de atributos y referencias, así como por un número de super clases (el símil con UML sigue siendo aplicable).

A continuación se comentan otros de los elementos aparecidos en el diagrama anterior:

- **EClassifier**: tipo abstracto que agrupa a todos los elementos que describen conceptos.

- **EDataType:** se utiliza para representar el tipo de un atributo. Un tipo de datos puede ser un tipo básico como int o float o un objeto, como por ejemplo java.util.Date

- **EAttribute**. Tipo que permite definir los atributos de una clase. Estos tienen nombre y tipo. Como especialización de ETypedElement, EAttribute hereda un conjunto de propiedades como cardinalidad (lowerBound, upperBound), si es un atributo requerido o no, si es derivado, etc.

- **EReference**: permite modelar las relaciones entre clases. En concreto EReference permite modelar las relaciones de asociación, agregación y composición que aparecen en UML. Al igual que EAttribute, es una especialización de ETypedElement, y hereda las mismas propiedades. Además define la propiedad *containment*

mediante la cual se modelan las agregaciones disjuntas (denominadas composiciones en UML).

- **EPackage:** agrupa un conjunto de clases en forma de módulo, de forma similar a un paquete en UML. Sus atributos más importantes son el nombre, el prefijo y la URI. La URI es un identificador único gracias al cual el paquete puede ser identificado unívocamente.

Una de las interfaces claves en Ecore es EObject, la cual es conceptualmente equivalente a java.lang.Object. Todos los objetos modelados implementan esta interfaz para proporcionar varias características importantes:

- de forma similar a Object.getClass(), utilizando el método eClass() se puede obtener los Metadatos de la instancia. En un objeto modelado ecore se pueden utilizar los métodos reflexivos del API (eGet(), eSet()) para acceder a sus datos

- de cualquier instancia de objeto se puede obtener su contenedor (padre) utilizando el método eContainer()

- EObject extiende Notifier, lo cual permite monitorizar todos los cambios en los datos de los objetos. La interfaz Notifier introduce una característica muy importante a cada elemento del modelo: notificaciones de cambios en el modelo como en el patrón de diseño Observador.

3.2.2 Graphical Modeling Project

El Graphical Modeling Project (GMP), proporciona un conjunto de componentes e infraestructuras para desarrollar editores gráficos basados en EMF y Graphical Editing Framework (GEF). Entre los más importantes podemos destacar Graphical Modeling Framework Tooling (GMF Tooling), Graphiti, etc.

3.2.3 Textual Modeling Framework

Proporciona herramientas y frameworks para desarrollar DSLs y sus correspondientes editores textuales basados en EMF. Podemos destacar TCS, Xtext y EMFText. Los dos últimos serán analizados en el siguiente apartado.

3.3 Herramientas de generación de artefactos

Existen muchas herramientas cuyo objetivo principal es la generación de artefactos siguiendo los principios MDE. Hemos destacado en la siguiente tabla algunas de entre las que proporcionan un entorno de modelado completo: sintaxis, validación, scoping, transformaciones, etc. indicando el tipo de licencia que utilizan; open source o comercial.

Herramienta	Tipo Licencia	Enlace
Olivanova Model Execution	comercial	http://www.care-t.com/
ObjectiF	comercial	http://www.microtool.de/objectif/en/
Moskitt	open source	http://www.moskitt.org/
AndroMDA	open source	http://www.andromda.org/index.html
oAW	open source	http://www.openarchitectureware.org/
EMFText	open source	http://www.emftext.org

Hay que resaltar que EMP no es realmente una herramienta de generación de artefactos, y sí un conjunto de herramientas y utilidades que otras herramientas pueden utilizar, por eso no se ha incorporado como tal en la tabla anterior.

A continuación se comentan algunas características relevantes sobre dichas herramientas, para finalmente poder extraer las conclusiones sobre la tecnología a adoptar:

3.3.1 Olivanova Model Execution

La tecnología Olivanova Model Execution, de CARE Technologies, es una implementación de la metodología OO-Method [7] centrada en el paradigma MDA. Según se explica en [16], dicha tecnología se basa en una serie de herramientas comerciales entre las que podemos destacar la herramienta de modelado OLIVANOVA Modeler y los compiladores de modelos llamados OLIVANOVA Transformation Engines.

Concretamente OLIVANOVA Modeler es una herramienta de edición y validación de modelos conceptuales OO-Method. En términos MDA diríamos que OLIVANOVA Modeler es una herramienta de edición y validación de PIMs. La herramienta implementa un repositorio de modelos que pueden ser exportados a formato XML para su intercambio con otras herramientas.

Un OLIVANOVA Transformation Engine diríamos que es, en términos MDA, una implementación de una herramienta que opera las transformaciones de PIM a PSM y las transformaciones de PSM a *Implementation Model* (IM).

3.3.2 ObjectiF

ObjectiF, de microTOOL, es un IDE comercial bajo el paradigma MDD que da soporte a todo el ciclo de vida del desarrollo: especificación de requisitos, definición del proceso de negocio, diseñar y refactorizar la arquitectura, modelar el comportamiento dinámico y generación del código.

Utiliza diagramas de clase UML y de estados para definir el PIM y hace una clara distinción entre el PIM y el PSM. Las transformaciones entre PIM y PSM están predefinidas y soporta tecnologías como .NET, Structs, JBoss o Hibernate.

Hay versiones para Visual Studio y para Eclipse, lo que permite generar código en C++, C# y/o Java. Además la herramienta tiene funcionalidades avanzadas como soporte parcial a ingeniería inversa y permite realizar generación incremental de artefactos aunque de un modo limitado.

3.3.3 Moskitt

MOdeling Software KITT (MOSKitt) es una herramienta CASE centrada en MDA y basada en Eclipse (bajo licencia EPL) que está siendo desarrollada por la Consejería de Infraestructuras, Territorio y Medio Ambiente para dar soporte a la metodología gvMétrica (una adaptación de Métrica III a sus propias necesidades). Comentar que gvMétrica utiliza técnicas basadas en el lenguaje de modelado UML.

Para dar soporte a gvMétrica se sigue un enfoque dirigido por modelos, de forma que las tareas principales a las que debe dar soporte MOSKitt son las siguientes:

- Edición gráfica de modelos.
- Soporte a la persistencia.
- Soporte al Trabajo colaborativo y versionado de modelos.
- Transformación, Trazabilidad y Sincronización de modelos.
- Generación de Documentación y de Código DDL a partir de modelos.
- Soporte al Proceso de Desarrollo definido por gvMétrica, guiando a los usuarios en los distintos pasos que deben realizar para llevar a cabo sus tareas.

Su arquitectura de plugins la convierte no sólo en una Herramienta CASE sino en toda una Plataforma de Modelado en Software Libre para la construcción de este tipo de herramientas.

3.3.4 AndroMDA

AndroMDA es un framework open source de generación extensible de código que se adhiere al paradigma MDA.

Soporta la utilización de UML y la importación/exportación mediante modelos serializados en XMI desde herramientas CASE como por ejemplo MagicDraw o Poseidon. Los modelos de las herramientas de UML son transformados en componentes que pueden ser desplegados en plataformas como J2EE, Spring framework y/o .NET.

AndroMDA utiliza unos artefactos denominados cartridges para la generación de código. Existen gran cantidad de cartuchos para ser utilizados en frameworks como Axis, jBPM, Struts, JSF, Spring, Hibernate, etc. aunque también te permite desarrollar tus propios cartuchos generadores de código, o personalizar los existentes, gracias al cartucho Meta, y es que con él puedes construir tu propio generador de código utilizando tu herramienta UML favorita.

3.2.5 openArchitectureWare

openArchitectureWare[7] (oAW) es un conjunto de herramientas integradas para trabajar bajo el paradigma MDE sobre la plataforma Eclipse. Según sus autores es "una herramienta para construir herramientas MDD/MDA". Por un lado utiliza tecnologías proporcionadas por el EMP, como el EMF o el GMF, y por otro lado aporta nuevas tecnologías al EMP, como las siguientes:

- Xtext 2.0 al Textual Modeling Framework (TMF)

Es un framework que proporciona un entorno completo para el desarrollo textual de lenguajes de programación y DSLs mediante metamodelos basados en Ecore, pues como se observa en la figura 9, donde se aprecia su arquitectura, está basado en EMF, entre otros componentes.

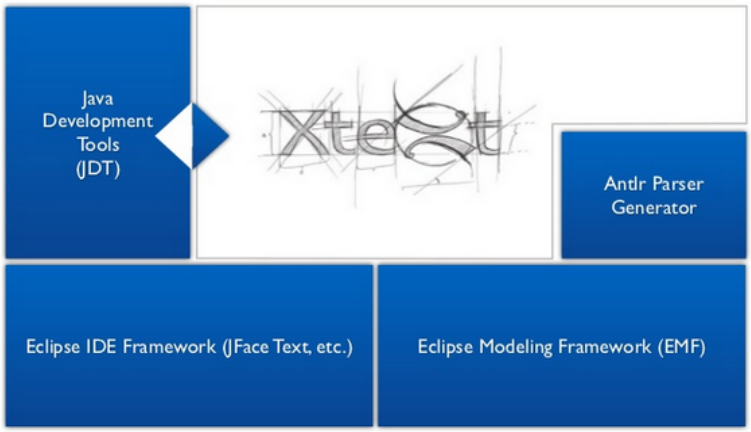


Figura 3.3: Arquitectura Xtext

- Xpand al Model to Text Transformation (M2T)

Es un lenguaje de plantillas que sirve para generar artefactos en función de un metamodelo.

- Xtend 2.0

Actualmente viene integrado con Xtext 2.0. Es un DSL que puede ser usado para la generación de código o para definir transformaciones entre modelos.

- Check

El lenguaje Check, muy próximo a Object Constraint Language (OCL), se utiliza para realizar restricciones en los modelos especificados.

- Modeling Workflow Engine 2 (MWE2)

Es un motor declarativo, completamente configurable, que sirve como punto central de las tareas que ejecutará el generador (Xtext Generator). Utiliza un lenguaje, con un sintaxis concisa, para describir los diferentes componentes que se ejecutarán secuencialmente.

[7] Dado su éxito, los principales componentes de oAW han sido migrados al Eclipse Modeling Project (EMP).

3.3.6 EMFText

EMFText es un plugin de Eclipse que permite definir DSLs mediante metamodelos basados en Ecore (EMF) de forma rápida y sin la necesidad de aprender nuevas tecnologías y conceptos por parte de los desarrolladores.

Según se indica en la documentación online, la herramienta EMFText "es indispensable para la definición de tu proceso MDD", aportando 10 razones[8] para su adopción.

3.4 Conclusión

Si bien anticipamos en el punto 2.2 que openArchitectureWare proporcionaba el ecosistema perfecto según el proceso planteado, es en este punto donde esclarecemos el porqué de dicha elección una vez se han analizado las distintas alternativas tecnológicas.

Entre las diferentes herramientas comerciales que puedan existir, ObjectiF, orientada a diferentes plataformas y adaptable a las necesidades en cuanto a generación de código, sería una buena opción a tener en cuenta. No ocurre lo mismo con Olivanova Model Execution, donde la generación de artefactos viene dada por el fabricante. Además corremos el riesgo de dejar de tener soporte con el paso del tiempo. Este ha sido el caso reciente de dos herramientas comerciales: OptimalJ, de Compuware, y ArcStyler, de io-software.

De todas formas se ha decidido descartar las soluciones comerciales debido a que uno de los objetivos perseguidos es utilizar herramientas basadas en código abierto. Dentro de ese contexto, tanto Moskitt como AndroMDA serían una excelente opción, en primer lugar porque son unas herramientas maduras (y open source), y en segundo lugar porque permiten implementar las transformaciones necesarias, en el caso de Moskitt, o extender los llamados *cartbridges*, en el caso de AndroMDA, necesarios para generar el código específico al framework elegido. Por ejemplo, djandromda es un cartucho diseñado para trabajar con los modelos diseñados en AndroMDA y convertirlos en código funcional para Django. O en el caso de Moskitt, mediante el uso del *Atlas Transformation Language* (ATL), que es un lenguaje declarativo e imperativo de transformación de modelos. Sin embargo, la hipótesis planteada al inicio, junto con las consideraciones establecidas en el punto 2.2.1, hace que nos inclinemos más hacia Xtext o EMFText.

Podemos afirmar que ambas herramientas, Xtext y EMFText, son muy parecidas [17], pero quizás al no tener ninguna preferencia por una herramienta de transformación en concreto, y el hecho de que Xtext viene integrado por defecto con Xtend y Xpand, es un motivo más que válido para declinarnos por éste último. Por el contrario, si quisiéramos optar por otras herramientas de transformación distintas a Xtend y Xpand, EMFText tendría

[8] http://www.emftext.org/index.php/EMFText_10_Reasons

mayor peso en nuestra elección.

Finalmente mencionar que Xtext puede integrarse con editores/visores gráficos, siendo la primera opción bastante más laboriosa. En el siguiente punto analizamos brevemente qué tipo de representación gráfica se ha decidido integrar con PLESG.

3.4.1 Editor/Visor Gráfico

Mientras las representaciones textuales se adaptan perfectamente para procesar los datos de un modelo de objetos, una representación gráfica puede ser muy útil para mostrar las relaciones entre dichos objetos. Por lo tanto el uso simultaneo de ambas formas de representación ayuda a mejorar la comprensión de los distintos modelos.

El ecosistema Eclipse provee una serie de editores gráficos (GEF, Graphiti, Spray, etc.) que podrían ser integrados en Xtext, pero encontramos un inconveniente en contraposición a un visor gráfico, y es debido a que la integración con un editor gráfico es bastante más laboriosa, al requerir que el mapeo sea bidireccional, lo cual implica, por ejemplo, tener la misma estructura tanto en la parte gráfica como en el lado semántico, introduciendo una mayor complejidad a nivel de compilación, generación, despliegue, etc. Es por ese motivo que hemos optado por un visor gráfico, concretamente por Generic Graph View (GGV) [18, 1]. GGV es un prototipo que permite proporcionar una representación gráfica de los diferentes modelos textuales basados en Xtext.

Actualmente PLESG proporciona una representación gráfica para el diagrama de transición de estados de un clase concreta y para el diagrama de clases. En el anexo 1 podemos ver como configurar GGV en un nuevo proyecto de desarrollo y como utilizarlo dentro de la especificación PLESG.

Capítulo 4 – Aplicación del proceso PLESG

En el presente capítulo vamos a ver como aplicar, a modo de ejemplo, las diferentes fases enumeradas que se han definido en el proceso PLESG (ver punto 2.2.2). Para ello vamos a modelar una aplicación que permite gestionar los prestamos que realizan los usuarios en una biblioteca. Como veremos a continuación, se ha decidido incluir la definición de la gramática como paso necesario para entender mejor qué tipo de modelos podemos llegar a construir. La implementación de los traductores puede verse en el punto 5.2.

1. Especificación PLESG
Veremos la especificación de los distintos modelos que conforman el sistema software a desarrollar para el caso de estudio mencionado mediante el uso de un DSL dado por la gramática especificada en el siguiente punto.

A. Definición de la gramática
Se detallará la sintaxis y las diferentes reglas que dan lugar a la gramática a partir de la cual se genera el DSL utilizado en la especificación anterior.

2. Generación de los artefactos y metadatos
Una vez se ha modelado el sistema, es posible realizar la invocación al traductor deseado con el fin de obtener los diferentes artefactos textuales.

3. Configuración del proyecto
Es necesario configurar los diferentes artefactos, generados y no generados, en la plataforma de servidor elegida (Django y/o Grails) y su integración en algún Sistema de Control de Versiones (VCS[9], de sus siglas en inglés). Sin embargo este último aspecto queda fuera del alcance del presente proyecto.

4. Extensiones específicas al código generado
En este punto se debe de realizar la implementación de la lógica de negocio correspondiente al modelo dinámico y funcional. Sería deseable su integración en algún VCS.

5. Generación del modelo de persistencia
Generación de la BD a partir del modelo de dominio generado para cada framework (Django y/o Grails) y gracias a los servicios que estos ofrecen. Adicionalmente se deberán incluir los metadatos del sistema generados que describen la parte dinámica de la aplicación.

[9] Los VCSs sirven para guardar versiones de los fuentes de desarrollo. No tendría sentido guardar los artefactos generados porque no son fuentes, los fuentes son los modelos y el código desarrollado.

6. Análisis de algunos requisitos no funcionales
Vamos a analizar ciertos requisitos no funcionales como es el caso de la seguridad en cuanto al acceso a la aplicación, la mantenibilidad y la usabilidad (análisis de la interfaz de usuario). El resto de requisitos no funcionales quedan fuera del ámbito del presente caso de estudio.

4.1 Especificación PLESG

Este punto se corresponde con la fase 1 del proceso PLESG. A continuación vamos a mostrar la especificación textual completa de la aplicación antes mencionada dentro del IDE que proporciona Xtext. El objetivo es ir viendo paso a paso como realizar una especificación PLESG mediante el DSL derivado a partir de la gramática, la cual será detallada en el apartado 4.2.2.

Los diferentes modelos del análisis (Modelo de Objetos, Dinámico, Funcional y de Presentación) que forman parte de la especificación PLESG deben de especificarse dentro de cada uno de los diferentes recursos con extensión *.model que aparecen dentro de la carpeta "model", según podemos ver en la siguiente figura, y que analizaremos a continuación.

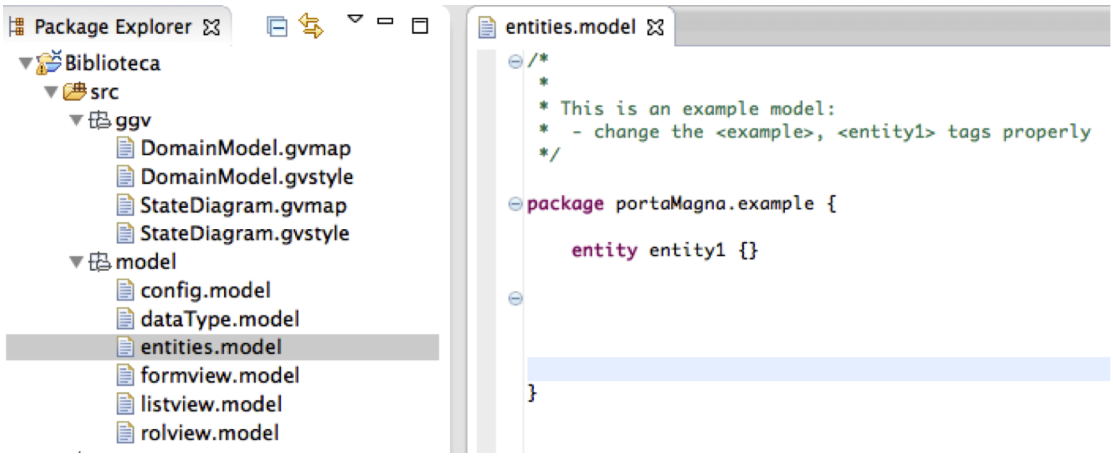


Figura 4.1: diagrama de clases – entities.model

El primer paso en la especificación PLESG es introducir el nombre del proyecto y de la aplicación en el recurso *config.model*.

Posteriormente debemos declarar en el fichero *dataType.model* los posibles tipos de datos enumerados a utilizar. En dicho recurso aparecerán definidos por defecto los posibles tipos de datos a utilizar en los diversos modelos.

```
datatype string
datatype email
datatype file
datatype integer
datatype decimal
datatype boolean
datatype date
```

```
datatype void

enum sex {
    (1 ,'M','masculino'), (2 ,'F','femenino')
}
```

En la siguiente figura se muestra la representación gráfica del diagrama de clases que queremos modelar textualmente mediante el nuevo lenguaje específico de dominio.

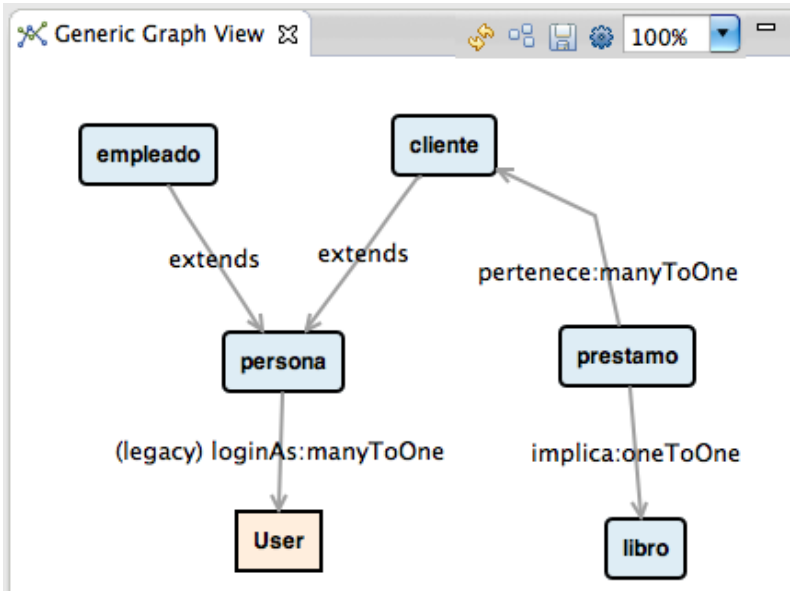


Figura 4.2: Diagrama de clases proporcionado por GGV

La representación textual del anterior diagrama de clases puede verse a continuación. Dicho contenido se debe de incluir en el fichero *entities.model*.

```
package portaMagna.biblioteca {

	legacy User

	entity persona {
		string nombre (alias: "Nombre", max_length:50)
		string apellidos (alias: "Apellidos", max_length:50)
		date nacimiento (alias: "F. nacimiento")

		edad (alias:"Edad") <= function getEdad("")
		fullname (alias:"Nombre completo") <= function getFullName("")

		sex sexo (alias:"Sexo")

		loginas manyToOne legacy User (blank:True)

		hasSessionProperty : True { sessionProperty : loginas }
	}

	entity empleado extends persona {
		integer codigo (
				alias:"Codigo",
				validation:{
					validate_even("Debe ser par"): string
				})
```

```
            estado (alias: "Estado", isSTDField: True)

            create (alias:"Guardar", type: C) : void
            update  (alias:"Actualizar", type: U) : void
            destroy  (alias:"Borrar", type: D) : void

            identity: { nombre }

            hasSTD:True{ field:estado,  alias:"Estado" }
      }

      entity cliente extends persona {
            estado (alias:"Estado", isSTDField: True)
            numero (alias:"Total prestamos") <= function getTotalPrestamos ("")

            //Servicios CRUD
            create (alias:"Guardar", type: C) : void
            update  (alias:"Actualizar", type: U) : void
            destroy  (alias:"Borrar", type: D) : void

            //Servicios Bussines Logic
            sancionar (alias:"Sancionar", type: B) : void

             //SERVICIOS COMPARTIDOS
            prestar (alias:"Prestar", type: B) : void
            devolver (alias:"Devolver", type: B) : void

            identity: { nombre }

            hasSTD: True{ field:estado, alias:"Estado" }
      }

      entity libro {
            string titulo (alias:"Titulo", max_length:50)
            string autor (alias:"Autor", max_length:50)
            estado (alias: "Estado", isSTDField:True)

            //Asociacion 1 a 1 (no aparece en GGV)
            reserva oneToOne cliente (unique:True, null:True, blank:True)

            //Servicios CRUD
            create (alias:"Guardar", type: C) : void
            update  (alias:"Actualizar", type: U) : void
            destroy  (alias:"Borrar", type: D) : void

            //SERVICIOS COMPARTIDOS
            prestar (alias:"Prestar", type: B) : void
            devolver (alias:"Devolver", type: B) : void

            //Servicios Bussines Logic
            reservar (alias:"Reservar", type:B) : void
            liberar (alias:"Liberar", type:B) : void

            //INFO. ADICIONAL TRADUCTORES
            identity: {titulo}

            hasSTD:True{ field: estado, alias: "Estado" }
      }

      entity prestamo {
            date fechaini (alias:"Fecha inicio", widget:widgetDate)
            date fechafin (alias:"Fecha fin", widget:widgetDate)
            integer renovacion (alias: "Total renovaciones", mandatory:False)
            string comentario (alias:"Comentario", max_length:255,
                  mandatory:False, widget:widgetTextarea)
```

```
            estado (alias: "Estado", isSTDField: True)
            diff(alias:"Vencimiento",help:"(dias)") <= function getDifFinIni("")

            pertenece manyToOne cliente (null:False)
            implica oneToOne libro (unique:True, null:False)

            //SERVICIOS COMPARTIDOS:
            // - Implica lanzar prestar/devolver en libro & cliente)
            prestar (alias:"Nuevo", type: C) : void
            devolver  (alias:"Cancelar", type: D) : void
            //Servicios Bussines Logic
            renovar (alias:"Renovar", type: B) : void

            hasSTD:True{ field:estado, alias:"Estado" }
    }
}
```

Lo primero que podemos destacar es que se ha creado el paquete *portaMagna.biblioteca* donde ubicar todas las entidades del diagrama de clases.

La entidad préstamo es la clase principal del sistema, pues almacenará los prestamos que realiza cada cliente. Tiene varios atributos; comentario, fecha ini./fin del préstamo y número total de renovaciones, dos atributos derivados; estado de la clase y número de días para el vencimiento del préstamo (diff), dos servicios CRUD; nuevo y cancelar, un servicio *custom* (renovar) y dos relaciones; en un préstamo interviene un libro (cardinalidad 1 - 1) y un lector (cardinalidad n - 1). Además se especifica que tiene un DTS, y que se utiliza el atributo estado para guardar las diferentes vidas válidas de dicha clase. El estado del préstamo en este caso sólo sirve para establecer el orden de ejecución de los servicios.

La entidad libro define el objeto a ser prestado a un lector (cliente) por parte de un empleado de la biblioteca. En este caso, un libro se compone de los atributos; título (que identifica al objeto) y autor, un atributo derivado (estado), tres servicios CRUD (create, update y delete) y dos servicios *custom* (reservar, liberar) y una relación con cardinalidad 1 - 1 con la clase cliente (reserva) de solo lectura. Además se especifica que tiene un DTS, y que se utiliza el atributo estado para guardar las diferentes vidas válidas de dicha clase.

La entidad persona tiene varios atributos; nombre, apellidos y fecha de nacimiento, un atributo enumerado (sexo) y dos atributos derivados (nombre completo y edad). Además una persona dada de alta en el sistema puede tener una cuenta de acceso a éste. Esto último se establece mediante la relación con la clase legada User, que contiene el login y password para acceder a la aplicación mediante un rol concreto. Puede ocurrir que un usuario tenga múltiples roles, en dicho caso, la aplicación solicitará con cual se desea acceder. Por último se ha especificado mediante la propiedad *sessionProperty* que dicha relación es usada para acceder al sistema por cualquier actor, con el fin de poder filtrar la información que aparece en los listados (en función del rol).

La entidad empleado extiende de la clase persona y por tanto hereda sus atributos, servicios y relaciones. Además tiene un atributo (código, de tipo entero) con una función de validación. El atributo heredado "nombre" es utilizado como identificador de la clase. También se especifica que tiene un DTS, y que se utiliza el atributo estado para guardar las diferentes vidas válidas de dicha clase. El estado de un empleado sólo sirve para establecer la secuencia de ejecución de los servicios.

Por otra parte, un lector (cliente), que también extiende de la clase persona, puede tener múltiples prestamos (máx. 5), así como renovar (máx. 2) los prestamos que ya tenga concedidos si no esta sancionado y además el libro no está reservado por otra persona. Si desea renovar él mismo por Internet, se le debe de asociar previamente una cuenta de usuario. La clase cliente tiene dos atributos derivados; número y estado. El primero indica el total de prestamos que tiene hasta la fecha y el segundo almacena el estado de la clase. Un lector "nuevo" no tiene prestamos asignados, pero cuando realiza alguno pasa a ser "deudor". Cuando no devuelve algún libro en la fecha indicada, el sistema/empleado puede sancionarlo, bloqueándole la cuenta hasta que devuelva todos los libros, en cuyo caso será "nuevo" otra vez.

Hay que prestar atención a los servicios custom (prestar, devolver) que son compartidos por las clases préstamo, cliente y libro. Esto quiere decir que la ejecución de dichos servicios debe ser atómica.

Respecto a los roles, se pueden definir en el fichero *src/model/drol.model*. En total se han creado cuatro roles:

- SUPER de tipo "admin", que por defecto tendrá total acceso a todos los elementos definidos en el sistema (clases, atributos, servicios, etc.)
- SOCIO, para que los clientes puedan ver sus datos, el catálogo de libros disponibles y sus prestamos, y puedan renovar éstos por Internet
- STAFF, propio de los trabajadores de la biblioteca
- GUEST, al cual sólo se le permite consultar el catálogo de libros por Internet.

```
import portaMagna.biblioteca.*

rol SUPER { //Administrador de la aplicacion
       type:admin, initialView:empleado
}
rol STAFF { //Personal que trabaja en la biblioteca
       type:user, initialView:cliente, referencedClass:empleado
}
rol SOCIO { //Usuario de la biblioteca
       type:user, initialView:libro, referencedClass:cliente
}
rol GUEST { type:user, initialView:libro }
```

Adicionalmente es necesario especificar para cada rol qué clase (initialView) aparecerá en el listado que se mostrará por defecto para la versión web "desktop" y opcionalmente a qué clase (referencedClass)

referencia dicho rol. Esto último está relacionado con el hecho de poder filtrar la información que aparece en los listados en función del rol.

Vemos que al comienzo del fichero *drol.model* se ha hecho necesario utilizar la directiva *import* con el fin de poder hacer referencia a los elementos definidos en el diagrama de clases (entidades, servicios, etc.). Esta directiva será necesaria incluirla en el resto de ficheros *.model que quedan por analizar.

Vamos a ver a continuación los diferentes diagramas de transición de estados (DTS, de sus siglas en inglés) para cada una de las clases anteriores, los cuales se han definido en el fichero *src/model/dts.model*. En la siguiente figura puede observarse el DTS corresponde a la clase cliente, donde es posible ver tanto su representación textual (imagen izquierda) como gráfica (imagen derecha).

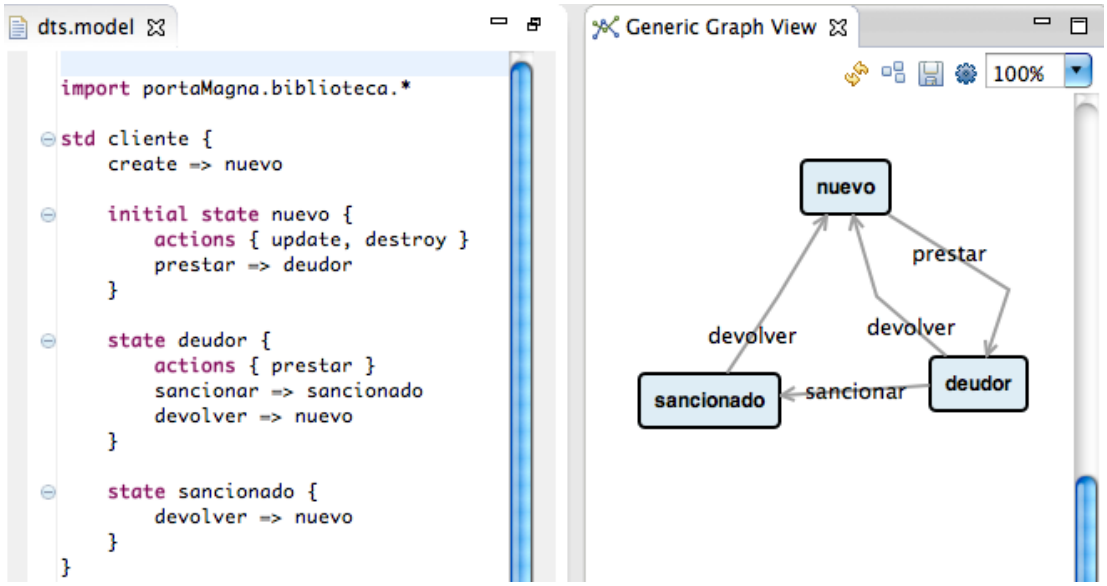


Figura 4.3: DTS clase cliente; representación textual y gráfica.

Al crear un cliente, este es “nuevo” y sólo en este estado puede ser eliminado o pueden ser actualizados sus datos (por defecto las acciones no son visibles en el GGV). El modelo funcional debe considerar que sólo cuando un cliente “deudor” o “sancionado” devuelve todos los libros, éste pasa al estado “nuevo”, es decir, sólo si el numero de préstamos actuales es igual a uno y se ejecuta devolver, se produce el cambio de estado a “nuevo”. Un cliente debe ser sancionado automáticamente o por los empleados (ante fallos eventuales del sistema) si no devuelve los libros prestados en el tiempo requerido (2 semanas, por ejemplo).

En la figura 4.4 puede observarse el DTS corresponde a la clase libro.

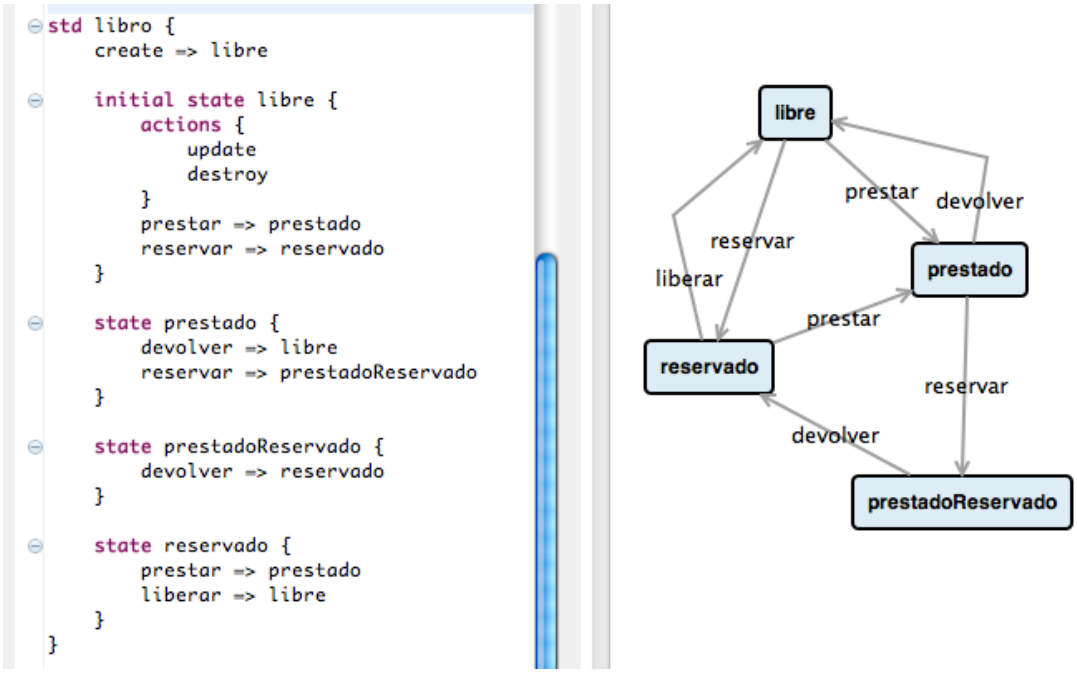


Figura 4.4: DTS clase libro; representación textual y gráfica.

Así pues, cuando un empleado crea un libro, este pasa automáticamente a estar disponible (libre). Solo entonces puede ser borrado o actualizados sus datos. Adicionalmente, cuando está libre puede ser prestado o reservado para ser recogido más tarde. También se puede reservar cuando está prestado y queremos impedir que se renueve por el actual dueño, en cuyo caso el libro pasará a estar "prestadoReservado" y sólo cuando es devuelto pasa a estar reservado. Se establece un tiempo máximo de reserva, a partir del cual el libro pasará a estar disponible.

Todas las clases, excepto la clase persona, tienen un atributo llamado "estado". En el caso de la clase empleado y préstamo, éste atributo sólo se utiliza para establecer el orden de ejecución de los diferentes servicios, por lo que sólo tienen un estado inicial.

En la siguiente figura puede observarse el DTS que corresponde a la clase préstamo.

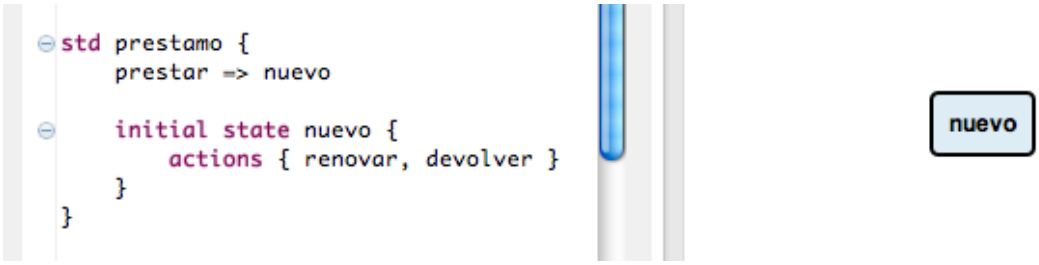


Figura 4.5: DTS clase préstamo; representación textual y gráfica.

Cuando se crea un préstamo por parte de un empleado, este es "nuevo" y solo entonces puede ser renovado o devuelto. Adicionalmente un cliente solo puede renovar un máx. de 2 veces y/o tener prestados un máximo de 5 libros.

Por último, en la siguiente figura puede observarse el DTS correspondiente a la clase empleado.

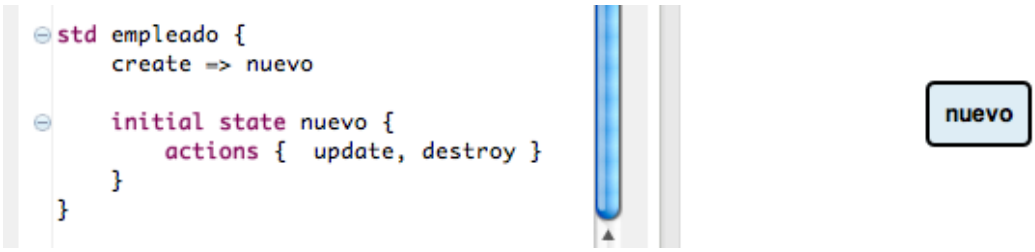


Figura 4.6: DTS clase empleado; representación textual y gráfica.

Cuando se crea un empleado por parte del administrador del sistema, este es "nuevo" y sólo entonces se pueden actualizar sus datos o puede ser despedido.

Uno de los aspectos más importantes es la visibilidad que tiene cada rol sobre los elementos anteriormente definidos. El rol de tipo "admin" se ha configurado para que tenga por defecto una total visibilidad sobre todos los elementos definidos en la especificación PLESG, con el fin de facilitar un rápido prototipado y poder empezar a trabajar de inmediato. Por el contrario, la visibilidad que el resto de roles puede tener sobre cualquier elemento debe ser declarada de manera explícita en el fichero *src/rolView.model*.

Como puede verse a continuación, para el rol SUPER (de tipo admin) se ha creado un servicio de navegación desde la clase cliente hacia la clase préstamo. La navegación entre clases puede ser utilizada para filtrar el listado de destino y así, por ejemplo, ver sólo los prestamos de un cliente en concreto.

```
/*
 * IMPORTANT:
 *      Rol ADMIN has TOTAL visibility over the model.
 *      (You must write his possibles navigations)
 */

import portaMagna.biblioteca.*

rolView super_persona (SUPER, persona, classview:False){
       properties:{ all }
       services:{ all }
}

rolView super_cliente (SUPER, cliente) {
       navigation : [
              {
                     prestamo,
                     "Ir a prestamos",
                     filter : { from:id,  to:pertenece }
              }
       ]
}
                                          ...
```

Curiosamente vemos como se ha establecido explícitamente acceso total a las propiedades y servicios de la clase persona para el rol SUPER, cuando no hace falta teóricamente. Esto se debe a que se ha decidido que el rol SUPER no pueda acceder al listado de dicha clase (classview:False), pero sí queremos que se tenga total visibilidad sobre sus atributos/servicios por parte de las clases que heredan de ella, y es que un aspecto importante a tener en cuenta es que hay que ser congruente en la especificación.

Ahora vamos a ir viendo la visibilidad establecida para el resto de roles. En el caso del rol SOCIO tenemos:

```
rolView socio_persona (SOCIO, persona, classview:False){
	properties:{ all }
}

rolView socio_cliente (SOCIO, cliente, classfilter:self) {
	properties:{ all }
	services:{ update }
	navigation : [
		{
			prestamo, "Ir a prestamos",
			filter : { from:id,  to:pertenece }
		}
	]
}

rolView socio_libro (SOCIO, libro){
	properties:{ all }
	services:{ reservar }
}

rolView socio_prestamos (SOCIO, prestamo, classfilter:pertenece){
	properties:{ all }
	services:{ renovar }
}
```

Un socio puede ver y actualizar todos los atributos, pero sólo de su propia instancia, según indica el parámetro *classfilter:self.* Ese mismo parámetro se ha utilizado para establecer que sólo pueda ver sus propios prestamos, *classfilter:pertenece,* así como renovar sólo los que tenga en su poder. Adicionalmente tiene la misma navegación entre clases que el rol SUPER. Por último un socio puede ver el catálogo de libros de la biblioteca y reservar los libros que le interesen. Importante mencionar que no puede acceder al listado de la clase persona (classView:False) y no tiene visibilidad alguna sobre un empleado (no se ha declarado explícitamente).

El siguiente rol a analizar es STAFF.

```
rolView staff_persona (STAFF, persona, classview:False){
	properties:{ all }
}

rolView staff_empleado (STAFF, empleado, classfilter:self) {
	properties:{ all }
	services:{ update }
}
```

```
rolView staff_cliente (STAFF, cliente){
	properties:{ all }
	services:{ create, update, destroy, sancionar }
	navigation : [
		{
			prestamo, "Ir a prestamos",
			filter : { from:id,  to:pertenece }
		}
	]
}

rolView staff_libro (STAFF, libro){
	properties:{ all }
	services:{ create, update, destroy, liberar }
}

rolView staff_prestamo (STAFF, prestamo){
	properties:{ all }
	services:{ prestar, devolver }
}
```

Al igual que ocurría con el rol SOCIO, un usuario con rol STAFF será un empleado y como tal solo podrá ver y actualizar los atributos de su propia instancia. Curiosamente, no puede dar de alta o eliminar a otros empleados, pues esto lo debe realizar el administrador. Por otra parte, puede ver a todos los clientes, sancionarlos si procede y ejecutar los servicios CRUD. También puede navegar a los prestamos de un cliente en concreto. Respecto a los libros, puede ejecutar los servicios CRUD y liberar los que estén reservados. El rol staff es el único capaz de realizar prestamos y cancelarlos (a excepción quizás del administrador). Estos servicios son compartidos con las clases libro y cliente, por lo que su ejecución puede implicar cambios en los estados de los objetos relacionados de forma atómica. Importante mencionar que no puede acceder al listado de la clase persona (classView:False).

Por último sólo nos queda por analizar el rol GUEST, que sólo puede consultar el catálogo de libros que existen en la biblioteca. Sin embargo no se le permite ver el estado de cada libro y tampoco ejecutar ningún servicio (no se han declarado explícitamente).

```
rolView guest_libro (GUEST, libro){
	properties:{ titulo, autor }
}
```

La última fase de la especificación PLESG consiste en diseñar los componentes visuales de la interfaz de usuario de forma particular a cada rol. Dichos componentes son el listado/grid y el formulario.

Un aspecto relevante para el tipo de componente grid es que es posible establecer un diseño adaptado según el dispositivo desde el cual se haga la conexión al sistema: desktop[10] y/o móvil. Por lo tanto, visualmente un listado/grid para la versión desktop es una matriz que contiene un conjunto de

[10] Desktop: este tipo de aplicaciones web se conocen con el nombre de Rich Internet Applications o RIA (en español "Aplicaciones de Internet Enriquecidas").

filas o instancias de una determinada clase y un número de columnas que corresponden a los atributos, atributos enumerados o relaciones que el analista debe especificar. En comparación con la versión móvil sólo difiere en el número de columnas, pues sólo se muestra una. Comentar que por ahora no es posible incluir atributos derivados en ambas vistas. Adicionalmente se puede especificar para cada columna/propiedad ciertas operaciones (ordenación y/o filtrado de instancias) aunque sólo para la versión desktop. Por defecto para la versión móvil sólo se permite el filtrado de la información (ver punto 7.1, trabajo futuro).

El componente formulario es utilizado por lo general para mostrar sólo aquellas propiedades estáticas definidas en una clase (atributos, atributos enumerados/derivados y relaciones) sobre las que un rol tenga visibilidad. Sin embargo, también es posible definir nuevos atributos exclusivos del formulario. Adicionalmente, todas las propiedades (de clase o exclusivas del formulario) pueden ser establecidas en modo de solo lectura o incluso pueden ser excluidas del formulario. Esto último no tendría sentido para los atributos añadidos. Comentar que no es necesario diseñar este tipo de componente para el rol de tipo admin ya que los traductores infieren dicha información a partir de la visibilidad declarada, a menos que se desee un diseño específico.

El diseño de los listados/grids se puede realizar en el fichero *listview.model,* creado por defecto por el asistente.

Empecemos mostrando el diseño del único listado para el rol GUEST, para ambos dispositivos.

```
import portaMagna.biblioteca.*

listView guest_libro (GUEST, libro){
	model : [
		{property:titulo, sortable:True, filter:True},
		{property:autor, sortable:False, filter:False}
	]
}

listView guest_libro2 (GUEST, libro, mobile){
	model : [
		{property:titulo} // puede filtrar por titulo
	]
}
```

Según se ha especificado, cuando un usuario invitado acceda al sistema desde un dispositivo móvil solo podrá ver el título de los libros, mientras que desde un ordenador podrá ver el título y el autor. Se ha decidido, para la versión desktop, que un usuario invitado pueda hacer peticiones de filtrado y/o ordenado sobre el título del libro, deshabilitando esos servicios sobre la propiedad autor. Recordemos que en la versión móvil se ofrece el filtro de información por defecto y no es posible ordenarla por el momento.

Para el rol STAFF se ha contemplado que no se va a conectar desde dispositivos móviles.

```
listView staff_empleado (STAFF, empleado){
       model : [all]
}

listView staff_libro (STAFF, libro){
       model : [
           {property:estado, sortable:True, filter:True},
           {property:titulo, sortable:True, filter:True},
           {property:autor, sortable:True, filter:True}
       ]
}

listView staff_cliente (STAFF, cliente){
       model : [
           {property:estado, sortable:True, filter:True},
           {property:nombre, sortable:True, filter:True},
           {property:apellidos, sortable:True, filter:True}
       ]
}

listView staff_prestamo (STAFF, prestamo){
       model : [
           {property:estado, sortable:True, filter:True},
           {property:renovacion, sortable:True, filter:True},
           {property:fechaini, sortable:True, filter:True},
           {property:fechafin, sortable:True, filter:True},
           {property:pertenece, sortable:True, filter:True},
           {property:implica, sortable:True, filter:True}
       ]
}
```

La sentencia *model:[all]* indica que el rol tiene acceso a las mismas propiedades sobre las que tiene visibilidad, sin embargo establece las propiedades de ordenado y filtrado a false, excepto para el rol admin. Por lo tanto si queremos habilitar alguna operación de ordenado/filtrado debemos declararla explícitamente.

El rol SUPER tampoco se va a conectar desde dispositivos móviles.

```
listView super_empleado (SUPER, empleado){
       model : [all]
}

listView super_cliente (SUPER, cliente){
       model : [all]
}

listView super_prestamo (SUPER, prestamo){
       model : [all]
}

listView super_libro (SUPER, libro){
       model : [all]
}
```

Respecto al rol SOCIO, sí que se contempla su acceso desde ambos dispositivos, pero teniendo en cuenta que en el listado para el dispositivo móvil no es recomendable mostrar más de dos propiedades.

```
listView socio_libro (SOCIO, libro){
       model : [
           {property:estado, sortable:True, filter:True},
           {property:titulo, sortable:True, filter:True},
           {property:autor, sortable:True, filter:True}
       ]
}

listView socio_cliente (SOCIO, cliente){
       model : [all]
}

listView socio_prestamo (SOCIO, prestamo){
       model : [all]
}
                                        /* MOBILE */
listView socio_libro2 (SOCIO, libro, mobile){
       model : [
              {property:estado},
              {property:titulo}
       ]
}

listView socio_cliente2 (SOCIO, cliente, mobile){
       model : [
           {property:nombre},
           {property:apellidos}
       ]
}

listView socio_prestamo2 (SOCIO, prestamo, mobile){
       model : [
           {property:fechafin},
           {property:pertenece}
       ]
}
```

Por último, al igual que se ha realizado con los listados, ahora se deben de declarar los formularios específicos para cada rol. En este caso, dicha declaración se puede realizar en el fichero *formview.model*.

Para el formulario de la clase libro y el rol GUEST (form <GUEST,libro>) se ha establecido que las propiedades visibles sean de solo lectura.

```
import portaMagna.biblioteca.*

formView guest_libro (GUEST, libro){
       readOnly: { titulo, autor }
}
```

Cuando un usuario invitado acceda con su dispositivo móvil desde el listado de libros (solo ve el título) a una instancia concreta, podrá saber el autor del libro.

Por último, los diversos formularios a los cuales tiene acceso el rol SOCIO y STAFF mostrarán las mismas propiedades establecidas según su

visibilidad. En todos ellos los atributos derivados estarán en modo de solo lectura.

```
//Usuario de la biblioteca
formView SOCIO_cliente (SOCIO, cliente){
	readOnly: {estado, numero, full_name, edad}
}
formView SOCIO_libro (SOCIO, libro){
	readOnly: {estado, reserva}
}
formView SOCIO_prestamo (SOCIO, prestamo){
	readOnly: {estado, renovacion, diff}
}

//Empleado de la biblioteca
formView STAFF_empleado (STAFF, empleado){
	readOnly: {estado, full_name, edad}
}
formView STAFF_cliente (STAFF, cliente){
	readOnly: {estado, numero, full_name}
}
formView STAFF_libro (STAFF, libro){
	readOnly: {estado, reserva}
}
formView STAFF_prestamo (STAFF, prestamo){
	readOnly: {estado, renovacion, diff}
}
```

En este caso (diseño de formularios) la omisión de la información implica delegar según lo establecido sobre la visibilidad que tiene cada rol. Recordemos que para el rol de tipo “admin” se infieren todos los formularios automáticamente.

Así pues, una vez se ha especificado el sistema software a desarrollar, el siguiente paso sería poder invocar a los distintos traductores con el fin de obtener los artefactos que formarán parte de la carrocería de la aplicación. Sin embargo en el siguiente apartado vamos a mostrar la gramática definida, basada en la notación Extended Backus-Naur Form-like (EBNF), que da lugar al DSL que es utilizado por los analistas funcionales para poder definir los distintos modelos del análisis según hemos podido observar.

4.2 Definición de la gramática

Este apartado se corresponde con la fase A del proceso PLESG. Antes de ver la definición de la gramática, que será analizada paso a paso en el punto 4.2.2 (gramática completa en el anexo 2), vamos a dar una guía rápida sobre la sintaxis[11] utilizada, con el fin de conocer las distintas reglas que se pueden definir en ella.

[11] Para más detalles ir al capítulo - 7.2 The Syntax: http://www.eclipse.org/Xtext/documentation/2_1_0/Xtext%202.1%20Documentation.pdf

4.2.1 Sintaxis

Las distintas reglas que se pueden definir en una gramática basada en Xtext son:

- Reglas terminales
- Reglas parser
- Reglas tipo de datos
- Reglas enumeración

Reglas terminales

- Son símbolos atómicos de las gramáticas
- Crean tokens[12] terminales en el Abstract Syntax Tree (AST)

En la gramática *org.eclipse.xtext.common.Terminals* se encuentran las reglas terminales que Xtext ofrece por defecto. En la siguiente figura podemos ver su declaración, donde vemos que se ha importado el EPackage con los tipos de datos del metamodelo Ecore.

Terminals.xtext

```
/*******************************************************************************
 * Copyright (c) 2008 itemis AG and others.
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *******************************************************************************/
grammar org.eclipse.xtext.common.Terminals hidden(WS, ML_COMMENT, SL_COMMENT)

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

terminal ID         : '^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;
terminal INT returns ecore::EInt: ('0'..'9')+;
terminal STRING :
            '"' ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"') )* '"' |
            "'" ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|"'") )* "'"
        ;
terminal ML_COMMENT : '/*' -> '*/';
terminal SL_COMMENT     : '//' !('\n'|'\r')* ('\r'? '\n')?;

terminal WS         : (' '|'\t'|'\r'|'\n')+;

terminal ANY_OTHER: .;
```

Figura 4.7: reglas terminales de Xtext

Cada regla terminal devuelve un valor atómico, un EDataType, que por defecto es un Ecore::EString. Sin embargo la regla terminal INT devuelve una instancia de Ecore::EInt.

Las reglas terminales son descritas usando expresiones Extended Backus-Naur Form-like (EBNF), dentro de las cuales se pueden utilizar, entre otros, los siguientes operadores:

[12] Token: secuencia de caracteres de entrada

Operador	Nombre
	Cardinalidad 1 (por defecto)
*	Cardinalidad 0 o más
+	Cardinalidad 1 o más
?	Cardinalidad 0 o 1
!	Token negación (negated)
->	Token until
|	Alternativa
&	Grupos desordenados
.	Comodín
..	Rango de valores

La única regla terminal definida en nuestra gramática es un separador entre sentencias:

```
terminal SEP: ',' (WS)?;
```

Reglas parser

- No son símbolos atómicos de las gramáticas
- Crean subárboles en el AST

No todas las expresiones que están disponibles en las reglas terminales pueden ser utilizadas en las reglas parser. Los operadores rango de valores, comodín y tokens “until”, “negated” y “End Of File” (EOF) sólo están disponibles para las reglas terminales. Sin embargo, los siguientes elementos están disponibles tanto para reglas parser como para reglas terminales:

- Grupos desordenados: los elementos de un grupo desordenado pueden ocurrir en cualquier orden pero sólo una vez. Un ejemplo sería:

```
Modifier: static?='static' & final?='final';
```

- Referencias cruzadas: son reglas donde existe un tipo de datos que referencia a otro a través de su ID (punto 5.1.1, linking).

- Llamadas a reglas sin asignación: son reglas que llaman a otras reglas parser.

```
Type: DataType | Entity | StateTransition | Rol | RolView | FormView | ListView;
```

- Reglas ocultas: son reglas que no se procesarán por el parser porque no tienen valor semántico. Para utilizarlas se emplea la sentencia *hidden.*

```
grammar org.eclipse.xtext.common.Terminals hidden (WS, ML_COMMENT, SL_COMMENT)
```

- Asignaciones: existen varios tipos de asignaciones, y su significado varía en función del tipo.

Tipo	Significado
=	El elemento de la derecha se le asigna al de la izquierda `DataType: 'datatype' name = ID;`
+=	Una lista de elementos de la derecha son asignados al elemento de la izquierda `Model: (elements += AbstractElement)*;`
?=	Asigna un EBoolean al elemento de la izquierda. Es "true" si el elemento de la derecha es consumido sino es "false". `Entity: (abstract?='abstract')? 'entity' name=ID ...`

Reglas tipo de datos

- Son símbolos atómicos de las gramáticas.
- No contienen ni llamadas a reglas parser ni asignaciones

Son muy similares a las reglas terminales, pero a diferencia de estas:

- Se recomiendan para combinar reglas terminales
- Permiten utilizar tokens ocultos
    - Por ejemplo se podrían introducir comentarios entre dos IDs

Ejemplos de reglas de tipo de datos son:

```
QualifiedName: ID ((ML_COMMENT)? '.' ID)*;
QualifiedNameWithWildcard: QualifiedName '.*'?;
```

Reglas enumeración

- Sirven para crear enumerados en una gramática
- Es posible definir literales alternativos

Las únicas reglas enumeración creadas son:

```
enum BOOL: True = 'True' | False = 'False';
enum IDIOM: es = "ES" | fr = "FR" | en = "EN";
enum WIDGET: textarea = "widgetTextarea" | date = "widgetDate";
enum ROLTYPE: admin = "admin" | user = "user";
```

4.2.2 Análisis de la gramática

Ahora sí, vamos a analizar en detalle y paso a paso lo que significan las diferentes reglas definidas en la gramática, aunque sin entrar en detalles específicos sobre como definir correctamente una gramática en Xtext, pues existe una buena documentación[13] online al respecto. Comentar que en

[13] Grammar language: http://www.eclipse.org/Xtext/documentation.html#grammarLanguage

algunos momentos haremos uso del caso de estudio visto en el punto anterior para ayudar en la comprensión de la gramática. Así pues, la gramática definida empieza según se muestra a continuación.

```
grammar org.xtext.portaMagna.Dsl with org.eclipse.xtext.common.Terminals
generate dsl "http://www.xtext.org/portaMagna/Dsl"
import "http://www.eclipse.org/emf/2002/Ecore"


Model:
        (elements += AbstractElement)*
;

AbstractElement:
        PackageDeclaration | Type | Import
;


PackageDeclaration:
        'package' name = QualifiedName '{'
                (elements += AbstractElement)*
        '}'
;

QualifiedName:
        projectName=ID ('.' appName=ID)?
;

Import:
        'import' importedNamespace = QualifiedNameWithWildcard
;

QualifiedNameWithWildcard:
        QualifiedName '.*'?
;

Type:
        DataType | EnumType | Entity |
        StateTransition |
        Rol | RolView |
        FormView | ListView
;
                                        ...
```

La primera línea indica el nombre de la gramática y mediante la sentencia *with* se especifica que es posible utilizar la gramática *org.eclipse.xtext.common.Terminals* dentro de la nuestra. Este último concepto se conoce como *grammar mixin*[14].

En la segunda línea se define el nombre “dsl” y el Uniform Resource Identifier (URI) “http://www.xtext.org/portaMagna/Dsl” del EPackage Ecore (ver figura 5.6 para más detalles).

Como se observa en la tercera línea, se ha importado un Epackage especial, es el Ecore, para poder hacer referencia al elemento *EObject* y así poder realizar *crosslink avanzado* (ver punto 5.1.1 para más detalles).

[14] Grammar mixin: http://www.eclipse.org/Xtext/documentation.html#grammarMixins

La primera regla en una gramática siempre se usa como punto de entrada o regla de inicio.

```
Model:
        (elements += AbstractElement)*
;
```

Podemos interpretar que *Model* contiene un número arbitrario de *AbstractElements,* que a su vez delega en las reglas *PackageDeclaration, Type, Import o Config*.

```
AbstractElement:
        PackageDeclaration | Type |
        Import | Config
;
```

La regla *PackageDeclaration* permite la definición de paquetes para evitar el conflicto de nombre de clases al igual que ocurre en lenguajes de programación como Java.

```
PackageDeclaration:
        'package' name = QualifiedName '{'
                (elements += AbstractElement)*
        '}'
;

QualifiedName: ID ('.' ID)*;
```

Dentro de la declaración de un paquete (*PackageDeclaration*) es posible declarar los tipos (*Type*) permitidos en la gramática, importar (*Import*) los elementos definidos en otros modelos o incluso anidar más paquetes de forma recursiva, así como establecer los parámetros de configuración (*Config*); nombre del proyecto y de la aplicación.

En Xtext las importaciones (*Imports*) pueden ser definidas de una forma muy práctica mediante el uso del nombre *importedNamespace* en la regla parser, ya que el framework tratará el valor como un *import* en Java.

```
Import: 'import' importedNamespace = QualifiedNameWithWildcard;

QualifiedNameWithWildcard: QualifiedName '.*'?;
```

La regla Type delega en las diferentes reglas parser que se muestran a continuación:

```
Type:
        DataType | EnumType | Legacy | Entity | StateTransition |
        Rol | RolView | FormView | ListView;
```

La regla *DataType* empieza con la palabra reservada “datatype”, seguido por un identificador, el cual es parseado por la regla terminal ID. El valor devuelto por el ID es asignado a la variable *name*.

```
DataType:
```

```
        'datatype' name = ID
;
```

Esta regla permite que se puedan definir los distintos tipos de datos a utilizar en la especificación del sistema PLESG, sin embargo en el anexo 3 puede verse realmente los tipos de datos soportados por PLESG.

Por otra parte, la regla *EnumType* permitirá definir los tipos de datos enumerados.

```
EnumType:
        'enum' name = ID '{'
                enumFields+=EnumField (SEP enumFields+=EnumField)*
        '}'
;

EnumField: '('id = INT SEP nombre = STRING SEP alias = STRING ')';
```

La regla *Legacy* permite definir clases legadas con las que el sistema deberá relacionarse.

```
Legacy:
        'legacy' name = ID
;
```

Un ejemplo es la clase legada User, que establece los parámetros de conexión (username y password) para los diferentes usuarios que puedan existir en el sistema. Según se verá más adelante será posible establecer ciertas asociaciones con estas clases desde las propias clases del sistema.

La regla *Entity,* al igual que las tres anteriores reglas, también empieza la definición con una palabra reservada seguida por un identificador. Opcionalmente, la clase puede ser declarada abstracta[15] y/o extender de otra clase. Entre llaves es posible definir cualquier número de atributos, atributos derivados/enumerados, relaciones (con clases legadas u otras clases del sistema), servicios y la función de identificación[16] de la clase.

```
extendedClass: entity=[Entity|QualifiedName];

Entity:
(abstract?='abstract')? 'entity' name=ID ('extends' superType=extendedClass)?
       '{'
                (attributes+=Attribute)*
                (derivatives+=AttrDerived)*
                (enumerators+=AttrEnum)*
                (relations+=Relation)*
                (services+=Service)*

                //ie: es como el metodo toString() para una clase Java
                (identity?='identity' ':' '{' (attribute+=[Attribute])+ '}')?

                /*
                 * INFO adicional para
```

[15] No es posible crear instancias de una clase abstracta.

[16] Es como el método toString() para una clase Java.

```
                 */
                // 1. optimizar los Traductores: default false
                ('hasSTD' ':' hasSTD?='True' '{'
                        'field' ':' fieldSTD=[AttrDerived]
                        'alias' ':' aliasSTD=STRING
                '}')?

                // 2. saber si alguna prop. de la clase es utilizada para conectarse al
                // sistema: default false
                // Esta directamente relacionado con classfilter en la regla RolView.
                ('hasSessionProperty' ':' hasSessionProperty?='True'
                        '{''sessionProperty' ':' sessionProperty=[EObject] '}'
                )?

        '}'
;
```

Al final de esta regla se ha incluido cierta información adicional. Por una parte, el objetivo de la etiqueta *hasSTD* ha sido optimizar los traductores de la parte servidora, indicando si la clase tiene asociado o no un Diagrama de Transición de Estados (DTE o STD de sus siglas en inglés) y así no tener que recorrer todos los atributos derivados para obtener dicha información. Por otra parte, es necesario establecer qué propiedad es utilizada para conectarse al sistema con el fin de poder filtrar la información que aparece en los listados en función del rol.

La regla *Attribute* permite definir los distintos atributos que contendrá la clase, indicando el tipo de dato (string, integer, boolean, etc.), su nombre y los parámetros/argumentos (ver anexo 3, punto 2.1 y 2.2), los cuales sirven para establecer:

- restricciones sobre las columnas de las tablas de la base de datos: nulo, único, etc.
- restricciones de validación en el servidor:
  por ejemplo, si dicho atributo debe cumplir con alguna/s función/es de validación, longitud máxima, número de decimales (si es de tipo decimal), etc.
- restricciones sobre la IU:
  - Sobre el formulario:
    - atributo de sólo lectura,
    - Información básica; alias, ayuda, etc.
  - Sobre el listado/grid: ordenable y/o filtrable

```
Attribute:
        type=[DataType] name=ID '('
                parameters+=attrParameter (SEP parameters+=attrParameter)*
        ')'
;

attrParameter:
        token="widget" ':' widget=WIDGET |
        token=TokenS ':' valueS=STRING |
        token=TokenI ':' valueI=INT |
        token=TokenB ':' valueB=BOOL |
        token=TokenFV':' '{'
                validations+=attrValidation (SEP validations+=attrValidation)*
        '}'
;
```

```
attrValidation:
        name=ID ('(' comment=STRING ')')? ':' type = [DataType]
;
```

Según se ha expresado la regla, un mismo parámetro podría repetirse en la definición de un mismo atributo. Esta misma situación también ocurre en las reglas que definen los atributos derivados/enumerados, relaciones y servicios de una clase. Para evitarlo se han implementado varias restricciones al modelo que imposibilita dicha situación, reportando el error correspondiente (ver punto 5.1.1, implementación de restricciones).

La regla *AttrDerived* permite definir, por una parte, un tipo de atributo especial, llamado *STDField*, que establece si la clase tendrá comportamiento dinámico y por tanto tendrá un diagrama de transición de estados, y por otra parte, permite definir los tipos de atributos derivados propiamente dichos; atributos que dependen del valor de los atributos de la propia clase o cuyo valor depende de consultas a otras clases o a sus atributos y cuyo valor no se almacena en la base de datos. Su declaración empieza con un identificador seguido entre paréntesis por los parámetros correspondientes.

```
AttrDerived: name=ID '(' parameters+=attrParameter (SEP parameters+=attrParameter)*
(
  SEP token='isSTDField' ':' isSTDField?=BOOL ')'
|
  (SEP token='refOtherClass' ':' refOtherClass?=BOOL)? ')' '<=' 'function'
                                        functionName=ID '(' functionComment=STRING ')'
);
```

De todos los parámetros existentes según la regla *attrParameter* sólo son válidos el *alias* (obligatorio) y *help* o texto de ayuda al usuario (opcional). Esto es debido a que se ha reutilizado la regla *attrParameter* para simplificar el código de los traductores, sin embargo se ha implementado una restricción al modelo que verificará el uso de sólo esos parámetros. Los otros dos parámetros (isSTDField y refOtherClass) aparecerán dependiendo del tipo de atributo derivado.

Es importante mencionar que un Atributo derivado es tratado como un *String*, por lo que la función que implemente su valor deberá tener en cuenta dicho requisito. Además existen dos aspectos a tener en cuenta con estos atributos respecto al diseño de la IU cuando el rol no es “admin”; en los formularios éstos deben incluirse dentro de la propiedad *readOnly*, pues no deben ser editables, y no aparecerán en los listados/grids porque no es posible recuperar su valor en un coste temporal aceptable. Esta última restricción no afecta al atributo *STDField*.

La regla *AttrEnum* permite definir los atributos enumerados que contendrá la clase, indicando el tipo de dato enumerado, su nombre y los parámetros pertinentes: el *alias* (obligatorio) y *help* o texto de ayuda al usuario (opcional).

```
AttrEnum:
        //NO IMPLEMENTADO AUN EN TRADUCTORES
```

```
        type=[EnumType] name=ID '('
            parameters+=attrParameter (SEP parameters+=attrParameter)*
        ')'
;
```

Al igual que antes, se ha utilizado la regla *attrParameter* para simplificar el código de los traductores, pues los parámetros permitidos son un subconjunto de los proporcionados por dicha regla. En este caso se debería implementar una restricción al modelo (check) para validar el uso exclusivo de dichos parámetros (ver punto 7.1, trabajo futuro).

La regla *Relation* permite establecer las asociaciones entre las diferentes clases que forman el Diagrama de Clases; indicando el nombre de la asociación, la cardinalidad y la clase destino, junto con los parámetros específicos de la relación.

```
Relation: name=ID
 (
  (relation='oneToOne' | relation= manytoOne) 'legacy'
    legacy=[Legacy|QualifiedName]
 |
  (relation='oneToOne'|relation='manyToOne'|relation='manyToMany'|relation='hasMany')
    entity=[Entity|QualifiedName]
 )
 ('(' parameters+=relParameter (SEP parameters+=relParameter)* ')')?
;

relParameter:
        token=TokenBRelation ':' valueB=BOOL |
        // ManyToMany
        token='through' ':' entity=[Entity|QualifiedName] |
        //ManyToMany REFLEXIVA
        token='related' ':' property=STRING //[Relation]
;
```

En el anexo 4 es posible ver una tabla resumen con todas las asociaciones, y sus correspondientes parámetros, permitidas en PLESG.

Tanto los atributos enumerados como las relaciones oneToOne y ManyToOne son representados de forma gráfica dentro de los formularios mediante el componente visual “dropbox” o lista desplegable. Por otra parte, las instancias de las relaciones manyToMany se muestran mediante un listado.

La regla *Service* permite definir tanto los servicios CRUD[17] como la interfaz de los servicios propios de la lógica de negocio, indicando el nombre del servicio, los parámetros y el tipo de dato simple ha devolver.

```
Service:
        name=ID ('('
            parameters+=servParameter (SEP parameters+=servParameter)*
        ')')? ':' type = [DataType]
;
```

[17] CRUD: el servicio de lectura (R) aparece implícitamente cuando un rol tiene visibilidad sobre los atributos de una clase.

```
servParameter:
        token='alias' ':' value=STRING |
        token='type' ':' value=("C" | "U" | "D" | "B") |
        token='customForm' ':' form=[FormView] |
        token=TokenBServ ':' valueB=BOOL
;
```

Los parámetros obligatorios son el alias y el tipo de servicio (CUD o Business). El parámetro *refreshParent* (por defecto a True) establece si el formulario se refrescará tras la ejecución del servicio. Por último, cuando estamos ante un servicio específico de la lógica de negocio (type = B), es posible decidir si ese servicio tendrá asociado un formulario personalizado, el cual debe ser definido previamente, y si su ejecución se realizará en una ventana modal (isModal, por defecto a True).

Si una clase estableció la etiqueta *hasSTD* a cierto, entonces la regla *StateTransition* permite describir las vidas válidas de los objetos de dicha clase mediante la creación de un diagrama de transición de estados (STD). La regla empieza la definición con la palabra reservada 'std' seguida por el nombre de la clase a la que hace referencia.

```
StateTransition:
  'std' entity=[Entity|QualifiedName] '{'
        (transIni=Transition) //accion x la cual llegamos al estado inicial
        (states+=State)*
 '}';

State:
  (initial='initial')? 'state' name=ID '{'
     /* actions = servicios, NO cambian el estado de la clase */
     ('actions' '{' actions+=Action (actions+=Action)* '}')?
     (transitions+=Transition)*
  '}';

Transition: action=Action '=>' state=[State];
Action: service=[Service];
```

Seguidamente se establece a través de qué servicio se llega al estado inicial del objeto, previamente definido. Por ejemplo, lo normal será que desde el método CRUD "*create*" se alcance el estado inicial (*initial*), se llame como se llame. Por lo tanto, un estado, inicial o no, es identificado con un nombre y se compone de un conjunto de acciones, las cuales no cambian el estado del objeto, y de un conjunto de transiciones, las cuales sí cambian el estado de dicho objeto.

Cabe comentar que el estado de un objeto no varía en función del rol, es decir, el comportamiento dinámico del objeto de una clase es único y está definido según su STD.

Cada rol puede ver la misma aplicación desde perspectivas diferentes. La regla *Rol* permite establecer cuales son las diferentes caras de ese prisma llamado aplicación, es decir, cuales son los roles/actores que van a intervenir en la aplicación. Dicha regla establece el nombre del rol, a qué tipo pertenece (admin o user, por ahora) y cual es el punto de entrada (entidad) de acceso

por defecto a la aplicación Web para dicho rol. Adicionalmente se debe de especificar a que posible clase referencia el rol.

```
Rol: 'rol' name = ID '{'
       'type' ':' type=RolType  SEP 'initialView' ':' entity=[Entity|QualifiedName]
       (SEP 'referencedClass' ':' refclass=[Entity|QualifiedName])?
     '}';
```

La visibilidad permite gestionar el acceso a la información de una manera controlada, dando los permisos adecuados a las personas/actores pertinentes. Así pues, la visibilidad que tiene cada rol sobre una entidad viene dada por la regla *RolView*. Opcionalmente se puede establecer un filtro mediante classfilter para recuperar aquellas instancias de las que el usuario conectado al sistema es dueño. Por ejemplo, un cliente no debe poder ver los prestamos y datos personales de otros clientes.

```
RolView: 'rolView' name=ID  '(' rol=[Rol] SEP entity=[Entity|QualifiedName]
(SEP 'classfilter' ':' (classfilter=[EObject])? ')'
'{'
       ('properties'  ':' '{'
         ( allProperties?='all' |
         properties+=declaredRolProperty (SEP properties+=declaredRolProperty)* )
       '}')?

       ('services'    ':' '{' (
         allServices?='all'   |
         services  +=declaredRolService  (SEP services  +=declaredRolService)*  )
       '}')?

       ('navigation'  ':' '['
         navigation+=declaredRolNavigation (SEP navigation+=declaredRolNavigation)*
       ']')?
'}';
declaredRolProperty: ref=[EObject];
declaredRolService: ref=[Service];
declaredRolNavigation:
       '{' // POR DEFECTO VAMOS AL GRID INDICADO -- OPCIONAL NAVEGAR CON FILTROS
              ref=[Entity|QualifiedName]
              text=STRING
              ('filter' ':' '{'
                     'from'  ':' from=[EObject] // id | this.Attribute
                     'to'    ':' to=[EObject] //grid.Attribute or grid.Relation
              '}')?
       '}';
```

La granularidad a la cual puede llegar la visibilidad de un rol sobre una clase es a nivel de propiedad y servicios. Se entiende por propiedad los atributos, atributos derivados/enumerados y relaciones que se hayan definido en un clase. A la hora de establecer la visibilidad sobre las propiedades y/o servicios que un rol tiene sobre una clase se ha incluido el operador “all”, el cual establece que dicho rol tendrá total visibilidad sobre todas las propiedades y/o servicios que se hayan declarado en la clase.

Adicionalmente también se puede establecer visibilidad sobre un tipo especial de servicios; navegaciones entre clases, con posibilidad de filtrar los resultados obtenidos en dicha navegación. Por defecto se muestran todas las instancias, pero se puede definir un filtro según las relaciones/atributos

establecidos. Un ejemplo para este último caso sería que un STAFF pueda navegar desde la clase Cliente, seleccionando previamente una instancia concreta, hacia la clase Prestamo, obteniendo así sólo los prestamos de dicho cliente.

Por último, vamos a comentar las reglas que nos permiten hacer un diseño simple, pero adaptable, de los artefactos que constituyen la IU atendiendo a la visibilidad en función del rol. Dichos artefactos son:

- los formularios (modelo de diseño de formularios, MDF) y
- los listados/grids (modelo de diseño de listados, MDL).

El hecho de que sea adaptable se refiere a que el analista funcional puede establecer para cada rol qué propiedades ocultar gráficamente y/o editar en un formulario, o incluso qué propiedades adicionales podrá tener. A nivel de listado, podrá establecer las propiedades a visualizar, así como ciertas operaciones[18] sobre ellos: ordenación y/o filtrado de instancias. Recordemos que no es posible por ahora mostrar el valor de los atributos derivados propiamente dichos en los listados.

La regla *FormView* permite definir dos tipos de formulario; un formulario genérico sobre el que se ejecutarán todos los servicios definidos en la clase y uno específico para los servicios "custom" que tenga la clase, los cuales están pensados para visualizarse y ejecutarse en una ventana modal. Ambos tipos de formularios atienden a servicios propios de una clase, y no a servicios interobjetuales, pues para ese tipo de servicios puede ser necesario el desarrollo de un formulario concreto.

```
FormView:
        'formView' name=ID '(' rol=[Rol] (SEP model=[Entity|QualifiedName])? ')' '{'

        // 1. (opcional) Atributos PROPIOS DEL FORMULARIO
        (formAttributes+=Attribute)*

        // 2. ReadOnly Properties, que IRAN en el fieldSet
        (roProperties=FormRoProperties)?

        // 3. Exclude Properties, que NO IRAN en el fieldSet
        (exProperties=FormExProperties)?
'}';

FormRoProperties:
        type='readOnly' ':' '{'
          properties+=declaredViewProperty (SEP properties+=declaredViewProperty)*
        '}'
;

FormExProperties:
        type='exclude' ':' '{'
          properties+=declaredViewProperty (SEP properties+=declaredViewProperty)*
        '}'
;

declaredViewProperty:
```

[18] Sólo para la versión desktop, para la versión móvil se ofrece por defecto el filtrado.

```
        // [Attribute] | [AttrDerived] | [AttrEnum] |
        // [Relation] | formAttributes=[Attribute]
        ref=[EObject]
;
```

La regla empieza la definición con la palabra reservada *formView* seguida por un identificador y entre paréntesis se define sobre que entidad tiene visibilidad un determinado rol. Cuando no se especifica la entidad estamos ante un formulario específico para un servicio concreto de una clase.

Por último, la regla *ListView* permite establecer las columnas que aparecerán en un listado/grid con respecto a la terna <rol-entidad-dispositivo>. El analista puede establecer las propiedades; atributos, atributos enumerados y/o relaciones a visualizar, así como las operaciones de ordenación y/o filtrado de instancias para cada una de las propiedades anteriores. El tipo de filtro dependerá del tipo de dato asociado a la propiedad. Por otra parte sino se especifica ningún dispositivo, por defecto es desktop.

```
ListView:
  'listView' name=ID '('
      rol=[Rol] SEP entity=[Entity|QualifiedName] (SEP device=('mobile' | 'desktop'))?
  ')' '{'
       (model=ListViewModel)?
  '}';

ListViewModel:
  'model' ':' '[' (
      allModel?='all' // <-- filter y sortable = True |
      properties+=declaredListViewProperty (SEP properties+=declaredListViewProperty)*
  ) ']'
;

declaredListViewProperty:
'{'
       'property' ':' ref=[EObject]
       (SEP 'sortable' ':' sortable=BOOL)?
       (SEP 'filter'   ':' filter=BOOL)?
'}';
```

Esta gramática establece los conceptos básicos para crear un lenguaje específico de dominio (DSL) que nos ha permitido definir los diferentes modelos del Análisis (objetual, dinámico, funcional y de presentación) vistos en el punto anterior, y que nos ha permitido obtener una visión casi completa del sistema.

A modo de resumen, esta gramática permite definir tipos de datos simples y/o enumerados, clases legadas, clases con sus propiedades, servicios, relaciones, etc., diagramas de transición de estados para cada clase, roles, formularios de clase y específicos de servicio, listados/grids adaptados al dispositivo (desktop y/o móvil) y visibilidad de los roles sobre:

- clases, junto con sus propiedades y servicios
- formularios y listados/grids

4.3 Generación de los artefactos y metadatos

Este punto se corresponde con la fase 2 del proceso PLESG. La forma de invocar a cada uno de los traductores, encargados de generar los diferentes artefactos propios de cada framework, es mediante un menú contextual que aparece al seleccionar la carpeta donde se ubican todos los modelos propios de la especificación PLESG. La implementación de dicho menú contextual se detalla dentro del punto 5.1.2.

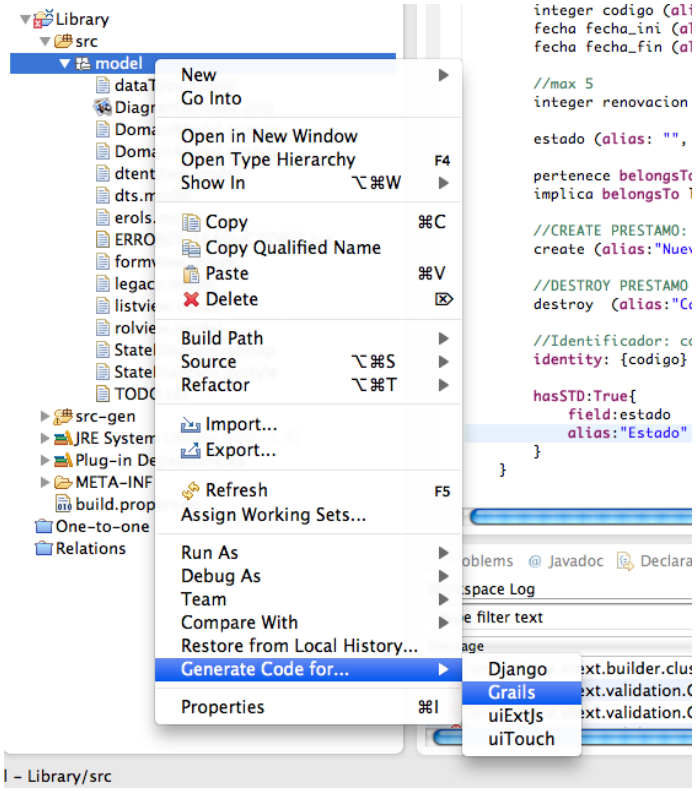
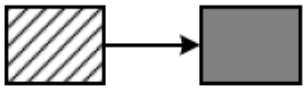


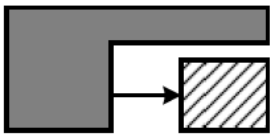
Figura 4.8: Invocación manual de un traductor mediante un menú contextual

Un aspecto importante que aparece en este punto es considerar la integración entre los artefactos generados y los no generados. A continuación se indican algunas de las pautas más utilizadas para la integración entre ambos:

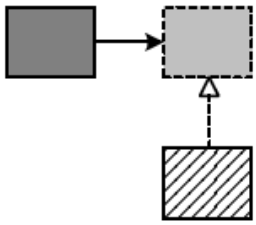
1. Se puede llamar a librerías desde dentro del código generado

código generado →  ← código no generado

2. Se pueden generar componentes que son llamados desde código no generado



3. Las partes no generadas pueden ser programadas contra interfaces que el código generado debe implementar



Estas técnicas pueden ser combinadas según las necesidades de los proyectos, pues lo importante es que queden bien delimitadas las responsabilidades y las interfaces, teniendo una arquitectura claramente definida. Todas estas técnicas han sido tenidas en cuenta a la hora de generar los diferentes artefactos software.

Hay que resaltar que a veces es inevitable modificar los artefactos generados, aunque esto pueda acarrear ciertos problemas de inconsistencia entre modelos y artefactos. Si finalmente se modifican, habría que guardar en algún VCS también esos artefactos generados. Por ejemplo, y según veremos a continuación, los servicios de la lógica de negocio definidos en Django no pueden programarse contra una interfaz porque no son servicios contenidos dentro de una clase.

Vamos hora a ver primero los artefactos generados para la capa servidora; traductor Django y Grails, y en último lugar y de manera conjunta, los artefactos que se generan para la capa cliente; ExtJs (desktop) y Touch (móvil).

Comentar que los metadatos, sentencias SQL encargadas de controlar la parte dinámica de la aplicación, son generados de forma específica para cada framework (Django y Grails), por lo que son generados en cada traductor.

4.3.1 Artefactos generados para la capa servidora

En la siguiente tabla podemos ver los artefactos generados por el traductor Django:

Traductor	Artefacto	Significado
Django	models.py	Define el modelo de datos. Un único fichero para todas las clases. Si una clase tiene DTS, éste se define en el método de clase setupFsm().
	common.py	Contiene varios métodos para obtener los parámetros de conexión; nombre de usuario, rol, etc.
	forms.py	Establece el diseño de los distintos formularios para cada uno de los roles definidos.
	views.py	Contiene toda la lógica de negocio, y es aquí donde se decide qué formularios mostrar o qué información enviar al cliente. Los métodos específicos de cada clase deben ser implementados por los desarrolladores.
	urls.py	Es el controlador, y establece el mapeo entre una URL y el servicio a invocar en views.py
	funcAttrDerived.py	Contiene las cabeceras de las funciones que establecen el valor de los atributos derivados de las clases. Su lógica debe ser implementada por los desarrolladores
	funcAttrValidations.py	Contiene las cabeceras de las funciones de validación de los atributos de las clases. Su lógica debe ser implementada por los desarrolladores
	funcFormValidations.py	Contiene las cabeceras de las funciones de validación de los formularios, se trata de validaciones globales al formulario. Su lógica puede ser modificada por los desarrolladores.
	funcFormAttrValidations.py	Contiene las cabeceras de las funciones de validación de los atributos propios de los formularios. Su lógica debe ser implementada por los desarrolladores
	insert_SQLiteDB_metamodel.sql	Contiene las sentencias SQL que insertan en la base de datos la meta información relativa al control de acceso a cada uno de los elementos; entidad, atributos, servicios, formularios, columnas en los listados/grids, etc.

Si bien los únicos artefactos textuales imprescindibles para Django son “models.py”, “views.py” y “urls.py”, pues establecen las bases del patrón de diseño MVC, el resto de artefactos se han creado para facilitar la tarea a los programadores y ofrecer una mejor integración del código a extender junto con el código no generado. Así pues, tanto la lógica de negocio (views.py) como las diferentes funciones (func*.py) deben ser implementadas por los desarrolladores (ver punto 4.5).

Toca el turno de describir los diferentes artefactos que genera el traductor Grails.

Traductor	Artefacto	Significado
Grails	/domain/<Class>.groovy	Define el modelo de datos. Un fichero por cada clase. Si una clase tiene DTS, este se define en el método de clase *setupFsm*(), al igual que en Django.
	/domain/<RolClass>.groovy	Establece el diseño de los distintos formularios para cada uno de los roles definidos. Sería como el fichero forms.py en Django.
	/controllers/<Class>.groovy /controllers/<RolClass>.groovy	Es el controlador, uno por cada clase y <rol,clase>. Es aquí donde se decide qué formularios mostrar o qué información enviar al cliente.
	/services/I<Class>Service.groovy	Interfaces que deben de implementar las clases que implementan la lógica de negocio
	/services/<Class>Service.groovy	Debe contener la lógica de negocio específica de cada clase. Su lógica debe ser implementada por los desarrolladores.
	src/groovy/validation/ ModelValidation.groovy	Clase donde se implementan tanto las funciones de validación definidas, así como las funciones de definición de los atributos derivados. Su lógica debe ser implementada por los desarrolladores.
	insert_SQLiteDB_metamodel.sql	Contiene las sentencias SQL que insertan en la base de datos la meta información relativa al control de acceso a cada uno de los elementos; entidad, atributos, servicios, formularios, columnas en los listados/grids, etc.
	/conf/UrlMappings.groovy	**GENERADO por el framework Grails**: es el correspondiente a url.py en Django. Establece el mapeo entre una URL y el servicio a invocar en /controllers/*.groovy. Su contenido se establece en el punto 4.4.2

Al igual que ocurre con Django, Grails también debe generar los artefactos que establecen el patrón MVC. En este caso son las clases ubicadas en /domain, /views[19] y /controller. El resto de artefactos se han creado para conseguir una mejor estructura y comprensión del código a extender para guiar a los desarrolladores. Así pues, tanto la lógica de negocio (/services/*.groovy) como las diferentes funciones (/validation/*.groovy) deben ser implementadas por éstos. Comentar que la lógica de negoció se ha generado contra unas interfaces, obligando a la implementación de los métodos ahí definidos.

4.3.2 Artefactos generados para la capa cliente

En la siguiente tabla mostramos conjuntamente los artefactos que generan los traductores (ExtJs y Touch) para la parte cliente, y cuyas

[19] Grails hace uso intensivo de la técnica scaffolding, capaz de generar los formularios a partir de las clases del dominio.

diferencias internas son mínimas, ya que tanto ExtJs (cliente desktop) como Touch (cliente móvil) pertenecen a la misma empresa: Sencha[20].

Traductor	Artefactos	Significado
ExtJs/ Touch	app/model/<Class>.js	Define el modelo de datos. Un fichero por cada clase.
	app/store/<Class>.js	Define el almacén de datos. Un fichero por cada clase.
	app/view/<Class>List.js	Define los listados/grids. Un fichero por cada clase. Para ExtJs (desktop) se establece uno por defecto para cada rol.
	app/controller/<Class>.js	Define el controlador. Un fichero por cada clase.

Todos los artefactos arriba mencionados han sido generados para la especificación del ejemplo expuesto.

4.4 Configuración del proyecto de desarrollo

Este punto se corresponde con la fase 3 del proceso PLESG. Vamos a ver como configurar los diferentes artefactos; generados y no generados, para cada uno de los frameworks de la capa servidora (Django y Grails). Hay que tener en cuenta que dichos frameworks se han instalado sobre la versión 4.2.2 de Eclipse y éste bajo el sistema operativo (SO) Mac OS X v10.6. Así pues, las instrucciones descritas en este capítulo podrían variar significativamente para otro SO.

4.4.1 Configuración en Django

Una vez se ha instalado Django[21] (versión 1.4.1) y todas sus dependencias, es posible integrarlo con PyDev[22] para poder trabajar cómodamente desde Eclipse. Para verificar la versión de Django instalada es posible ejecutar las siguientes líneas desde el terminal de Python:

```
local:~ davifer$ python
Python 2.6.5 (r265:79359, Mar 24 2010, 01:32:55)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "help", "copyright", "credits" or "license" for more information.

>>> import django
>>> print(django.get_version())
1.4.1
```

Adicionalmente es necesario instalar un componente en el *path* donde se ha instalado Python para la correcta ejecución de la aplicación generada.

[20] www.sencha.com
[21] Django: https://docs.djangoproject.com/en/dev/intro/install/
[22] PyDev: http://pydev.org/manual_adv_django.html

Componente	url	Descripción
crispy_forms	https://github.com/maraujop/django-crispy-forms	Es una aplicación Django que permite fácilmente construir, personalizar y reutilizar formularios utilizando tus CSS favoritas, como bootstrap, etc.

Las manera más fácil de instalar un componente en el path de python es ejecutando desde el terminal/consola la siguiente sentencia, donde el fichero *setup.py* se encuentra dentro de la carpeta principal de cada componente.

```
$> python <path_componente>/setup.py install
```

Si la integración con PyDev ha sido correctamente realizada, podremos crear/ejecutar un proyecto Django desde Eclipse. Para crear un nuevo proyecto se puede utilizar un asistente pulsando las teclas "Ctrl+N". Tras ello aparecera la siguiente figura, donde se debe seleccionar "PyDev Django Project" y pulsar *Next*.

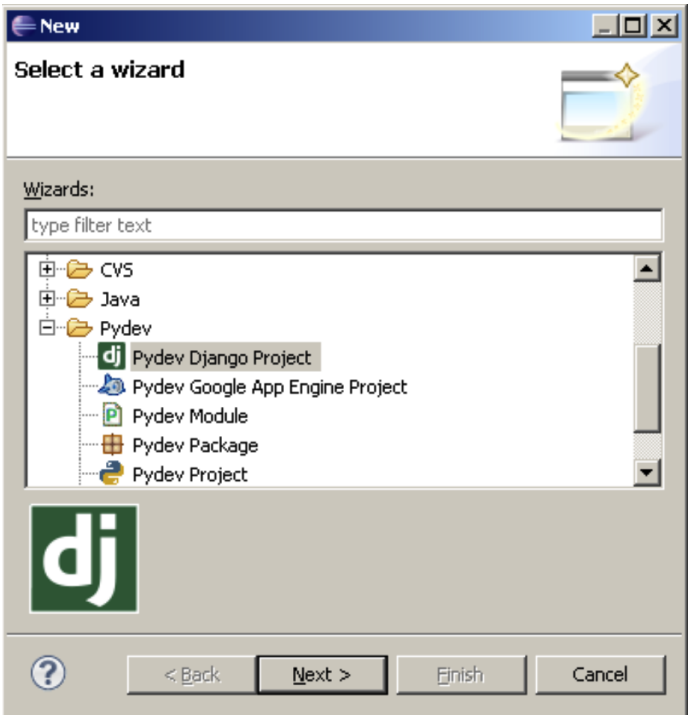


Figura 4.9: Nuevo proyecto Django desde Eclipse

Tras ello, aparecerá un nuevo diálogo donde especificar el nombre del nuevo proyecto. A modo de ejemplo hemos introducido "portaMagnaDjango". Finalmente tras pulsar *Finish* se creará el nuevo proyecto en Eclipse con la siguiente estructura.

```
portaMagnaDjango/
          manage.py
          portaMagnaDjango/
                    __init__.py
                    settings.py
                    urls.py
                    wsgi.py
```

Estructura de un nuevo proyecto Django

Sin entrar en los detalles sobre la estructura de un proyecto Django, este puede contener múltiples aplicaciones. Ahora vamos a crear una nueva aplicación, llamada "biblioteca", según se especificó en el fichero *config.model*. Para ello, desde el terminal vamos a */portaMagnaDjango* y ejecutamos la siguiente sentencia.

```
python manage.py startapp biblioteca
```

Esa misma acción se podría realizar desde Eclipse, seleccionando la carpeta raíz del proyecto y con el botón secundario seleccionamos *Django* → *Create application*, según vemos en la siguiente figura.

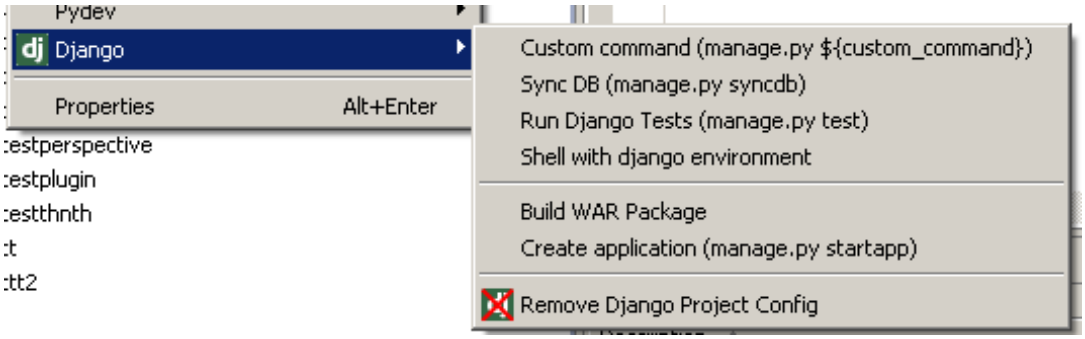


Figura 4.10: Acciones Django desde el menú contextual

Independientemente del modo elegido, se habrá creado una nueva carpeta llamada biblioteca, que contendrá varios ficheros, los cuales deberán ser sustituidos por los artefactos generados por el traductor Django, según podemos apreciar en la siguiente figura. Comentar que la carpeta se ha movido manualmente a "*/portaMagnaDjango/portaMagnaDjango*".

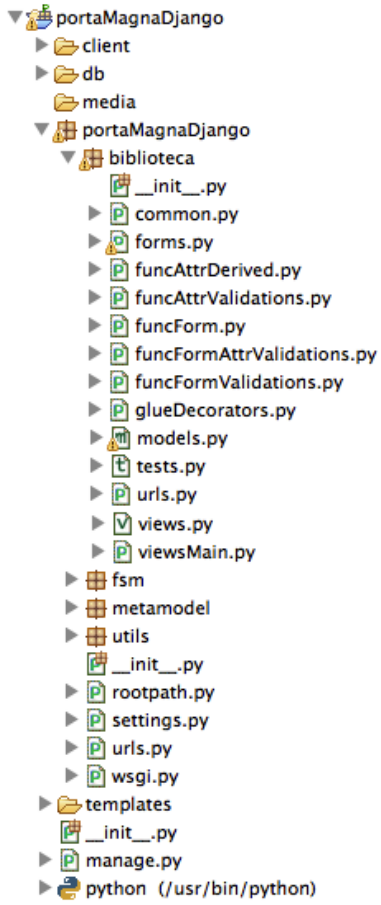


Figura 4.11: Nueva aplicación dentro del proyecto portaMagnaDjango

Como se comentó al principio del capítulo 4.3, no es una buena idea modificar los artefactos generados pero a veces es inevitable. Este es el caso de Django, donde en el recurso *views.py* se deben de implementar los servicios de la lógica de negocio de la aplicación (punto 4.5 para más detalles).

Adicionalmente se deberán incluir ciertos artefactos no generados que establecen la función de "glue code" o "código pegamento". Dichos artefactos no generados se comentan a continuación.

Artefacto no generado	Descripción
biblioteca/glueDecorators.py	Define el *decorator* @service que preprocesa los parámetros enviados desde el cliente para que el servicio invocado tenga ya instanciado los objetos necesarios.
biblioteca/funcForm.py	Es el único artefacto no generado que se invoca desde código no generado, concretamente desde el fichero *glueDecorators.py*. Este fichero alberga funciones de diferente ámbito, como es el caso de *setModifiedForm* (ver punto 4.5)
biblioteca/viewsMain.py	Contiene ciertos servicios clave para el funcionamiento dinámico de la aplicación desde los cuales se invocan a las funciones definidas en el fichero *metamodel/rolView.py*, y cuyo significado se explicará en breve
utils/enum.py	Define la clase Enumeration para poder crear atributos de tipo enumerado en Django.

utils/middleware.py	Define la clase *MobileDetectionMiddleware* que permite saber desde qué dispositivo se ha realizado la petición de conexión para redireccionar al fichero index.html adecuado.
utils/readOnly.py	Permite establecer que puedan existir en un formulario atributos de solo lectura.
utils/serializer.py	Se encarga de serializar la información que viaja desde el servidor al cliente a formato JSON. En el cliente, esa información es deserializada a objetos JavaScript.
utils/jsonview.py	Se encarga de deserializar automáticamente las peticiones en formato JSON desde el cliente a sus correspondientes objetos en python.
fsm/grammar.py	Proporciona soporte para poder definir una máquina de estados finita en Django.
fsm/fsm.py	Define la clase *FiniteStateMachine*

También se debe incluir una aplicación especial llamada *metamodel*, en la cual existen dos ficheros

- *models.py*: donde se han definido ciertas clases que definen los metadatos; son la propia especificación del sistema encargada de gestionar la parte dinámica de la aplicación. Entre dichas gestiones podemos destacar:

Gestión	en función del
Visibilidad sobre las entidades	rol
Gestionar los servicios que aparecen en cada formulario	rol, entidad y estado de la clase
Gestión de las propiedades a mostrar en los listados	rol, entidad y dispositivo
Gestión de las propiedades a editar/mostrar en los formularios	rol y entidad

- *rolView.py*: establece las funciones de consulta hacia la base de datos para obtener los metadatos específicos del sistema. Recordemos que estos metadatos establecen la parte dinámica de la aplicación.

Existen algunos aspectos adicionales a configurar de forma manual en la parte servidora:

1. se debe de modificar el fichero de configuración del proyecto (*setttings.py*) para configurar ciertos parámetros, entre ellos:
   - añadir el middleware *MobileDetectionMiddleware*
   - modificar la zona horaria
   - instalar las diversas aplicaciones creadas
   - configuración del motor de base de datos (ver punto 4.6.1)
   - y algunos parámetros adicionales

2. se debe incluir un fichero especial llamado *rootpath.py* que define una variable llamada *ROOTPATH*, la cual debe contener la ruta absoluta al proyecto. Esta variable es utilizada en el fichero de configuración del proyecto para establecer:
   - la ruta absoluta donde se encuentran instalados los ficheros estáticos (plantillas, css, imágenes, …) de la aplicación

- la ruta absoluta donde se guardarán los ficheros multimedia que puedan eventualmente ser subidos en tiempo de ejecución
- la ruta absoluta donde se han instalado las plantillas *.html

Por último, quedan los pasos relativos a la configuración de la parte cliente en el servidor:

a) incluir los diferentes artefactos generados y no generados pertenecientes a la capa cliente

b) establecer el orden de carga de los anteriores artefactos en el fichero *index.html*, que establece el punto de entrada a la aplicación web

c) incluir las plantillas para la generación dinámica de formularios y autenticación contra el sistema.

Respecto al punto a), en la siguiente tabla podemos ver los diferentes artefactos, generados y no generados, que constituyen la capa cliente y los cuales se han establecido dentro de la ruta del proyecto */portaMagnaDjango/client*.

Artefactos	Ruta	Ver.
Framework ExtJs: contiene los archivos SDK de Ext JS 4	/client/ext	4.1.1a
Framework Touch: contiene los archivos SDK de Sencha Touch 2	/client/touch	2.1.1
Artefactos generados por el traductor uiExtJs donde el estilo de nombres debe seguir la convención que figura en la guía del Sistema de clases de ExtJs	/client/js-biblioteca/app/	1.0
Middleware entre los artefactos generados por el traductor uiExtJs y la parte servidora	/client/js-biblioteca/*.js	1.0
Artefactos generados por el traductor uiTouch donde el estilo de nombres debe seguir la convención que figura en la guía del Sistema de clases de Sencha Touch	/client/js-biblioteca-touch/ app/	1.0
Middleware entre los artefactos generados por el traductor uiTouch y la parte servidora	/client/js-biblioteca-touch/*.js	1.0
Framework Bootstrap	/client/bootstrap	2.2.1
Framework JQuery	/client/jquery	1.9.1
Contiene los archivos CSS adicionales y las imágenes que son responsables de la apariencia común de la aplicación	/client/css	1.0
	/client/imagenes	-

Anteriormente ya se comentaron los diferentes artefactos generados para la capa cliente (traductores uiExtJs y uiTouch) en el punto 4.3.2. Así pues, ahora vamos a centrarnos en comentar los diferentes artefactos textuales no generados.

Empezaremos por los artefactos ubicados en */client/js-biblioteca/*.js,* que como bien se ha definido anteriormente en la tabla, son el middleware entre el cliente (desktop) y el servidor, y adicionalmente permiten la extensión organizada de la parte cliente de la aplicación. Veamos mediante la siguiente tabla la funcionalidad correspondiente a cada uno:

Recurso	Significado
app.js	Es el punto de entrada a toda aplicación ExtJs. En él se debe de establecer manualmente el nombre de la aplicación, la ruta hacia la carpeta donde se han ubicado los artefactos generados (/client/js-biblioteca/app), así como el nombre de los diferentes controladores generados. Para obtener más información ir a: http://docs.sencha.com/extjs/4.1.1/#!/api/Ext.app.Application
soGex.js	**Establece el middleware entre la aplicación cliente y el servidor**. Define el namespace global y sobre él se apoyan el resto de artefactos. Además permite, entre otras cosas, establecer extensiones al modelo de objetos Javascript, así como posibles parches y correcciones a ExtJS.
soGex.locale-es.js	Recurso donde se define el valor de las etiquetas y ciertos mensajes de aviso al usuario en el idioma español. Se deberá crear un recurso por idioma y carga el adecuado según las necesidades.
soGex.Utils.js	Contiene diferentes funciones javascript de propósito general.
soGex.data.js	En este fichero se definen todas las llamadas ajax específicas hacia el servidor, con la excepción de la llamada ajax para refrescar las instancias de cualquier grid, definida en app/store/*.js
soGex.Actions.js	Si se desea extender la funcionalidad de la aplicación, es en este recurso donde se deben definir las diferentes funciones que contendrán la lógica de la aplicación de la parte cliente.
soGex.dynamicTab.js	Se ha definido una aplicación ExtJs basada en pestañas. Este recurso controla la carga y borrado dinámico de los diversos listados y formularios en dichas pestañas.

En la siguiente figura podemos ver como se integran los diferentes artefactos generados y no generados para la versión desktop, incluido el framework ExtJs.

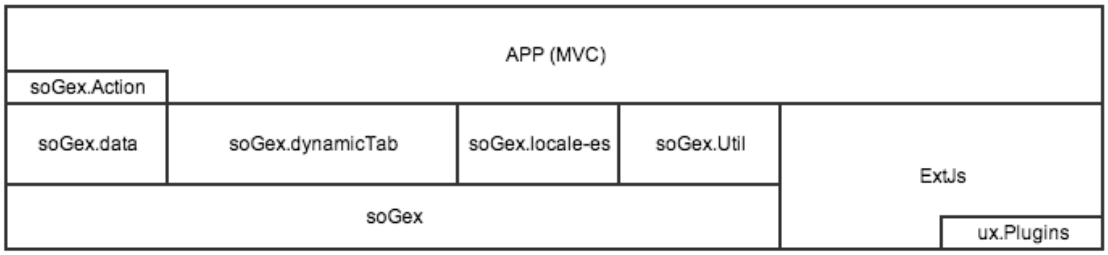


Figura 4.12: Integración entre artefactos de la capa cliente (ExtJs: desktop)

Recordemos que en APP (MVC) se ubican los artefactos generados.

Respecto a los artefactos ubicados en *client/js-biblioteca-touch/*.js, éstos* son los mismos que se definieron para ExtJs, aunque se han incluido dos más, encargados de mostrar los diferentes tipos de formularios. En la siguiente tabla mostramos sólo esos dos nuevos artefactos.

Recurso	Significado
app.js	Existen algunas diferencias respecto a su framework hermano (extjs) que hace que sea necesario especificar además de los controladores, los modelos, las vistas y los almacenes (stores). Para obtener más información ir a: http://docs.sencha.com/touch/2.1.1/#!/api/Ext.app.Application
soGex.htmlPanel.js	Encargado de mostrar los formularios de instancia de clase
soGex.htmlSubPanel.js	Desde un formulario de instancia de clase es posible ir a un formulario específico para un servicio concreto de una clase.

En la siguiente figura podemos ver la integración entre los diferentes artefactos generados y no generados para la versión móvil, junto con el framework Sencha Touch.

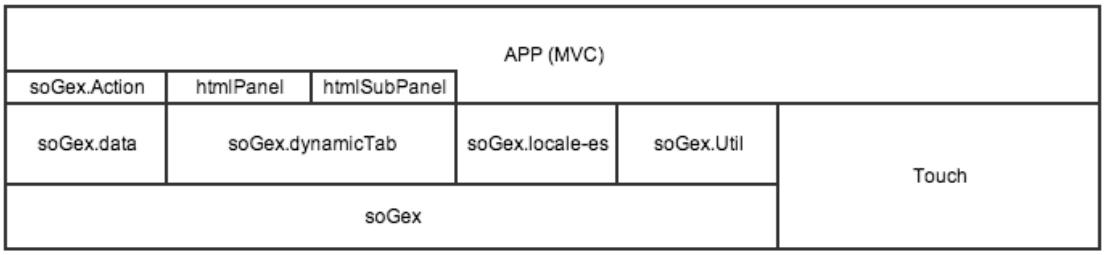


Figura 4.13: Integración entre artefactos de la capa cliente (Touch: móvil)

Por último, se han definido un conjunto de ficheros con extensión **.html*, ubicados dentro del proyecto */portaMagnaDjango/templates,* que engloban los aspectos definidos en los dos últimos puntos anteriores; b) y c). En la siguiente figura pueden contemplarse todos ellos.

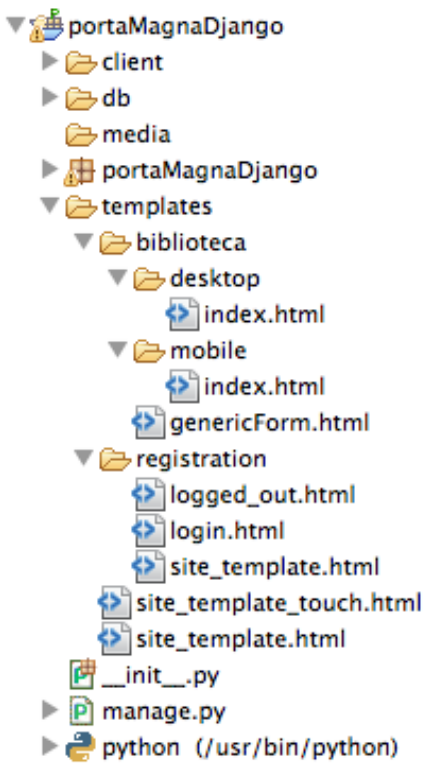


Figura 4.14: Estructura de la carpeta templates

Django ofrece la posibilidad de definir formularios a partir de los modelos definidos en *models.py* mediante el uso de la clase ModelForm[23]. Adicionalmente, y para facilitar aún más la tarea, se instaló la aplicación *crispy_forms*, según se indicó al comienzo del apartado, con el objetivo de poder crear una plantilla genérica, llamada *genericForm.html*, que permite generar dinámicamente cualquier formulario html solicitado por parte del usuario conectado a la aplicación.

[23] ModelForm: https://docs.djangoproject.com/en/dev/topics/forms/modelforms/

Un vez se han configurado todos los artefactos, generados y no generados, dentro de la ruta */portaMagnaDjango/client*, es necesario establecer el orden de carga de éstos dentro del fichero *index.html* tanto para la versión de desktop como para la versión móvil. Estos ficheros heredan de las plantillas site_template.html y site_template_touch.html respectivamente.

Por último, hay que verificar que no hay errores de compilación, por ejemplo importando desde la consola de python los diferentes recursos declarados (ir a la documentación de Django para más información).

4.4.2 Configuración en Grails

El primer paso es instalar Grails[24] (v2.2.3), junto con todas sus dependencias, y configurarlo correctamente para poder trabajar desde Eclipse.

Una vez configurado, es posible crear un nuevo proyecto mediante el uso de un asistente, pulsando las teclas “Ctrl+N” y seleccionando “Grails Project”.

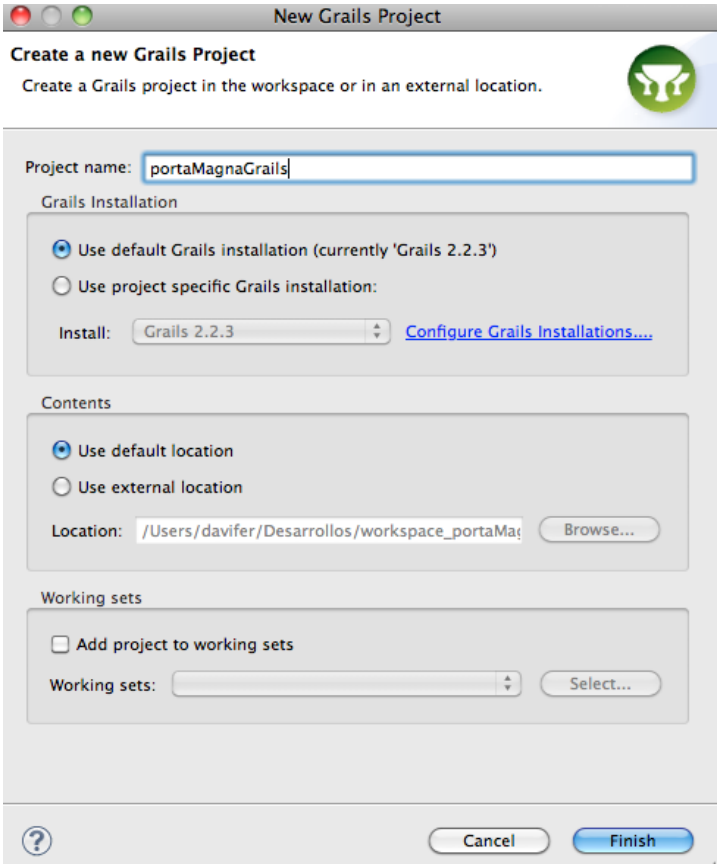


Figura 4.15: Nuevo proyecto Grails desde Eclipse

Tras especificar el nombre del nuevo proyecto, a modo de ejemplo hemos introducido “portaMagnaGrails”, y pulsar *Finish*, se lanzará un proceso cuyas acciones pueden verse en la consola de Eclipse.

[24] http://grails.org/doc/latest/guide/gettingStarted.html#requirements

```
Loading Grails 2.2.3
| Configuring classpath
| Downloading: resources-1.2.pom
| Downloading: resources-1.2.zip.

| Environment set to development.....

| Installing zip tomcat-2.2.3.zip.....
| Installed plugin tomcat-2.2.3.....
| Installing zip cache-1.0.1.zip.....
| Installed plugin cache-1.0.1...
| Installing zip webxml-1.4.1.zip.....
| Installed plugin webxml-1.4.1...
| Installing zip hibernate-2.2.3.zip.....
| Installed plugin hibernate-2.2.3...
| Installing zip jquery-1.8.3.zip.....
| Installed plugin jquery-1.8.3...
| Installing zip resources-1.2.zip.....
| Installed plugin resources-1.2...
| Installing zip database-migration-1.3.2.zip.....
| Installed plugin database-migration-1.3.2.....

| Compiling 121 source files
| Compiling 121 source files..
| Compiling 8 source files.
```

Básicamente se configura el *classpath* para el nuevo proyecto, donde se instalan ciertas dependencias según la versión concreta de Grails. Tras finalizar el proceso aparecerá en Eclipse el nuevo proyecto con la siguiente estructura.

```
portaMagnaGrails/
    grails-app/
        conf/
        controllers/
        domain/
        i18n/
        services/
        taglib/
        utils/
        views/
    src/
        groovy/
        java/
    web-app/
    scripts/
    test/
    lib/
```

Estructura de un nuevo proyecto Grails (vista desde el explorador)

Adicionalmente es necesario instalar el plugin *spring-mobile* para la correcta ejecución de la aplicación generada.

Componente	url	Descripción
spring-mobile	http://grails.org/plugin/spring-mobile	La principal característica es la detección del dispositivo (desktop o móvil) a partir de la petición *httpRequest*.

Las manera más fácil de instalar dicho plugin es mediante el gestor de plugins que ofrece Grails desde Eclipse.

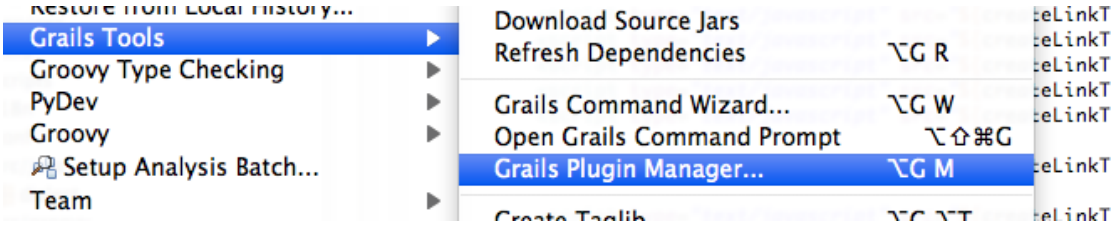


Figura 4.16: Acciones para Grails desde Eclipse

Según se muestra en la figura anterior, tras seleccionar *Grails Plugin Manager* aparecerá el gestor que permitirá instalarlo. Una vez hecho hay que configurarlo en el fichero *grails-app/conf/BuildConf.groovy* según los pasos que se muestran a continuación.

1. descomentar los “repositories” que existen y añadir

```
repositories {
                ...
  mavenRepo "http://maven.springframework.org/milestone/"
```

2. incluir en “dependences”

```
dependences {
  compile "org.springframework.mobile:spring-mobile-device:1.0.0.M3"
  runtime "net.sourceforge.wurfl:wurfl:1.2"
}
```

3. incluir en “plugins”

```
compile ":spring-mobile:0.4"
```

e incluir también en conf/Config.groovy las siguientes líneas

```
springMobile {
        deviceResolver="wurfl"
}
```

Finalmente hay que refrescar las dependencias, para ello desde *Grails Tools* seleccionar la acción *Refresh Dependencies*. En la consola de Eclipse se mostrará dicho proceso.

```
Loading Grails 2.2.3
| Configuring classpath.
| Environment set to development.....
| Installing zip spring-mobile-0.4.zip.....
| Installed plugin spring-mobile-0.4.....
| Compiling 121 source files
| Compiling 121 source files..
| Compiling 8 source files.
```

Sin entrar en detalles sobre la estructura de un proyecto Grails, debemos crear varios paquetes donde ubicar los distintos artefactos generados por el traductor Grails tal y como se indica en la siguiente tabla.

Crear los paquetes	Incluir artefactos generados
grails-app/domain/**biblioteca**	Grails/domain/*.groovy
grails-app /controllers/**biblioteca**	Grails/controllers/*Controller.groovy
grails-app /services/**biblioteca**	Grails/services/*Service.groovy
src/groovy/**enumerators**	Grails/enumerators/*.groovy
src/groovy/**validation**	Grails/validation/*.groovy
src/groovy/**util**	Grails/util/util.groovy

En Grails/services/ se han generado las interfaces "I<domainclass>Service.groovy" que el código no generado debe implementar. Realmente estos servicios (<domainClass>Service.groovy) sí se han generado, pero sólo como ayuda a los desarrolladores, por si estos deciden implementar directamente ahí la lógica de negocio, o se opta por otra estrategia, como utilizar procedimientos almacenados, etc., aunque su invocación podría establecerse desde dichos servicios.

Por otra parte se deberán de incluir ciertos artefactos no generados para la correcta ejecución de la aplicación. Dichos artefactos se comentan a continuación.

Artefacto no generado	Descripción
grails-app/controllers/ metamodel/MainController.groovy	Contiene ciertos servicios clave para el funcionamiento dinámico de la aplicación desde los cuales se invocan a las funciones definidas en src/groovy/metamodel/ *MetaModel.groovy*
src/groovy/metamodel/ MetaModel.groovy	Establece las funciones de consulta hacia la base de datos para obtener los metadatos específicos del sistema. Recordemos que estos metadatos establecen la parte dinámica de la aplicación
src/groovy/metamodel/ DataType.groovy	Esta clase forma parte del paquete especial llamado grails-app/domain/metamodel que se verá a continuación. Al ser una clase abstracta se debe ubicar en src/groovy
grails-app/controllers/common/ SecureBaseController.groovy	Cierra la sesión cuando ésta caduca. Todos los controladores generados heredan de esta clase.
src/groovy/fsm/Grammar.groovy	Proporciona soporte para poder definir una máquina de estados finita en Grails.
src/groovy/fsm/ FiniteStateMachine.groovy	Define la clase *FiniteStateMachine*
src/groovy/util/Common.groovy	Define la clase Common con varios métodos estáticos invocados desde grails-app/controllers/metamodel/ MainController.groovy
src/java/dialect/SQLiteDialect.java	Clase que representa el dialecto usado por Hibernate para comunicarse con la base de datos SQLite
lib/sqlite-jdbc-3.7.2.jar	Driver para conectar con la base de datos SQLite

Adicionalmente se debe incluir un paquete especial llamado grails-app/domain/*metamodel*, en la cual existen un conjunto de clases que definen los metadatos especiales; son la propia especificación del sistema encargada de gestionar la parte dinámica de la aplicación. Dichas gestiones son las mismas que se comentaron anteriormente para Django:

Gestión	en función del
Visibilidad sobre las entidades	rol
Gestionar los servicios que aparecen en cada formulario	rol, entidad y estado de la

	clase
Gestión de las propiedades a mostrar en los listados	rol, entidad y dispositivo
Gestión de las propiedades a editar/mostrar en los formularios	rol y entidad

Existen tres últimos aspectos adicionales a configurar de forma manual en la parte servidora:

1. Incluir en el fichero *conf/UrlMappings.groovy*, que controla las peticiones desde el cliente, las siguientes líneas de código.

```
static mappings = {
        "/$controller/$action?"{
                format = "json"
        }

        "/$controller/$action?/$id?"{
                constraints {
                        // apply constraints here
                }
        }

        "/main/setRol"(controller:'main', action:'setRol'){
                format = "json"
        }

        "/main/getRolViewEntities"(controller:'main', action:'getEntities'){
                format = "json"
        }

        "/main/getListModel"(controller:'main', action:'getListModel'){
                format = "json"
        }

        "/main/search_paged"(controller:'main', action:'search_paged'){
                format = "json"
        }

        "/main/get"(controller:'main', action:'getForm'){
                format = "json"
        }

        "/main/getServices"(controller:'main', action:'getServices'){
                format = "json"
        }

        "/main/deleteList"(controller:'main', action:'deleteList'){
                format = "json"
        }

        "/user/loggedOut"(controller:'user', action:'doLogout')
        "/"(controller:'user', action:'doLogin')
        "500"(view:'/error')
}
```

2. Es necesario cambiar el nombre de la aplicación del proyecto Grails, ubicada en el fichero *portaMagnaGrails/application.properties*, por el que se especificó en la variable appName del fichero *Config.model* de la especificación PLESG. Esto es debido a que para Grails el proyecto es una aplicación, por lo que le asigna el nombre del proyecto a la aplicación.

```
#Grails Metadata file
#Wed Sep 04 18:15:21 CEST 2013
app.grails.version=2.2.3
app.name=biblioteca
app.version=0.1
```

3. Grails proporciona soporte para la internacionalización basada en la configuración regional del usuario según el sistema operativo. Dicha configuración puede forzarse a un idioma determinado (en concreto a "es"), estableciendo el siguiente código en el recurso *conf/spring/resources.groovy*, según vemos a continuación:

```
import org.springframework.web.servlet.i18n.SessionLocaleResolver

beans = {
  localeResolver(SessionLocaleResolver) {
      defaultLocale = new Locale('es');
      Locale.setDefault (defaultLocale)
  }
}
```

En la ruta del proyecto */grails-app/i18n/* existen múltiples ficheros con el formato "messages_<ID>.properties" que contienen la misma información pero en distintos idiomas, en los cuales se parametrizan entre otras cosas:

- los nombres y el texto de ayuda de las propiedades de las clases
- los mensajes de error a mostrar cuando se produce un fallo en la validación de un formulario
- el formato de la fecha
- etc…

En consecuencia, tanto el alias como el texto de ayuda de los atributos de las clases deben de especificarse en el fichero "messages_es.properties" de forma manual.

Por último, quedan los pasos relativos a la configuración de la parte cliente en el servidor:

a) instalar los diferentes artefactos generados y no generados pertenecientes a la capa cliente

b) establecer el orden de carga de los anteriores artefactos en el fichero de entrada de la aplicación *index.gsp* (desktop/móvil)

c) instalar las plantillas para la generación dinámica de formularios y autenticación contra el sistema.

Respecto al punto a), en la siguiente tabla podemos ver los diferentes artefactos, generados y no generados, que constituyen la capa cliente y los cuales se han establecido dentro de la ruta del proyecto */portaMagnaGrails/web-app/*.

Artefacto	Ruta	Ver.
Framework ExtJs: contiene los archivos SDK de Ext JS 4	web-app/js/ext	4.1.1a
Framework Touch: contiene los archivos SDK de Sencha Touch 2	web-app/js/touch	2.2.1
Artefactos generados por el traductor uiExtJs donde el estilo de nombres debe seguir la convención que figura en la guía del Sistema de clases de Ext JS	web-app/js/js-biblioteca/app	1.0
Middleware entre los artefactos generados por el traductor uiExtJs y el servidor	web-app/js/js-biblioteca/*.js	1.0
Artefactos generados por el traductor uiTouch donde el estilo de nombres debe seguir la convención que figura en la guía del Sistema de clases de Touch	web-app/js/js-biblioteca-touch/app	1.0
Middleware entre los artefactos generados por el traductor uiTouch y el servidor	web-app/**js**/js-biblioteca-touch/*.js	1.0
Framework Bootstrap	/web-app/bootstrap	2.2.1
Framework JQuery	/web-app/jquery	1.9.1
Contiene los archivos CSS adicionales y las imágenes que son responsables de la apariencia común de la aplicación	/web-app/css	1.0
	/web-app/imagenes	-

Los diferentes artefactos textuales no generados son exactamente los mismos que se comentaron en Django, por lo tanto no se van a volver a explicar.

Por último, se han definido un conjunto de ficheros con extensión **.gsp*, que deben instalarse en */portaMagnaGrails/grails-app/views,* y engloban parte de los aspectos definidos en los dos últimos puntos anteriores; b) y c). Estos deben ubicarse según se observa en la siguiente figura, eliminando los ficheros existentes que hubieran tras la creación del proyecto.

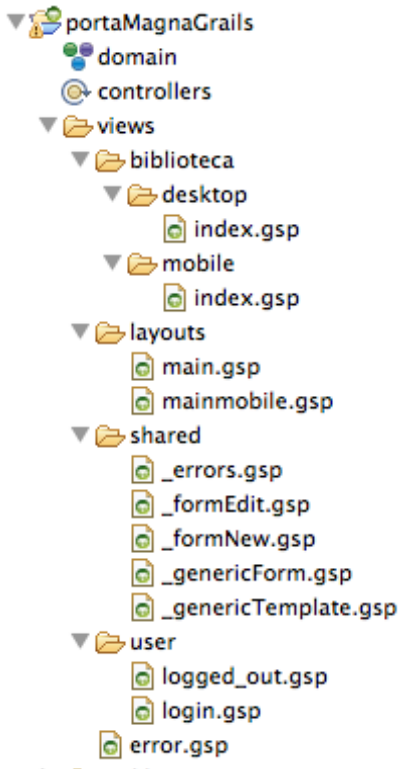


Figura 4.17: Estructura de la carpeta views en Grails

Grails ofrece la posibilidad de inferir los formularios CRUD a partir de los modelos definidos mediante la técnica de *scaffolding*. Sin embargo, para usos más avanzados se hace necesario instalar las plantillas en las que se basa dicha técnica y adaptarlas a las necesidades de la propuesta PLESG, por ejemplo, para poder ver los atributos derivados en los formularios.

Para instalar dichas plantillas debemos seleccionar el proyecto con el botón secundario y pulsar *Grails Tools → Grails Command Wizard…* Tras ello aparecerá un asistente, seleccionamos *install-templates* y pulsamos *Finish*. En la consola de Eclipse se mostrará el proceso de instalación.

```
Loading Grails 2.2.3

| Configuring classpath.
| Environment set to development.....
| Templates installed successfully
```

Si todo ha ido bien se habrán instalado en *src/templates* las plantillas utilizadas por Grails para la generación de artefactos (controladores, vistas, servicios, clases, etc.) y formularios CRUD. El siguiente paso es modificar la plantilla "*_form.gsp"* ubicada en *src/templates/scaffolding* para:

- para poder visualizar todos los atributos
- incluir una línea que indique el final del formulario
- mostrar el texto de ayuda de cualquier campo en el formulario

Concretamente hay que:

- sustituir

```
                                        ...
4.  persistentPropNames = domainClass.persistentProperties*.name
```

por

```
    persistentPropNames = domainClass.properties*.name
```

- e incluir el siguiente código en la línea 21 y a partir de la 44

```
21. %><legend></legend><%
                                        ...
44. <help for="${prefix}${p.name}">
45. <g:message code="${domainClass.propertyName}.${prefix}${p.name}.help" default=""
/>
46. </help>
                                        ...
```

Un vez se han configurado todos los artefactos (generados y no generados) dentro de la ruta */portaMagnaGrails/web-app*, es necesario establecer el orden de carga de éstos dentro del fichero *index.gsp*, tanto para la versión de escritorio como la de móvil.

Por último, hay que verificar que no hay errores de compilación. Para ello seleccionamos *Grails Tools → Grails Command Wizard.* Tras ello aparecerá un asistente, seleccionamos *compile* y pulsamos *Finish*.

```
Loading Grails 2.2.3
| Environment set to development.....
```

De aparecer algún error deberemos solucionarlo antes de pasar al siguiente apartado, en el cual vamos a ver cómo y dónde se han implementado las extensiones específicas para conseguir la funcionalidad requerida según las especificaciones expuestas en el punto 4.1.

4.5 Implementación de las extensiones específicas

Este punto se corresponde con la fase 4 del proceso PLESG. Según el proceso definido, una vez se han generado y configurado los diferentes artefactos, es el momento en el que los programadores cualificados en cada uno de los frameworks de desarrollo implementen la funcionalidad requerida. Estos aplicarán los estándares necesarios para garantizar la correcta implementación de la lógica de negocio en función de los requisitos establecidos, por ejemplo mediante el uso de patrones de diseño bien conocidos [19], así como de los requerimientos no funcionales necesarios; multiidioma, seguridad, escalabilidad, etc.

Se ha decidido simplificar la complejidad del modelo funcional reduciendo una parte del conjunto de las especificaciones globales sobre la gestión de prestamos de libros, expuestas en el punto 4.1, según se detalla a continuación:

- no se considera la gestión de las reservas de libros por parte de los clientes. Así pues, un libro puede ser prestado o devuelto.

- la sanción a los clientes que no devuelven los libros prestados a tiempo es realizada manualmente por los empleados, no por el sistema. Lo mismo ocurre con el levantamiento de dicha sanción.

- El sistema no avisa a los usuarios ante ningún cambio de estado de los objetos. Por ejemplo, un sistema de aviso deseable sería que el sistema alerte al cliente que hizo una reserva de un libro prestado que ha sido devuelto, con el fin de no perder dicha reserva.

Dicha simplificación es debida a que el alcance del presente apartado es dar una guía de dónde implementar las partes más importantes de la funcionalidad requerida, mostrando ejemplos al respecto, pero sin entrar en detalles sobre cómo se han implementado.

Para facilitar dicha tarea se han extraído varios puntos importantes a implementar en toda aplicación de gestión PLESG. Éstos se han especificado en la siguiente tabla, mostrando un ejemplo concreto sobre la lógica a implementar y dónde hacerlo en cada uno de los entornos. Hay que tener presente que el modo en el que se ha desarrollado la implementación de cada ejemplo mostrado no pretende ser la única forma de hacerlo, y muy posiblemente no sea la mejor de ellas, y sólo debe tomarse como una sugerencia.

	Aspecto	Ejemplo de lógica a implementar		Recurso donde implementar la lógica
1	Modificar dinámicamente los valores de un combo en un formulario	Al crear un préstamo, debemos de poder seleccionar un libro en estado “libre” y un cliente “no sancionado”	django	funcForm.py
			grails	Múltiples ficheros. Se verá a continuación.
2	Implementar la lógica de negocio	Servicio “prestar” de la clase préstamo	django	views.py
			grails	<clase>Service.groovy
3	Implementar las funciones de los atributos derivados	Función del atributo derivado *full_name* de la clase persona y *numero* de la clase cliente	django	funcAttrDerived.py y/o models.py

			grails	validation/ModelValidation.* y/o domain/<clase>.*
4	Implementar las funciones de validación de los atributos	Función de evaluación validator_even() del atributo código de la clase empleado	django	funcAttrValidations.py
			grails	domain/<clase>.groovy
5	Validaciones de los atributos propios de los formularios	Si se incluye un atributo adicional en un formulario, éste también puede tener validaciones.	django	funcFormAttrValidations.py
			grails	domain/<rol><clase>. groovy
6	Otras validaciones. Opcional y exclusivo para Django.	Permite modificar los errores de validación de los atributos que se muestran en la cabecera del formulario.	django	funcFormValidations.py
7	Modificación de los ficheros CSS e imágenes	Es posible adaptar los estilos e imágenes a mostrar en la aplicación.	django	client/css client/imagenes
			grails	web-app/css web-app/images

Veamos ahora en detalle cada uno de los puntos anteriores:

1. Modificar dinámicamente los valores de un combo en un formulario

    a. Lógica: cuando un empleado crea un préstamo, debe de poder seleccionar un libro en estado “libre” y un cliente “no sancionado”, por lo que se hace necesario filtrar el valor de los combos mostrados por defecto, pues estos devuelven por defecto todas las instancias de la clase.

- En Django este requisito se puede implementar en la función *setModifiedForm*, ubicada en el fichero *funcForm.py*.

```
from workflow.libreria.models import *

def setModifiedForm(request, className, form, isNew):
  if (className == "prestamo"):
      if(isNew):
        libro_choices=[(c.id,c.titulo)for c in libro.objects.filter(estado="libre")]
        form.fields['implica'].choices = libro_choices
                                          ...
```

El parámetro *isNew* indica si el formulario *form* es de nueva instancia y el parámetro *className* indica el nombre de la clase a la que pertenece el formulario que queremos modificar. El parámetro *request* contiene entre

otros; la variable de sesión del sistema y los diferentes parámetros enviados desde el cliente.

- En Grails debemos de realizar varios pasos, algunos de los cuales implican modificar diferentes artefactos generados por parte del framework:

  - primero debemos generar la vista a partir de la clase StaffPrestamo, ejecutando el comando "Generate Views" desde eclipse o por línea de comandos.

```
Loading Grails 2.2.3
| Configuring classpath.
| Environment set to development.....
| Packaging Grails application.....
| Packaging Grails application.....
| Generating views for domain class biblioteca.StaffPrestamo

> File /grails-app/views/staffprestamo/_form.gsp already exists. Overwrite?[y,n,a] a

| Finished generation for domain class biblioteca.StaffPrestamo
```

  - una vez hemos generado las plantillas para la clase StaffPrestamo, vamos a "_form.gsp" y cambiamos el valor del parámetro "from" del Tag "g:select" por la variable que deseamos y la cual será instanciada desde el método show() del Controlador StaffPrestamo, según los requisitos funcionales. En este caso, cambiamos:

```
<g:select id="implica" name="implica.id" from="${biblioteca.Libro.list()}" ...
```

por

```
<g:select id="implica" name="implica.id" from="${implicaList}" ...
```

  - en el método show del controlador *StaffPrestamoController.groovy*, instanciamos la variable implicaList según vemos en el siguiente código.

```
                                        ...
def show(Long id) {
        def objInstance = new StaffPrestamo()
        def implicaLista = null
        if(id > 0){
                //OBTENEMOS LA INSTANCIA DE LA CLASE PADRE
                def parent = Prestamo.get(id)
                                        ...

                def args = [objInstance, parent.properties]
                bind.invoke(objInstance, 'bind', (Object[])args)
        } else {
                //Nuevo prestamo
                implicaLista = Libro.findAllWhere(estado:"libre")
        }

        render(template:"/shared/genericTemplate", model:[legend: "PRESTAMO",
```

```
className: "prestamo", implicaList: implicaLista, staffPrestamoInstance: objInstance,
objInstance: objInstance])
}
                                        ...
```

Estas mismas extensiones deberán desarrollarse también para el cliente que interviene en un préstamo.

2. Implementar la lógica de negocio

- En Django toda la lógica se puede implementar en el fichero views.py, donde se han generado los servicios para facilitar la tarea. Un ejemplo de ello podemos verlo en el servicio *prestamo_prestar()*, donde en gris se muestra el desarrollo manual.

```
@service()
def prestamo_prestar(request, obj, form):
    results = {}
    try:
        if form.is_valid():
            # SE DEBE MODIFICAR:
            #  1. cambiar el libro a prestado
            objlibro = libro.objects.get(id=int(form.data['implica']))
            if(objlibro.estado == "prestado"):
                raise Exception('Operacion no permitida: libro no disponible')

            #  2. lector es deudor
            objlector = cliente.objects.get(id=int(form.data['pertenece']))
            if(objlector.estado == "sancionado"):
                raise Exception('Operacion no permitida: lector sancionado')
            objlibro.prestar()
            objlector.prestar()

            obj = form.save()
            obj.prestar()

            results['success'] = True
            results['id'] = obj.id
            results['text'] = "create OK"
            return HttpResponse(simplejson.dumps( results ), mimetype='json')

        # Form NO VALIDO, repintamos el formulario con los errores
        return render(request, WEBAPP + '/genericForm.html', {'form': form,})
    except Exception, e:
        #ERRROR: ie; x restricciones del objeto al tener relacion con otras entidades
        context = {
            'success': False,
            'text': e.message,
            'id': obj.id
        }
        return HttpResponse(simplejson.dumps(context), mimetype="application/json")
```

- En Grails la lógica de negocio de los servicios propios a cada clase hay que implementarla en el fichero services/biblioteca/<clase>Service.groovy que implementa la interfaz /I<clase>Service.groovy generada. En gris se muestra el desarrollo manual del método prestar de la clase PrestamoService.

```
package libreria
import grails.validation.ValidationException

class PrestamoService implements IPrestamoService {
       int prestar(Object params){
              def obj = null

              try{
                params.id = params.int('id') //si es id = nuevo.. --> exception
              } catch(ex){ params.id = null }

              try{
                  obj = new Prestamo(params)

                 //TODO: Additional logic in your service go here
                 //CAMBIAMOS EL ESTADO DEL LIBRO
                 def libro = Libro.get(params['implica.id'])
                 libro.fireEvent("prestar")

                 //CAMBIAMOS EL ESTADO DEL LECTOR
                 def lector = Cliente.get(params['pertenece.id'])
                 lector.fireEvent("prestar")

                 obj.fireEvent("prestar")
                 libro.save(libro)
                 lector.save(lector)
                 obj.save(obj)
              } catch (Exception e){
                 throw new ValidationException("Prestamo is not valid",obj.errors)
              }
              return obj.id
       }
                                          ...
}
```

Otra opción igualmente válida sería trasladar la lógica de negocio a la base de datos, por ejemplo, mediante la implementación de procedimientos almacenados, triggers, etc.

3. Implementar las diversas funciones de los atributos derivados

Se han definido varios atributos derivados cuyas funciones deben implementarse. Vamos a ver los ejemplos para *full_name* en la clase persona y *numero* en la clase cliente.

Cuando los valores de un atributo derivado dependen de otros atributos de la misma clase (*refOtherClasss = False, valor por defecto*), dichas funciones pueden implementarse fuera del fichero donde se define la clase.

- En el caso de Django, la primera función se ha implementado en el fichero *funcAttrDerived.py*. En el siguiente cuadro podemos ver el código implementado.

```
        #function name generated: write your custom code
        def getFullName(self):
        ""
                #Please, write the code of this function
                #return '%s' % ("not implemented")
```

```
            return '%s %s' % (self.nombre, self.apellidos)
```

Fichero funcAttrDerived.py

Sin embargo, cuando los valores de un atributo derivado dependen de otras clases, la función se debe de implementar en la propia clase, al menos para Django. Así, la función getTotalPrestamos, asociada al atributo derivado *numero* se debe de implementar dentro del fichero *models.py*.

```
    class cliente(persona):
        estado = FSMField("Estado",default='-')
        def getTotalPrestamos(self):
                #Please, write the code of this function
                query = prestamo.objects.values().filter(pertenece=self.id)
                return '%s' % (str(query.count()))
        numero = property(getTotalPrestamos)
                                                ...
```

Fichero models.py

- En Grails, la primera función se implementa en el fichero *validation/ModelValidation.groovy*.

```
class ModelValidation {
        //Requerimientos: "persona"
        static public Object getFullName(Persona self){
                //TODO: function name generated: write your custom code
                //Please, write the code of this function as is said in the requirement
                return ("not implemented")
       }
                                                ...
```

Fichero ModelValidation.py

Respecto a la segunda función, Grails ha seguido la misma estrategia que Django, aunque el nombre de la función debe seguir la nomenclatura especial de los métodos getters & setters de java. Así, la función getNumero(), asociada al atributo derivado *numero,* se puede implementar en la propia clase, dentro del fichero *Cliente.groovy*.

```
class Cliente extends Persona {

        String estado = "-"

        //ATRIBUTOS DERIVADOS
        String getNumero(){
                //"not implemented"
                def total = "0"
                try{
                        if (this.id != null){
                                def client = Cliente.get(this.id)
                                def prestamos = Prestamo.findAllWhere(pertenece:client)
                                if(prestamos != null){
                                        total = prestamos.size().toString()
                                }
                        }
                } catch(Exception e){
                        total = "-1"
                } finally{
                        return total
                }
```

```
        }

        static transients =  ['numero']
                                                ...
```

4. Funciones de validación de los atributos

La validación de los datos enviados al servidor se realiza para garantizar la integridad de la base de datos, por lo que si existe algún error con los valores que los usuarios introducen en cada campo del formulario se debe mostrar una advertencia para que el usuario pueda rectificar.

Vamos a ver donde se ha implementado la función de validación del atributo "*codigo"* de la clase empleado en ambos entornos; django y grails.

- En Django existe un fichero generado llamado *funcAttrValidations.py* donde implementar este tipo de funciones. En el siguiente cuadro se muestra el código de validación asociado al atributo *codigo.*

```
#function name generated: write your custom code
def validate_even(value):
    "Debe ser par"
    if value % 2 != 0:
        raise ValidationError(u'%s is not an even number' % value)
```

- En Grails este tipo de validaciones se deben implementar en la propia clase generada, dentro de la estructura *constraints,* como se observa a continuación.

```
class Prestamo {

        String estado = "-"
        Integer codigo
                                        ...
        static constraints = {
                codigo(validator:{val, obj ->
                        //return ['invalidValue']
                })
                                        ...
```

Toda función de validación de un atributo declarada en PLESG será generada en cualquier entorno para que por defecto devuelva un error de validación (antes de ser implementada). En el caso de Grails, la implementación ha sido simplemente comentar la línea para que no produzca error alguno al usuario y se pueda realizar el préstamo.

5. Función de validación de los atributos de los formularios

Existe la posibilidad de incluir atributos adicionales a un formulario en la fase de diseño de formularios en PLESG.

- En Django, estos atributos aparecen en *forms.py* y las funciones de validación definidas en ellos se pueden implementar en el fichero *funcFormAttrValidations.py*.

- En el caso de Grails, se deben de implementar dentro de la estructura *constraints* del propio fichero donde se declaran, concretamente en *domain/<paquete>/<rol><clase>.groovy.*

6. Validación conjunta de los errores en la cabecera del formulario (opcional)

Es un opción exclusiva para Django. Permite modificar que los errores de validación de los atributos de un formulario se muestren (por defecto) o no en la cabecera del formulario (así es como funciona Grails). Dicho comportamiento reside en las funciones que se han generado en el recurso *funcFormValidations.py*, según se muestra en el siguiente recuadro.

```
def validation_form_<rol>_<clase>(self, cleaned_data):
 #    field = cleaned_data.get("<field_name>")
 #    if field:
 #          setErrorField(self, cleaned_data, "<field_name>", u"error text")
 #
 #          #Custom error on TOP of PAGE
 #          raise forms.ValidationError("error text")
    if self.errors.__len__() > 0:
           raise forms.ValidationError(str(self.errors))
    return
```

7. Modificar los ficheros de definición de estilos e imágenes (opcional)

Si se hace necesario modificar los estilos, imágenes, iconos, etc. para adaptarlos a los requerimientos de marketing/imagen de la empresa que requiere el nuevo software de gestión, existen ciertos recursos para dicho propósito en ambos frameworks. En la siguiente tabla vemos cuales son en función del entorno.

Recurso	Entorno	Ruta	Descripción
CSS	Django	client/css/	Hojas de estilos que permiten parametrizar los iconos, imágenes, etc. visibles en la aplicación
	Grails	web-app/css/	
imagenes	Django	client/imágenes	Recursos gráficos que pueden ser cambiados según las necesidades empresariales
	Grails	web-app/images	
plantillas	Django	templates	Ficheros con extensión html/gsp en función del framework
	Grails	views y/o src/templates	

Por último, hay que verificar que no hay errores de implementación, por ejemplo mediante el desarrollo de tests unitarios, siendo los desarrolladores los responsables de este aspecto.

4.6 Generación del modelo de persistencia

Una vez se han configurado/implementado de forma adecuada los diferentes artefactos que constituyen el proyecto, es posible generar la capa de persistencia para cada framework gracias a que estos ofrecen un comando/acción específico para ello.

Sin embargo, antes de ejecutar cualquier acción es necesario configurar el motor de base de datos a utilizar. Para el ejemplo de la gestión de los prestamos de un biblioteca se ha decidido utilizar SQLite3[25] por su sencillez a la hora de configurarlo tanto en Django como en Grails, aunque otras opciones hubieran sido igualmente válidas (MySQL, PostgreSQL, Oracle, etc.) o incluso preferibles si hablamos de aplicaciones más complejas; concurrencia, uso de tareas planificadas (oracle jobs), etc.

Adicionalmente, y tras generar la BD, se deberán de incluir los metadatos que establecen el comportamiento dinámico de la aplicación ejecutando el script *insert_SQLiteDB_metamodel.sql* (generado por cada framework) desde cualquier administrador de base de datos. El hecho de que se hayan generado diferentes scripts para cada framework es debido a que éstos presentan pequeñas diferencias en la generación del esquema, como es el caso de la nomenclatura que dan a las claves ajenas.

Otro aspecto a tener en cuenta ha sido el utilizar la misma estrategia de generación de tablas en el caso de la herencia entre clases para ambos frameworks: se generan tablas tanto para la clase padre como para las clases hijas. Esto se ha establecido directamente en los traductores, y quizás una opción de mejora sería poder configurarlo en un futuro desde algún fichero de configuración del traductor (punto 7.1).

4.6.1 Django

La configuración global de cualquier proyecto Django se realiza en el fichero *settings.py*. En él se deben especificar aspectos como:

- la configuración del motor de base de datos
- las aplicaciones instaladas
- ruta donde se han instalado las plantillas
- la zona horaria, etc…

En el siguiente cuadro podemos ver un ejemplo de configuración del motor de BD para la aplicación biblioteca.

```
                                        ...
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.sqlite3', # Add 'postgresql','mysql' or 'oracle'.
        'NAME': ROOTPATH + '/db/biblioteca.db', # path to database file using sqlite3.
        'USER': '',       # Not used with sqlite3.
        'PASSWORD': '',   # Not used with sqlite3.
```

[25] ¿Cuando usar SQLite3?: http://www.sqlite.org/whentouse.html

```
        'HOST': '',       # Set to empty string for localhost. Not used with sqlite3.
        'PORT': '',       # Set to empty string for default. Not used with sqlite3.
    }
}
```

Configuración del motor de BD en el fichero settings.py

Tras establecer dicha configuración es posible ejecutar la acción encargada de generar la base de datos con todas las tablas para cada una de las aplicaciones instaladas. Para ello y según se puede observar en la figura 4.10, se debe seleccionar la carpeta raíz del proyecto y con el botón secundario seleccionar *Django → Sync DB*. Si no existe ningún error, ya sea en la definición del modelo de dominio de la aplicación o en el fichero de configuración del proyecto, deberían de aparecer las diferentes secuencias de creación de tablas en función de las aplicaciones.

```
Plataforma: darwin - Mac OS X (10.6)
Creating tables ...
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_user_permissions
Creating table auth_user_groups
Creating table auth_user
Creating table django_content_type
Creating table django_session
Creating table django_site
Creating table django_admin_log
Creating table metamodel_app
Creating table metamodel_entity
Creating table metamodel_rol
Creating table metamodel_rolview
Creating table metamodel_service
Creating table metamodel_property
Creating table metamodel_navigation
Creating table metamodel_device
Creating table metamodel_gridproperty
Creating table biblioteca_persona
Creating table biblioteca_libro
Creating table biblioteca_cliente
Creating table biblioteca_prestamo
Creating table biblioteca_empleado

You just installed Django's auth system, which means you don't have any superusers
defined. Would you like to create one now? (yes/no): yes
Username (leave blank to use 'davifer'):
E-mail address: davifer80@gmail.com
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)
```

Tras finalizar de forma correcta la ejecución anterior, se habrá creado la base de datos correspondiente en */workflow/db/biblioteca.db*. Hay que tener en cuenta que si es la primera vez que se ejecuta dicha acción se solicitará la creación de una cuenta de administrador, con el fin de administrar la base de datos. Cualquier cambio en el modelo de dominio hace necesaria la reejecución de este comando para refrescar el esquema de base de datos.

4.6.2 Grails

La configuración global de cualquier proyecto Grails se realiza en los distintos ficheros que existen en la ruta *grails-app/conf/*. En concreto, la configuración del motor de base de datos se realiza en *conf/DataSource.groovy*. En el siguiente cuadro podemos ver un ejemplo de configuración del motor de BD en el entorno de desarrollo para la aplicación biblioteca.

```
// environment specific settings
environments {
  development {
    dataSource {
        dbCreate="update"
        url="jdbc:sqlite:/Users/workspace/biblioteca/web-app/db/dev.sqlite3"
        logSql="true"
        dialect="dialect.SQLiteDialect"   // <-- src/java/dialect/SQLiteDialect.java
        driverClassName="org.sqlite.JDBC" // <-- lib/sqlite-jdbc-3.7.2.jar
        readOnly="false"
     }
                                              ...
 }
}
```

Configuración del motor de BD en el fichero conf/DataSource.groovy

Tras establecer dicha configuración, y verificando que no existan errores de compilación, debemos de generar la base de datos. Grails ofrece dos opciones, ambas desde Eclipse o desde el terminal:

- generar el esquema de BD mediante el comando *schema-export* y ejecutarlo desde el administrador de BD correspondiente

```
local:portaMagnaGrails davifer$ grails schema-export
| Packaging Grails application.....
Generating script to
/Users/Desarrollos/workspace_portaMagna/portaMagnaGrails/target/ddl.sql
in environment 'development' for the default DataSource
```

- generar directamente la BD al poner en marcha la aplicación mediante el comando *run-app*

```
local:portaMagnaGrails davifer$ grails run-app
| Running Grails application
06-sep-2013 21:27:02 org.apache.coyote.AbstractProtocol init
INFO: Initializing ProtocolHandler ["http-bio-8080"]
06-sep-2013 21:27:02 org.apache.catalina.core.StandardService startInternal
INFO: Starting service Tomcat
06-sep-2013 21:27:02 org.apache.catalina.core.StandardEngine startInternal
INFO: Starting Servlet Engine: Apache Tomcat/7.0.39
06-sep-2013 21:27:02 org.apache.catalina.startup.ContextConfig
getDefaultWebXmlFragment
INFO: No global web.xml found
06-sep-2013 21:27:03 org.apache.catalina.core.ApplicationContext log
| Server running. Browse to http://localhost:8080/biblioteca
```

La segunda opción es más rápida, pues compila, crea la BD y arranca la aplicación, permitiendo comprobar de inmediato el funcionamiento de la aplicación web en cualquier navegador.

Así, cuando un usuario abre el navegador e introduce, por ejemplo, la siguiente URL http://localhost:<puerto>/biblioteca para acceder a la aplicación web, el servidor es capaz de detectar el tipo de dispositivo desde el cual se ha realizado la petición, gracias al plugin spring-mobile (Grails) o *MobileDetectionMiddleware (Django)*, redireccionando hacia el fichero *index.** adecuado; desktop o mobile, siempre y cuando el usuario se haya autenticado previamente en el sistema, de lo contrario le redireccionará a la página de login.

4.7 Requisitos no funcionales

Existen ciertos requisitos no funcionales que deben ser siempre tomados en cuenta a la hora de desarrollar una aplicación web. De entre todos ellos, vamos a analizar los que tienen un impacto directo según el proceso definido: seguridad, mantenibilidad y usabilidad.

Por otra parte, el uso de este tipo de frameworks de desarrollo ágil nos proporciona una infraestructura que facilita en muchos casos la configuración de la mayoría de estos requisitos, entre los cuales se encuentran la seguridad y la mantenibilidad.

4.7.1 Seguridad: proceso de autenticación

Una parte de la seguridad en toda aplicación web es el control del acceso de los usuarios a ésta. Para ello hay que tener en cuenta aspectos como el diseño de los formularios que permiten la autenticación (ver figura 4.24), la lógica de acceso (comprobar que los datos enviados son correctos) y el alta de los usuarios (persistencia del login y password). Vamos a ver como se han resuelto estos dos últimos aspectos tanto en Django como en Grails.

Para dar de alta a dichos usuarios (login y password) en la base de datos, junto con sus roles correspondiente, quizás la opción común a ambos frameworks hubiera sido crear las sentencias SQL (insert into) adecuadas y ejecutarlas en el administrador de base de datos correspondiente, pero como se verá a continuación, se ha optado por utilizar otros métodos.

Según la aplicación del ejemplo, deberemos de crear algunos usuarios y asociarles alguno de los roles (SUPER, STAFF, SOCIO y GUEST) que previamente también se deben haber dado de alta.

Django viene preconfigurado con una aplicación llamada “django.contrib.admin” que permite delegar la gestión de los usuarios/roles que se conectan a la aplicación. Debido a su facilidad de uso se ha utilizado dentro de las soluciones desarrolladas para Django (más información en https://docs.djangoproject.com/en/dev/ref/contrib/admin)

Para acceder a dicha aplicación sólo es necesario descomentarla en el fichero de configuración del proyecto (settings.py) y arrancar el proyecto (ver punto 4.7.3, apartado puesta en marcha de la aplicación) en modo *debug* (depuración). Tras ello podremos acceder bajo la URL: http://localhost:<puerto>/admin.

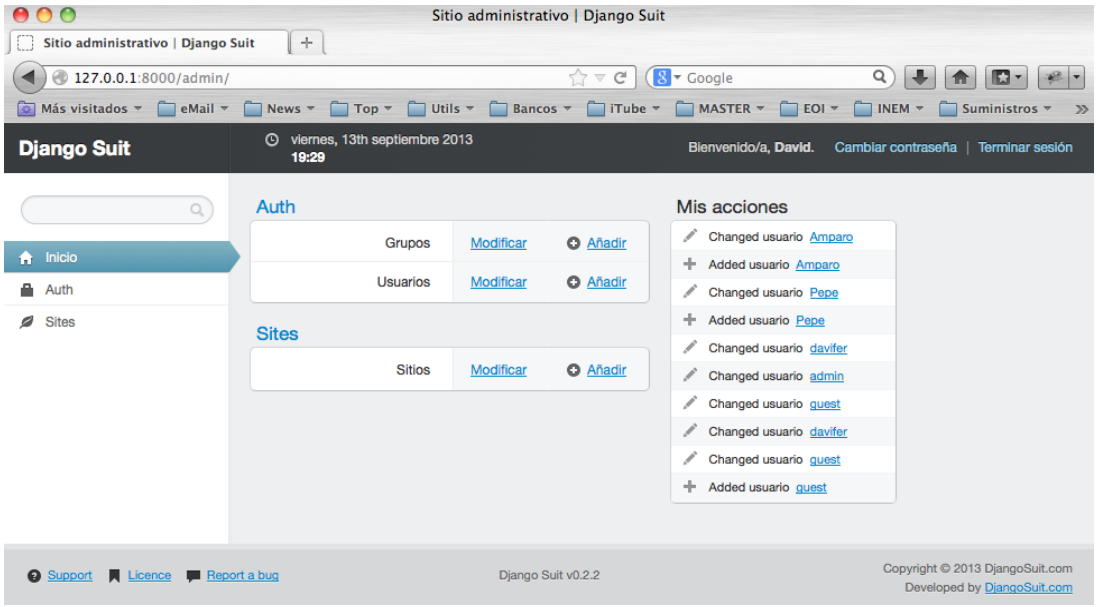


Figura 4.18: vista de la aplicación Admin para Django

Recordemos que en el caso de Django, al crear la BD se creó un súper usuario. Ese usuario tiene permisos de administrador para acceder a la aplicación "admin" y lo utilizaremos ahora para dar de alta a los usuarios que deseamos que se conecten al sistema, con los roles correspondientes.

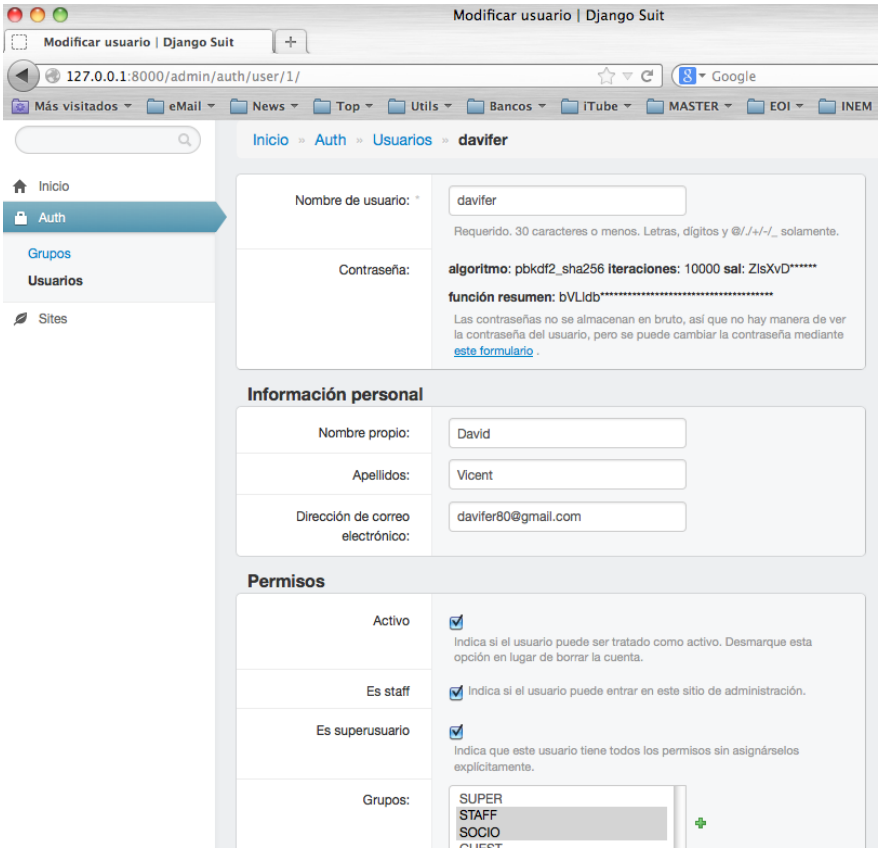


Figura 4.19: asociar un grupo/rol a un usuario desde Admin

En la imagen anterior, al usuario "davifer" se le han asociado los roles STAFF y SOCIO. También se ha creado un usuario "admin" con el rol "SUPER" que deberá dar de alta a los empleados de la biblioteca y asociarles su cuenta de acceso.

Respecto a la lógica de control de acceso, sólo es necesario colocar el *decorator*[26] @login_required encima del método *index* ubicado en el fichero view.py, que es el punto de entrada a la aplicación.

Con respecto a Grails, éste no viene preconfigurado con una aplicación similar a Django, pero sí existen plugins que implementan dicha funcionalidad (Spring Security Plugin, Authentication Plugin, etc.). Sin embargo hemos optado en este caso por no delegar en ellos la gestión de la seguridad en cuanto a la autenticación. Es por ello que en el paquete domain/metamodel existen las clases User y Rol, entre las cuales se ha implementado una asociación con cardinalidad n a n que representa los roles de un usuario y viceversa.

En este caso, el alta de los usuarios y la asignación de los roles pertinentes se ha realizado en el fichero *conf/BootStrap.groovy.* Será durante el arranque de la aplicación cuando se ejecuten las sentencias expuestas en dicho recurso, según se aprecia a continuación.

[26] https://docs.djangoproject.com/en/1.5/topics/auth/default/#the-login-required-decorator

```
class BootStrap {
    def init = { servletContext ->
                def ra = Rol.get("biblioteca_SUPER")
                def rs = Rol.get("biblioteca_STAFF")
                def rsocio = Rol.get("biblioteca_SOCIO")

                new User(username: 'admin', enabled: true, password: 'secreto')
                        .addToRols(ra)
                        .save()

                new User(username: 'davifer', enabled: true, password: 'secreto')
                        .addToRols(rs)
                        .addToRols(rsocio)
                        .save()
    }
    def destroy = {}
}
```

Respecto a la lógica de control de acceso, se ha implementado en el método doLogin() del controlador metamodel/UserController.groovy.

Mencionar que se han creado los mismos usuarios, con los mismos permisos que en Django.

4.7.2 Mantenibilidad

La facilidad del mantenimiento recae, por una parte, en la estructura de cada framework, los cuales siguen el patrón de diseño MVC, y por otra, en los programadores. Éstos deben seguir, en la medida de los posible, los criterios establecidos en el punto 4.5, y por supuesto, ciertos criterios profesionales [19].

Es el momento de realizar un análisis detallado de los aspectos clave de la interfaz de usuario (IU) para poder comprobar la usabilidad de la aplicación en función del dispositivo. Comentar que da igual bajo qué entorno estemos operando, Django o Grails, pues las diferentes interfaces de usuario (desktop/móvil) son iguales para ambos frameworks, a excepción de algunos pequeños matices en los formularios html, que son generados dinámicamente en la parte servidora.

4.7.3 Usabilidad: análisis de la interfaz de usuario

La página de inicio *index.** además de establecer la carga de los recursos necesarios, establece la estructura de toda la aplicación, la cual se ha dividido en varias regiones (norte, sur y centro). Cada región muestra cierto tipo de información relevante:

- Norte

  Se encarga de mostrar, en la parte izquierda, el usuario conectado al sistema junto con el rol que desempeña, y en la parte derecha, se muestra la acción de cerrar la sesión, que redirecciona a la página de *logout.*

- Centro

Es la región principal, donde se presenta toda la información relevante de la aplicación. Comentar que existen grandes diferencias visuales en función del dispositivo desde el cual se accede. Así, para la versión desktop, los diferentes elementos (listados / formularios) serán presentados cada uno de ellos en una pestaña propia, accesible de forma directa. En la versión móvil existen dos pestañas por defecto; la primera, que aparece activada, muestra un listado donde se visualizan todas las entidades del modelo, y la segunda pestaña, muestra información relativa sobre los derechos de autor (@copyright) y la versión de la aplicación.

- Sur (opcional)

Esta región es exclusiva para la versión desktop, y muestra el @copyright de la aplicación.

En la siguiente imagen podemos ver las regiones de la aplicación vista desde el navegador de un ordenador. Se han marcado en diferentes colores cada una de las regiones; [norte: rojo, centro:azul, sur:amarillo].

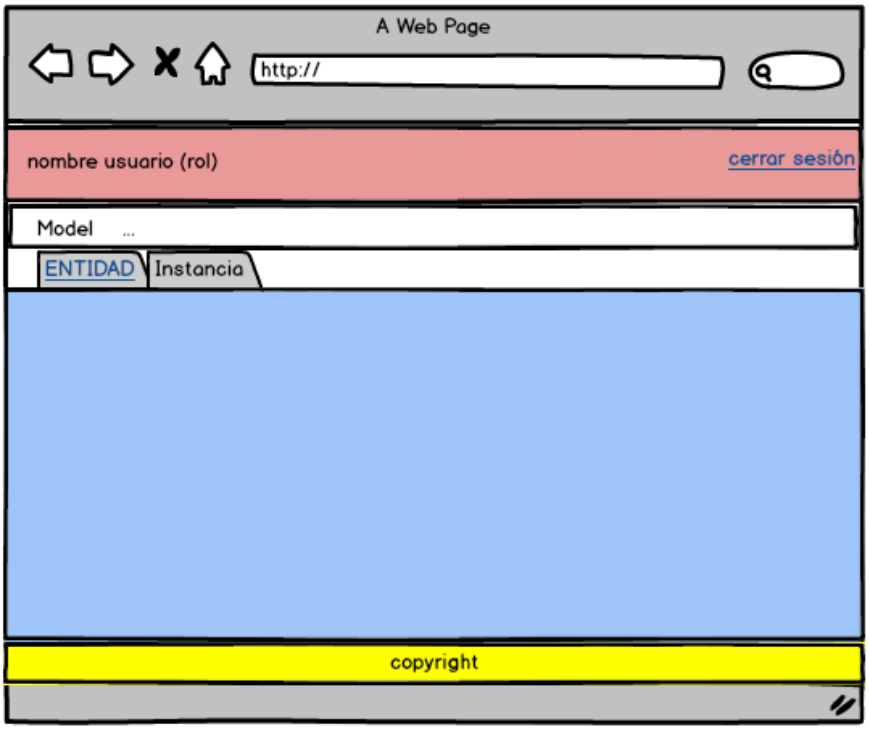


Figura 4.20: Diseño de la IU de la página de inicio (desktop)

Por el contrario, cuando la aplicación es vista desde el navegador de un dispositivo móvil ésta se divide sólo en dos regiones: norte y centro, según se aprecia en la siguiente figura.

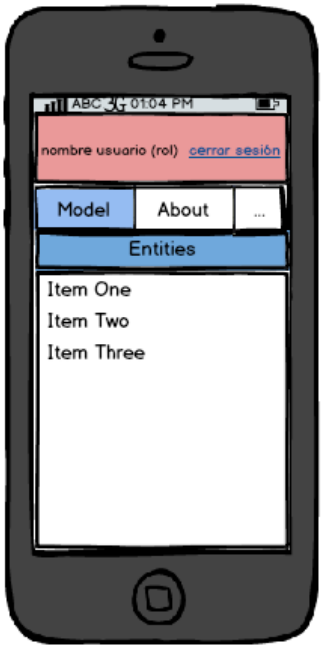


Figura 4.21: Diseño de la IU de la página de inicio (móvil)

Llegados a este punto ya podemos abrir un navegador (desktop y/o móvil) e introducir la dirección URL http://127.0.0.1:<puerto>/biblioteca con el fin de conectarnos a la aplicación para ver los aspectos clave de la interfaz de usuario (IU) y poder ejecutar parte de la funcionalidad desarrollada, por ejemplo crear/editar/borrar un préstamo.

Sin embargo, antes es necesario arrancar la aplicación bajo la plataforma tecnológica preferida (Grails/Django). Vamos a ver rápidamente como.

Puesta en marcha de la aplicación

Una vez se han realizado todos los pasos anteriores sólo nos resta explicar como poner en marcha la aplicación en cada uno de los entornos.

En Grails es tan fácil como seleccionar el proyecto desde Eclipse y con el botón secundario seleccionar *Run As → Crails Command (run-app)*, según se puede observar en la siguiente figura.

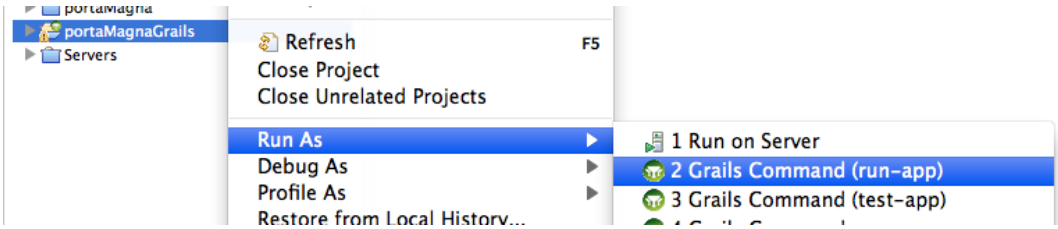


Figura 4.22: puesta en marcha de la aplicación Grails

Dicho comando lanza la aplicación en modo “developer”. Si no existen errores de compilación, configuración, etc., se pondrá en marcha el servidor de aplicaciones disponible, configurando los diferentes componentes necesarios para la aplicación y dejándola accesible desde una URL concreta,

visible en la consola de Eclipse, según podemos apreciar a continuación.

```
| Loading Grails 2.2.3
| Configuring classpath.
| Environment set to development.....
| Packaging Grails application.....
| Running Grails application14-sep-2013 11:57:33 org.apache.coyote.AbstractProtocol
init

INFO: Initializing ProtocolHandler ["http-bio-8080"]
14-sep-2013 11:57:35 org.apache.catalina.core.StandardService startInternal
INFO: Starting service Tomcat
14-sep-2013 11:57:35 org.apache.catalina.core.StandardEngine startInternal
INFO: Starting Servlet Engine: Apache Tomcat/7.0.39
14-sep-2013 11:57:35 org.apache.catalina.startup.ContextConfig
getDefaultWebXmlFragment
INFO: No global web.xml found
14-sep-2013 11:57:36 org.apache.catalina.core.ApplicationContext log
INFO: Initializing Spring root WebApplicationContext

Hibernate: select rol0_.pk as pk3_0_, rol0_.app_id as app2_3_0_, rol0_.name as
name3_0_ from metamodel_rol rol0_ where rol0_.pk=?
Hibernate: select this_.id as id0_0_, this_.version as version0_0_,
this_.account_expired as account3_0_0_, this_.account_locked as account4_0_0_,
this_.enabled as enabled0_0_, this_.password as password0_0_, this_.password_expired
as password7_0_0_, this_.username as username0_0_ from metamodel_user this_

| Server running. Browse to http://localhost:8080/biblioteca
```

En Django es igual de fácil. Seleccionamos el proyecto desde Eclipse y con el botón secundario seleccionar *Run As → PyDev:Django*, según se puede observar en la siguiente figura.

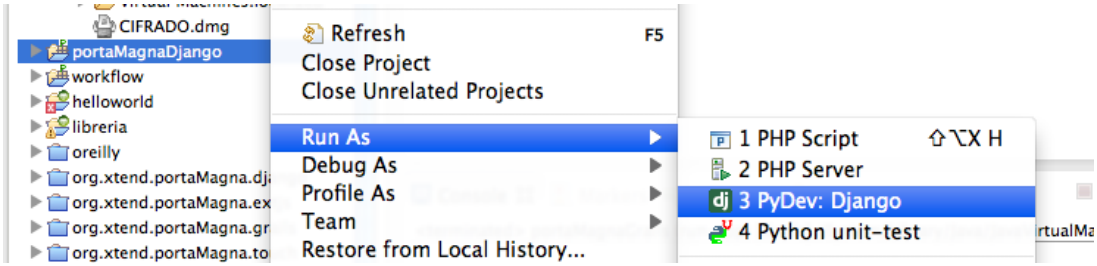


Figura 4.23: puesta en marcha de la aplicación en Django

Al igual que en Grails, tras lanzar dicho comando, y si no existen errores de compilación o similares, se pondrá en marcha el servidor de aplicaciones disponible, configurando la aplicación desarrollada y dejándola accesible desde una URL concreta, visible también en la consola de Eclipse según podemos apreciar a continuación.

```
Plataforma: darwin - Mac OS X (10.6)
Validating models...
0 errors found
Django version 1.4.1, using settings 'portaMagnaDjango.settings'
Development server is running at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Ahora sí, ya podemos introducir la URL anterior en un navegador (desktop/móvil). La aplicación nos redireccionará a la página de

autenticación, tal y como se muestra en la siguiente figura.

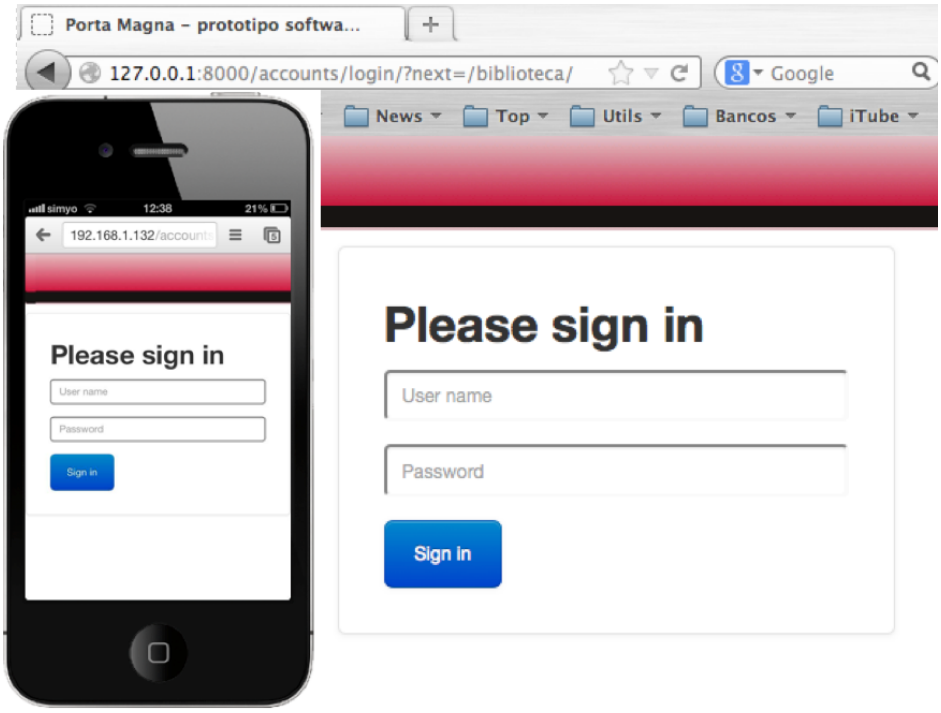


Figura 4.24: página de autenticación (desktop/móvil)

Para acceder a la aplicación desde el navegador de un dispositivo móvil se ha tenido que instalar la aplicación en un servidor de aplicaciones (ver anexo 8).

Tras autenticarnos correctamente con el usuario “davifer”, recordemos que ese usuario tiene asociados dos roles diferentes, nos aparecerá un cuadro de dialogo para que decidamos con qué rol entrar en la aplicación según podemos ver en la siguiente figura.

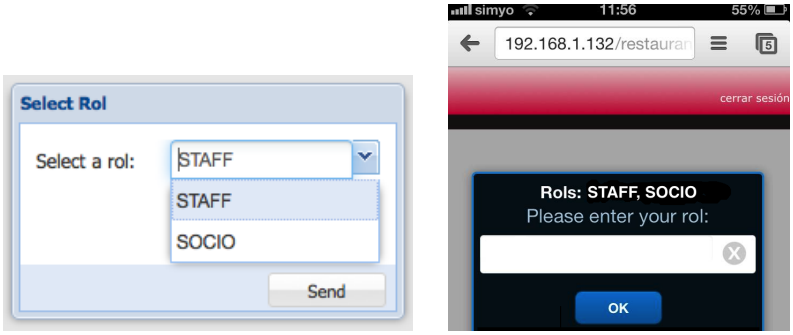


Figura 4.25: cuadro de diálogo para la selección del rol (desktop/móvil)

Llegados a este punto empezaremos a ver notables diferencias en las interfaces gráficas en función del dispositivo. La primera de ellas es el modo de elegir el rol. Esto es debido a ciertas restricciones que impone el framework Sencha Touch, donde por ejemplo no es posible utilizar el componente visual “combobox” o lista desplegable en una ventana modal. Por lo tanto, para la versión móvil se ha decidido que el usuario deba escribir el rol deseado.

Tras seleccionar/escribir el rol STAFF y pulsar el botón correspondiente, aparecerá la página de inicio, donde podremos observar las diferentes regiones en función del dispositivo.

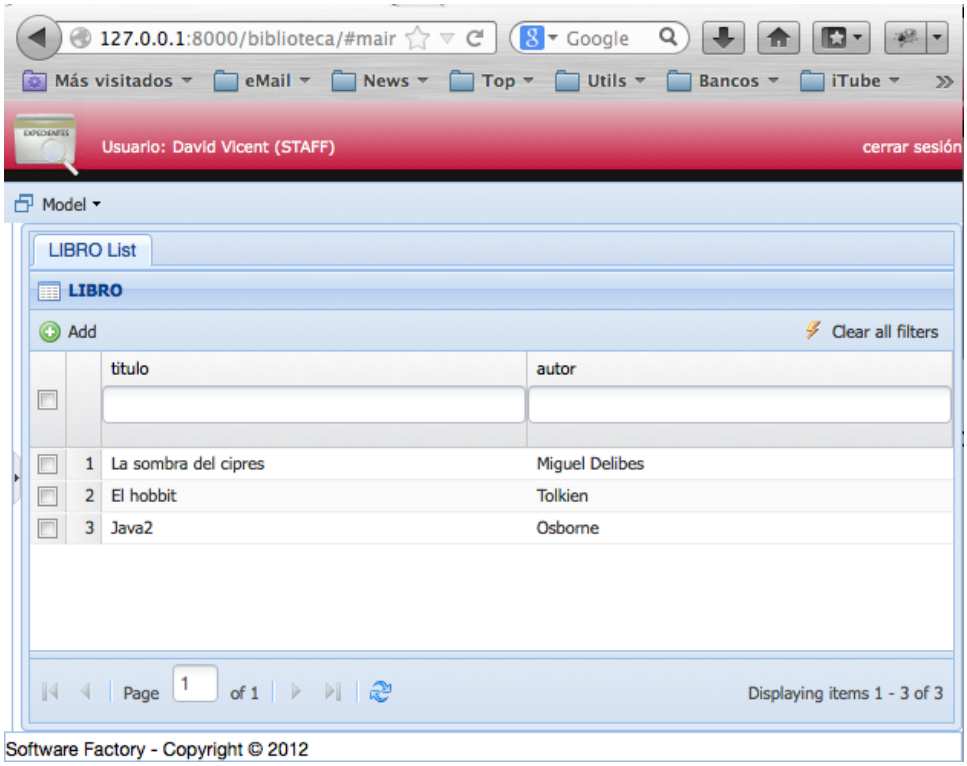


Figura 4.26: página de inicio en función del rol (desktop)

Según se definió en la especificación PLESG, al rol STAFF se le asoció por defecto la entidad libro. Es por ese motivo que en la pantalla de inicio de la aplicación para la versión desktop aparece automáticamente una pestaña con el grid "listado de libros" para dicho rol. La versión móvil difiere sustancialmente y muestra el conjunto de entidades disponibles según el rol.

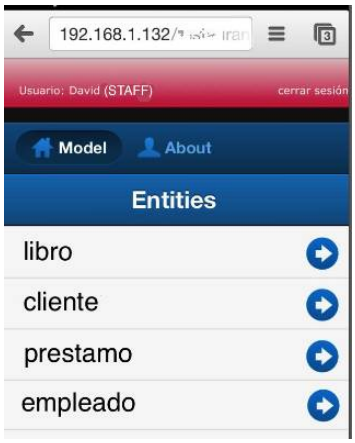


Figura 4.27: página de inicio en función del rol (móvil)

Dadas las diferencias entre ambas interfaces, a partir de ahora vamos a realizar un análisis de la interfaz de usuario por separado. Primero analizaremos la versión desktop y más adelante veremos la IU para la versión móvil.

Versión desktop

Primeramente vamos a comentar como van a mostrarse los dos componentes visuales que se definen en la especificación PLESG; el grid/listado y el formulario.

Recordemos que un grid o listado es un componente visual que agrupa a todas las instancias de una misma clase. Por defecto éste se ha configurado para que sólo muestre 20 instancias como máximo, es decir, la información esta paginada en grupos de 20 elementos. Este mecanismo evita recuperar de una sola vez toda la información relativa a una tabla de la BD, lo que provocaría la caída del sistema cuando hubieran miles de instancias almacenadas.

Por otra parte, un formulario es un componente visual que permite mostrar la información relativa a una instancia en concreto de una clase determinada.

Ambos componentes visuales se muestran siempre dentro de otro componente visual llamado *tab* o pestaña. Un tab sólo podrá cerrarse en el caso de contener un formulario. Además dicho componente (tab) puede tener dos barras de acciones, una en la parte superior y otra en la parte inferior. Sin embargo los formularios que se hayan declarado como modales serán mostrados dentro de un componente visual llamado ventana modal.

Llegados a este punto, surge la siguiente pregunta ¿qué acciones podemos realizar desde la página de inicio?. Vamos a enumerar el conjunto total de acciones disponibles, visibles en la interfaz de usuario:

1. Acceder a otros listados
Desde el menú general, ubicado debajo de la región norte, existe una opción llamada “Model”, que al seleccionarlo mostrará el resto de entidades sobre las que tiene acceso el rol conectado al sistema. Seleccionando una de ellas se abrirá una nueva pestaña que contendrá el nuevo grid de la clase seleccionada con 20 instancias como máximo.

2. Recuperar más instancias
Es posible recuperar el resto de instancias de una clase, y siempre de 20 en 20, mediante las acciones disponibles en la barra de paginación del grid, según podemos ver en la siguiente imagen.

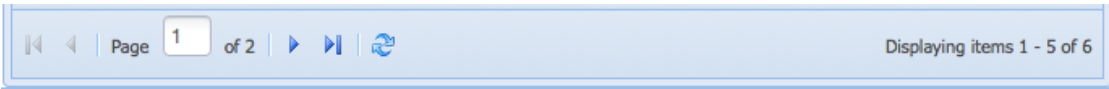


Figura 4.28: acciones disponibles en la barra de paginación del grid (desktop)

Vamos a explicar de forma más detallada las acciones que se ven en la figura anterior:

Icono	Significado
	Recuperar los siguientes 20 registros
	Avanzar hasta los últimos registros
	Retroceder 20 registros
	Recuperar los 20 primeros registros como máximo
	Refrescar la información. Muy importante cuando se ha eliminado/creado algún registro

3. Crear/Editar una instancia de una entidad

Es posible crear nuevas instancias de una entidad siempre que el rol tenga visibilidad sobre el método de tipo "C" (Create) de la clase en cuestión. De ser así, aparecerá el botón 

en la barra de acciones superior del tab del listado. Sin embargo para acceder a un instancia ya existente solo hay que hacer doble-clic sobre una de las filas que aparecen en el grid. En ambos casos la acción resultante en la misma, se abrirá un nuevo tab dentro del cual aparecerá un formulario. Si estamos ante la creación de una nueva instancia, esa información deberá ser completada y enviada al servidor y en caso de ser una instancia existente se recuperará toda la información relativa al elemento seleccionado desde el servidor.

Las acciones que se puedan realizar sobre el formulario aparecerán en la barra de acciones superior del tab. Estas serán el resultado de la intersección entre las acciones/transiciones definidas en el diagrama de transición de estados asociado a la clase y los servicios de la clase sobre los que el rol tenga visibilidad. Por ejemplo, para el caso de un préstamo nuevo, solo se puede ejecutar la acción prestar y si se edita uno existente, se podrá cancelar, eliminando dicho préstamo y cambiando el estado del libro y del cliente, siempre que sólo tuviera dicho préstamo.

4. Filtrado rápido de la información visible en el grid

Existen dos tipos de filtrado, excluyentes entre si, y disponibles siempre que se haya definido la propiedad "filter" a cierto en la especificación PLESG mediante la regla *listView*.

El primer filtro es de acceso rápido, pues aparece directamente debajo del nombre de la columna a filtrar, y permite hacer búsquedas exactas de la información. Adicionalmente también se utiliza para filtrar información en las navegaciones entre clases (ir al punto 7 de este apartado). En la siguiente imagen podemos observar este tipo de filtro rápido.

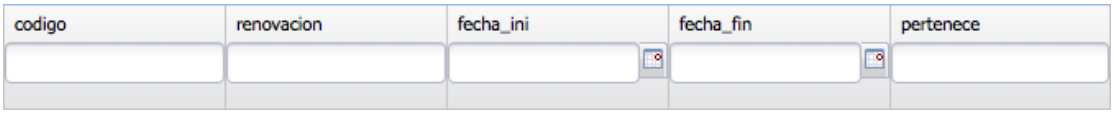


Figura 4.29: filtro rápido para realizar búsquedas exactas (desktop)

Para activar dicho filtro se debe de introducir el valor en el cuadro de texto y pulsar la tecla *Enter*. El segundo filtro se comenta al final del siguiente punto.

5. Conjunto de acciones disponibles para una columna en un grid

Las siguientes acciones son visibles pulsando sobre la flecha que aparece al lado del nombre de la columna, según se aprecia en la siguiente figura.

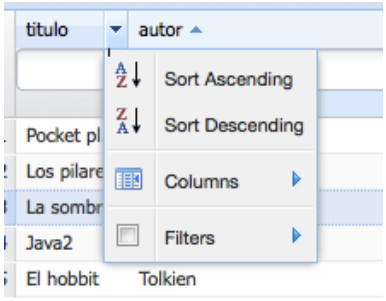


Figura 4.29: acciones disponibles para una columna del grid (desktop)

En total podemos:

a. Ordenación Asc/Des

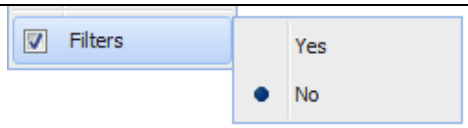
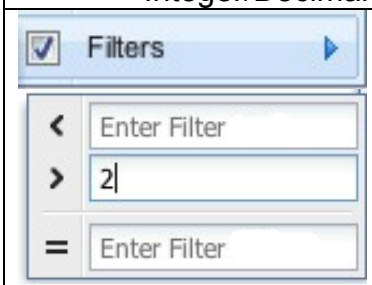
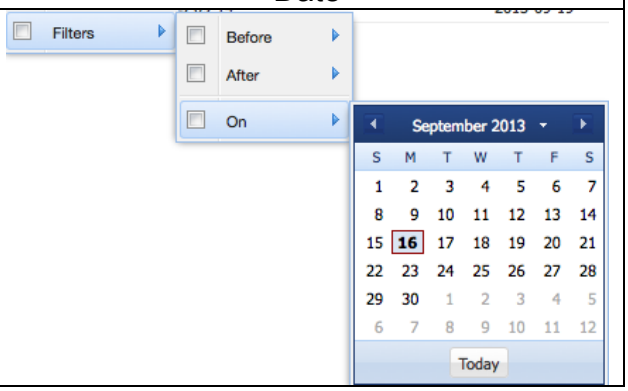
Permite ordenar la información disponible en el grid de manera ascendente/descendente.

b. Mostrar/Ocultar columnas

Mediante la acción "Columns" es posible mostrar/ocultar las columnas; atributos, atributos enumerados y/o relaciones visibles para el rol conectado, pero sólo en la sesión actual.

c. Filtrar la información a mostrar en el grid

Mediante la acción "Filters" es posible realizar filtros más complejos en función del tipo de datos de la columna, es decir, cada columna tiene su propio filtro en función del tipo de dato que sea, según podemos apreciar en la siguiente tabla.

String	Boolean
	
Integer/Decimal	**Date**
	

Sin embargo, las acciones a) y c), deben ser declaradas explícitamente mediante la propiedad 'sortable' y 'filter' de la regla *listView* (en la especificación PLESG) para que estén habilitadas.

6. Borrar todos los filtros establecidos en el grid
Independientemente de qué tipo de filtro se utilice en un listado, es posible dehacerlos todos mediante la acción "clear all filters" ubicada en la parte derecha de la barra de acciones superior del tab.

7. Navegaciones entre clases
Las navegaciones entre clases aparecen en el grid, concretamente en la barra de acciones inferior del tab. Este tipo de acciones permite navegar desde una clase origen a otra destino mediante el uso (o no) de filtros, siempre que se hayan establecido mediante la propiedad 'filter' que existe en la regla *declaredRolNavigation* de la especificación PLESG. Por ejemplo se ha creado un filtro para navegar desde la clase cliente a la clase préstamo para mostrar sólo los prestamos del cliente seleccionado.

Versión móvil

Está claro que van a existir unas diferencias significativas en la IU para la versión móvil debido a las dimensiones reducidas de la pantalla del dispositivo.

Al igual que para la versión desktop, ahora vamos a comentar también como van a mostrarse los dos componentes visuales que se definen en la especificación PLESG; el grid/listado y el formulario. Sin embargo, para comprender mejor el diseño gráfico establecido para la versión móvil, vamos a conocer previamente el diseño navegacional establecido (figura 4.30).

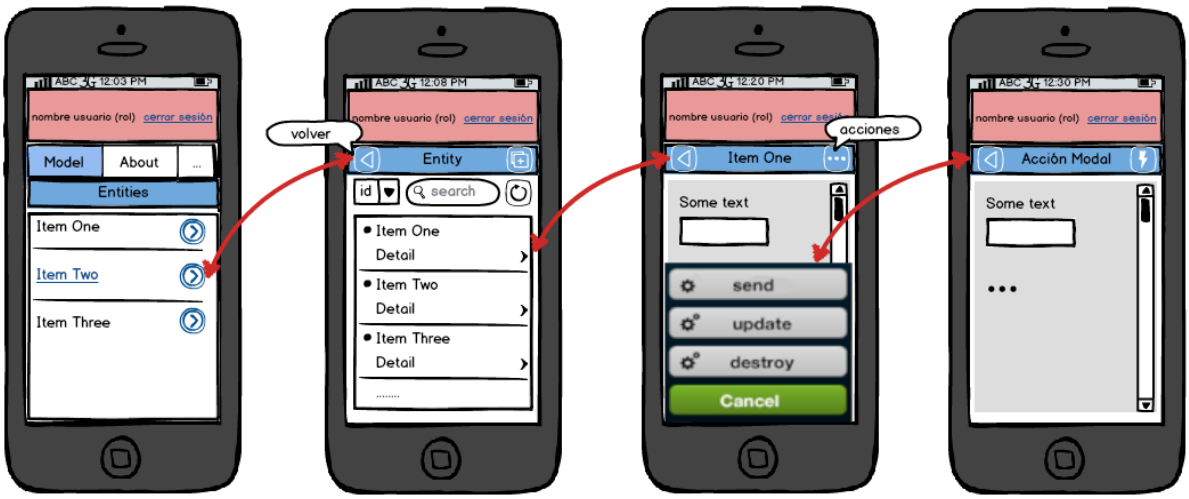


Figura 4.30: diseño navegacional (móvil)

Podemos ver que como máximo podemos navegar de manera secuencial hasta cuatro niveles de profundidad.

El primer componente visual que aparece en la pantalla inicial (nivel 1) se llama *TabPanel*. Este componente se divide en diferentes *tabs* o pestañas, el cual es posible extenderlo de forma manual. Inicialmente el *TabPanel* contiene dos *tabs*; *Model* y *About*.

La pestaña Model, contiene a su vez un componente visual llamado *ListContainer* dentro del cual se incluye un grid/listado, que en este caso muestra el conjunto de entidades accesibles para el rol. Esta pestaña se muestra por defecto al iniciar la aplicación.

Eligiendo una entidad pasamos al segundo nivel, donde aparece otro grid o listado dentro de otro componente *ListContainer*. Este grid muestra las instancias concretas para una clase determinada. Más adelante detallaremos el conjunto de acciones posibles en cada nivel de profundidad.

Seleccionando un ítem o instancia de una clase, pasamos al tercer nivel, donde se muestra la información relativa a un objeto concreto dentro de un componente visual llamado *Panel*. Si alguno de los servicios definidos para dicho objeto es de tipo modal, este se abrirá en otro *Panel*, llegando al último nivel de profundidad.

Por lo tanto, un grid/listado se muestra siempre dentro de un *ListContainer,* mientras que un formulario se muestra siempre dentro de un *Panel.* Vamos ahora a enumerar el conjunto total de acciones disponibles en la interfaz de usuario según el nivel de profundidad en la navegación:

Nivel 1

1. Acceder a una entidad
   Desde la pestaña Model, seleccionando alguna de las entidades que se muestran, pasamos al segundo nivel donde aparecerá un

grid/listado con 5 instancias como máximo de la clase seleccionada. El grid que aparece en este nivel también tiene paginación, pero ésta es incremental.

2. Ver la información sobre la aplicación
Pulsando sobre el tab *About* es posible ver la información relativa a la aplicación; versión, copyright, etc.

Nivel 2

3. Crear/Editar una instancia de una entidad
Es posible crear nuevas instancias sobre una entidad siempre que el rol tenga visibilidad sobre el método de tipo "C" (Create) de la clase en cuestión. De ser así aparecerá el botón en la barra de acciones superior del *ListContainer*. Sin embargo para acceder a un instancia ya existente solo hay que hacer clic sobre una de las filas que aparecen en el grid. En ambos casos la acción resultante en la misma, se pasará al tercer nivel donde se mostrará un *Panel* dentro del cual aparecerá un formulario. Si estamos ante la creación de una nueva instancia, esa información deberá ser completada y enviada al servidor y en caso de ser una instancia existente se recuperará toda la información relativa a la instancia seleccionada desde el servidor. Por ejemplo, para el caso de un préstamo nuevo, solo se puede ejecutar la acción prestar y si se edita uno existente, se podrá cancelar, eliminando dicho préstamo y cambiando el estado del libro y del cliente, siempre que sólo tuviera dicho préstamo.

4. Conjunto de acciones sobre el grid
El conjunto total de acciones a realizar sobre un grid se sitúan en una barra propia, según se puede apreciar en la figura 4.31:

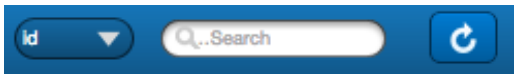


Figura 4.31: filtrar la información en un listado (móvil)

Desde esta barra se puede realizar:

a. Recuperar más instancias
Es posible recuperar más instancias pulsando el botón . Como se comentó en el primer punto, este grid también está paginado, pero a diferencia del grid para la versión desktop, esta paginación es incremental.

b. Filtrado rápido de la información visible en el grid
El filtro se aplica inmediatamente tras seleccionar la propiedad sobre la que aplicar el filtro e introducir su valor en el campo de búsqueda que aparece a la derecha de este.

c. Borrar el filtro establecido en el grid
Una vez introducido un valor en el campo de búsqueda, este puede ser eliminado pulsando el botón que aparece en la

parte derecha del texto introducido. Tras esta acción se recuperarán los registros acumulados previamente.

En comparación con la versión desktop, la ordenación viene dada por el id de la entidad, no es posible mostrar/ocultar columnas y existe sólo un tipo de filtro para realizar búsquedas rápidas, el cual aparece siempre (ver punto 7.1).

5. Navegaciones entre clases
Funcionan exactamente igual que en la versión desktop, solo que el conjunto de navegaciones aparecen, si han sido declaradas, tras pulsar el botón  , ubicado en la barra de acciones superior del *ListContainer.*

Nivel 3

6. Acciones propias del formulario
Las acciones a realizar sobre el formulario aparecerán seleccionando el botón  ubicado en la barra de acciones superior del *ListContainer*. Estas son el resultado de obtener la intersección entre las acciones/transiciones definidas en el diagrama de transición de estados asociado a la clase y los servicios de la clase sobre los que el rol tenga visibilidad. Si se ha declarado alguna acción modal, su ejecución será delegada en un segundo *Panel,* pasando al cuarto nivel.

Nivel 4

7. Ejecución de la acción modal
Estamos ante un formulario *custom* donde por lo general se habrá declarado algún atributo exclusivo para ese servicio, por ejemplo, en el servicio de sancionar a un cliente, podría haberse declarado un check que en el caso de marcarse implicaría avisar al usuario por email.

A partir del primer nivel aparece siempre la acción “volver” hacia atrás  , ubicada a la izquierda de la barra de acciones superior del *ListContainer* y del *Panel.*

En el capítulo 6 pondremos a prueba PLESG con un ejemplo real (gestión de los datos médicos y radiografías de una clínica dental). El cumplimiento de parte de algunos requisitos no funcionales directamente relacionados con éste proceso, tales como la usabilidad, seguridad (autenticación), etc. quedarán validados por la satisfacción de los usuarios de dicha aplicación.

Vamos a ver ahora los pasos requeridos para llevar a cabo la implementación del prototipo.

Capítulo 5 - Construcción del prototipo

En este capítulo veremos todos los pasos necesarios que se han realizado para llevar a cabo la construcción del prototipo PLESG y así poder cumplir con el objetivo general del proyecto:

1. Nuevo proyecto Xtext
Xtext permite crear, mediante un asistente, un entorno inicial formado por varios proyectos. Los dos más importantes son:
   - El proyecto del lenguaje
   Es el proyecto principal donde se debe especificar, además de la gramática vista en el punto 4.2.2, lo siguiente:
     - Linking (referencias)
     - Implementación de restricciones en los modelos
     - Generación de los artefactos del lenguaje (DSL)
   - El proyecto de interfaz de usuario (IU)
   Permite especificar:
     - Implementación de un menú contextual para invocar a los traductores
     - Configuración adicional del wizard

2. Implementación de los traductores
Éstos son los encargados de realizar la generación de los artefactos software a partir de la especificación PLESG para los diversos frameworks de desarrollo ágil. En total se han implementado dos traductores para la parte servidora (Django y Grails) y dos traductores para la parte cliente (Sencha: desktop y/o móvil), empleando Xtend como herramienta para la construcción de dichos traductores.

3. Configuración de la especificación PLESG
Es necesario abordar una serie de pasos adicionales antes de poder realizar la especificación de los modelos del análisis vistos en el punto 4.1.

5.1 Nuevo proyecto Xtext

Una vez se ha instalado[27] Eclipse con Xtext[28] (versión 2.3.1), es posible crear un nuevo proyecto Xtext mediante la utilización de un wizard. Para ello vamos a File → New → Project, y seleccionamos Xtext Project.

[27] Existen dos opciones: Eclipse preconfigurado con Xtext o instalar Xtext en Eclipse. http://www.eclipse.org/Xtext/download.html
[28] Xtext 2.4.x supone realizar cambios en la gramática relativos al "importedNamespace"

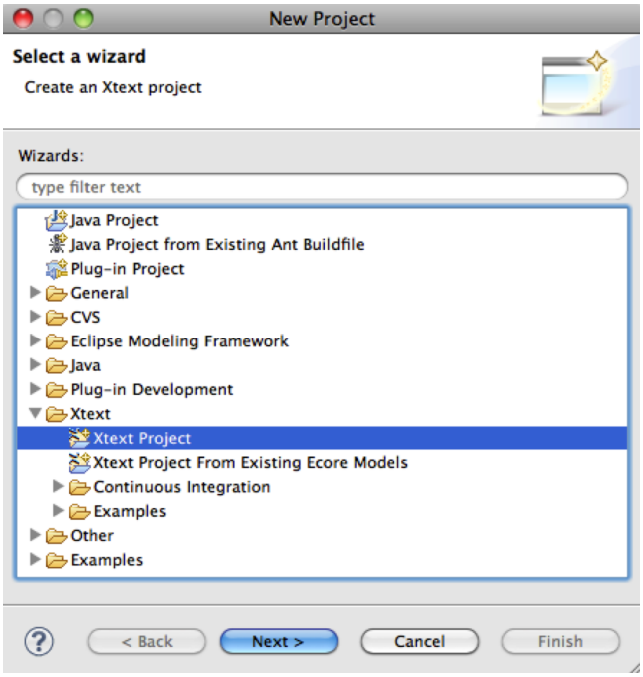


Figura 5.1: crear un nuevo proyecto Xtext en Eclipse

Tras pulsar *Next* aparece un asistente que nos pide cierta información necesaria para crear el nuevo lenguaje específico de dominio (DSL).

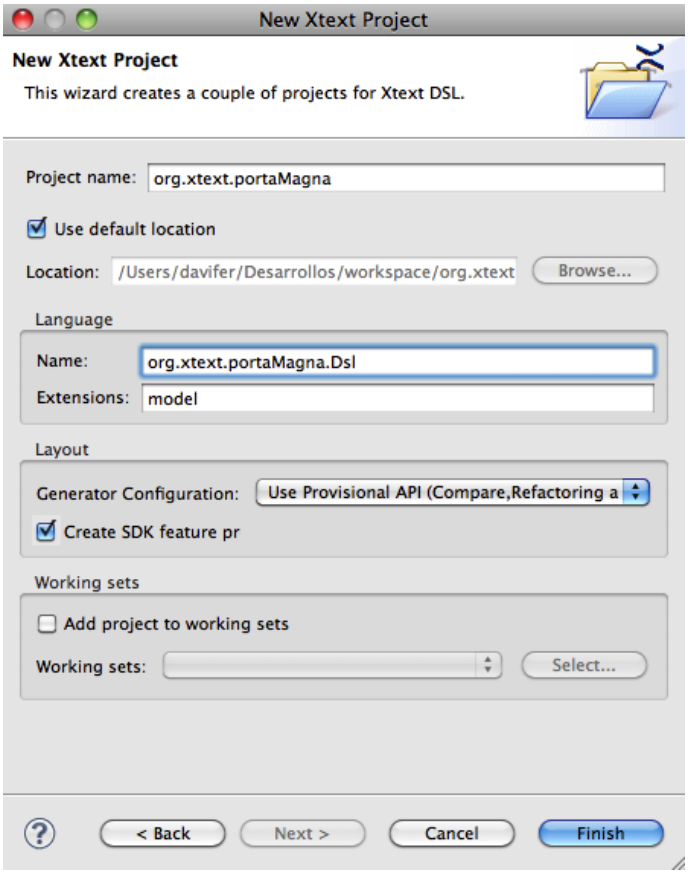


Figura 5.2: nombre del proyecto Xtext y del DSL

Más concretamente, y según se ve en la figura de arriba, el asistente nos pregunta principalmente por:

- Project Name: org.xtext.portaMagna[29]

  Es el nombre del proyecto Xtext donde se definirá la gramática (vista en el punto 4.2.2) que dará lugar al nuevo lenguaje

- Language Name: org.xtext.portaMagna.Dsl

  Es el nombre del nuevo lenguaje.

- Language Extensions: model

  La extensión de los archivos que trabajan con el lenguaje en el editor. Dichos archivos representarán los distintos modelos del sistema software a especificar.

Adicionalmente es posible modificar la ruta donde ubicar el proyecto, así como crear un proyecto adicional (SDK feature project) por si se desean establecer características adicionales al proyecto.

Tras pulsar *Finish,* el asistente genera por defecto cuatro proyectos en el workspace, visibles desde el *Package Explorer.*

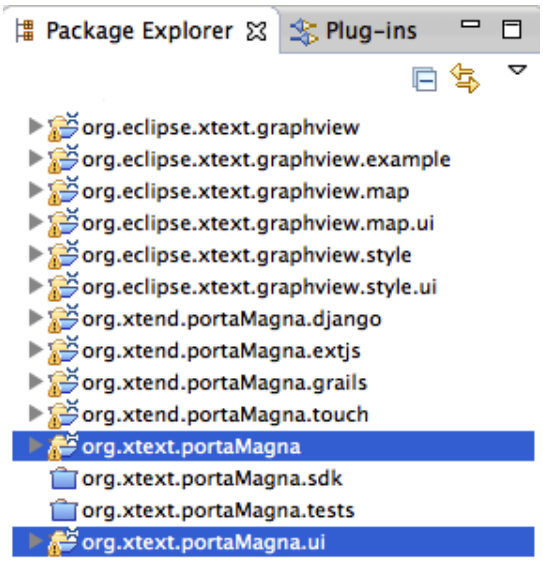


Figura 5.3: proyecto Xtext y proyecto de la IU

En la siguiente tabla podemos ver una descripción de los cuatro proyectos.

Proyecto	Descripción
org.xtext.portaMagna	Contiene la definición de la gramática y todos los componentes “runtime” (parser, lexer, validación, etc.)
org.xtext.portaMagna.sdk	Posibilidad de establecer características adicionales al proyecto (No contemplado)

[29] Porta Magna: puerta principal del Arsenal de Venecia, donde se puso en marcha en el s.XVI la primera línea de ensamblado. http://es.wikipedia.org/wiki/Arsenal_de_Venecia

org.xtext.portaMagna.tests	Unit tests (No contemplado)
org.xtext.portaMagna.ui	Permite configurar el editor de Eclipse y todas las demás funcionalidades asociadas al workbench

Los dos proyectos más importantes y que vamos a analizar a continuación son:

- org.xtext.portaMagna → Proyecto del lenguaje
- org.xtext.portaMagna.ui → Proyecto de la interfaz de usuario

Así pues, los proyectos con extensión *.sdk y *.tests, generados por defecto, se han cerrado pues no son utilizados en el proceso de construcción del prototipo PLESG.

5.1.1 Proyecto del lenguaje

Sin duda alguna el aspecto más importante dentro del proyecto del lenguaje es la definición de la gramática, vista en el punto 4.2, la cual debe ser especificada en el fichero *src/org.xtext.portaMagna/Dsl.xtext* del presente proyecto (gramática completa en el anexo 2).

Vamos a ver con más detalle el resto de los puntos definidos al principio del capítulo para este apartado, algunos de los cuales son opcionales:

- Linking (o referencias)
- Implementación de restricciones (opcional)
- Generación de los artefactos del lenguaje

5.1.1.1 Linking

Los distintos modelos del Análisis (objetual, dinámico, funcional y de presentación) permiten obtener una visión completa del sistema. En este contexto, se hace necesario crear “links” entre las distintas reglas de la gramática, por ejemplo, cuando un tipo de datos referencia a otro a través de su identificador.

Un diferencia entre Xtext y otros *parsers*, es que éste no solo proporciona mecanismos para describir la sintaxis del lenguaje de modelado, sino que además implementa aspectos como “linking”. La característica “linking” permite especificar referencias cruzadas dentro de una gramática Xtext (ver apartado 7.2 sobre incidencias, punto 1). Para ello se necesita:

- declarar un crosslink en la gramática (referencia cruzada), especificando entre corchetes la referencia en cuestión.

- especificar la semántica de vinculación (contexto), es decir, definir qué elementos son atribuibles por una cierta referencia. Implementado mediante un mecanismo llamado “scoping”.

Es muy útil permitir que un tipo de datos referencie a otro a través de su ID (mecanismo por defecto). Un ejemplo de crosslink se puede observar en la siguiente regla, donde una entidad se identifica en función de algunos de los atributos que se hayan definido en ella.

```
Attribute:
      type=[DataType] name=ID '(' parameters+=attrParameter (COMMA
                                                    parameters+=attrParameter)* ')'
;

Entity:
  (abstract?='abstract')? 'entity' name=ID ('extends' superType=extendedClass)? '{
               (attributes+=Attribute)*
               (derivatives+=AttrDerived)*
               (enumerators+=AttrEnum)*
               (relations+=Relation)*
               (services+=Service)*

               //ie: es el metodo toString() para una clase de Java.
               //Por defecto el ID (se utiliza como referencia desde otras entidades)
               (identity?='identity' ':' '{'
                     attribute+=[Attribute]  (SEP attribute+=[Attribute])*
               '}')?
                                  ...
;
```

Sin embargo, esta simple declaración de crosslink no dice nada sobre dónde encontrar los atributos, ya que estos pueden ser no sólo de la propia clase sino también de la clase padre, y así sucesivamente. Por lo tanto, es necesario implementar el mecanismo conocido como Scoping[30]. Esto se hace en el paquete src/*.scoping del proyecto, dentro de la clase *DslScopeProvider.java*.

Para el caso analizado, a continuación se muestra el método implementado, que devuelve una lista con todos los atributos de la propia clase y de su árbol de herencia.

```
IScope scope_Entity_attribute(Entity context, EReference reference) {
       List<EObject> crossRefTargets = new ArrayList<EObject>();
       getAllScopeEntityElements(context, "Attribute", crossRefTargets);
       return Scopes.scopeFor(crossRefTargets);
}
```

La función *getAllScopeEntityElements* es recursiva y para esa llamada en concreto concatena en la lista *crossRefTargets* todos los atributos que encuentra en el árbol de herencia, haciendo una búsqueda *bottom – up*, desde la clase pasada por referencia.

En algunos contextos es posible querer hacer una referencia cruzada al elemento EObject de Ecore para, en tiempo de ejecución, poder obtener los tipos de datos deseados y posiblemente de diferente tipo cada uno.

[30] Scoping API: http://www.eclipse.org/Xtext/documentation.html#scoping

Veamos un ejemplo, donde queremos especificar las propiedades; atributos, atributos derivados (solo STDField), enumerados y relaciones que deben de aparecer en un grid/listado para una determinada clase. Queremos que *ref* haga referencia, por tanto, a Attributes, AttrDerived, AttrEnum y Relations:

```
declaredListViewProperty: '{'
	'property' ':' ref=[EObject] SEP
	'sortable' ':' sortable=BOOL SEP
	'filter' ':' filter=BOOL
'}';
```

Al igual que antes, se ha hecho necesario utilizar el mecanismo de scoping, pero ahora *crossRefTargets* concatena los cuatro tipos de datos diferentes.

```
IScope scope_declaredListViewProperty_ref(declaredListViewProperty context,
EReference eRef) {
	EObject parent = (EObject) context.eContainer();
	...
	getAllScopeEntityElements(refEntity, "Attribute", crossRefTargets);
	getAllScopeEntityElements(refEntity, "AttrDerived", crossRefTargets);
	getAllScopeEntityElements(refEntity, "AttrEnum", crossRefTargets);
	getAllScopeEntityElements(refEntity, "Relation", crossRefTargets);
	return Scopes.scopeFor(crossRefTargets);
}
```

5.1.1.2 Implementación de restricciones

La sintaxis abstracta de un lenguaje especifica su estructura, es decir, las construcciones, propiedades y conectores que puede tener dicho lenguaje. Generalmente también se deben de especificar las reglas del lenguaje en el metamodelo para así evitar la mala práctica de tener que validar los modelos en los generadores de código. Cuanto antes se detecten anomalías, más sencilla será la tarea de los demás componentes.

El lenguaje Check se utiliza para establecer restricciones en los modelos. Las restricciones son reglas (expresiones booleanas) que los modelos deben cumplir para ser válidos.

Hay que tener en cuenta que el lenguaje Check se puede utilizar dentro de Xtext o en editores de modelado externos. Dentro de Xtext, la implementación de las distintas restricciones se hace en la clase <nombre_dsl>JavaValidator.java ubicada en el paquete src/*.validation.

Por ejemplo, se ha implementado (anexo 5) una restricción que imposibilita incluir dos veces un mismo parámetro en relaciones, servicios, atributos, atributos enumerados y atributos derivados. Otra restricción implementada especifica los parámetros que no son válidos en la definición de un atributo derivado.

Otra restricción a contemplar es no permitir establecer parámetros para atributos cuyo tipo de dato no sea el adecuado, por ejemplo, no es correcto

incluir el parámetro *decimal_places* cuando el parámetro es de tipo String. Sin embargo en esta primera versión se opta por documentar todas la posibles combinaciones de dichos parámetros en función del tipo de dato, de modo que los analistas funcionales puedan establecer siempre modelos válidos (ver punto 7.1).

5.1.1.3 Generación de los artefactos del lenguaje

Una vez se ha definido la gramática, es necesario ejecutar el generador Xtext (Xtext Generator) para derivar de ese modo los distintos artefactos del lenguaje (figura 5.4) disponibles en tiempo de ejecución:

- asistente para la creación de proyectos PLESG (opcional)
- un generador de código (descartado)
- parser (analizador sintáctico) para procesar el DSL, y así poder determinar si una sentencia o palabra escrita en el modelo pertenece al dominio PLESG.
- el metamodelo Ecore del DSL, que describe la estructura del Abstract Syntax Tree (AST)
- el Entorno de Desarrollo Integrado (IDE, de sus siglas en ingles) sobre el cual trabajarán los analistas funcionales o expertos del dominio.

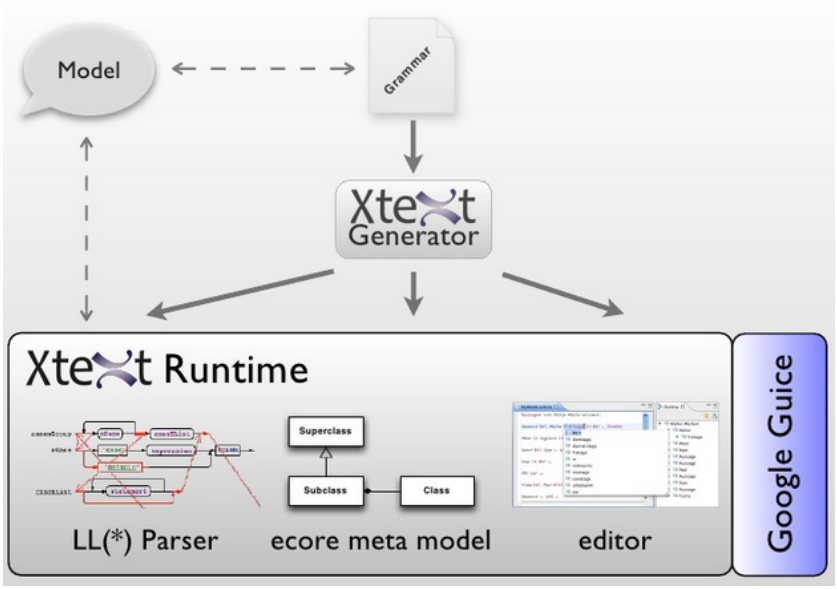


Figura 5.4: artefactos del lenguaje (Xtext Generator)

Para ello se debe ejecutar el fichero *GenerateDsl.mwe2* (Modeling Workflow Engine 2), según se puede observar en la figura 5.5. Sin embargo, antes de ejecutarlo es posible habilitar un wizard que facilite la creación de "proyectos PLESG" en una nueva instancia de Eclipse. Para ello hay que descomentar las siguientes líneas, ubicadas en el fichero anterior.

```
fragment = projectWizard.SimpleProjectWizardFragment { // project wizard (optional)
    generatorProjectName = "${projectName}.generator"
    modelFileExtension = file.extensions
}
```

Adicionalmente también se puede configurar el wizard para que genere, mediante el uso de Xpand, diferentes artefactos textuales dentro del nuevo proyecto PLESG (ver punto 5.1.2.2 – configuración adicional del wizard).

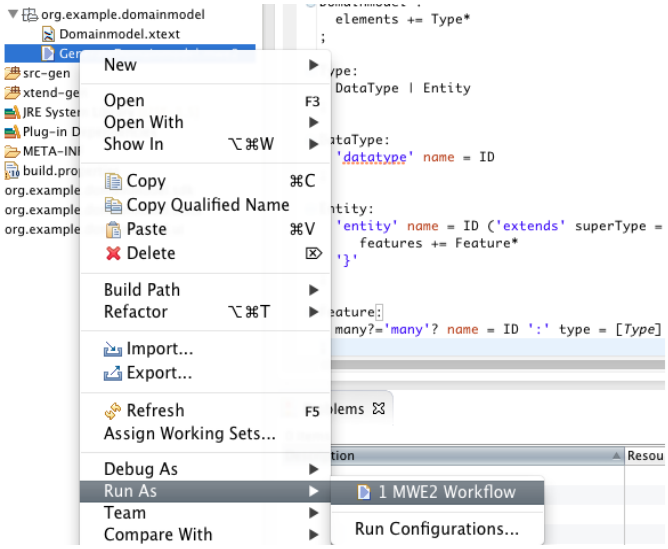


Figura 5.5: Generación de los artefactos del lenguaje

Esta ejecución es necesaria realizarla siempre que se produzca un cambio en la gramática. El resultado de su ejecución se puede observar en la consola de Eclipse.

```
                                                ...
    57905 [main] INFO  .emf.mwe2.runtime.workflow.Workflow  - Done.
```

Al finalizar correctamente la ejecución del proceso anterior, éste añade en el proyecto un generador de código basado en Xtend dentro del paquete *xtend-gen/org.xtext.portaMagna.generator*. Dicho generador ha sido eliminado por la necesidad de implementar varios traductores, por lo que se deberá eliminar la siguiente entrada del fichero **.mwe2* anterior.

```
// generator API
fragment = generator.GeneratorFragment {
  generateMwe = false
  generateJavaMain = false
}
```

La generación del metamodelo Ecore, a partir del proceso lanzado, es debido al uso de la directiva *generate* incluida en línea 2 de la gramática. Dicha directiva le dice a Xtext que derive un EPackage[31] vacío llamado dsl (ver figura 5.6) a partir de la gramática. Xtext agregará posteriormente al EPackage una EClass con EAttributes y EReferences por cada regla parser definida en la gramática, según se describe en el *Ecore model inference*[32].

[31] http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/EPackage.html
[32] http://www.eclipse.org/Xtext/documentation.html#metamodelInference

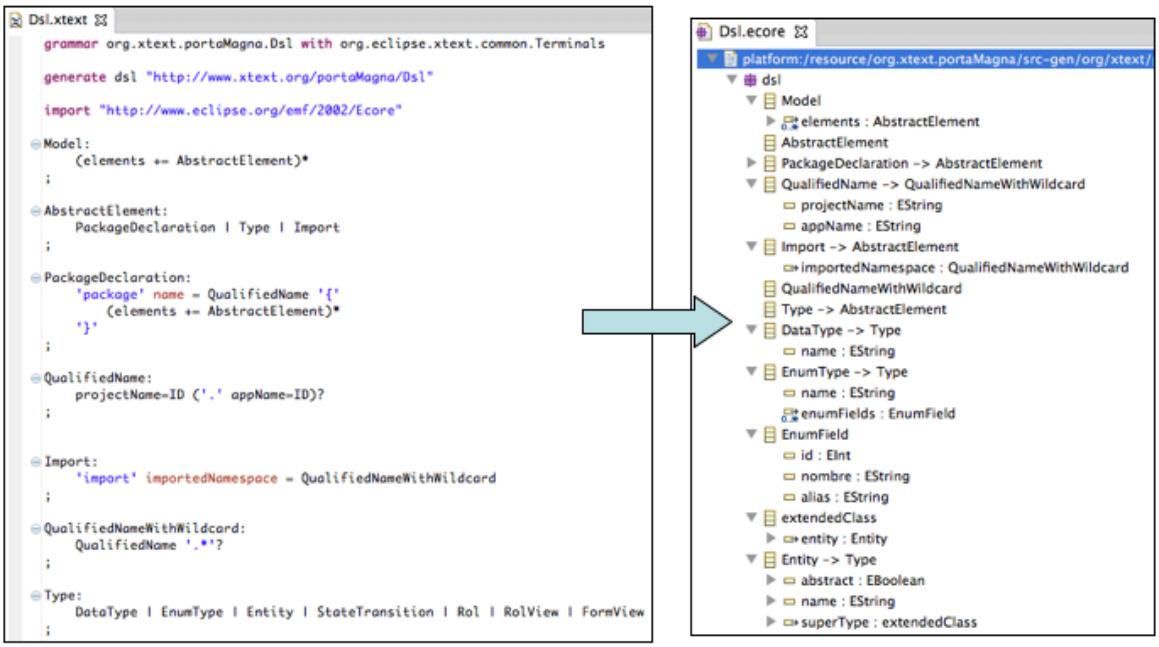


Figura 5.6: generación del metamodelo Ecore

En el siguiente punto se comenta el proyecto de la interfaz de usuario, donde se verá la implementación de un menú contextual para poder invocar a los traductores y la configuración adicional del asistente para facilitar la creación de proyectos PLESG.

5.1.2 Proyecto de la interfaz de usuario

El proyecto de la interfaz de usuario, llamado *org.xtext.portaMagna.ui*, se genera automáticamente al crear un nuevo proyecto Xtext mediante el correspondiente asistente, y contiene todo el código relacionado con la IU relativo al DSL; la implementación de un menú contextual para invocar al generador de código correspondiente (traductor), configuración inicial del IDE, asistente de contenido, etc.

Como se comentó en el punto anterior, tras ejecutar el fichero *GenerateDsl.mwe2* aparece un generador de código basado en Xtend. Dicho generador está configurado por defecto para ser invocado automáticamente cada vez que se guarda un fichero *.model desde el editor proporcionado por Xtext Generator.

Así pues, la forma actual que por defecto se ha establecido en Xtext para invocar al generador no es la adecuada cuando se pretende tener varios traductores, pues el analista no puede decidir para qué arquitectura generar, pero, ¿cómo hacer entonces si se quiere llamar explícitamente a un traductor determinado?. Una posible solución sería la mostrada en la siguiente imagen, donde aparece un menú contextual al seleccionar la carpeta donde se ubican todos los modelos de la especificación PLESG.

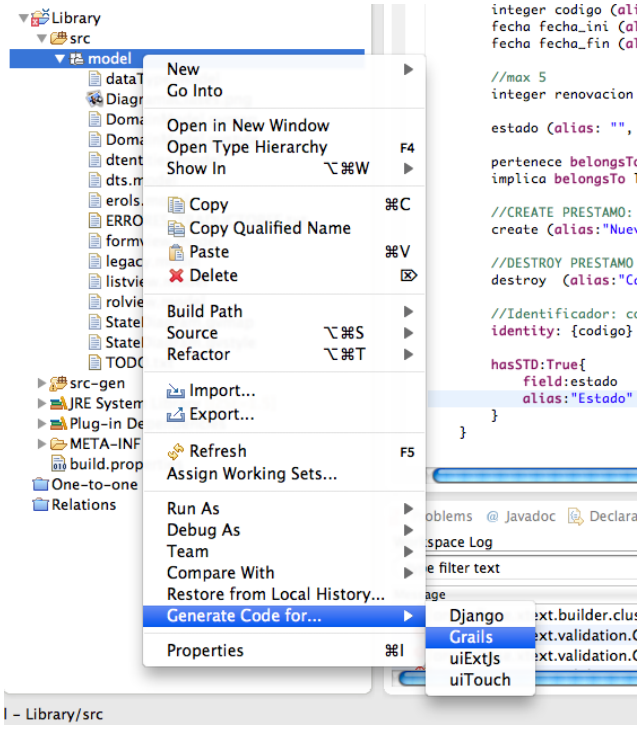


Figura 5.7: Invocación manual de un generador mediante un menú contextual

En el siguiente punto vamos a comentar los pasos necesarios para implementar dicho menú contextual.

5.1.2.1 Menú contextual

El primer paso necesario sería deshabilitar el punto de extensión encargado de realizar la invocación por defecto. Para ello se deben comentar las siguientes líneas, ubicadas en el fichero *plugin.xml* del actual proyecto.

```
                                        ...
<!-- extension point="org.eclipse.xtext.builder.participant">
  <participant
class="org.xtext.portaMagna.ui.DslExecutableExtensionFactory:org.eclipse.xtext.buil
der.IXtextBuilderParticipant">
  </participant>
</extension -->
                                        ...
```

Según podremos ver en el punto 5.2, Xtext ofrece la posibilidad de implementar en otro proyecto (plugin de Eclipse) una clase Xtend que implemente la interfaz *IGenerator,* obligando a dicha clase a sobrescribir el método *doGenerate*, el cual implementa toda la lógica del traductor/generador. El problema que surge es como invocar dicho método, ubicado en otro proyecto/plugin, desde un menú contextual que aparecerá en el editor que proporciona Xtext Generator.

La solución adoptada ha consistido en usar Google Guice[33] e implementar varios manejadores (handlers), uno por cada traductor, ubicados en el paquete *org.xtext.portaMagna.ui.handler*. En el anexo 6 podemos ver el código completo del manejador *GenerationHandlerGrails.java* implementado para invocar al traductor Grails. El resto de manejadores son iguales, ha excepción de las líneas 1 y 3 de la anterior clase.

```
                                        ...

1. DslInjectorProviderGrails injectorProvider = new DslInjectorProviderGrails();
2. Injector injector = injectorProvider.getInjector();
3. generator = injector.getInstance(DslGeneratorGrails.class);
4. if(resource != null) generator.doGenerate(resource, fsa);

                                        ...
```

En la línea 1 se crea una instancia del proveedor de dependencias específico a cada traductor, y en la línea 3 se obtiene una instancia concreta del generador correspondiente para poder invocar el método *doGenerate()*.

Otro aspecto a tener en cuenta es la ruta donde guardar los artefactos generados por los traductores, la cual se ha establecido por defecto a "src-gen" en el método *execute* de cada handler.

```
                                        ...

IFolder srcGenFolder = project.getFolder("src-gen");
if (!srcGenFolder.exists()) {
  try {
    srcGenFolder.create(true, true, (IProgressMonitor) new NullProgressMonitor());
  } catch (CoreException e) { return null; }
}                                       }
fsa.setOutputPath(srcGenFolder.getFullPath().toString());

                                        ...
```

Por último, y no menos importante, se debe de establecer la conexión entre el menú contextual del editor proporcionado por *Xtext Generator* y el *handler* correspondiente, encargado de invocar a su traductor, tal y como podemos observar en la figura 5.8.

[33] Google Guice es un framework de inyección de dependencias que puede ser utilizado en aplicaciones hechas con Java.

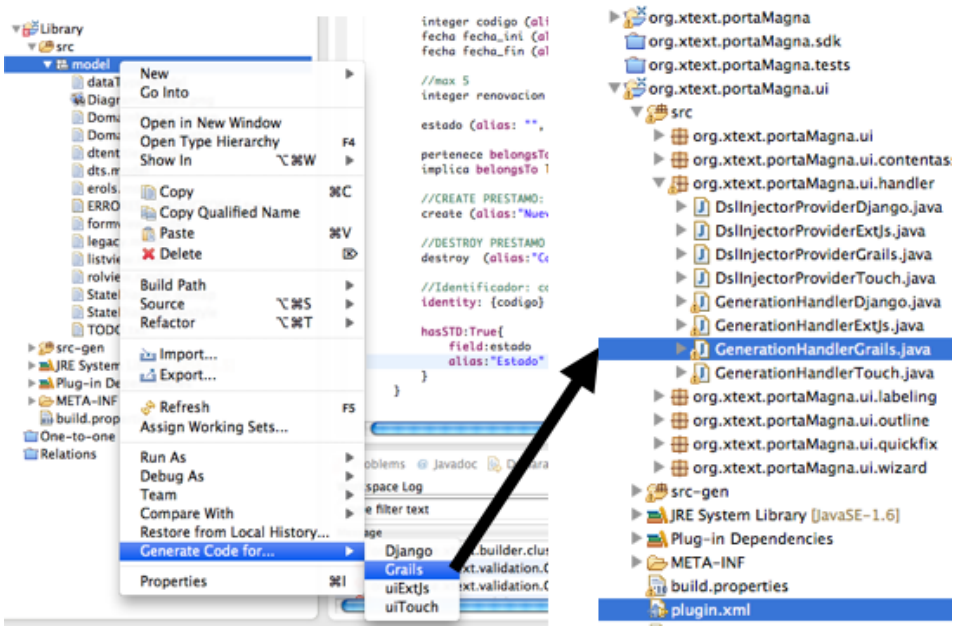


Figura 5.8: Relación entre el menú contextual y el handler del traductor

Para establecer tal conexión se deben de incluir los siguientes puntos de extensión dentro del fichero *plugin.xml*, por ejemplo, justo detrás de las líneas antes comentadas.

```
<!-- extension point="org.eclipse.xtext.builder.participant">
  <participant
class="org.xtext.portaMagna.ui.DslExecutableExtensionFactory:org.eclipse.xtext.builder
.IXtextBuilderParticipant">
  </participant>
</extension-->

<!-- Calling the Generator from a Context Menu -->
<extension point="org.eclipse.ui.handlers">
     <handler class="org.xtext.portaMagna.ui.DslExecutableExtensionFactory:
org.xtext.portaMagna.ui.handler.GenerationHandlerDjango"
           commandId="org.xtext.portaMagna.ui.handler.GenerationCommandDjango">
     </handler>
</extension>

<extension point="org.eclipse.ui.handlers">
     <handler class="org.xtext.portaMagna.ui.DslExecutableExtensionFactory:
org.xtext.portaMagna.ui.handler.GenerationHandlerExtJs"
           commandId="org.xtext.portaMagna.ui.handler.GenerationCommandExtJs">
     </handler>
</extension>

<extension point="org.eclipse.ui.handlers">
     <handler class="org.xtext.portaMagna.ui.DslExecutableExtensionFactory:
org.xtext.portaMagna.ui.handler.GenerationHandlerTouch"
           commandId="org.xtext.portaMagna.ui.handler.GenerationCommandTouch">
     </handler>
</extension>

<extension point="org.eclipse.ui.handlers">
     <handler class="org.xtext.portaMagna.ui.DslExecutableExtensionFactory:
org.xtext.portaMagna.ui.handler.GenerationHandlerGrails"
           commandId="org.xtext.portaMagna.ui.handler.GenerationCommandGrails">
     </handler>
</extension>
```

```
<extension point="org.eclipse.ui.commands">
        <command name="Django"
              id="org.xtext.portaMagna.ui.handler.GenerationCommandDjango">
        </command>
</extension>

<extension point="org.eclipse.ui.commands">
        <command name="Grails"
              id="org.xtext.portaMagna.ui.handler.GenerationCommandGrails">
        </command>
</extension>

<extension point="org.eclipse.ui.commands">
        <command name="uiExtJs"
              id="org.xtext.portaMagna.ui.handler.GenerationCommandExtJs">
        </command>
</extension>

<extension point="org.eclipse.ui.commands">
        <command name="uiTouch"
              id="org.xtext.portaMagna.ui.handler.GenerationCommandTouch">
        </command>
</extension>

<extension point="org.eclipse.ui.menus">
    <menuContribution locationURI="popup:org.eclipse.jdt.ui.PackageExplorer">

        <menu
          id="fileMenu"
          label="Generate Code for...">
            <command
               commandId="org.xtext.portaMagna.ui.handler.GenerationCommandDjango"
               style="push">
               <visibleWhen checkEnabled="false">
                  <iterate>
                      <adapt type="org.eclipse.core.resources.IResource">
                      <test property="org.eclipse.core.resources.name" value="model"/>
                      </adapt>
                   </iterate>
               </visibleWhen>
               </command>
               <command
               commandId="org.xtext.portaMagna.ui.handler.GenerationCommandGrails"
               style="push">
               <visibleWhen checkEnabled="false">
                  <iterate>
                    <adapt type="org.eclipse.core.resources.IResource">
                      <test property="org.eclipse.core.resources.name" value="model"/>
                    </adapt>
                  </iterate>
               </visibleWhen>
               </command>

               <command
               commandId="org.xtext.portaMagna.ui.handler.GenerationCommandExtJs"
               style="push">
               <visibleWhen checkEnabled="false">
                  <iterate>
                    <adapt type="org.eclipse.core.resources.IResource">
                      <test property="org.eclipse.core.resources.name" value="model"/>
                    </adapt>
                  </iterate>
               </visibleWhen>
               </command>
```

```
                <command
                  commandId="org.xtext.portaMagna.ui.handler.GenerationCommandTouch"
                  style="push">
                <visibleWhen checkEnabled="false">
                   <iterate>
                       <adapt type="org.eclipse.core.resources.IResource">
                        <test property="org.eclipse.core.resources.name" value="model"/>
                     </adapt>
                   </iterate>
                </visibleWhen>
                </command>
        </menu>
    </menuContribution>
</extension>
<!-- END Calling the Generator from a Context Menu -->
```

Finalmente, según se comentó en el punto 5.1.1.3 (generación de los artefactos del lenguaje), se habilitó un asistente para facilitar la creación de proyectos PLESG, sin embargo existen configuraciones adicionales que deben de realizarse en el presente proyecto.

5.1.2.2 Configuración adicional del wizard

Además de lo expuesto anteriormente, es necesario incluir las siguientes líneas en el fichero *plugin.xml* del presente proyecto.

```
<extension point="org.eclipse.ui.newWizards">
      <wizard
            category="org.eclipse.xtext.projectwiz"
            class="org.xtext.portaMagna.ui.DslExecutableExtensionFactory:
org.xtext.portaMagna.ui.wizard.DslNewProjectWizard"
            id="org.xtext.portaMagna.ui.wizard.DslNewProjectWizard"
            name="PLESG Project"
            project="true">
      </wizard>
</extension>
```

Por otra parte, también es posible configurar el fichero <MYDSL>NewProject.xpt[34], ubicado en el paquete *org.xtext.portaMagna.ui.wizard*, para inicializar el nuevo proyecto PLESG con ciertos modelos que los analistas funcionales pueden tomar como punto de partida para especificar el sistema software a desarrollar. De hecho se ha configurado la plantilla anterior (anexo 7) para que tras finalizar el asistente se generen varios recursos dentro de “src/” en el nuevo proyecto PLESG.

Ruta	Fichero	Artefacto	Significado
src/model/	config.model	Información	Permite establecer cierta información importante, como es el nombre del proyecto y de la aplicación, así como la versión y el idioma
	dataType.model	Tipo de datos	Fichero donde especificar los distintos tipos de datos y tipos enumerados

[34] Es una plantilla Xpand que permite generar artefactos textuales en el nuevo proyecto PLESG

	dentities.model	Diagrama de clases	Fichero donde especificar el diagrama de clases
	drol.model	Roles	Recurso donde especificar los actores del sistema
	dts.model	Diagrama de Transición de Estados	Fichero donde especificar el conjunto de estados que puede tener cada una de las clases
	rolview.model	Visibilidad	Fichero donde establecer la visibilidad que tiene cada rol sobre los diferentes elementos
	formview.model	Diseño de formularios	Fichero donde se especifica el diseño de los diferentes formularios, en función del rol
	listview.model	Diseño de listados/grids	Fichero donde se especifica el diseño de los diferentes listados; uno por cada terna <entidad, rol y dispositivo> (desktop y/o móvil)
src/ggv	DomainModel.map	Mapeo	Establece el mapeo entre el diagrama de clases y el DSL interno del GGV para poder realizar una representación gráfica en el visor
	DomainModel.style	Estilo	Estable los estilos, es decir, como serán representados las clases, junto con sus propiedades y servicios en el visor
	StateDiagram.map	Mapeo	Establece el mapeo entre un DTS y el DSL interno del GGV para poder realizar una representación gráfica en el visor
	StateDiagram.style	Estilo	Estable los estilos, es decir, como será representado el DTS en el visor

5.2 Implementación de los traductores

Una vez se han generado los distintos artefactos del lenguaje específico de dominio, aparecerá automáticamente un generador de código basado en Xtend. Debido a las necesidades de la presente propuesta (recordemos que queremos generar artefactos para varios frameworks) el generador de código que proporciona *Xtext Generator* ha sido descartado.

La solución adoptada pasa por explotar el hecho que Eclipse está basado en una arquitectura de plugins, y por ello se ha decidido desarrollar varios, uno por cada traductor y cuya estructura es prácticamente común en todos ellos. En concreto se han creado cuatro traductores; dos para la parte cliente y dos para la parte servidora.

En la siguiente tabla podemos ver los frameworks soportados por PLESG, junto con su versión y cierta información adicional: capa de la arquitectura y tecnología en la que está basada el traductor.

Capa	Traductor basado en	Framework	Versión
Cliente	Xtend	ExtJs (desktop)	4.1.1.a
		Sencha Touch (móvil)	2.X[35]
Servidor		Django	1.4.1
		Grails	2.2
Persistencia[36]	Scripts propios de cada framework; Django y Grails	SQLite	3
		SQLite	3

Cada capa es responsable de un conjunto diferente de actividades: desde el acceso a datos, la lógica de dominio y/o las reglas de negocio, hasta la lógica de presentación de la aplicación web ejecutada en el navegador del usuario. Las capas cliente/servidor siguen el patrón de diseño Model View Controller (MVC), lo cual facilita la extensión y el mantenimiento de las aplicaciones generadas debido a la buena organización y estructura interna del código.

Esas mismas características son deseables en los traductores desarrollados. Es importante señalar que la estrategia común a cada traductor ha sido analizar todas las entidades, elementos de diseño de la IU y servicios definidos en la especificación PLESG para generar los **modelos**, **vistas** y **controladores** propios a cada capa de la arquitectura cliente/servidor. Así pues, en el presente apartado comentaremos la estructura interna común a los cuatro traductores desarrollados sin entrar en detalles específicos sobre cada uno.

En la siguiente figura podemos ver los cuatro plugins (proyectos de Eclipse) que implementan a cada uno de los traductores mencionados. Puede observarse como el proyecto que implementa el traductor para Django se ve desplegado, mostrando el recurso que implementa la lógica del traductor.

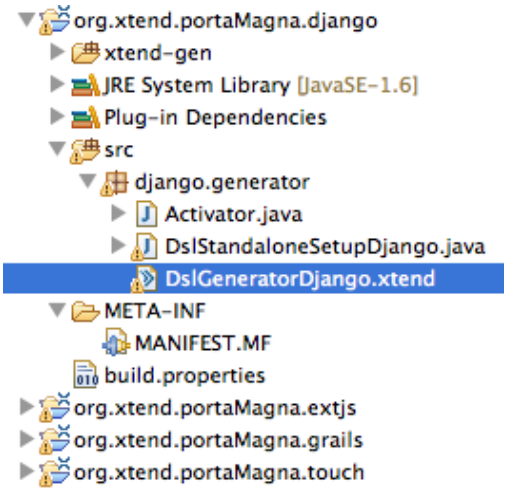


Figura 5.9: Proyectos Eclipse que implementan los cuatro traductores

[35] Existen discrepancias en la versión del framework Sencha Touch (ver punto 7.2 para más detalles)

[36] La capa de persistencia se ha incluido en la tabla anterior sólo para tener una visión completa de la arquitectura (ver punto 4.6).

Antes de nada localizamos el fichero *src/X.generator DslGeneratorX.xtend,* tal que X = [**django**, grails, extjs, touch], y siendo X = "django" en la figura anterior. Debido a que los traductores comparten cierta estructura común, se utilizará el proyecto "org.xtend.portaMagna.django" a partir de ahora como ejemplo para describir las partes iguales a todos ellos.

Así pues, la clase *DslGeneratorDjango*, ubicada en el paquete *django.generator*, es la encargada de generar los diferentes artefactos a partir de los modelos definidos en el editor proporcionado por una nueva instancia de Eclipse (punto 5.3.1 – selección del entorno de especificación/desarrollo de proyectos PLESG). Dicha clase implementa la interfaz *IGenerator* por lo que debe sobrescribir el método *doGenerate*, que es donde se ha implementado toda la lógica del traductor.

```
1.    package django.generator
2.
3.    import org.eclipse.emf.ecore.resource.Resource
4.    import org.eclipse.xtext.generator.IGenerator
5.    import org.eclipse.xtext.generator.IfileSystemAccess
6.    import org.xtext.portaMagna.dsl.*
7.    import org.xtext.portaMagna.data.*
8.
9.    class DslGeneratorDjango implements IGenerator {
10.                    ...
11.     private static String srcGenPath = "Django";
12.
13.     override void doGenerate(Resource resource, IFileSystemAccess fsa) {
14.                     ...
15.     }
16.   }
```

Según se observa en la tabla anterior, dicho método tiene dos parámetros:

- El parámetro *fsa* permite tener acceso al sistema de ficheros del proyecto de desarrollo para poder guardar los diferentes artefactos generados. Por defecto, para todos los traductores, la ruta a guardar los diferentes artefactos se ha configurado en *"src-gen/"* (punto 5.1.2 – menú contextual) + <*srcGenPath*>, donde la variable estática *srcGenPath* (línea 11) puede modificarse a placer en cada traductor.

- El parámetro *resource* contiene todos los elementos definidos en los distintos modelos PLESG (modelo de objetos, dinámico, etc.) según el DSL desarrollado. Además dichos elementos son accesibles gracias a que se han importado (línea 6).

Es importante señalar que se han creado unas estructuras de datos comunes a todos los traductores y accesibles a ellos gracias a su importación (línea 7). Dichas estructuras de datos se encuentran en el paquete *org.xtext.portaMagna.data* del proyecto del lenguaje (figura 5.10) y su utilidad se comenta a continuación.

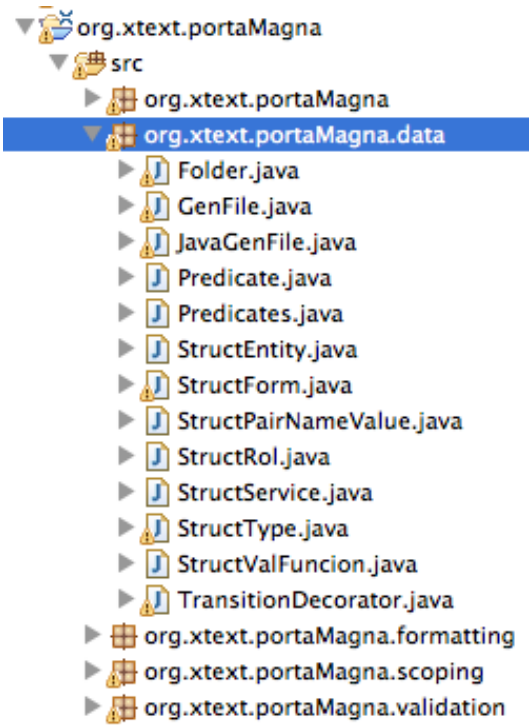


Figura 5.10: ubicación de las estructuras de datos comunes a los traductores

Las tres primeras clases están relacionas con el parámetro fsa anterior, e implementan los métodos que permiten crear nuevos directorios dentro de la ruta "src-gen/" del proyecto de desarrollo donde depositar los artefactos generados por los traductores. El resto de clases definen estructuras que los traductores utilizan para cargar en ellas los elementos definidos en los modelos, accesibles mediante el parámetro *resource*, y así poder acceder a la información adecuada en el momento adecuado gracias al uso de predicados (acceso a las estructuras de datos mediante filtros: en función del rol), con el fin mejorar el mantenimiento de los traductores y facilitar la generación de los distintos artefactos.

A continuación puede verse la relación entre dichas estructuras, donde un modelo se compone de una serie de elementos con un nombre, alias y un array que establece el control de acceso (qué roles tienen permisos sobre los diferentes elementos) a excepción de los elementos TransitionDecorator y StructRol, que no utilizan dicho control de acceso.

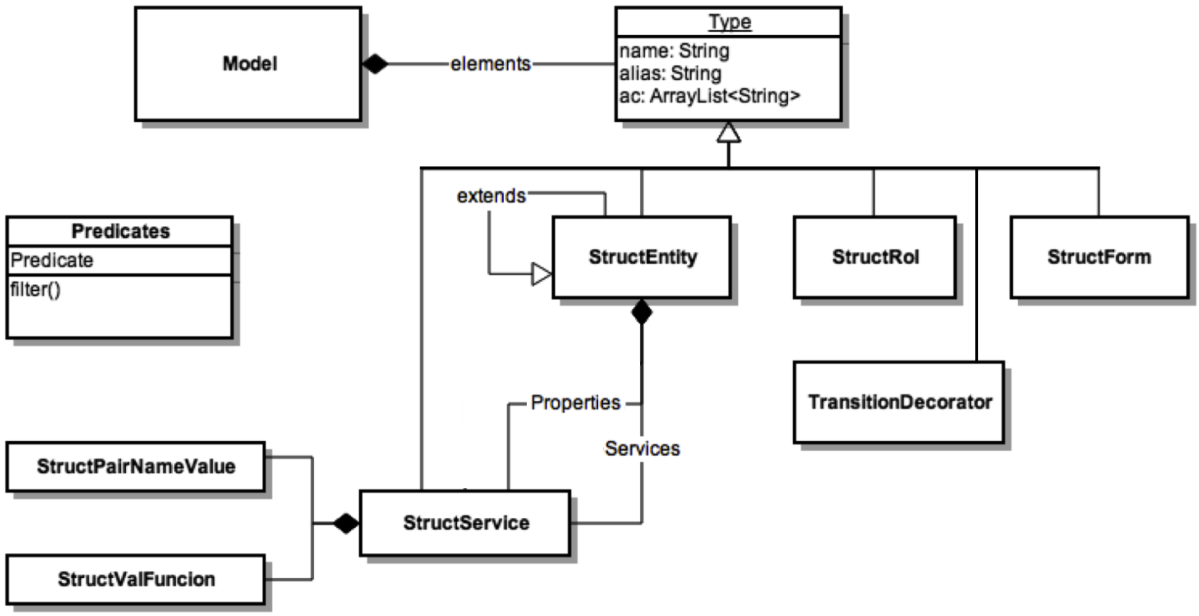


Figura 5.11: Relaciones entre las estructuras de datos internas

Así pues, lo primero que hace todo traductor, a parte de inicializar ciertas variables locales, entre ellas el nombre de la aplicación y del proyecto, es cargar los diferentes elementos definidos en cada unos de los modelos PLESG en las estructuras anteriores. Esto se realiza en la función *srcModelVisitor*.

```
...

//CARGAMOS EL MODELO/S EN LAS ESTRUCTURAS DE DATOS
srcModelVisitor(resource);

fsa.generateFile("models.py", pythonSrcDomainModelVisitor(resource))

fsa.generateFile("common.py", pythonSrcCommonVisitor(resource))

//Establecemos el Control de Acceso en las estructuras
srcRolViewVisitor(resource)

...
```

El siguiente paso sería establecer el Control de Acceso (AC) en función del rol en dichas estructuras mediante la llamada a la función *srcRolViewVisitor*. Sin embargo, como excepción en el traductor Django, este paso se realiza tras generar los ficheros “models.py” y “common.py”. Las funciones que generan dichos ficheros, así como el resto de funciones que aparecen en el método *doGenerate()* son propias a cada traductor y dan lugar a una serie de artefactos que ya han sido analizadas en el punto 4.3.

En el siguiente apartado vamos a comentar los últimos pasos necesarios antes de poder empezar a realizar la especificación PLESG.

5.3 Configuración de la especificación PLESG

Antes de poder realizar la especificación PLESG (vista en el punto 4.1) se hacen necesarios ciertos pasos adicionales, los cuales son comentados a continuación.

5.3.1 Selección del entorno de desarrollo

Una vez se han generado los diferentes artefactos del lenguaje, se hace necesario disponer de un editor donde realizar la especificación PLESG. Existen dos opciones:

- instalar el proyecto con el editor del lenguaje en un entorno Eclipse
- desde el proyecto del lenguaje, ejecutar una segunda instancia de Eclipse

Se ha escogido la segunda opción debido a que agiliza bastante la fase de desarrollo.

Para lanzar el nuevo entorno sólo hay que seleccionar el proyecto del lenguaje *org.xtext.portaMagna*, pulsar el botón Run y seleccionar *Launch Runtime Eclipse* según se muestra en la siguiente figura.

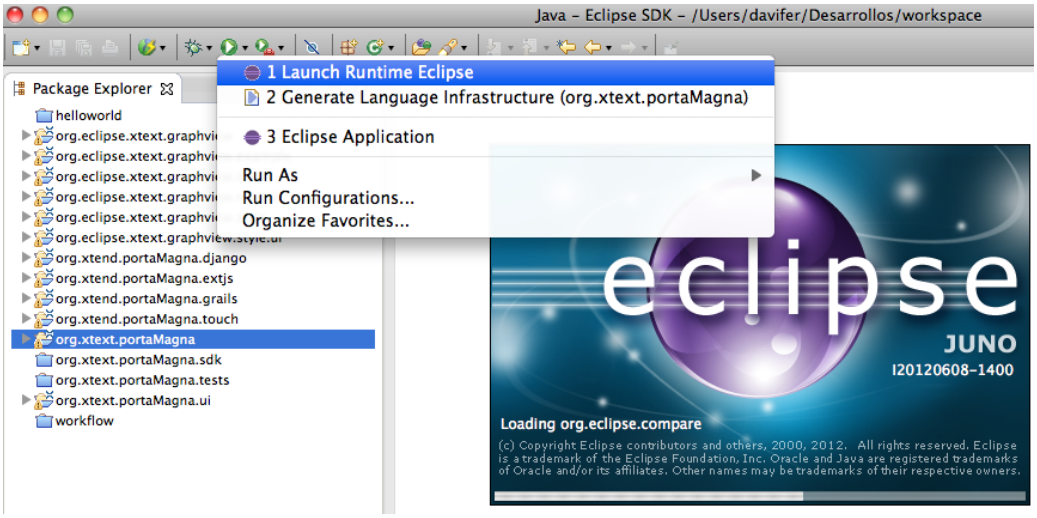


Figura 5.12: Lanzar un nuevo entorno de desarrollo PLESG

Según se puede observar, tras seleccionar dicha opción, arrancará una nueva instancia de Eclipse, aunque es muy probable que se deban modificar los parámetros de la máquina virtual de java para asignar más memoria RAM al nuevo entorno.

5.3.2 Creación y configuración de un nuevo proyecto PLESG

Gracias a la configuración realizada en los puntos 5.1.1 y 5.1.2 relativa al wizard, es posible crear nuevos proyectos PLESG en el nuevo workbench mediante la ayuda de un asistente. Para ello ir a *File* → *New Proyect*, seleccionar "PLESG Project" e introducir el nombre del nuevo proyecto según se muestra en la figura 5.13.

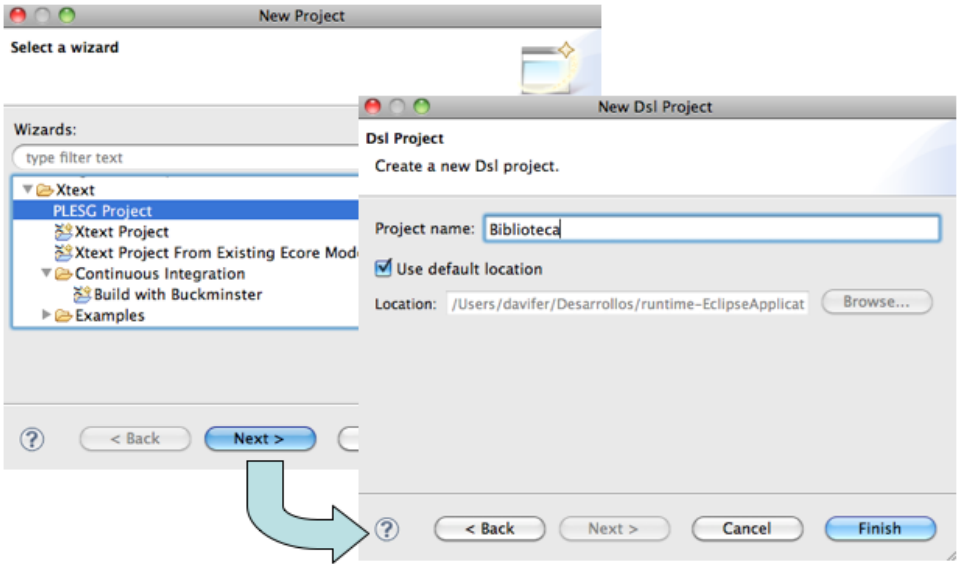


Figura 5.13: Nuevo proyecto PLESG

Tras pulsar en botón *Finish*, aparecerán en el nuevo proyecto los artefactos textuales comentados al final del punto 5.1.2, a partir de los cuales podremos empezar a definir la especificación del sistema software requerido.

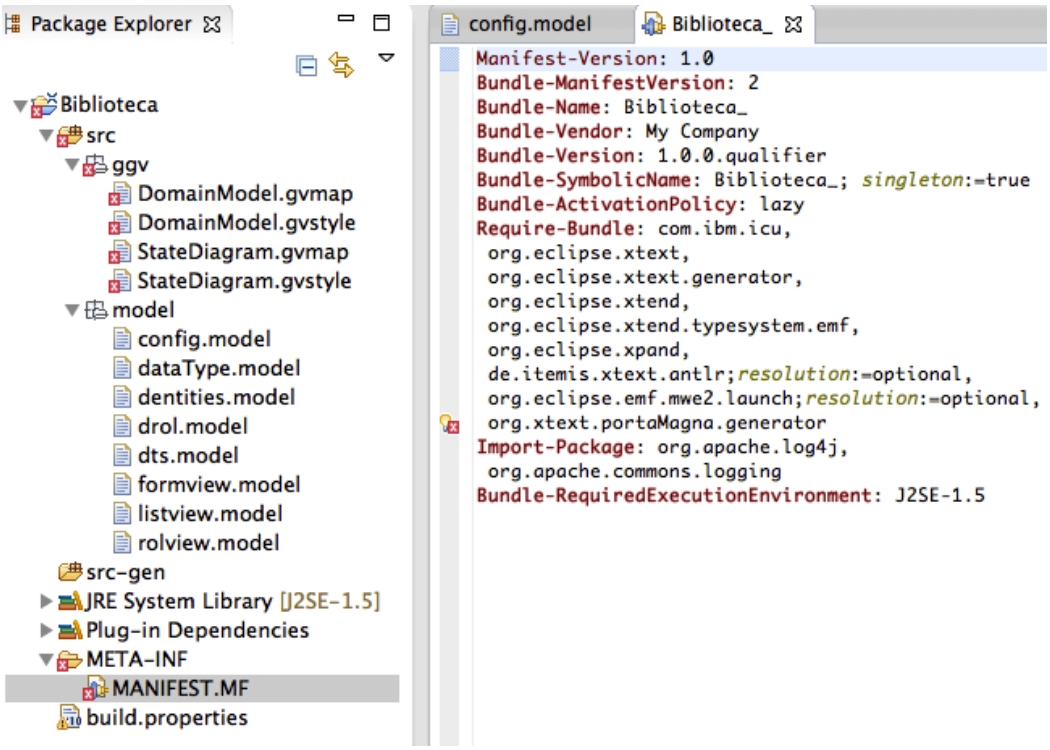


Figura 5.14: Configuración del nuevo proyecto PLESG

Antes de entrar en materia, debemos fijarnos en un error que aparece en el fichero MANIFEST.MF, como puede observarse en la figura anterior:

Bundle 'org.xtext.portaMagna.generator' cannot be resolved

Recordemos que Xtext crea por defecto un generador de código en el proyecto del lenguaje y que éste se eliminó intencionadamente. Así pues, ese error es debido a que por defecto el wizard está configurado para generar proyectos que utilicen dicho generador, haciendo referencia a él en el fichero MANIFEST. Por lo tanto, antes de seguir, hay que eliminar "manualmente" esa referencia, e incluir otras, según se muestra en la siguiente tabla:

Referencias a eliminar	Referencias a incluir
org.xtext.portaMagna.generator	org.eclipse.xtext.xbase.lib, org.eclipse.emf.ecore, org.eclipse.jdt.core, org.eclipse.emf.common, org.eclipse.xtext.common.types, org.eclipse.xtext.graphview, org.xtext.portaMagna

Las referencias a incluir son las dependencias necesarias para poder usar el visor Generic Graph View (GGV) en el proyecto PLESG. Adicionalmente se necesitan los recursos que aparecen en la ruta src/ggv/*, lo cuales contienen el mapeo y estilo correspondientes para poder visualizar el diagrama de clases y el DTS en el visor GGV.

Existe otra alternativa para eliminar/añadir las referencias anteriores de forma automática. Para ello habría que ir al método *getRequiredBundles()* de la clase *DslProjectCreator,* ubicada en el paquete *src-gen/org.xtext.portaMagna.ui.wizard,* y sustituir su código por las líneas que aparecen a continuación.

```
public class DslProjectCreator extends AbstractPluginProjectCreator {
                                        ...
@Override
protected List<String> getRequiredBundles() {
        List<String> result = Lists.newArrayList(super.getRequiredBundles());
        //result.add(DSL_GENERATOR_PROJECT_NAME);
        result.add("org.eclipse.xtext.xbase.lib");
        result.add("org.eclipse.emf.ecore");
        ...
        result.add("org.eclipse.xtext.graphview");
        result.add("org.xtext.portaMagna");
        return result;
}
```

Sin embargo, debido a que dicha clase está ubicada en la carpeta "src-gen", dicha sustitución habría que realizarla cada vez que se regeneren los artefactos del lenguaje tras ejecutar el fichero .mwe2 (ver figura 5.5).

5.3.3 Mejora del asistente de contenido (opcional)

Por otra parte es posible implementar ciertas plantillas, asociadas a un contexto, para autocompletar algunas partes de la especificación PLESG y así acelerar la creación de los distintos modelos del análisis. Para ello es necesario ir a *Eclipse → Preferencias → Dsl → Templates*, tras lo cual aparecerá un editor donde implementarlas, según se muestra en la siguiente figura.

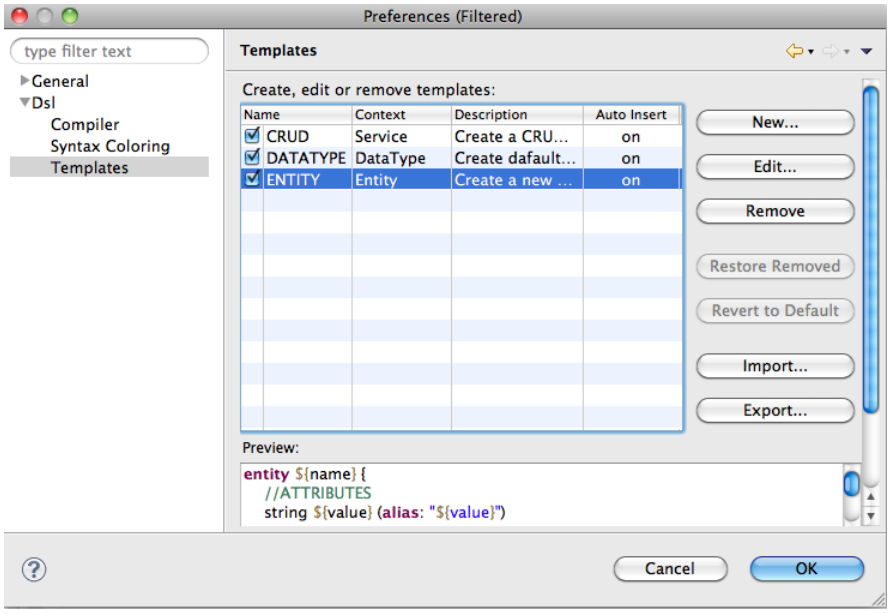


Figura 5.15: mejorar el asistente de contenido en PLESG

Dichas plantillas aparecerán después al final del asistente de contenido, el cual se invoca pulsando las teclas “ctrl + space” desde cualquier fichero con extensión **.model*. En función de donde este el cursor en ese momento (en qué contexto) se mostrarán unas plantillas u otras para evitar crear modelos erróneos. Por ejemplo, en la siguiente figura no aparece la plantilla CRUD y es debido a que estamos en el contexto de un paquete.

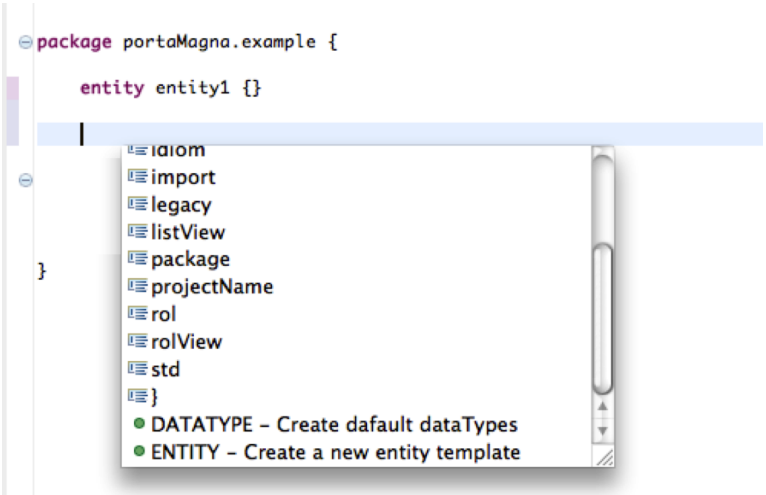


Figura 5.16: Asistente de contenido en Xtext

En el siguiente capítulo vamos a tener la oportunidad de poder validar el proceso definido.

Capítulo 6 – Validación del proceso PLESG

Ha surgido la oportunidad de poder validar el proceso PLESG mediante el desarrollo de una aplicación para la gestión de pacientes en una clínica dental. Concretamente es la Clínica Dental Fernández – Traver, en Vila-real (Castellón). Con el fin de poder hacer mención a la empresa citada se ha establecido un acuerdo de colaboración, firmado por cada una de las partes implicadas (anexo 9).

Como veremos a continuación, la complejidad de la aplicación es muy baja, solo consta de cuatro clases y tres relaciones, pero ha exigido implementar/analizar ciertas funcionalidades adicionales que no se habían contemplado abordar inicialmente. Éstas serán comentadas en el último punto, donde también se incluye un análisis sobre la ganancia en cuanto al aumento de la productividad.

Vamos a ver ahora cada una de las fases del proceso PLESG para la nueva aplicación.

6.1 Especificación PLESG

Actualmente los trabajadores de la clínica dental rellenan a mano unos documentos en papel (anexo 10) relativos a la ficha del paciente y sus datos médicos. Después pasan parte de esa información a una hoja Excel por si se extravían dichos documentos. Se ha utilizado dicha documentación para extraer parte de los requisitos de la aplicación, pues adicionalmente también necesitan poder gestionar tanto las radiografías como las citas/evoluciones de cada paciente.

Como se comentó en el punto 4.1, el primer paso en la especificación PLESG es introducir el nombre del proyecto y de la aplicación en el recurso config.model.

```
projectName portaMagnaDjango
appName fernandez_traver
version 1.0
idiom ES
```

En este caso no ha sido necesario declarar nuevos tipos de datos en el fichero *dataType.model*.

En la siguiente figura se muestra la representación gráfica del diagrama de clases que muestra GGV tras modelar textualmente las diferentes entidades.

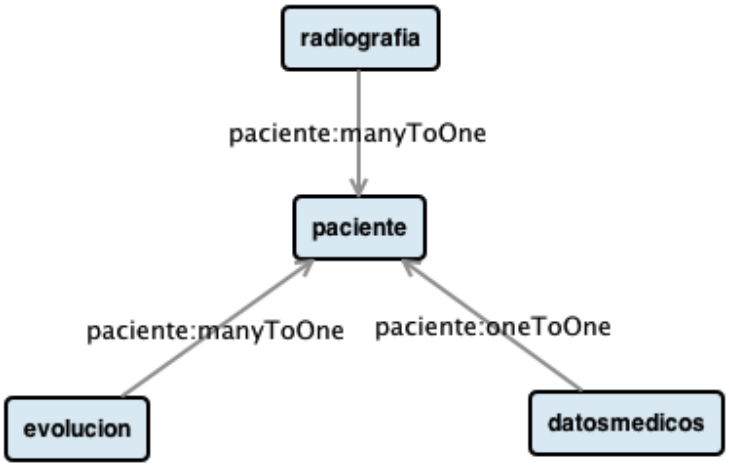


Figura 6.1: Diagrama de clases "Clínica Dental"

Ahora sí, vamos a mostrar la representación textual de la anterior figura según se ve en el editor que proporciona *Xtext Runtime*.

```
package clinicaDental.fernandez_traver {

  entity paciente {
        string nombre (alias:"Nombre", max_length:25)
        string apellidos (alias: "Apellidos", max_length:50)
        string nacionalidad (alias: "Nacionalidad", max_length:50)
        date nacimiento (alias: "Fecha de nacimiento", widget:widgetDate)
        string dni (alias:"DNI/NIE", unique:True, max_length:15, mandatory:False)

        //direccion
        string calle (alias:"Calle/Avda.", max_length:100)
        string numero (alias:"N°", max_length:5, mandatory:False)
        integer cp (alias:"CP", mandatory:False)
        string poblacion (alias:"Población", max_length:50, mandatory:False)

        string telfijo (alias:"Tlf. Fijo", max_length:12, mandatory:False)
        string telmovil (alias:"Tlf. Movil", max_length:12, mandatory:False)

        email email (alias:"Correo electronico", mandatory:False)

        string seguro (alias:"Seguro dental", max_length:75, mandatory:False)
        string poliza (alias:"N° Poliza", max_length:75, mandatory:False)

        string contacto (alias:"Persona contacto", max_length:75, mandatory:False)
        string contactotelf (alias:"Tlf. contacto",max_length:12, mandatory:False)

         /*
          * Fechas especiales, mantenidas por los frameworks de destino
          */
         date dateCreated (alias:"F. alta", mandatory:False)
         date lastUpdated (alias:"F. actualizacion", mandatory:False)

        edad(alias: "Edad") <= function getEdad("Calculo de la edad de nacimiento")
        estado (alias:"Estado", isSTDField:True)

        create (alias:"Guardar", type: C) : void
        update  (alias:"Actualizar", type: U) : void
        destroy  (alias:"Borrar", type: D) : void
```

```
        identity: { apellidos, nombre }

        hasSTD:True{
                field:estado, alias:"Estado"
        }
}

  entity datosmedicos {
        //ATTRIBUTES
        string alergias (alias: "Alergias", max_length:1050, mandatory:False,
                widget:widgetTextarea)
        string antecedentes (alias:"Antecedentes", max_length:1050,
                mandatory:False, widget:widgetTextarea)
        string antfamiliares (alias:"Ant. familiares", max_length:1050,
                mandatory:False, widget:widgetTextarea)

        //procedimientos de higiene
        boolean cepillado (alias:"Cepillado", default:"False")
        string freqcepillado (alias:"Frec. cepillado", max_length:50,
                mandatory:False)
        boolean seda (alias:"Seda", default:"False")
        string freqseda (alias:"Frec. seda", max_length:50, mandatory:False)
        boolean colutorio (alias:"Colutorios", default:"False")
        string freqcolutorio (alias:"Frec. colutorio", max_length:50,
                mandatory:False)
        string nombrecolutorio (alias:"Nombre colutorio", max_length:50,
                mandatory:False)

        string sintomas (alias:"Sintomas que refiere el paciente", max_length:1050,
                mandatory:False, widget:widgetTextarea)
        string medicacion (alias:"Medicacion actual", max_length:1050,
                mandatory:False, widget:widgetTextarea)
        string enfermedadesinfec (alias:"Enfermedades infecciosas", max_length:1050,
                mandatory:False, widget:widgetTextarea)

        /*
         * Fechas especiales, mantenidas por los frameworks de destino
         */
        date dateCreated (alias:"F. alta", mandatory:False)
        date lastUpdated (alias:"F. actualizacion", mandatory:False)

        estado (alias:"Estado", isSTDField:True)

        paciente oneToOne paciente

        //SERVICES
        create (alias:"Guardar", type: C) : void
        update (alias:"Actualizar", type: U) : void
        destroy (alias:"Borrar", type: D) : void

        hasSTD:True{
                field:estado, alias:"Estado"
        }
  }

  entity radiografia {
        //ATTRIBUTES
        string nombre (alias:"Nombre", max_length:50)
        string descripcion (alias:"Descripcion", max_length:1050, mandatory:False,
                widget:widgetTextarea)
        file fichero (alias: "Seleccionar fichero")

        /*
         * Fechas especiales, mantenidas por los frameworks de destino
         */
```

```
        date dateCreated (alias:"F. alta", mandatory:False)
        date lastUpdated (alias:"F. actualizacion", mandatory:False)

        estado (alias:"Estado", isSTDField:True)

        paciente manyToOne paciente


        //SERVICES
        create (alias:"Guardar", type: C) : void
        update (alias:"Actualizar", type: U) : void
        destroy (alias:"Borrar", type: D) : void

        hasSTD:True{
                field:estado, alias:"Estado"
        }

  }

  entity evolucion {
        //ATTRIBUTES
        string nombre (alias:"Nombre", max_length:50)
        string descripcion (alias:"Descripcion", max_length:1050, mandatory:False,
                widget:widgetTextarea)

        /*
         * Fechas especiales, mantenidas por los frameworks de destino
         */
        date dateCreated (alias:"F. alta", mandatory:False)
        date lastUpdated (alias:"F. actualizacion", mandatory:False)

        estado (alias:"Estado", isSTDField:True)

        paciente manyToOne paciente

        //SERVICES
        create (alias:"Guardar", type: C) : void
        update (alias:"Actualizar", type: U) : void
        destroy (alias:"Borrar", type: D) : void

        hasSTD:True{
                field:estado, alias:"Estado"
        }
  }

}
```

Como se puede apreciar, las diferentes clases tienen varios aspectos en común:

- solo constan de servicios CRUD
- tienen dos fechas especiales, dateCreated y lastUpdated, mantenidas por cada framework de destino y utilizadas para futuros análisis estadísticos
- cada entidad tiene un sencillo STD (declaración especificada más adelante)

En la clínica trabajan dos socias y una auxiliar. Ambas gerentes desean que su empleada no pueda borrar ningún tipo de información. Ante este planteamiento se decide crear dos roles en el recurso *drol.model*, según se muestra a continuación.

```
import clinicaDental.fernandez_traver.*

rol ADMIN {
	type : admin, initialView: paciente
}

rol EMPLEADO {
	type : user, initialView: paciente
}
```

Por lo tanto, cuando se cree la visibilidad que cada rol tiene sobre los diferentes servicios de cada una de las entidades, se excluirá el servicio CRUD “delete” al rol EMPLEADO.

A continuación se muestra la especificación de los diferentes Diagramas de Transición de Estados (STD, de sus siglas en ingles) para cada clase, con el único fin de controlar el orden de ejecución de los servicios CRUD.

```
import clinicaDental.fernandez_traver.*

std paciente {
	create => nuevo

	initial state nuevo {
		actions {  update, destroy }
	}
}

std datosmedicos {
	create => nuevo

	initial state nuevo {
		actions {  update, destroy }
	}
}

std radiografia {
	create => nuevo

	initial state nuevo {
		actions {  update, destroy }
	}
}

std evolucion {
	create => nuevo

	initial state nuevo {
		actions {  update, destroy }
	}
}
```

La visibilidad que tiene cada rol sobre las diferentes clases se muestra a continuación. Recordemos que el rol de tipo “admin” tiene total visibilidad sobre todos los elementos de la especificación, y sólo hay que declarar las posibles navegaciones que deseemos que tenga.

```
import clinicaDental.fernandez_traver.*
```

```
rolView admin_paciente (ADMIN, paciente){
  navigation : [
        { datosmedicos, "Ir a datos Medicos", filter : { from:id,  to:paciente } },
        { radiografia, "Ir a radiografias", filter : { from:id,  to:paciente } },
        { evolucion, "Ir a evolucion", filter : { from:id,  to:paciente } }
  ]
}

/*
 * EMPLEADO
 */
rolView empleado_paciente (EMPLEADO, paciente){
  properties:{ all }
  services:{ create, update }
  navigation : [
        { datosmedicos, "Ir a Datos Medicos", filter : {from:id, to:paciente} },
        { radiografia, "Ir a Radiografias", filter : { from:id,  to:paciente } },
        { evolucion, "Ir a Evolucion", filter : { from:id,  to:paciente } }
  ]
}

rolView empleado_datosMedicos (EMPLEADO, datosmedicos){
       properties:{ all }
       services:{ create, update }
}

rolView empleado_evolucion (EMPLEADO, evolucion){
       properties:{ all }
       services:{ create, update }
}

rolView empleado_radiografia (EMPLEADO, radiografia){
       properties:{ all }
       services:{ create, update }
}
```

Como se comentó anteriormente, se le ha excluido al rol EMPLEADO la posibilidad de ejecutar el servicio CRUD “Delete”, llamado “destroy” en este caso.

La última fase de la especificación PLESG consiste en diseñar los componentes visuales de la interfaz de usuario de forma particular a cada rol. Dichos componentes son los listados/grids y los formularios.

El diseño de los listados/grids se puede realizar en el fichero *listview.model,* creado por defecto por el asistente.

```
/*
 * IMPORTANTE:
 *      Necesario especificar para cada GRID y dispositivo (desktop o movil) que
 *     columnas puede ver cada rol
 */

import clinicaDental.fernandez_traver.*

listView admin_paciente (ADMIN, paciente){
       model:[
              {property:apellidos, sortable:True, filter:True},
              {property:nombre, sortable:True, filter:True},
              {property:dni, sortable:True, filter:True},
              {property:email, sortable:True, filter:True},
```

```
                {property:telfijo, sortable:True, filter:True},
                {property:telmovil, sortable:True, filter:True},
                {property:seguro, sortable:True, filter:True},
                {property:poliza, sortable:True, filter:True},
                {property:dateCreated, sortable:True, filter:True}]
}

listView admin_datos (ADMIN, datosmedicos){
        model:[
                {property:alergias, sortable:True, filter:True},
                {property:antecedentes, sortable:True, filter:True},
                {property:dateCreated, sortable:True, filter:True},
                {property:paciente, sortable:True, filter:True}]
}

listView admin_radiografia (ADMIN, radiografia){
        model:[
                {property:nombre, sortable:True, filter:True},
                {property:dateCreated, sortable:True, filter:True},
                {property:paciente, sortable:True, filter:True}]
}

listView admin_evolucion (ADMIN, evolucion){
        model:[
                {property:nombre, sortable:True, filter:True},
                {property:dateCreated, sortable:True, filter:True},
                {property:paciente, sortable:True, filter:True}]
}


/*
 * EMPLEADO
 */
listView empleado_paciente (EMPLEADO, paciente){
        model:[
                {property:apellidos, sortable:True, filter:True},
                {property:nombre, sortable:True, filter:True},
                {property:dni, sortable:True, filter:True},
                {property:email, sortable:True, filter:True},
                {property:telfijo, sortable:True, filter:True},
                {property:telmovil, sortable:True, filter:True},
                {property:seguro, sortable:True, filter:True},
                {property:poliza, sortable:True, filter:True},
                {property:dateCreated, sortable:True, filter:True}]
}

listView empleado_datos (EMPLEADO, datosmedicos){
        model:[
                {property:alergias, sortable:True, filter:True},
                {property:antecedentes, sortable:True, filter:True},
                {property:dateCreated, sortable:True, filter:True},
                {property:paciente, sortable:True, filter:True}]
}

listView empleado_radiografia (EMPLEADO, radiografia){
        model:[
                {property:nombre, sortable:True, filter:True},
                {property:dateCreated, sortable:True, filter:True},
                {property:paciente, sortable:True, filter:True}
        ]
}

listView empleado_evolucion (EMPLEADO, evolucion){
        model:[
                {property:nombre, sortable:True, filter:True},
```

```
                {property:dateCreated, sortable:True, filter:True},
                {property:paciente, sortable:True, filter:True}
        ]
}
```

Según se puede observar sólo se ha realizado la especificación para el dispositivo desktop (opción por defecto), habilitando en todas las columnas de todos los listados las opciones de ordenación y filtro.

Vamos ahora a ver el diseño de los formularios para el rol EMPLEADO, pues para el rol ADMIN se infieren automáticamente en función de la visibilidad establecida.

```
/*
 * IMPORTANTE: Rol ADMIN tendra generado todos los forms automaticamente (by default)
 */
import clinicaDental.fernandez_traver.*

formView EMPLEADO_paciente (EMPLEADO, paciente){
        readOnly: { edad }
        exclude: { estado }

        fieldSet p_datos {
                legend: "DATOS PERSONALES",
                properties: { nombre, apellidos, nacimiento, edad, dni, nacionalidad,
                        telfijo, telmovil, email }
        }

        fieldSet p_direccion {
                legend: "DIRECCION",
                properties: { calle, numero, cp, poblacion }
        }

        fieldSet p_seguro {
                legend: "DATOS ASEGURADORA",
                properties: { seguro, poliza }
        }

        fieldSet p_contacto {
                legend: "PERSONA O FAMILIAR DE CONTACTO",
                properties: { contacto, contactotelf }
        }
}
formView EMPLEADO_datos (EMPLEADO, datosmedicos){
        exclude: { estado }

        fieldSet d_paciente {
                legend: "PACIENTE",
                properties: { paciente }
        }

        fieldSet d_antecedentes {
                legend: "ANTECEDENTES",
                properties: { alergias, enfermedadesinfec, antecedentes,antfamiliares}
        }

        fieldSet d_higiene {
                legend: "PROCEDIMIENTOS DE HIGIENE",
                properties: { cepillado, freqcepillado, seda, freqseda, colutorio,
                        freqcolutorio, nombrecolutorio }
        }
```

```
        fieldSet d_sintomas {
                legend: "SINTOMATOLOGIA",
                properties: { sintomas, medicacion }
        }
}

formView EMPLEADO_evolucion (EMPLEADO, evolucion){
        exclude: { estado }

        fieldSet e_paciente {
                legend: "PACIENTE",
                properties: { paciente }
        }

        fieldSet e_nombre {
                legend: "EVOLUCION",
                properties: { nombre, descripcion }
        }
}

formView EMPLEADO_radiografia (EMPLEADO, radiografia){
        exclude: { estado }

        fieldSet r_paciente {
                legend: "PACIENTE",
                properties: { paciente }
        }

        fieldSet r_nombre {
                legend: "RADIOGRAFIA",
                properties: { nombre, descripcion }
        }

        fieldSet r_fichero {
                legend: "FICHERO",
                properties: { fichero }
        }
}
```

Vemos como se ha excluido el atributo “estado” en todos ellos, pues sólo se utiliza para establecer el orden de ejecución de los servicios ofrecidos.

Otro aspecto interesante es la aparición de los fieldSets, utilizados para agrupar visualmente aquellos atributos que tengan información en común. Sin embargo, esta opción sólo está soportada de base por Django, en Grails habría que modificar tanto el traductor como las plantillas del framework relacionadas con la técnica del *scaffolding* (ver punto 7.1 – Trabajo futuro).

6.2 Generación de los artefactos y metadatos

Debido a las limitaciones y necesidades arriba comentadas, sólo se han generado los artefactos textuales para el servidor Django y el cliente desktop (ExtJs), según se describe en el punto 4.3.

6.3 Configuración del proyecto de desarrollo

La configuración del proyecto se ha realizado según el punto 4.4.1, configuración en Django.

6.4 Implementación de las extensiones específicas

Según el proceso definido, una vez se han generado y configurado los diferentes artefactos, es el momento de desarrollar la funcionalidad requerida. Ésta solo se ha desarrollado para el framework de destino Django, pues por ahora es el único que soporta los agrupadores de atributos o *fieldSets*.

Para este proyecto ha sido necesario, por una parte, dar soporte al tipo de dato "file", con el fin de poder gestionar las radiografías de un paciente, y por otra parte, implementar algunos de los puntos mencionados en la tabla expuesta al comienzo del apartado 4.5. En concreto, sólo ha sido necesario:

- Implementar la función del atributo derivado "edad" en el fichero *funcAttrDerived.py* (punto 3 – apartado 4.5)

```
#function name generated: write your custom code
def getEdad(self):
        "Calculo de la edad de nacimiento"

        #Please, write the code of this function as is said in the requirement
        #showed above
        today = date.today()
        born = self.nacimiento

        if born == None:
                return ""

        try:
                birthday = born.replace(year=today.year)
        except ValueError:
        # raised when birth date is February 29 and the current year is not a leap year
                birthday = born.replace(year=today.year, day=born.day-1)
        if birthday > today:
                return str(today.year - born.year - 1)
        else:
                return str(today.year - born.year)
```

- Modificar los ficheros CSS e imágenes para adaptarlas a la imagen de la empresa (punto 7 – apartado 4.5)

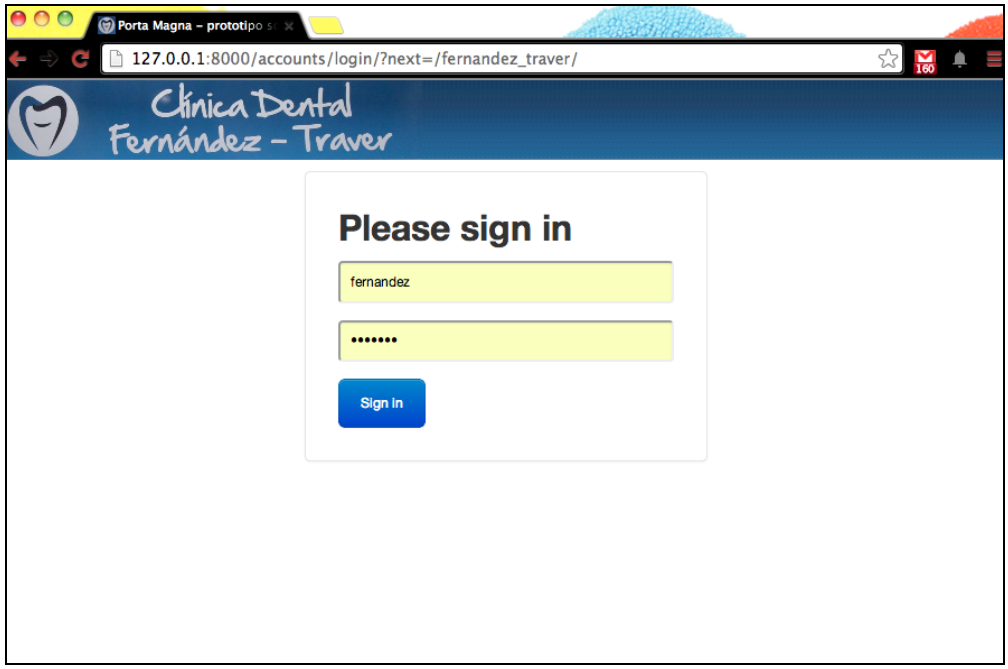


Figura 6.2: página de inicio "Clínica Dental Fernández - Traver"

6.5 Generación del modelo de persistencia

En este caso también se ha decidido utilizar SQLite3 (configuración en el punto 4.6.1 - Django), aunque no se descarta migrar la información a MySQL en función de las necesidades de la empresa.

6.6 Requisitos no funcionales

Al tratarse de un ejemplo real aparecen nuevos requisitos no funcionales que no se habían discutido en el apartado 4.7. Vamos a analizar factores como el tiempo de respuesta, concurrencia y disponibilidad.

Por una parte se ha analizado el ordenador personal (PC) que tienen en la empresa y sobre el cual debe instalarse un servidor de aplicaciones que de acceso a la aplicación desarrollada. Dicho PC es utilizado por la auxiliar contratada, la cual se encuentra en la recepción de la empresa. Actualmente dicho ordenador opera bajo el SO Windows XP y sólo cuenta con 2GB de memoria RAM. Creemos que no es suficiente, aunque se intentará instalar y evaluar el rendimiento para ver si se puede cumplir con un mínimo de calidad en cuando a tiempo de respuesta.

Respecto a la concurrencia, es posible que existan varios accesos múltiples a la aplicación (máx. 2), pues cuentan con un portátil desde el que desean poder conectarse para pasar consulta al paciente.

En cuanto a la disponibilidad, desean únicamente acceso local, por lo que la aplicación sólo será accesible desde la red Wifi de la oficina.

6.7 Conclusiones

El hecho de enfrentarnos al desarrollo de un ejemplo real nos ha exigido, como se comentó al comienzo del capítulo, implementar/analizar ciertas funcionalidades adicionales que no se habían contemplado abordar inicialmente. Estas son:

- Funcionalidad adicional
  - Implementar el soporte al tipo de dato "file"
  - Incluir fechas de creación/actualización de instancias de una clase, de forma transparente al usuario, con el fin de poder hacer análisis estadístico a posteriori
  - Listados (sólo versión desktop)
    - exportación a Excel de la información visible
    - permitir modificar el nivel de paginación

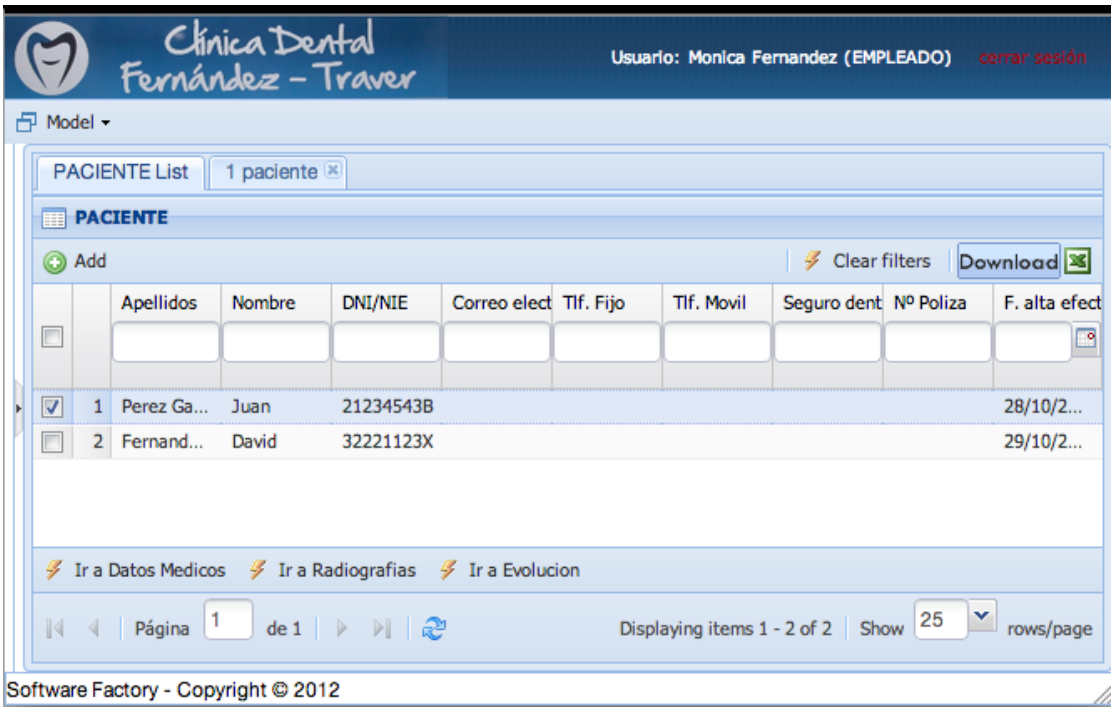


Figura 6.3: funcionalidad adicional en listados

- Seguridad
  - Implementar el cambio de contraseña del usuario conectado a la aplicación (sólo versión desktop)

- Usabilidad
  - Permitir incluir fieldSets[37] en el diseño de los formularios. Se ha modificado la regla *Formview* de la gramática para permitirlo, aunque por el momento sólo esta soportado para Django

[37] fieldSet: agrupador visual en un formulario de los atributos de una clase que tienen información en común.

    - Cambiar el control visual para introducir una fecha mediante un calendario

- Análisis de nuevos requisitos no funcionales
    - Tiempo de respuesta, concurrencia y disponibilidad

Por último vamos a ver un cuadro comparativo con los resultados temporales del proceso de desarrollo de la aplicación, sin tener en cuenta el despliegue de ésta en un servidor de aplicaciones, entre PLESG y un proceso de desarrollo tradicional, centrados ambos en el framework Django.

		Tiempo (horas)	
Fase	Actor	PLESG	Tradicional
F1: Especificación	Analista/ Programador	*2	2
F2: Generación de artefactos y metadatos	Programador	0	**10
F3: Configuración del proyecto	Programador	0.5	0.5
F4: Extensiones	Programador/ Diseñador gráfico	2	2
***F5: Generación de la BD	Programador	0	0
F6: Análisis de los NFR	Programador	0.5	0.5
	TOTAL	4	15

NOTA*: del cómputo total de horas efectivas sólo se ha tenido en cuenta la mitad, pues en dicha fase han participado activamente los gerentes de la clínica dental, aunque con la ayuda de un programador.

NOTA**: cómputo total aproximado, basado en la experiencia profesional y sin contar con posibles errores de programación.

NOTA***: en ambos casos es 0 porque ese servicio lo ofrece el framework de desarrollo ágil.

Tras el análisis de la anterior tabla podemos concluir que la ganancia real estimada es, sólo en este caso, de un x3.75 (37.5%), pero creemos que en proyectos de mayor complejidad podemos llegar a alcanzar los incrementos antes mencionados (x10).

Llegados a este punto sólo queda por establecer las conclusiones generales del trabajo final de master.

Capítulo 7 - Conclusiones

En el presente trabajo final de máster se han realizado las siguientes tareas:

- Se ha analizado el planteamiento del problema, las causas y posibles soluciones (véase capítulo 1)
- Se han analizado las bases teóricas sobre las que construir la presente propuesta (véase capítulo 2)
- Se ha realizado un análisis técnico para decidir sobre qué plataforma tecnológica apoyar la propuesta (véase capítulo 3)
- Se han analizado cada una de las fases definidas en el proceso planteado mediante la adopción de un caso de estudio concreto (véase capítulo 4)
- Se ha desarrollado un prototipo, analizando con detalle cada uno de los pasos necesarios para su construcción (véase capítulo 5)
- Se ha validado el proceso mediante un ejemplo real, analizando la ganancia en cuanto al incremento de la productividad (véase capítulo 6).

Todo ello con el fin de poder cumplir con el objetivo global de la investigación, es decir:

obtener una nuevo proceso (PLESG), que facilite la tarea de crear o modificar una aplicación por parte de los analistas funcionales de pequeñas y/o medianas empresas mediante el uso de un lenguaje específico de dominio que les permita cierta independencia con los expertos informáticos

Con el trabajo realizado (mediante el prototipo desarrollado), el cliente o usuario final del sistema, si tiene los conocimientos suficientes del dominio de análisis, podrá comenzar a trabajar en desarrollos dirigidos por modelos como si de desarrollos tradicionales se tratara. Los artefactos generados irán evolucionando con los cambios de los modelos, evitando tener que regenerar todos los artefactos con cada modificación y adaptándose la generación al estado del sistema en cada momento. Así pues, podrá hacer cambios en su aplicación de forma transparente para él y con el mínimo impacto posible.

Las ventajas derivadas del cumplimiento con el objetivo son claras pero, pese a ello, debe efectuarse un ejercicio de sano escepticismo para cualquier nueva metodología que profetice un cambio dramático y cuestionar cualquier aparente exageración lanzada. También debe cuestionarse si la metodología es de veras efectiva y si el esfuerzo requerido para desarrollar este prototipo ha merecido la pena, pues sólo la implementación de esta propuesta ha exigido un total aproximado de 180 jornadas de desarrollo.

Sin embargo creemos que el resultado ha sido una línea de producción casi totalmente automatizada que podría llevar a esos niveles de incremento de la productividad anteriormente mencionados (x10), aunque todavía hay un

gran camino que recorrer. En el siguiente apartado podemos ver algunas de las diferentes mejoras a contemplar.

7.1 Trabajo futuro

Estamos ante un primer prototipo y aún quedan muchos aspectos por mejorar. Algunas de las líneas de mejora que planteamos, en función del área de actuación, son:

- Global
  1. Seguir elevando el nivel de abstracción, por ejemplo, mediante el uso del PLN [9].
  2. Análisis de las diferentes herramientas que:
     - permitan gestionar las diferentes versiones de cada proyecto de desarrollo (modelos, código extendido, artefactos generados y modificados, etc.)
     - faciliten la migración de la base de datos en futuras evoluciones del software
- Especificación PLESG:
  3. Incluir atributos derivados en los listados: se ha conseguido para Django con tiempo cuadrático pero aún no ha sido posible en Grails
  4. Poder especificar la clave primaria (PK) en una clase, pues actualmente se delega en los frameworks
  5. Adaptar la expresión del valor por defecto de los atributos en función del tipo, para que no hayan errores en los artefactos generados. Por ejemplo, en el caso de una fecha.
  6. Implementar el resto de las restricciones al modelo necesarias, por ejemplo:
     - Uso de parámetros correctos en función del tipo de datos de la propiedad
- Traductores:
  7. Poder configurar la estrategia de generación de tablas en el caso de la relación de herencia (ahora es una tabla por clase) desde algún fichero de configuración del traductor.
  8. Implementar los servicios de ordenación y filtrado en los listados para el dispositivo móvil a partir de las propiedades especificadas en el modelo *listview.model* de PLESG.
  9. Adaptar tanto el traductor como el framework Grails para que soporte los *fieldSets*.
- Interfaz de Usuario:
  10. Realizar los cambios oportunos para permitir aplicaciones multiidioma.
  11. Incluir validación JavaScript en el cliente antes de enviar la información al servidor.

Por último, enumeramos una serie de problemas que han aparecido en la presente propuesta y esperemos sean resueltas en futuras versiones de PLESG y de los diferentes frameworks en uso (Xtext, Django, Grails y Sencha).

7.2 Problemas detectados

- Relativos a Xtext, GGV y traductores

1. Se ha detectado que el orden de carga inicial de los ficheros *.model en los traductores afecta al funcionamiento interno del crosslink. Una posible solución detectada sería crear todos los elementos de los diferentes modelos de análisis en un mismo fichero *.model.

2. El nombre de las acciones en el DTS que aparece en el GGV es un ID interno a Xtext y no el alias correspondiente.

3. La última actualización de Xtext (versión 2.4.x) fuerza a introducir cambios importantes en la gramática que deben de evaluarse.

4. Generalmente falla la primera invocación a un traductor.

- Relativos a Django, Grails y Sencha

5. Diferencias en la versión cliente para móvil

Framework servidor	Sencha Touch	Sencha ExtJs
Django	**2.1.1**	4.1.1a
Grails	**2.2.1**	4.1.1a

Existe una diferencia de versión para Grails debido a que hay un problema a la hora de cargar en el listado inicial las entidades recibidas desde el servidor con la versión 2.1.1 (solo carga la primera entidad). Por ese motivo se actualizó a la versión 2.2.1, que resolvió el problema. Curiosamente al actualizar la parte cliente a la versión 2.2.1 para Django aparece un problema con las CSS y no se muestran las instancias de los objetos en el grid para ninguna de las clases del modelo.

6. Actualización de ExtJs
Versiones superiores de ExtJs presentan problemas con el Plugin *HeaderFilter*, tanto para Grails como para Django. Este plugin ha sido actualizado a mano desde la versión 2.x para adaptarlo a la versión 4.1.1a. y sirve para filtrar instancias; tanto en las navegaciones entre entidades, según se haya especificado en el modelo, como introduciendo un valor directamente.

- Relativos a la aplicación (en tiempo de ejecución)

7. Las navegaciones con filtros
No funcionan la primera vez que se ejecutan para el cliente desktop, es necesario que el grid de destino este cargado previamente.

8. Generación dinámica de los formularios
Grails produce un error al generar cualquier formulario de una clase que herede de una clase legada. Hasta que este error no se subsane no se permitirá la herencia de clases legadas por parte de las clases del modelo de dominio.

Anexo 1: Generic Graph View

Generic Graph View (GGV) [18, 1] es un prototipo disponible bajo los términos del Eclipse Public License v1.0 que utiliza dos DSLs basados en Xtext:

- uno para el mapeo semántico entre los elementos definidos por un DSL (en nuestro caso el DSL utilizado para definir la especificación PLESG) y los elementos gráficos [20] y
- otro para proporcionar los estilos adecuados a esos elementos que queremos visualizar en el visor gráfico.

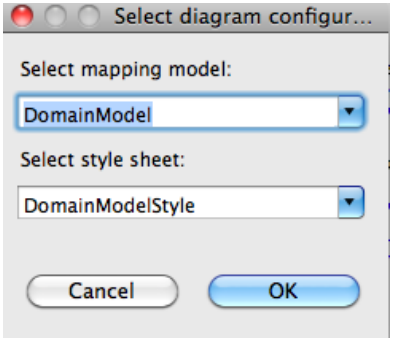
El objetivo ha sido poder visualizar múltiples diagramas a partir de cualquier modelo definido textualmente mediante un DSL basado en Xtext.

En [18, 2] se explica claramente como realizar la integración de GGV en un proyecto Xtext, junto con los requisitos necesarios. En el presente proyecto hemos realizado la integración de GGV con Eclipse 4.2.2 y Xtext 2.3.1. Actualmente GGV ya ofrece soporte para la versión 2.4 de Xtext.

Una vez se ha configurado correctamente GGV dentro del proyecto del lenguaje, es necesario activarlo dentro de la nueva instancia de Eclipse para poder ver gráficamente el diagrama de clases y cada uno de los diferentes DTS de la especificación PLESG. Para ello, desde la barra de tareas de Eclipse, ir a *Window* > *Show View* > *Other* y seleccionar la entrada *Xtext* > *Generic Graph View*. Tras ello éste aparecerá en una nueva pestaña junto con una barra con varias acciones, que puede ubicarse en cualquier zona de eclipse.

barra de acciones disponibles para GGV

El siguiente paso es pulsar sobre la acción de configuración (botón con forma de engranaje) y seleccionar el tipo de diagrama a mostrar junto con el estilo correspondiente según se muestra en la siguiente imagen.



selección del tipo de diagrama a mostrar

Tras pulsar el botón *OK*, deberemos abrir el fichero *.model correspondiente y situar el cursor al comienzo del diagrama a mostrar. El resto de acciones que aparecen no son relevantes, sin embargo vamos a

comentar ciertas operaciones que aparecen al seleccionar alguna de las figuras o componentes gráficos que aparecen en el visor GGV.

Al seleccionar un componente en el visor, por ejemplo una clase correspondiente al diagrama de clases o un estado relativo al DTS, es posible realizar varias operaciones:

- mostrar los elementos ocultos propios del componente seleccionado mediante el botón [+]. Por ejemplo, mostrar las propiedades y/o servicios ocultos de una entidad o las acciones ocultas que tenga un estado concreto de una clase.
- ocultar cualquier componente visible mediante el botón **x**
- mostrar otros componentes ocultos mediante el botón **+**, y que están relacionados con el componente seleccionado. Por ejemplo, mostrar las entidades no visibles que están relacionadas mediante una asociación, relación de herencia, etc., con la clase seleccionada o mostrar el resto de estados propios de una clase si no están visibles.

ejemplo de acciones visibles al seleccionar una entidad

Una vez mostrados los elementos ocultos de un componente tras pulsar el botón [+] , es posible volverlos a ocultar mediante el botón **x** que aparece en la parte superior derecha.

ejemplo de acciones visibles al seleccionar las propiedades de una entidad

Existen ciertos problemas a la hora de mostrar el nombre de las acciones correspondiente en un estado de un DTS dentro del visor GGV. En vez de mostrar el nombre, GGV muestra un ID interno, y es por ese motivo por el cual no se muestran las acciones por defecto (punto 7.2).

Anexo 2: Gramática

A continuación podemos observar la gramática desarrollada, basada en la notación Extended Backus-Naur Form-like (EBNF), necesaria para crear el DSL utilizado en la especificación PLESG.

```
grammar org.xtext.portaMagna.Dsl with org.eclipse.xtext.common.Terminals
generate dsl "http://www.xtext.org/portaMagna/Dsl"
import "http://www.eclipse.org/emf/2002/Ecore"

Model:
        (elements += AbstractElement)*
;

AbstractElement:
        PackageDeclaration | Type | Import | Config
;

Config:
        'projectName' projectName = ID &      // Proyecto
        'appName' appName = ID &              // Nombre de la aplicacion (app)
        'version' version = INT ('.' INT)? &  // Version de la app
        'idiom' idiom = IDIOM                 // Idioma de la app
;

PackageDeclaration:
        'package' name = QualifiedName '{'
                (elements += AbstractElement)*
        '}'
;

QualifiedName:
        ID ('.' ID)*
;

Import:
        'import' importedNamespace = QualifiedNameWithWildcard
;

QualifiedNameWithWildcard:
        QualifiedName '.*'?
;

Type:
        DataType | EnumType | Legacy | Entity | StateTransition | Rol | RolView |
FormView | ListView
;

/* TIPOS SIMPLES */
DataType: 'datatype' name = ID;

/* TIPOS ENUMERADOS */

EnumType:
        'enum' name = ID '{'
                enumFields+=EnumField (SEP enumFields+=EnumField)*
        '}'
;

EnumField:
        '('id = INT SEP nombre = STRING SEP alias = STRING ')'
```

```
;

/* ENTIDADES LEGADAS (ejemplo: User) */
Legacy:
	'legacy' name = ID
;

/* ENTIDADES */
extendedClass:
	entity=[Entity|QualifiedName]
;

Entity:
	(abstract?='abstract')? 'entity' name=ID ('extends' superType=extendedClass)?
'{'
		(attributes+=Attribute)*
		(derivatives+=AttrDerived)*
		(enumerators+=AttrEnum)*
		(relations+=Relation)*
		(services+=Service)*

		//Es el metodo toString() para una clase de Java: (ID por defecto)
		(identity?='identity' ':' '{'
			attribute+=[Attribute] (SEP attribute+=[Attribute])*
		'}')?

		/* INFO para mejorar los Traductores */
		('hasSTD' ':' hasSTD?='True' '{'
			'field' ':' fieldSTD=[AttrDerived] SEP
			'alias' ':' aliasSTD=STRING
		'}')?

		('hasSessionProperty' ':' hasSessionProperty?='True' '{'
			'sessionProperty' ':' sessionProperty=[EObject]
		'}')?
	'}'
;

/* ENTIDADES: ATTRIBUTES & VALIDATIONS */
AttrEnum:
	type=[EnumType] name=ID '(' parameters+=attrParameter (SEP
parameters+=attrParameter)* ')'
;

AttrDerived:
	name=ID '(' parameters+=attrParameter (SEP parameters+=attrParameter)*  (SEP
'isSTDField' ':' isSTDField?=BOOL ')' | (SEP 'refOtherClass' ':' refOtherClass?=BOOL)?
')' '<=' 'function' functionName=ID '(' functionComment=STRING ')')
;

Attribute:
	type=[DataType] name=ID '(' parameters+=attrParameter (SEP
					parameters+=attrParameter)* ')'
;

attrParameter:
	token='widget' ':' widget=WIDGET | token=TokenS ':' valueS=STRING |
	token=TokenI ':' valueI=INT | token=TokenB ':' valueB=BOOL |
	token=TokenFV ':' '{'
		validations+=attrValidation (SEP validations+=attrValidation)*
	'}'
;

attrValidation:
	name=ID ('(' comment=STRING ')')? ':' type = [DataType]
```

```
;


/* ENTIDADES: RELATIONS */

Relation: name=ID
(
 (relation='oneToOne' | relation='manyToOne') 'legacy' legacy=[Legacy|QualifiedName] |
 (relation='oneToOne' | relation='manyToOne' |
  relation='manyToMany' | relation= 'hasMany') entity=[Entity|QualifiedName]
) ('(' parameters+=relParameter (SEP parameters+=relParameter)* ')')?
;

relParameter:
        token=TokenBRelation ':' valueB=BOOL |
        token='through' ':' entity=[Entity|QualifiedName] |  // ← ManyToMany
        token='related' ':' property=STRING                  // ← ManyToMany REFLEXIVA
;

/* ENTIDADES: SERVICES */
Service:
        name=ID ('(' parameters+=servParameter (SEP parameters+=servParameter)* ')')?
':' type = [DataType]
;

servParameter:
        token='alias' ':' value=STRING |
        token='type' ':' value=("C" | "U" | "D" | "B") |
        token='customForm' ':' form=[FormView] |
        token=TokenBServ ':' valueB=BOOL
;

/* STATE TRANSITION */
StateTransition:
  'std' entity=[Entity|QualifiedName] '{'
        (transIni=Transition)
        (states+=State)*
 '}'
;

State:
  (initial='initial')? 'state' name=ID '{'
         /* actions = servicios que NO cambian el estado de la clase */
     ('actions' '{' actions+=Action (SEP actions+=Action)* '}')?
     (transitions+=Transition)*
  '}'
;

Transition:
        action=Action '=>' state=[State]
;

Action:
        service=[Service]
;


/* ROL */
Rol:
        'rol' name = ID '{'
                'type' ':' type=RolType  SEP
                'initialView' ':' entity=[Entity|QualifiedName]
                (SEP 'referencedClass' ':' refclass=[Entity|QualifiedName])?
        '}'
;
```

```
/* ROL VISIBILITY */
RolView:
	'rolView' name=ID  '(' rol=[Rol] SEP entity=[Entity|QualifiedName] (SEP
'classview' ':' classview='False')? (SEP 'classfilter' ':' classfilter=[EObject])? ')'
'{'
	('properties'  ':' '{' ( allProperties?='all' | properties+=declaredRolProperty
					(SEP properties+=declaredRolProperty)* ) '}' )?
	('services'    ':' '{' ( allServices?='all'   | services  +=declaredRolService
					(SEP services  +=declaredRolService)*  ) '}' )?
	('navigation'  ':' '[' navigation+=declaredRolNavigation
				(SEP navigation+=declaredRolNavigation)* ']' )?
'}';

declaredRolProperty: ref=[EObject];

declaredRolService: ref=[Service];

declaredRolNavigation:
	'{'
		ref=[Entity|QualifiedName] SEP
		text=STRING SEP
		('filter' ':' '{'
			'from'  ':' from=[EObject] SEP
			'to'    ':' to=[EObject]
		'}')?
	'}'
;


/* DESIGN (Form) VIEW */
FormView: 'formView' name=ID '(' rol=[Rol] (SEP model=[Entity|QualifiedName])? ')' '{'
	// model = null		--> formulario de servicio, CUSTOM
	// model = [entity]	--> formulario de instancia de clase

	// 1. (opcional) Atributos PROPIOS DEL FORMULARIO
	(formAttributes+=Attribute)*

	// 2. ReadOnly Properties: hay que incluir explicitamente
	// los attr q tienen el param. editable:false
	(roProperties=FormRoProperties)?

	// 3. Exclude Properties, que NO iran en el fieldSet
	(exProperties=FormExProperties)?

	// ------ DISENYO IU (AZUCAR) --------
	//(opcional) Grails solo ofrece un fieldset para todos los campos
	(fieldSets+=FieldSet)*
'}';

declaredViewProperty:  ref=[EObject];

FormRoProperties:
	type='readOnly' ':' '{'
		properties+=declaredViewProperty (SEP
		properties+=declaredViewProperty)*
	'}'
;

FormExProperties:
	type='exclude' ':' '{'
		properties+=declaredViewProperty (SEP
		properties+=declaredViewProperty)*
	'}'
```

```
;

FieldSet:
        'fieldSet' name=ID '{'
                'legend' ':' legend=ID SEP
                'properties' ':' '{'
                        properties+=declaredViewProperty (SEP
                        properties+=declaredViewProperty)*
                '}'
        '}'
;

/* DESIGN (List) VIEW: */
ListView:
        'listView' name=ID '(' rol=[Rol] SEP entity=[Entity|QualifiedName] (SEP
device=('mobile' | 'desktop'))? ')' '{'
                (model=ListViewModel)?
        '}'
;

ListViewModel:
        'model' ':' '[' ( allModel?='all' | properties+=declaredListViewProperty (SEP
                                properties+=declaredListViewProperty)* ) ']'
;

declaredListViewProperty:
        '{'
                'property' ':' ref=[EObject]
                (SEP 'sortable' ':' sortable=BOOL)?
                (SEP 'filter' ':' filter=BOOL)?
        '}'
;

/* Lexer rules only below here*/
TokenS: 'alias' | 'help' | 'default';
TokenB: 'mandatory' | 'unique' | 'editable' | 'sortable' | 'filter';
TokenFV: 'validation' | 'function';
TokenI: 'max_length' | 'max_digits' | 'decimal_places';
TokenBRelation:
        'symmetrical' |         // para relaciones ManyToMany reflexivas (default True)
        'unique' |              // --> restriccion
        'null' |                // --> permite null
        'blank'                 // --> permite introducir cadena vacia
;

TokenBServ:
        'isModal' |             //dafault FALSE
        'refreshParent'         //default TRUE
;

terminal SEP: ',' (WS)?;

enum RolType: admin = "admin" | user = "user";
enum BOOL: False = "False" | True = "True";
enum IDIOM: es = "ES" | fr = "FR" | en = "EN";
enum WIDGET: textarea = "widgetTextarea" | date = "widgetDate";
```

Anexo 3: Tipos de datos

Aunque el DSL permite declarar un atributo como un tipo de dato (DataType), sólo los tipos de datos; simples o complejos, que pueden verse reflejados en la primera columna de las siguientes dos tablas son soportados actualmente por PLESG. Las otras dos columnas muestran el mapeo al tipo correspondiente en función del framework de destino seleccionado.

Los tipos de datos simples se muestran en la siguiente tabla.

Tipo de dato simple		
PLESG	Django	Grails
string	CharField	String
integer	IntegerField	Integer
decimal	DecimalField	BigDecimal
boolean	BooleanField	Boolean
date	DateField	Date
void	-	void

Los tipos de datos complejos se muestran en esta segunda tabla.

Tipo de dato complejo		
PLESG	Django	Grails
email	EmailField	String con restricción (email:true)
file*	FileField	String

Actualmente PLESG no da soporte completo al tipo de dato "*file*", para el cual es necesario realizar ciertas configuraciones y desarrollos manuales (ver la documentación específica de cada framework).

En función del tipo de dato al que pertenezca un atributo es posible definir argumentos genéricos y/o específicos. Veamos a continuación cuales son esos argumentos a declarar en cada atributo.

2.1 Argumentos genéricos

Los siguientes argumentos están disponibles para cualquier atributo, independientemente del tipo de dato que sea, siendo algunos obligatorios (+). Podemos dividirlos en tres grupos;

- Restricciones en la base de datos
- Restricciones de validación en el servidor: debe cumplir con alguna/s función/es de validación, longitud máxima, número de decimales (si es de tipo decimal), etc.
- Restricciones sobre la IU:
    - Sobre el formulario:
        - sólo lectura,
        - información básica; alias, texto de ayuda, etc.
    - Sobre el listado/grid: ordenable y/o filtrable

En la siguiente tabla vemos los argumentos que afectan al comportamiento de la base de datos.

Restricción de DB	Descripción	Django	Grails
null	Permite que la propiedad pueda ser establecida a null	null	nullable
	ejemplo	age(null:true)	age(nullable:true)
unique	Establece que el atributo debe ser único a nivel de tabla de BD	unique	unique
	ejemplo	dni(unique:true)	dni(unique:true)

En la siguiente tabla vemos los argumentos que establecen restricciones de validación en el servidor.

Restricciones validación en el Servidor	Descripción	Django	Grails
mandatory	Valida que el valor del dato no sea vació	blank	blank
validation	Permite establecer validaciones específicas al atributo	validators	validator

Aunque a priori pueda no ser una buena práctica incluir información sobre la IU en el modelo de dominio (diagrama de clases), es necesario si se hace un uso exhaustivo de la técnica de *scaffolding* o generación dinámica de formularios.

Restricciones visuales sobre la IU	Descripción	Django	Grails
*alias+	Define el alias para el atributo en el formulario (**obligatorio**)	alias (en el Modelo del Dominio – MD)	label (en i18n/messages_XX. Properties)
*help	Texto de ayuda del atributo visible para el usuario	help_text (en el MD) fecha=DateField(" F. nacimiento", help_text:"**El formato debe ser dd/mm/yyyy**")	help (i18n/messages_XX. properties) adminPrestamo.fecha.help=**El formato de la fecha debe ser dd/mm/yyyy**
*editable	Establece que el campo sea de solo lectura. Afecta a todos los roles → mejor especificar la propiedad ReadOnly en el formulario específico a cada rol	**readonly** (*en el Modelo de Diseño de Formularios*)	editable (en el MD)
*widget Dos tipos: - date - textarea	Permite adaptar el componente visual que va asociado al tipo de dato del atributo según las necesidades de la IU.	widget (*en el Modelo de Diseño de Formularios*)	widget (en el MD)

	Por ejemplo, el atributo “descripcion” de tipo String queremos que se visualice en un componente “TextArea” y no en un campo de texto normal. String desc (widget:textarea)	MD: Desc = CharField(“Desc.”) MDF: desc **=** CharField (widget**=**Textarea)	MD: String desc Constrainst [desc (widget: 'textarea')]

Como se puede observar en la tabla, existen algunas diferencias (*) que comentamos a continuación:

- El alias debe ser el primer argumento y es el único obligatorio, sin embargo en Grails tanto el alias como el texto de ayuda de los atributos (este último opcional) quedará especificado fuera del modelo de dominio, exactamente se especificará manualmente en el fichero messages_XX.properties, donde XX es el idioma declarado en la configuración de la aplicación.

- Debido a las diferencias entre los frameworks soportados por la especificación PLESG (Django sólo permite el uso del argumento editable y widget en el MDF), si se utiliza el argumento editable:false en el MD es necesario incluir manualmente dicha propiedad dentro del parámetro readOnly en el modelo del diseño de formularios. Sin embargo el uso del argumento widget será procesado automáticamente por el traductor Django, el cual hará la adaptación necesaria.

2.2 Argumentos específicos

Los siguientes argumentos son específicos y obligatorios al tipo de dato que sea el atributo, y establecen restricciones de validación en el servidor:

		Framework	
Tipo de dato	Argumentos específicos PLESG	Django	Grails
decimal	max_digits	max_digits	size*
	decimal_places	decimal_places	scale
string	max_length	max_length	size*
email			

Dichos argumentos específicos pueden combinarse adicionalmente junto con los ya expuestos arriba, a excepción de una limitación a tener en cuenta respecto a Grails:

- actualmente “size” no puede ser usado junto con “blank” o “null”. Si se desean usar éstos últimos, es posible utilizar el parámetro “validator” y que implemente la restricción que debería hacer “size”.

Finalmente podemos concluir que, si bien existen más argumentos en ambos frameworks, hemos considerado los mínimos argumentos imprescindibles para una primera versión de PLESG.

Anexo 4: Asociaciones

En los sucesivos puntos del presente anexo veremos en varias tablas cuales son las asociaciones permitidas entre las entidades definidas dentro del diagrama de clases y su correspondiente mapeo al framework de destino. Aunque dichas tablas parezcan muy complejas sólo muestran los tipos de asociaciones permitidas en PLESG, una en cada tabla, las cuales resumimos aquí:

Tipo		Cardinalidad
oneToOne		1 - 1: uno a uno
manyToOne		n - 1: muchos a uno
manyToMany	hasMany[38]	n - n: muchos a muchos

Cualquiera que haya trabajado anteriormente con Django podrá pensar que la especificación PLESG es muy parecida a éste en lo que se refiere a asociaciones. De hecho, PLESG toma muchas consideraciones de Django por su simplicidad. Por ejemplo, según se observa en la anterior tabla, si se desea especificar una relación de cardinalidad 1 – n ("uno a muchos") se deberá utilizar su homologa inversa. Como se sabe, Django no da soporte a ese tipo de relaciones y PLESG a optado por no ofrecerla.

Cuando se definan clases legadas, las únicas asociaciones posibles entre estas y las clases propias del sistema son oneToOne y manyToOne. Tampoco tiene sentido establecer relaciones reflexivas sobre este tipo de clases.

Los parámetros de la siguiente tabla son aplicables en cualquier tipo de relación, independientemente de la cardinalidad, y permiten establecer restricciones y/o configurar de forma adecuada la relación entre las clases correspondientes, o entre una misma (relación reflexiva).

Parámetro	Tipo	Restricción
unique	booleano	Permite establecer que la asociación sea única a nivel de tabla base de datos (por defecto false). **Obligatorio cuando la relación es oneToOne** [**unique:true]**
null	booleano	Posibilidad de establecer una asociación a null (por defecto false)
blank	booleano	Posibilidad de establecer una asociación a cadena vacía (por defecto false)
related	string	La propiedad que define "related" será tratado como el sentido inverso de la asociación. **Obligatorio cuando la relación es reflexiva**

Los siguientes parámetros sólo son aplicables al tipo de relación manyToMany.

Parámetro	Significado
through	Es un parámetro obligatorio y establece el nombre de la entidad intermedia que aparece en cualquier relación muchos a muchos

[38] hasMany: es la relación inversa a manyToMany (bidireccional)

symmetrical	Establece si una relación muchos a muchos es simétrica (por defecto true). Obligatorio cuando la relación es reflexiva, y su valor debe ser “false” obligatoriamente.

Adicionalmente se ha considerado que todas las asociaciones sean unidireccionales a excepción de la relación *manyToMany*. Para los expertos en Grails, esto tiene efectos importantes, pues en dicho framework la relación oneToOne es forzosamente bidireccional. Para hacerla unidireccional, se fuerza a utilizar la relación propia del framework, *belongsTo,* junto con el parámetro “unique = true”. Esta solución afecta directamente a la política de borrado en cascada, que se detalla a continuación.

Ambos frameworks permiten controlar la política de borrado en cascada entre las asociaciones donde existe una relación maestro – detalle. Sin embargo la especificación PLESG no la gestiona por motivos de incompatibilidad entre los frameworks soportados, y es que:

- Django tiene configurado por defecto el borrado en cascada desde la clase maestra hacia la clase detalle, siendo la clase donde se establece la asociación la clase “detalle”. Mediante el uso del parámetro *on_delete* es posible modificar el tipo de borrado en cascada (nothing, set null, etc).

- Grails no tiene establecido un borrado en cascada por defecto, éste depende de la combinación de ciertos parámetros y asociaciones:

  1. El uso de la asociación *belongsTo* (claseA “belongsTo” claseB) junto con *hasOne/hasMany* (claseB hasOne/hasMany A) establece automáticamente la política de borrado en cascada desde la clase B (maestra) hacia la clase A (detalle), y nada desde la clase A hacia B. Pero la dirección del borrado en cascada cambia si no se utiliza la asociación *hasOne/hasMany*, es decir, la asociación donde aparece *belongsTo* se considera ahora la clase maestra y ésta queda desactivada (este es el caso para las asociaciones oneToOne/manyToOne en Grails)

  2. El parámetro cascade:[all, none, lock,…] permite controlar deforma manual el borrado en cascada.

  En consecuencia, debido a estas diferencias, hay que tener en cuenta que PLESG delega la política de borrado en cascada al framework de destino, por lo que los programadores deberán actuar en consecuencia, teniendo en cuenta que:

  - django tiene activado por defecto el borrado en cascada desde la clase maestra hacia la clase detalle.

- en grails, la dirección del borrado en cascada para oneToOne/manyToOne ha quedado desactivado[39]. Si se desea que el borrado en cascada sea como en Django, la solución es que se programe manualmente.

Es hora de ver con detalle cada una de las asociaciones descritas anteriormente en PLESG y su transformación en cada uno de los frameworks soportados.

3.1 Asociación one-to-one

<table>
<tr><th>UML</th><th>PLESG</th><th colspan="2">Frameworks</th></tr>
<tr><td rowspan="2">A one to one B
(1 — 1)</td><td rowspan="2">A {
x oneToOne B (unique:True
[,*params])
}

Notas:
1. unique:True (obligatorio)
2. B LEGACY (opcional)

<u>Ejemplo:</u>
Prestamo {
implica oneToOne Libro(
unique:True,null:False)
}</td><td>Django</td><td>a:
x = models.OneToOneField(b,unique[,*params])

<u>ejemplo</u>
prestamo:
implica = models.OneToOneField(libro, unique:True,
null:False)</td></tr>
<tr><td>Grails</td><td>A {
static belongsTo = [x:B]
static constraits = {
x (unique:true)
}
}

<u>ejemplo</u>
Prestamo {
static belongsTo = [implica: Libro]
static constraints = {
implica (unique:true, null:False)
}
}</td></tr>
<tr><td rowspan="2">A one to one A
REFLEXIVA</td><td rowspan="2">A {
x oneToOne A (unique:True,
related:”none”[,*params])
}

Notas:
1. unique:True (obligatorio)
2. related:”none” (oblig.)
<u>-related en Django</u>
“none” => “+”
<u>-related en Grails</u>
“none” => “none”

<u>Ejemplo:</u>
Persona {
Estado situacion
pareja oneToOne Persona(
unique:True,related:“none”
,blank:true)
}</td><td>Django</td><td>a:
x = models.OneToOneField(‘self’,unique:True,
related_name=”+”[,*params])

<u>ejemplo</u>
persona:
pareja = models.OneToOneField(‘self’, unique:True,
related_name=”+”, blank:True)</td></tr>
<tr><td>Grails</td><td>A {
static belongsTo = [x:A]
static constraits = {
x (unique:true)
}
static mappedBy = [x:”none”]
}

<u>ejemplo</u>
Persona {
static belongsTo = [pareja:Persona]
static constraints = {
pareja (unique:true, blank:true)
}
static mappedBy = [pareja:”none”]
}</td></tr>
</table>

Siguiendo con el ejemplo de la gestión de prestamos bibliotecarios, para el caso del borrado en cascada de un libro y su préstamo, este no se permite

[39] Advertencia: en caso de activarla con el parámetro cascade:”all”, la dirección de borrado sería contraria a Django.

según la especificación (no es visible el servicio de borrado de un libro que ha sido prestado). Si finalmente se permitiera el borrado en cascada del libro y el préstamo asociado, habría que programarlo en Grails.

3.2 Asociación many-to-one

<table>
<tr><th>UML</th><th>PLESG</th><th colspan="2">Frameworks</th></tr>
<tr><td rowspan="2">A many to one B
(A n — 1 B)</td><td rowspan="2">A {
x manyToOne B ([*params])
}

NOTA:
1. si se pone “unique:True” es una relacion 1 — 1.
2. B LEGACY (opcional)

Ejemplos:

1. prestamo {
pertenece manyToOne Cliente
}

2. persona {
loginas manyToOne User (
blank:True)
}

Un usuario puede actuar en la misma aplicación con roles diferentes (Cliente y/o Staff) Siendo la misma clave en ambos. Al entrar en la aplicacion se debera escoger el rol deseado.</td><td>Django</td><td>a:
x = models.ForeignKey(b [,*params])

ejemplo 1
prestamo:
pertenece = models.ForeignKey(cliente)</td></tr>
<tr><td>Grails</td><td>A {
static belongsTo = [x:A]
static constraits = {
x ([,*params])
}
}

ejemplo 1
Prestamo {
static belongsTo = [pertenece:Cliente]
static constrainst = {
pertenece()
}
}

ejemplo 2
Persona {
static belongsTo = [loginas:User]
static constrainst = {
loginas (blank:true)
}
}</td></tr>
<tr><td rowspan="2">A many to one A
REFLEXIVA</td><td rowspan="2">A {
x manyToOne A (related
[,*params])
}

NOTA:
1. si se pone “unique:True” es una relacion 1 — 1.
2. related:”string” (oblig.)
-related en Django
“string” => “string”
-related en Grails
“string” => “none”

Ejemplo:
Persona {
works manyToOne Persona(
related:“manager”, blank:true)</td><td>Django</td><td>a:
x = models.ForeignKey(‘self’, related_name:”string”
[,*params])

ejemplo
persona:
works = models.ForeignKey(‘self’,
related_name:”manager”, blank:True)</td></tr>
<tr><td>Grails</td><td>A {
static belongsTo = [x:A]
static constraits = {
x (unique:true)
}
static mappedBy = [x:”none”]
}

ejemplo
Persona {
static belongsTo = [works:Persona]
static constraints = {
works (blank:true)
}
static mappedBy = [work:”none”]
}</td></tr>
</table>

Puede observarse como el parámetro *related*, exclusivo de las relaciones reflexivas, se establece siempre a “none” en Grails para oneToOne y manyToOne. Esto cambia en las relaciones reflexivas manyToMany.

3.3 Asociación many-to-many

En la especificación PLESG, cuando se define una relación many-to-many, es necesario definir la tercera entidad que formará parte de la relación. Con esa nueva entidad se establecen cuatro asociaciones; dos manyToOne y una manyToMany junto con su inversa hasMany, según se detalla en la siguiente tabla.

<table>
<tr><th>UML</th><th>PLESG</th><th colspan="2">Frameworks</th></tr>
<tr><td rowspan="2">A manyTomany B
(A n — n B)</td><td rowspan="2">(A 1-n C n-1 B)

A {
1. w = manyToMany B(through C)
}

B {
2. x = hasMany C
}

C {
3. y = manyToOne A
4. z = manyToOne B
}

Notas:
1. through (obligatorio)

ejemplo
User {
rol manyToMany Rol(through UserRol)
}

Rol {
user hasMany UserRol
}

UserRol {
user manyToOne User
rol manyToOne Rol
}</td><td>Django</td><td>a:
1. w = models.ManyToManyField(b,c)

c:
3. y = models.ForeignKey(A)
4. z = models.ForeignKey(B)</td></tr>
<tr><td>Grails</td><td>A {
1. static hasMany = [w:C]
}

B {
2. static hasMany = [x:C]
}

C {
3,4. static belongsTo = [y:A, z:B]
}</td></tr>
<tr><td rowspan="2">A manyTomany A
REFLEXIVA</td><td rowspan="2">Notas:
1. related:”string” (obligatorio)
2. simmetrical:False (obligatorio)

ejemplo
Airport {
out manyToMany Airport(through F,
related:“departure”)

in hasMany Flight(
related:“destination”)
}

Flight {
departure manyToOne Airport
destination manyToOne Airport
}</td><td>Django</td><td>Airport {
out = models.ManyToManyField(‘self’, Flight,
symmetrical:False, related_name:”not_departure”)
}

Flight {
departure = models.Foreignkey(A)
destination = models.Foreignkey(A)
}</td></tr>
<tr><td>Grails</td><td>Airport {
hasMany [out:Flight, in:Flight]
mappedBy[
out:”departure”,
in:”destination”]
}

Flight {
static belongsTo = [departure:A, destination:A]
}</td></tr>
</table>

Consideraciones a Django:

- al no existir la relación inversa *hasMany,* cuando la relación manyToMany utilice el parámetro *related* se ha decidido, por simplicidad, establecerlo a su negación, para poder realizar la consulta inversa.

- la relación manyToMany utiliza por defecto un parámetro llamado “symmetrical” cuyo valor por defecto es “true”. Sin embargo en las relaciones reflexivas se debe de establecer a “false”, y es que según el ejemplo mostrado, en un aeropuerto, el número de vuelos entrantes puede ser diferente del número de vuelos salientes.

Consideraciones a Grails:

- la política de borrado en las relaciones manyToMany se establece automáticamente en “cascada” desde la clase maestra (hasMany) hacia la clase detalle (belongsTo), y “nada” desde la clase detalle hacia la clase maestra.

Anexo 5: Restricciones al modelo (código)

Las restricciones al modelo se deben de implementar en el fichero *DslJavaValidator.java* ubicado en el proyecto del lenguaje. Concretamente se han implementado dos restricciones:

1. La siguiente restricción imposibilita incluir dos veces un mismo parámetro en relaciones, servicios, atributos, atributos enumerados y atributos derivados.

```
@Check
public void checkNotParamenterRepeated(Relation rel){
	Set<String> tokens = new HashSet<String>();
	Boolean inserted = false;
	for(relParameter param: rel.getParameters()){
		try{
			inserted = tokens.add(param.getToken());
			if(!inserted)
				error("No es posible incluir dos parametros iguales",
param.eContainingFeature());
			}catch(Exception e){
				error("No es posible incluir dos parametros iguales",
param.eContainingFeature());
		}
	}
}

@Check
public void checkNotParamenterRepeated(Attribute obj){
	Set<String> tokens = new HashSet<String>();
	Boolean inserted = false;
	for(attrParameter param: obj.getParameters()){
		try{
			inserted = tokens.add(param.getToken());
			if(!inserted)
				error("No es posible incluir dos parametros iguales",
param.eContainingFeature());
			}catch(Exception e){
				error("No es posible incluir dos parametros iguales",
param.eContainingFeature());
		}
	}
}

@Check
public void checkNotParamenterRepeated(AttrEnum obj){
	Set<String> tokens = new HashSet<String>();
	Boolean inserted = false;
	for(attrParameter param: obj.getParameters()){
		try{
			inserted = tokens.add(param.getToken());
			if(!inserted)
				error("No es posible incluir dos parametros iguales",
param.eContainingFeature());
			}catch(Exception e){
				error("No es posible incluir dos parametros iguales",
param.eContainingFeature());
		}
	}
}
```

```
@Check
public void checkNotParamenterRepeated(AttrDerived obj){
	Set<String> tokens = new HashSet<String>();
	Boolean inserted = false;
	for(attrParameter param: obj.getParameters()){
		try{
			inserted = tokens.add(param.getToken());
			if(!inserted)
				error("No es posible incluir dos parametros iguales",
param.eContainingFeature());
			}catch(Exception e){
				error("No es posible incluir dos parametros iguales",
param.eContainingFeature());
		}
	}
}

@Check
public void checkNotParamenterRepeated(Service obj){
	Set<String> tokens = new HashSet<String>();
	Boolean inserted = false;

	for(servParameter param: obj.getParameters()){
		try{
			inserted = tokens.add(param.getToken());

			if(!inserted)
				error("No es posible incluir dos parametros iguales",
param.eContainingFeature());
			}catch(Exception e){
				error("No es posible incluir dos parametros iguales",
param.eContainingFeature());
		}
	}
}
```

2. La siguiente restricción implementada especifica los parámetros que no son válidos en la definición de un atributo derivado.

```
@Check
public void checkValidParamenter(AttrDerived obj){

		for(attrParameter param: obj.getParameters()){

			if(param.getToken().equalsIgnoreCase("mandatory") ||
			   param.getToken().equalsIgnoreCase("unique") ||
			   param.getToken().equalsIgnoreCase("editable") ||
			   param.getToken().equalsIgnoreCase("sortable") ||
			   param.getToken().equalsIgnoreCase("filter") ||
			   param.getToken().equalsIgnoreCase("widget") ||
			   param.getToken().equalsIgnoreCase("validation") ||
			   param.getToken().equalsIgnoreCase("function") ||
			   param.getToken().equalsIgnoreCase("max_digits") ||
			   param.getToken().equalsIgnoreCase("max_length") ||
			   param.getToken().equalsIgnoreCase("decimal_places"))
			{
				error("Parámetro no válido. Ver documentación",
param.eContainingFeature());
			}

		}
}
```

Anexo 6: Manejadores (código)

A continuación podemos ver el código completo del manejador *GenerationHandlerGrails.java,* desde el cual se invoca al método doGenerate(resource,fsa) del traductor Grails pasando los argumentos requeridos.

```
@InjectWith(DslInjectorProviderGrails.class)
public class GenerationHandlerGrails extends AbstractHandler implements IHandler {

private IGenerator generator;

@Inject
private Provider<EclipseResourceFileSystemAccess> fileAccessProvider;

@Inject
IResourceDescriptions resourceDescriptions;

@Inject
IResourceSetProvider resourceSetProvider;

@Override
public Object execute(ExecutionEvent event) throws ExecutionException {

ISelection selection = HandlerUtil.getCurrentSelection(event);
  if (selection instanceof IStructuredSelection) {
   IStructuredSelection structuredSelection = (IStructuredSelection) selection;
   Object firstElement = structuredSelection.getFirstElement();
   if(firstElement instanceof PackageFragment){
       try {
        PackageFragment pg = (PackageFragment) firstElement;
        IProject project = (IProject) pg.getJavaProject().getProject();
        IFolder srcGenFolder = project.getFolder("src-gen");
        if (!srcGenFolder.exists()) {
          try {
            srcGenFolder.create(true, true, (IProgressMonitor) new
NullProgressMonitor());
          } catch (CoreException e) {
            return null;
          }
        }

        final EclipseResourceFileSystemAccess fsa = fileAccessProvider.get();
        fsa.setOutputPath(srcGenFolder.getFullPath().toString());
        XtextResourceSet set = (XtextResourceSet) resourceSetProvider.get(project);
        set.addLoadOption(XtextResource.OPTION_RESOLVE_ALL, Boolean.TRUE);

        Object[] files = pg.getNonJavaResources();
        Resource resource = null;

        //EL ORDEN ES IMPORTANT
        for(Object obj : files){
          if (obj instanceof IFile) {
               IFile fileName = (IFile) obj;

               if(fileName.getFileExtension().toLowerCase().equals("model")){
                   if (resource == null) {
                       resource = set.getResource(URI.createURI(
fileName.getFullPath().toString()), true);
                   } else {
                       Resource modelResource = set.getResource(URI.createURI(
```

```
fileName.getFullPath().toString()), true);
                        resource.getContents().addAll(modelResource.getContents());
                    }
        } } }

        DslInjectorProviderGrails injectorProvider = new DslInjectorProviderGrails();
        Injector injector = injectorProvider.getInjector();
        generator = injector.getInstance(DslGeneratorGrails.class);

        if(resource != null)
                generator.doGenerate(resource, fsa);

    } catch (JavaModelException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
   }
  }
  return null;
 }

 @Override
 public boolean isEnabled() {
        return true;
 }
}
```

Anexo 7: Plantilla Xpand (código)

A continuación podemos ver el código del fichero *DslNewProject.xpt*, ubicado en el paquete *org.xtext.portaMagna.ui.wizard*, que se encarga de inicializar cada nuevo proyecto PLESG con ciertos recursos que los analistas funcionales pueden tomar como punto de partida para especificar el sistema software a desarrollar.

```
«IMPORT org::xtext::portaMagna::ui::wizard»

«DEFINE main FOR DslProjectInfo»
«EXPAND model FOR this»
«ENDDEFINE»

«DEFINE model FOR DslProjectInfo»
«FILE "src/model/dataType.model"-»
/*
 * NOTE:
 *      These are the accepted types.
 *  You can witer your enumerated types here.
 */

datatype string
datatype email
datatype file  // <-- this type requires programatic code
datatype integer
datatype decimal
datatype boolean
datatype date
datatype void

«ENDFILE»

«FILE "src/model/config.model"-»
/*
 * NOTE:
 *      Config APP
 */

projectName portaMagna
appName example
idiom ES

«ENDFILE»

«FILE "src/model/dentities.model"-»
/*
 *
 * This is an example model:
 *      - change the <example>, <entity1> tags properly
 */

package portaMagna.example {

        entity entity1 {}

}
«ENDFILE»

«FILE "src/model/drol.model"-»
/*
```

```
 *
 * This is an example model:
 *      - change the <example>, <entity1> tags properly
 */

import portaMagna.example.*

rol ADMIN {
        type : admin, initialView: entity1
}

«ENDFILE»

«FILE "src/model/dts.model"-»
/*
 * IMPORTANT:
 *      Define the STD for each class
 */
«ENDFILE»

«FILE "src/model/rolview.model"-»
/*
 * IMPORTANT:
 *      Rol ADMIN has TOTAL visibility over the model.
 *      (You must write his possible navigations)
 */
«ENDFILE»

«FILE "src/model/formview.model"-»
/*
 * IMPORTANT:
 *      Rol ADMIN will has generated all forms automatically (by default)
 */
«ENDFILE»

«FILE "src/model/listview.model"-»
/*
 * IMPORTANT:
 *      Necesario especificar para cada GRID y dispositivo (desktop o movil) que
columnas puede ver
 *  cada rol
 */
«ENDFILE»

«FILE "src/ggv/DomainModel.gvmap"-»

import org.eclipse.emf.ecore.*
import org.eclipse.xtext.common.types.*
import org.eclipse.xtext.EcoreUtil2
import org.xtext.portaMagna.dsl.*

diagram DomainModel type PackageDeclaration {

        node EntityNode for each elements.filter(typeof(Entity)) {
                label Name for this

                edge SuperType for superType {
                        => call EntityNode for entity
                        label extendedClass for "extends"
                } unless (superType == null)

                hidden node Attributes for this {
                        label Attribute for each attributes.filter(typeof(Attribute))
                } unless attributes.filter(typeof(Attribute)).empty
```

```
		// 3. edge --> permite establecer "flechas", en este caso
		//                       RelationEdge desde EntityNode call EntityNode
		edge RelationEdge for each relations {
			=> call EntityNode for entity
			label RelationName for name + (":" + relation)
		}

		edge LegacyRelationEdge for each relations {
			=> call LegacyNode for legacy
			label LegacyRelationName for "(legacy) " + name + (":" +
relation)
		}

		// 1. hidden --> inicialmente NO aparecen en el GGD,
		//                              al seleccionar la clase aparece un boton
+ para mostrar
		//                              los elementos ocultos
		// 2. unless --> No muestra los Servicios "vacios"
		hidden node Services for this {
			label Service for each services.filter(typeof(Service))
		} unless services.filter(typeof(Service)).empty
	}
	node LegacyNode for each elements.filter(typeof(Legacy)) {
		label Name for this
	}
	node PackageNode for each elements.filter(typeof(PackageDeclaration)) {
		label Name for this
		open DomainModel for this
	}
}
«ENDFILE»

«FILE "src/ggv/DomainModel.gvstyle"-»

import org.eclipse.xtext.graphview.shape.*
import org.eclipse.xtext.graphview.behavior.layout.*
import org.eclipse.draw2d.*
import org.eclipse.swt.SWT
import DomainModel.*

stylesheet DomainModelStyle for DomainModel

style DomainModel {
	autoLayoutManager = new KielerAutoLayout
	font = font("Helvetica", 14, SWT::BOLD)
}

style LegacyNode as RectangleShape {
	lineStyle = SWT::LINE_DOT
	lineWidth = 33
	foregroundColor = color(#8a8a8a)
	border = new MarginBorder(16,16,16,16)
}

style EntityNode.SuperType as SplineConnectionShape {
	val targetDeco = new DiamondDecoration //ClosedArrowDecoration
	targetDeco.backgroundColor = color(#ffffff)
	targetDecoration = targetDeco
}

style EntityNode.Attributes as RectangleShape {
	outline = false
	backgroundColor = color(#ffffff)
}
```

```
style EntityNode.Attributes.Attribute {
	text = element.name + ": " + element.type
}

style EntityNode.RelationEdge.RelationName {
	opaque = true
	foregroundColor = color(#ff2222)
}

style EntityNode.LegacyRelationEdge.LegacyRelationName {
	opaque = false
	foregroundColor = color(#aaaaaa)
}

style EntityNode.Services as RectangleShape {
	outline = false
	backgroundColor = color(#ffaffa)
}

style EntityNode.Services.Service {
	text = "-" + element.parameters.map[if(it.token == "alias") it.value else "-
"].join(" ") + ": " + element.type
}
«ENDFILE»

«FILE "src/ggv/StateDiagram.gvmap"-»

import org.eclipse.emf.ecore.*
import org.eclipse.xtext.common.types.*
import org.eclipse.xtext.EcoreUtil2
import org.xtext.portaMagna.dsl.*


diagram StateDiagram type StateTransition {
	node StateNode for each states {
		label Name for this

		edge Transition for each transitions {
			=> ref StateNode for state
			label ActionLabel for action.service
		}

		hidden node Actions for this {
			label ActionsLabel for each actions.filter(typeof(Action))
		} unless actions.filter(typeof(Action)).empty

	} unless (this.entity.hasSTD != true)
}
«ENDFILE»

«FILE "src/ggv/StateDiagram.gvstyle"-»

import org.eclipse.draw2d.*
import org.eclipse.xtext.graphview.shape.*
import org.eclipse.xtext.graphview.behavior.layout.*
import org.xtext.softwareFactory.std.std.*
import org.eclipse.swt.SWT

import StateDiagram.*
import StateDiagram.StateNode.*
import StateDiagram.PackageNode.*

stylesheet StateDiagramStyle for StateDiagram

style StateDiagram {
```

```
        autoLayoutManager = new KielerAutoLayout()
        font = font("Helvetica", 14, SWT::BOLD)
}

style Transition.ActionLabel {
        opaque = true
        text = element.name
}

style Actions as RectangleShape {
        outline = true
        backgroundColor = color(#aafaff)
        foregroundColor = color(#8a8a8a)
}

style Actions.ActionsLabel {
        opaque = true
        text = element.service.name
}
«ENDFILE»

«ENDDEFINE»
```

Anexo 8: Configurar Apache para Django

La presente configuración se ha realizado bajo el sistema operativo (SO) Mac OS X (v10.6), en el cual viene instalado por defecto el servidor de aplicaciones Apache. Así pues, los pasos descritos a continuación podrían variar significativamente bajo otro SO:

1. Arrancar Apache Web Server.

   Ir a *System Preferences → Compartir → Compartir Web*

2. Incluir un nuevo dominio en /etc/hosts. Para ello abrir el terminal (consola):

```
$> sudo nano /etc/hosts
```

   Una vez abierto el fichero, es necesario añadir un nuevo dominio, por ejemplo

```
127.0.0.1       localhost
192.168.1.132   local.portamagna
```

3. Instalar mod_wsgi-3.3. Para ello hacer los pasos descritos en

   https://code.google.com/p/modwsgi/downloads/detail?name=mod_wsgi-macosx106-ap22py26-3.3.so

4. Crear un Virtual HOST donde establecer la aplicación “biblioteca” desarrollada para Django. Para ello abrir el terminal:

```
$> sudo nano /etc/apache2/extra/httpd-vhosts.conf
```

   e incluir las siguientes líneas al final del fichero:

```
<VirtualHost 192.168.1.132:80>
Alias /client/uniforms/
"/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/
site-packages/crispy_forms/static"

<Directory
"/Users/davifer/Desarrollos/workspace/portaMagnaDjango/client">
      Order allow,deny
      Options Indexes
      Allow from all
      IndexOptions FancyIndexing
</Directory>

Alias /client/
"/Users/davifer/Desarrollos/workspace/portaMagnaDjango/client/"
<Directory
"/Users/davifer/Desarrollos/workspace/portaMagnaDjango/client">
      Order allow,deny
```

```
      Options Indexes
      Allow from all
      IndexOptions FancyIndexing
</Directory>

Alias /media/
"/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/
site-packages/django/contrib/admin/media/
<Directory
"/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/
site-packages/django/contrib/adm$
            Order allow,deny
            Options Indexes
            Allow from all
            IndexOptions FancyIndexing
</Directory>

ServerName 192.168.1.132
ErrorLog "/Library/WebServer/logs/local.portamagna.txt"

<Directory />
        #Options FollowSymLinks
        Options Indexes FollowSymLinks Includes ExecCGI
        AllowOverride All
        Order deny,allow
        Allow from all
</Directory>

<Directory /Users/davifer/Desarrollos/workspace/portaMagnaDjango>
        AllowOverride All
        Options Indexes FollowSymLinks
        Order allow,deny
        Allow from all
</Directory>

WSGIDaemonProcess 192.168.1.132 processes=2 threads=15 maximum-
requests=10 display-name=%{GROUP}
WSGIProcessGroup 192.168.1.132
WSGIScriptAlias /
"/Users/davifer/Desarrollos/workspace/portaMagnaDjango/django.wsg
i"
</VirtualHost>
```

Por último reiniciar el servidor APACHE, abrir el navegador e ir a http://local.portamagna/biblioteca

Anexo 9: Acuerdo de colaboración

ACUERDO DE COLABORACIÓN EN PRO DE LA INVESTIGACIÓN ACADÉMICA

En Valencia, a 15 de Noviembre de 2013

REUNIDOS

De una parte, Don David Vicent Ferrer, con DNI: 24380218B, alumno del master en Ingeniería del Software, Métodos Formales y Sistemas de Información de la Universidad Politécnica de Valencia, y autor del Trabajo Final de Master (TFM) bajo el título **PLESG: Propuesta de Línea de Ensamblado de Software de Gestión**.

Y de otra parte, Doña Mónica Fernández Mafé, gerente de la Clínica Dental Fernández Traver, con C.I.F.: E-12903639, en adelante Entidad Colaboradora, en representación de la misma, con domicilio social en Av. Francisco Tárrega 2, Vila-Real (Castellón).

Intervienen como tales y en la representación que ostentan se reconocen entre sí la capacidad legal necesaria para suscribir el presente convenio y

EXPONEN

Que es voluntad de las partes colaborar en la formación práctica de los estudiantes de postrado, cuyo objetivo es permitir a los mismos aplicar y complementar los conocimientos adquiridos en su formación académica, favoreciendo la adquisición de competencias que les preparen para el ejercicio de actividades profesionales, faciliten su empleabilidad y fomenten su capacidad de emprendimiento.

Por todo ello, deciden concertar el presente Convenio de Colaboración de acuerdo con las siguientes

CLÁUSULAS

PRIMERA.- Se ha puesto en práctica el citado TFM mediante el desarrollo de una aplicación real para la gestión de los datos médicos, personales, radiografías y citas de los pacientes de la Entidad Colaboradora.

SEGUNDA.- Que dicha aplicación cubre los requisitos funcionales y no funcionales establecidos en el TFM (capítulo 6).

TERCERA.- Que la Entidad Colaboradora permite la cesión de su imagen en el citado TFM con el fin de poder difundir los objetivos perseguidos en éste.

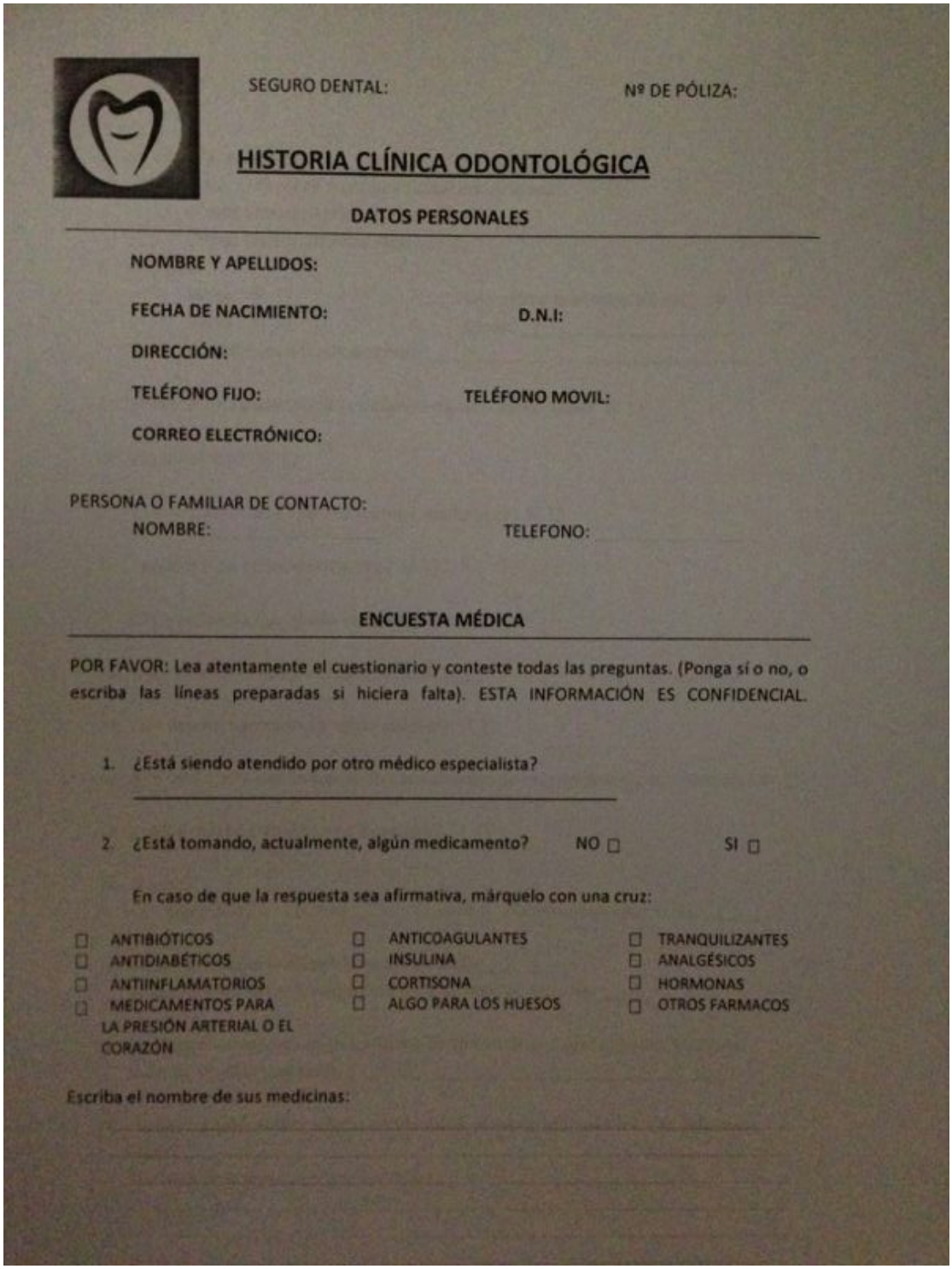
Autor del TFM

En representación de la entidad colaboradora

Anexo 10: Requisitos Clínica Dental Fernández - Traver

A continuación se muestran los documentos utilizados por la clínica dental analizada para obtener los datos personales y médicos de un nuevo paciente. A partir de dicha información se ha extraído parte de la especificación PLESG vista en el apartado 6.1.

Paciente: datos personales y médicos



SEGURO DENTAL: Nº DE PÓLIZA:

HISTORIA CLÍNICA ODONTOLÓGICA

DATOS PERSONALES

NOMBRE Y APELLIDOS:

FECHA DE NACIMIENTO: D.N.I:

DIRECCIÓN:

TELÉFONO FIJO: TELÉFONO MOVIL:

CORREO ELECTRÓNICO:

PERSONA O FAMILIAR DE CONTACTO:

NOMBRE: TELEFONO:

ENCUESTA MÉDICA

POR FAVOR: Lea atentamente el cuestionario y conteste todas las preguntas. (Ponga sí o no, o escriba las líneas preparadas si hiciera falta). ESTA INFORMACIÓN ES CONFIDENCIAL.

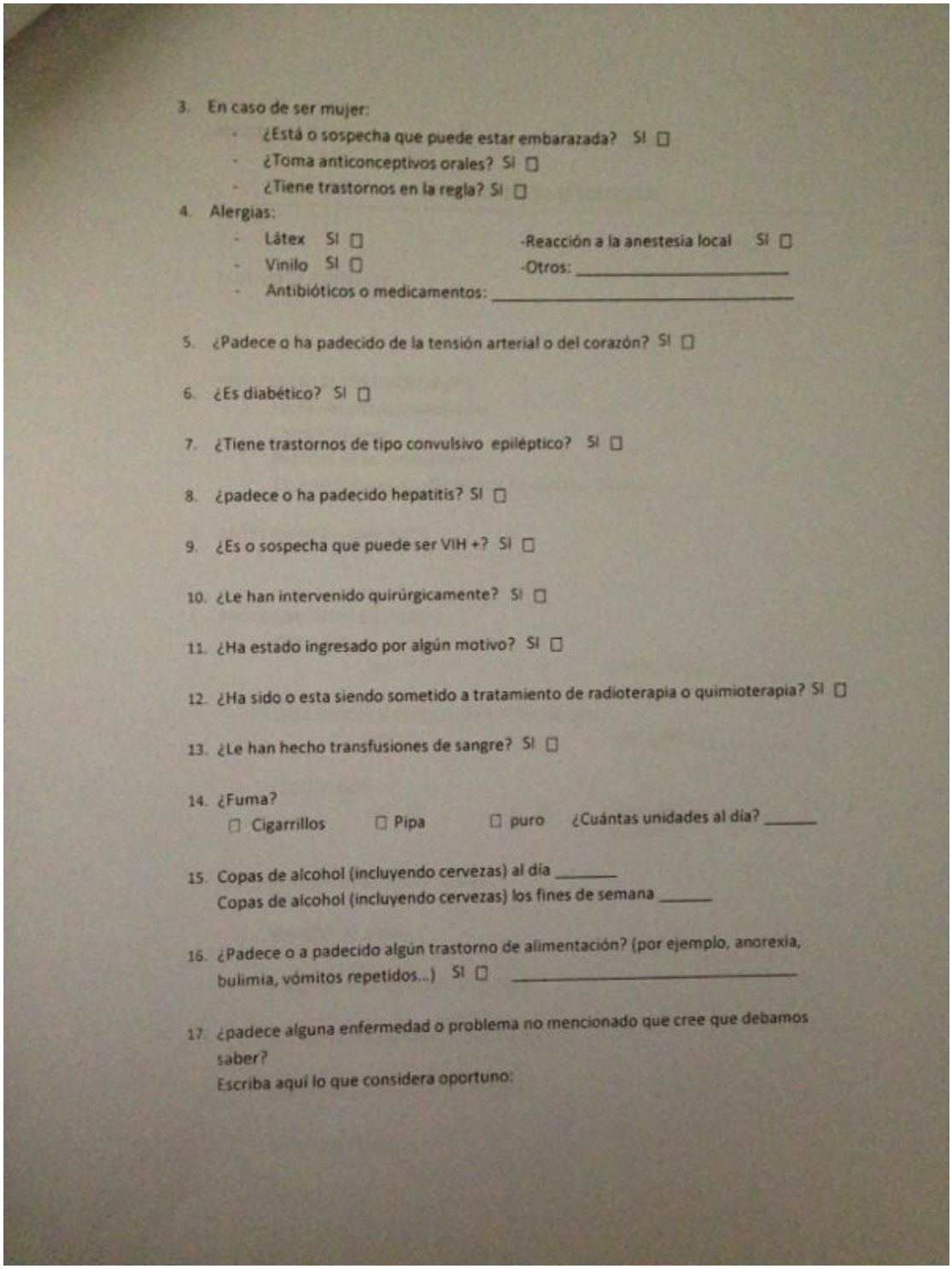
1. ¿Está siendo atendido por otro médico especialista?

2. ¿Está tomando, actualmente, algún medicamento? NO ☐ SI ☐

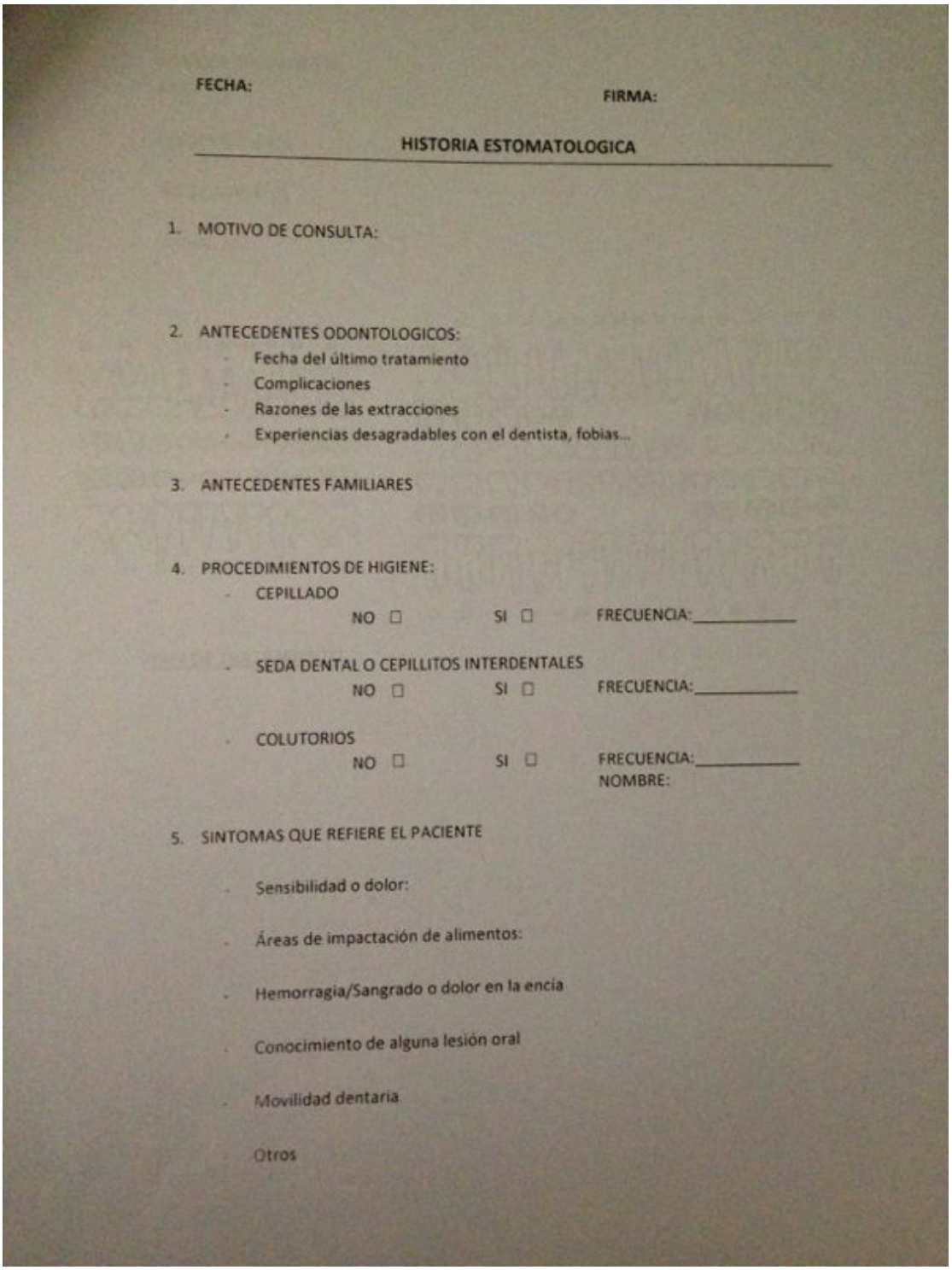
En caso de que la respuesta sea afirmativa, márquelo con una cruz:

☐ ANTIBIÓTICOS	☐ ANTICOAGULANTES	☐ TRANQUILIZANTES
☐ ANTIDIABÉTICOS	☐ INSULINA	☐ ANALGÉSICOS
☐ ANTIINFLAMATORIOS	☐ CORTISONA	☐ HORMONAS
☐ MEDICAMENTOS PARA LA PRESIÓN ARTERIAL O EL CORAZÓN	☐ ALGO PARA LOS HUESOS	☐ OTROS FARMACOS

Escriba el nombre de sus medicinas:



3. En caso de ser mujer:
   - ¿Está o sospecha que puede estar embarazada? SI ☐
   - ¿Toma anticonceptivos orales? SI ☐
   - ¿Tiene trastornos en la regla? SI ☐
4. Alergias:
   - Látex SI ☐ -Reacción a la anestesia local SI ☐
   - Vinilo SI ☐ -Otros: _____
   - Antibióticos o medicamentos: _____
5. ¿Padece o ha padecido de la tensión arterial o del corazón? SI ☐
6. ¿Es diabético? SI ☐
7. ¿Tiene trastornos de tipo convulsivo epiléptico? SI ☐
8. ¿padece o ha padecido hepatitis? SI ☐
9. ¿Es o sospecha que puede ser VIH +? SI ☐
10. ¿Le han intervenido quirúrgicamente? SI ☐
11. ¿Ha estado ingresado por algún motivo? SI ☐
12. ¿Ha sido o esta siendo sometido a tratamiento de radioterapia o quimioterapia? SI ☐
13. ¿Le han hecho transfusiones de sangre? SI ☐
14. ¿Fuma?
    ☐ Cigarrillos ☐ Pipa ☐ puro ¿Cuántas unidades al día? _____
15. Copas de alcohol (incluyendo cervezas) al día _____
    Copas de alcohol (incluyendo cervezas) los fines de semana _____
16. ¿Padece o a padecido algún trastorno de alimentación? (por ejemplo, anorexia, bulimia, vómitos repetidos...) SI ☐ _____
17. ¿padece alguna enfermedad o problema no mencionado que cree que debamos saber?
    Escriba aquí lo que considera oportuno:



FECHA: FIRMA:

HISTORIA ESTOMATOLOGICA

1. MOTIVO DE CONSULTA:

2. ANTECEDENTES ODONTOLOGICOS:
   - Fecha del último tratamiento
   - Complicaciones
   - Razones de las extracciones
   - Experiencias desagradables con el dentista, fobias...

3. ANTECEDENTES FAMILIARES

4. PROCEDIMIENTOS DE HIGIENE:
   - CEPILLADO
     NO ☐ SI ☐ FRECUENCIA:_____
   - SEDA DENTAL O CEPILLITOS INTERDENTALES
     NO ☐ SI ☐ FRECUENCIA:_____
   - COLUTORIOS
     NO ☐ SI ☐ FRECUENCIA:_____
     NOMBRE:

5. SINTOMAS QUE REFIERE EL PACIENTE
   - Sensibilidad o dolor:
   - Áreas de impactación de alimentos:
   - Hemorragia/Sangrado o dolor en la encía
   - Conocimiento de alguna lesión oral
   - Movilidad dentaria
   - Otros

Referencias

[1. R. Hopkins, 2008]
Richard Hopkins, Kevin Jenkins.
Eating the IT Elephant: Moving from Greenfield Development to Brownfield.
IBM Press; 2008

[2. F. Brooks, 1995]
Frederick P. Brooks, The Mythical Man Month (Anniversary Edition)
Chap. 16 ("No Silver Bullet - Essence and Accident"), F. P., Addison Wesley, 1995

[3. Middleware Company, 2003]
Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach

[4. Hibbs Curt, 2007]
ONLamp : Rolling with Ruby on Rails,
http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html

[5. M. Fowler, 2002]
Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford.
Patterns of Enterprise Application Architecture, Addison Wesley, 2002

[6. Amberpoint, 2008]
Amberpoint Inc., BearingPoint Inc., Composite Software, Inc., MomentumSI, Progress Software Corporation, and Luc Clem-Ent, SOA Getting It Right;
An Implementor's Guide to Service Oriented Architecture, 2008

[7. O. Pastor, 2001]
O. Pastor, J. Gomez, E. Insfrán and V. Pelechano,
"The OO-Method approach for information systems modeling: from object-oriented conceptual modeling to automated programming", Information Systems, 26, 2001, pp. 507-534.

[8. O. Pator, J. C. Molina, 2007]
Oscar Pastor, Juan Carlos Molina
Model-Driven Architecture in Practice: A Software Production Environment Based on Conceptual Modeling. Springer, ISBN: 978-3-540-71867-3

[9. Imran Sarwar Bajwa, 2009]
Object Oriented Software Modeling Using NLP Based Knowledge Extraction

[10. Jordi Cabot, 2010]
Dealing with Non-Functional Requirements in Model-Driven Development

[11. Miguel Llàcer, 2008]
Eclipse - Tesis de Master ISMFSI, DSIC (UPV);
DISEÑO Y DESARROLLO DE MECANISMOS PARA LA CONFIGURACIÓN DE TRANSFORMACIONES DE MODELOS EN MOSKITT

[12. Johan den Haan]
1) 8 reasons why Model-Driven Development is dangerous, 25 June 2009
http://www.theenterprisearchitect.eu/archive/2009/06/25/8-reasons-why-model-driven-development-is-dangerous

2) 5 types of Model Driven Software Development
http://www.theenterprisearchitect.eu/archive/2009/03/31/5-types-of-model-driven-software-development

[13. Greenfield and Short, 2003]
Jack Greenfield, Keith Short, Steve Cook, Stuart Kent, John Crupi,
Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools

[14. Lenz et al., 2008]
Gunther Lenz, Microsoft Corporation; Christoph Wienands, Siemens Corporate Research, Making the Business Case for Software Factories, Microsoft, 2008
http://msdn.microsoft.com/en-us/library/cc496679.aspx#anchor6

[15. J. Cabot, V. Pelechano]
Automatic Generation of Basic Behavior Schemas from UML Class Diagrams
http://modeling-languages.com/la-regla-de-pareto-aplicada-mdd/

[16. J. C. Molina, O. Pastor]
Juan Carlos Molina, Oscar Pastor
MDA, OO-Method y la Tecnología OLIVANOVA Model Execution.

[17. P. Graff, 2009]
Xtext vs. EMFText
http://pettergraff.blogspot.com.es/2009/11/xtext-vs-emftext-code-generation.html

[18. Jan Köhnlein, 2012]
1) Discovery Diagrams for the Generic Graphical View
http://koehnlein.blogspot.com.es/2012/01/discovery-diagrams-for-generic.html

2) Integración de GGV en Xtext (README)
https://github.com/JanKoehnlein/Generic-Graph-View/tree/master/org.eclipse.xtext.graphview

[19, Gamma, 1995]
Gamma, Erich. Helm, Richard. Johnson, Ralph. Vlissides, John (Año 1995).
"Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley,
ISBN: 0-20-163361-2.

[20, KIELER project]
http://www.informatik.uni-kiel.de/rtsys/kieler/

Acrónimos

API	Application Programming Interface
ATL	ATLAS Transformation Language
CASE	Computer-Aided Software Engineering
CRUD	Create, read, update and delete
CSS	Cascading Style Sheets
DDL	Data Definition Language
DSL	Domain-Specic Language
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modeling Framework
EMOF	Essential Meta-Object Facility
EMP	Eclipse Modeling Project
EPL	Eclipse Public Licence
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
HTML	HyperText Markup Language
IDE	Integrated Development Environment
M2T	Model-to-Text
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MDSD	Model-Driven Software Development
MOF	Meta-Object Facility
MVC	Model-View-Controller
MWE	Modeling Workflow Engine
oAW	openArchitectureWare
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform-Independent Model
PSM	Platform-Specific Model
NFR	Non-Functional Requirement
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TMF	Textual Modeling Framework
UML	Unified Modeling Language
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
VCS	Version Control System
XMI	XML Metadata Interchange
XML	Extensible Markup Language