Máster Universitario de Investigación en
Ingeniería de Software y Sistemas Informáticos

Service Builder, software que escribe y evoluciona aplicaciones basadas en Liferay Portal

Autor: Manuel de la Peña Peña
Director. Ismael Abad Cardiel

Curso Académico 2016/2017
Convocatoria Septiembre

Máster Universitario de Investigación en Ingeniería de Software y Sistemas Informáticos

ITINERARIO: Ingeniería del Software

CÓDIGO DE ASIGNATURA: 31105128

TÍTULO DEL TRABAJO: Service Builder, software que escribe y evoluciona aplicaciones basadas en Liferay Portal

TIPO DE TRABAJO: Tipo B, proyecto específico propuesto por el alumno.

AUTOR: Manuel de la Peña Peña

DNI: 53430012-T

DIRECTOR: Ismael Abad Cardiel

DECLARACIÓN JURADA DE AUTORÍA DEL TRABAJO CIENTÍFICO, PARA LA DEFENSA DEL TRABAJO FIN DE MÁSTER

Fecha: 10/09/2017

Quien suscribe:
AUTOR: Manuel de la Peña Peña
DNI: 53430012-T

Hace constar que es el autor del trabajo: “Service Builder, software que escribe y evoluciona aplicaciones basadas en Liferay Portal”

En tal sentido, manifiesto la originalidad de la conceptualización del trabajo, interpretación de datos y la elaboración de las conclusiones, dejando establecido que aquellos aportes intelectuales de otros autores, se han referenciado debidamente en el texto de dicho trabajo.

DECLARACIÓN:

✔ Garantizo que el trabajo que remito es un documento original y no ha sido publicado, total ni parcialmente por otros autores, en soporte papel ni en formato digital.

✔ Certifico que he contribuido directamente al contenido intelectual de este manuscrito, a la génesis y análisis de sus datos, por lo cual estoy en condiciones de hacerme públicamente responsable de él.

✔ No he incurrido en fraude científico, plagio o vicios de autoría; en caso contrario, aceptaré las medidas disciplinarias sancionadoras que correspondan.

Firmado:

DE LA PEÑA PEÑA MANUEL - 53430012T

Digitally signed by DE LA PEÑA PEÑA MANUEL - 53430012T
DN: c=ES, serialNumber=IDCES-53430012T, givenName=MANUEL, sn=DE LA PEÑA PEÑA, cn=DE LA PEÑA PEÑA MANUEL - 53430012T
Date: 2017.09.12 14:52:49 +02'00'

IMPRESO TFDM04_DEFENSA
INFORME DE DIRECCIÓN
Y DEFENSA DEL TFDM

Impreso TFdM04_Defensa. Informe del Director para proceder a la defensa del TFdM

El PROFESOR
D./Da Ismael Abad Cardiel del Departamento de Ingeniería de Software y Sistemas Informáticos,

INFORMA favorablemente para el depósito y posterior defensa del Trabajo Fin de Máster realizado por:

D./Dña.: .MANUEL DE LA PEÑA PEÑA
DNI: 53430012-T

del tipo:
☐ Tipo A: Trabajo específico propuesto por un profesor.
✔ Tipo B: Trabajo específico propuesto por el alumno.

con el Título: SERVICE BUILDER , software que escribe y evoluciona aplicaciones basadas en Liferay Portal

Fecha: 13/09/2017 (El informe sólo es vigente para las siguientes convocatorias del curso correspondiente a esta fecha)

Firma del Director

Juan del Rosal, 16
28040, Madrid
Tel: 91 398 89 10
Fax: 91 398 89 09
www.issi.uned.es

IMPRESO TFDM05_AUTOR
AUTORIZACIÓN DE PUBLICACIÓN
CON FINES ACADÉMICOS

Impreso TFdM05_Autor. Autorización de publicación
y difusión del TFdM para fines académicos

Autorización

Autorizo a la Universidad Nacional de Educación a Distancia a difundir y utilizar, con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la memoria de este Trabajo Fin de Máster, como el código, la documentación y/o el prototipo desarrollado.

Firma del Autor
Manuel de la Peña Peña

DE LA PEÑA PEÑA MANUEL - 53430012T

Digitally signed by DE LA PEÑA PEÑA MANUEL - 53430012T
DN: c=ES, serialNumber=IDCES-53430012T, givenName=MANUEL, sn=DE LA PEÑA PEÑA, cn=DE LA PEÑA PEÑA MANUEL - 53430012T
Date: 2017.09.12 14:53:13 +02'00'

Juan del Rosal, 16
28040, Madrid
Tel: 91 398 89 10
Fax: 91 398 89 09
www.issi.uned.es

Índice de contenidos

1. Introducción

A la hora de acometer un nuevo desarrollo, no muchas veces se plantea el uso de un generador de código, puesto que el esfuerzo inicial de creación de éste suele hacer contemplarlo como gasto frente al esfuerzo de creación del propio desarrollo. Sin embargo, según avanza el proyecto y se van reutilizando estructuras de código, los equipos empiezan a considerar la creciente deuda técnica que tienen debido al mantenimiento de todos esos bloques con cada modificación que los afecte. Lo que en un principio se consideraba un coste, es en realidad una inversión, puesto que de haber utilizado un generador de código, las modificaciones se realizarían en un único lugar y se aplicarían al resto de la base del código mediante la ejecución del generador, reduciendo considerablemente el tiempo de desarrollo.

Sin embargo, esta generación es unidireccional, en el sentido que se tienen unas plantillas que dirigen la generación: el generador recibe como input las plantillas, cierta configuración, y genera un código fuente resultante, siempre el mismo. Si quisiéramos cambiar el código generado deberíamos, o bien cambiar las plantillas o bien cambiar la configuración.

Si en su lugar se utilizara un enfoque orientado a la composición, las estructuras se podrían componer para generar el código resultante. Mediante esta composición podríamos “envolver” las plantillas en operaciones, de manera que mediante la composición de éstas, el programa resultante pudiera tener un comportamiento parcialmente distinto según se invocara o no cierta operación.

Estas operaciones que envuelven a las plantillas de los generadores conformarían una especie de lenguaje propio para trabajar con los generadores, de manera que conociendo el lenguaje se podría trabajar con el generador apropiado de una manera mucho más abstracta. Este lenguaje especializado es lo que en ingeniería del software suele denominarse como DSL, o lenguaje específico de dominio, y supone un hilo conductor entre los expertos del dominio y los desarrolladores. O incluso yendo más allá, los DSL permiten convertir los unos en los otros.

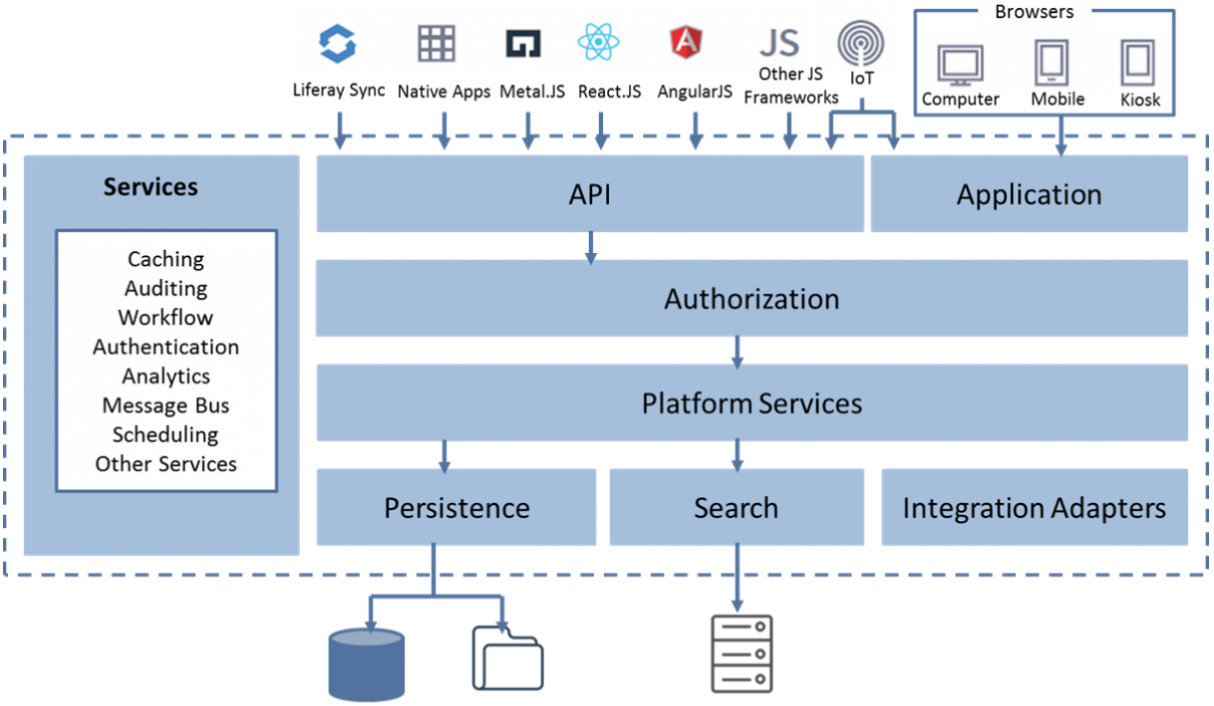
Si nos embarcamos en un proyecto ya consolidado, en el que se utilizan frameworks propios de desarrollo, el problema es aún mayor, puesto que la opción de añadir un generador de código o un DSL propio se antoja complicado, puesto que es preciso conocer el dominio del problema y cómo el framework se aplica sobre él.

Por el contrario, si nos encontrásemos con una plataforma que ofreciera ya de por sí capacidades de generación de código, podríamos empezar a trabajar en modelar el

conocimiento en busca de un DSL que acerque esas posturas entre los analistas del dominio y los desarrolladores. En este punto aparece Liferay Portal [1] como plataforma de desarrollo en forma de portal, con herramientas de generación de código, así como el uso de frameworks de desarrollo estándar.

Liferay Portal [1] es una aplicación escrita en Java que permite desplegar sobre ella aplicaciones que sigan las especificaciones de Portlet (JSR-168 [27] y JSR-286 [28]). Por tanto, es considerada una plataforma de desarrollo, en la que los desarrolladores pueden escribir aplicaciones satisfaciendo sus propias reglas de negocio. Además, Liferay Portal incluye de manera adicional un número muy elevado de aplicaciones propias que añaden funcionalidades relacionadas con herramientas de colaboración (p.e.: wikis, blogs, chat), gestión de contenidos (p.e. publicador de artículos, galerías de imágenes, gestión de archivos), o gestión de usuarios, roles y permisos, entre otras.

Para facilitar el desarrollo de estas aplicaciones, la plataforma ofrece a los desarrolladores diferentes servicios de infraestructura para optimizar el uso de los recursos, así como para aprovechar al completo el potencial de la misma, como podría ser el acceso a las aplicaciones mencionadas con anterioridad, el uso de cachés, transacciones, servicio de gestión documental, búsqueda e indexación de entidades del modelo, entre otros.



Detalle de la arquitectura de Liferay Portal

Y todos estos servicios se ofrecen mediante una herramienta de generación de código, denominada Service Builder, que permite la adición de lógica propia de la aplicación sobre estos servicios.

Sin embargo, este proceso de creación de aplicaciones basadas en Liferay Portal con Service Builder se basa en un proceso de modelado de las entidades del dominio basado en XML, como veremos más adelante, de manera que es necesario configurar un descriptor XML con todas las operaciones necesarias para crear la infraestructura antes mencionada en las aplicaciones. La creación y edición de este descriptor puede ser bastante tediosa, de aquí la búsqueda de una solución mucho más moderna en cuanto a tecnología, y que aplique conceptos de ingeniería del software para construir las aplicaciones de una manera más sencilla y menos propensa a errores.

Uniendo mejoras en cuanto a calidad del producto y del proceso software, con conceptos del diseño de aplicaciones dirigidas por el modelo, podemos llegar a diseñar una herramienta que permita generar el código de las aplicaciones basadas en Liferay Portal de una manera mucho más eficiente. El presente documento pretende demostrarlo.

1.1. Estructura del documento

En este primer capítulo de introducción se expondrán tanto la motivación para la realización de este trabajo, como una breve descripción de la solución propuesta, en formato de justificación y listado de entregables del trabajo.

En el capítulo número dos pasaremos a determinar los objetivos del trabajo de investigación.

El capítulo tres consistirá en la planificación realizada para acometer el trabajo, indicando medios técnicos utilizados, alcance y planificación temporal.

El cuarto capítulo consistirá en una fundamentación teórica y un análisis pormenorizado del estado del arte en cuanto a generación de código, calidad de proyecto software y lenguajes de dominios específicos, o DSLs. Además se describirá qué es Liferay Portal, así como se detallará el proceso de definición del modelo de entidades para las aplicaciones basadas en él, de manera que se pueda comprender la problemática actual en esta definición. Para ello será necesario entender el vocabulario del dominio y así, a modo de ejemplo, comprender que una

Entidad está compuesta de un número determinado de columnas, así como de otros metadatos que la describen.

En el quinto punto se expondrán los resultados obtenidos mediante una versión esquematizada del proyecto realizado. Se trasladarán las operaciones del dominio a un lenguaje DSL, en el que se buscarán facilitar a los desarrolladores la expresión de dichas operaciones de una manera muy cercana al desarrollador. En este punto el diseño de un DSL adecuado ayudará sobremanera.

En el capítulo seis se detallará la estructura del código fuente que representa la solución, que no es otra cosa que un programa escrito en Java, con una batería de pruebas verificando los requisitos funcionales establecidos. Se abordarán temas como la construcción del proyecto, ejecución de los tests, y se analizará la calidad del código en base a las buenas prácticas de la industria, apoyándose en servicios cloud de integración continua y de análisis estático de código.

Para terminar, se cerrará el trabajo con las conclusiones y los trabajos futuros a realizar tras este proyecto, dado que el alcance limitado del mismo; y las lecturas y referencias relacionadas con el presente trabajo.

1.2. Motivación

La motivación surgió de la necesidad de realizar el Trabajo Fin de Máster a partir de los conocimientos obtenidos por el alumno al cursar las asignaturas del Máster. La idea principal del trabajo consiste en definir una manera mucho más robusta para escribir aplicaciones basadas en Liferay Portal, puesto que la manera actual es propensa a errores al utilizar un lenguaje no compilado. La alternativa elegida es la de diseñar un lenguaje DSL interno homogéneo, que permita apoyarse en la potencia del compilador para restringir operaciones no válidas en la definición de estas aplicaciones.

Esta definición de las aplicaciones se realiza a partir de la creación de un descriptor del modelo, en formato XML, y a partir de él, mediante unos procesos de generación de código, se generan las capas de persistencia y servicio de las aplicaciones.

El alumno ha cursado las asignaturas de Generación Automática de Código, Arquitecturas Orientadas a Servicios, Gestión y Mejora de Procesos Software, Arquitecturas para Sistemas Software, y Computación Ubicua, que le han permitido adquirir conocimientos sobre el IoT, las aplicaciones basadas en servicios, gestión

de la calidad de los procesos software y diseño de aplicaciones dirigido por el modelo.

Además, en el trabajo para la asignatura Computación Ubicua presentó un trabajo que analizaba el proceso de generación de código de aplicaciones basadas en Liferay Portal [1] para extenderlo y generar el código de clientes IoT que se conectaran a la plataforma, realizando una gestión de estos dispositivos a la hora de establecer las conexiones, utilizando los distintos protocolos para cada uno de ellos.

Aplicando los conocimientos obtenidos tras cursar estas asignaturas, el alumno ha sido capaz de desarrollar una implementación de un lenguaje DSL para construir aplicaciones basadas en Liferay Portal, buscando la originalidad al acercar áreas tecnológicas en auge como la generación automática de código y el diseño dirigido por el modelo. Además, ha sido capaz de aplicar las mejores prácticas de la industria para garantizar que la calidad del software desarrollado sea lo más alta posible.

1.3. Solución propuesta

1.3.1. Justificación

A continuación se comentan los motivos para la mejora de Service Builder como herramienta de generación de código de aplicaciones basadas en Liferay Portal:

- Las mejoras propuestas simplificarían la escritura de aplicaciones basadas en Liferay Portal no sólo a la compañía que desarrolla el producto, Liferay Inc., sino también a su base de clientes, que desarrollan sus propias aplicaciones, así como a la gran comunidad de usuarios distribuidas por todo el mundo.

- Las mejoras propuestas modernizarían la herramienta de generación de código actuales, pues pasaría de utilizar un descriptor del modelo basado en XML, al uso de un lenguaje robusto como Java, que se apoye en el sistema de tipos y en el compilador para garantizar las operaciones válidas de construcción del modelo.

- Un motivo para la elección de Service Builder es que el alumno ya realizó un trabajo basado en esta herramienta para la asignatura de Computación Ubicua, realizando una extensión de la misma para generar clientes IoT y gestionar la conexión con ellos desde la plataforma Liferay Portal.

- El alumno trabaja actualmente para Liferay Inc, empresa de desarrollo de software con base en Los Ángeles encargada del mantenimiento y desarrollo de Liferay Portal.

1.3.2. Entregables del trabajo

Los entregables del trabajo son los siguientes:

- Memoria resumen.
- Código Java generado para la solución.
- Cobertura de código de la solución propuesta.
- Código Javadoc de la solución.

2. Objetivos del trabajo de investigación

En el siguiente apartado se describen los objetivos que se desean cumplir con la realización del trabajo.

2.1. Objetivos principales

Los objetivos principales para acometer este trabajo son los siguientes:

- Realizar un análisis pormenorizado de la generación de código en Liferay Portal.
- Implementar un lenguaje específico de dominio DSL interno y homogéneo para describir las operaciones de definición del modelo en aplicaciones basadas en Liferay Portal.

2.2. Otros objetivos

Otros objetivos a cumplir con el desarrollo de este trabajo de investigación serán:

- Detectar las carencias actuales en cuanto a generación de código.
- Realizar un análisis de la creciente relación entre la generación de código y los lenguajes de dominio específicos.
- Dar valor a los procesos de desarrollo del software desde el punto de cualitativo.
- Validar la solución propuesta frente a las carencias detectadas.
- Integrar la solución en el proceso de desarrollo de Liferay Portal.

3. Planificación y documentación

En este apartado se muestra el resultado de la planificación del trabajo indicando los medios a utilizar, el alcance y la estimación de horas asignadas a cada tarea.

3.1. Medios técnicos

Los medios técnicos utilizados para desarrollar el proyecto han sido los siguientes:

- Para el desarrollo del software, portátil MacBook Pro Retina 15'' con procesador Intel® Core™ i7 @ 2.6 GHz, 16Gb de memoria RAM, y sistema operativo Mac OS X 10.12.6 x86_64.

- Para realizar la memoria se ha utilizado el procesador de textos cloud Google Docs.

- Para el desarrollo del código se ha utilizado IntelliJ Idea Ultimate 2017.1.3 [2].

- Para la generación de gráficos UML se ha utilizado IntelliJ Idea Ultimate 2017.1.3 [2].

- Para diseñar el lenguaje se ha utilizado Java, en su versión de Oracle JDK 1.8.0_144.

- Para la construcción del código se ha utilizado Gradle en su versión 3.3. [3]

- Para la ejecución de las pruebas unitarias se ha utilizado jUnit en su versión 4. [4]

- Para la integración continua del proyecto se ha utilizado el servicio Travis CI. [5]

- Para la medición de cobertura de código se ha utilizado el servicio Codecov.io. [6]

3.2. Alcance

Al ser Liferay Portal un proyecto Open Source, el alcance de la herramienta propuesta en este trabajo es el de integrar dicha herramienta en el proyecto Liferay Portal en su versión 7.0, de modo que cualquier usuario pudiera utilizarla para crear sus desarrollos sobre Liferay Portal.

La funcionalidad implementada es la siguiente:

- Diseño y codificación de un lenguaje DSL que soporte las mismas operaciones existentes en el sistema actual.

- Generación del XML descriptor del modelo a partir del modelo definido con el nuevo lenguaje DSL.

3.3. Planificación

Como ya se ha comentado anteriormente, la asignatura Trabajo Fin de Máster tiene un valor de 15 créditos ECTS. Si cada crédito corresponde a 25 horas de trabajo, la extensión máxima en horas del trabajo debe ser de unas 375 horas, siendo la fecha máxima de entrega el día 10 de Septiembre de 2017.

La idea original fue haber presentado el Trabajo de Fin de Máster en el mes de junio pero, por circunstancias personales -voy a ser padre por primera vez en octubre- y profesionales -los meses de mayo y junio fueron complicados al marcharse el líder de mi equipo y reubicarme por ello a un equipo localizado en Los Ángeles a 9 horas de distancia- no pude dedicarle todo el tiempo requerido. Por tanto, en consenso con Ismael Abad, tutor del proyecto, decidimos presentarlo en septiembre y hacer uso del verano para sacar adelante el proyecto.

La fecha de inicio de los trabajos comenzaron a primeros del año 2017, momento en que empiezo la labor de investigación y búsqueda de información. A partir de ese momento, gracias a la información recopilada, las ideas van surgiendo en la cabeza y voy modelando el diseño y análisis del proyecto. El 18 de mayo de 2017 (fecha del primer commit en el proyecto) comienza el desarrollo del DSL, proceso que no termina hasta el mismo día de hoy. Me declaro seguidor de las metodologías ágiles, por tanto no concibo una fase “final” de pruebas, sino que durante todo el proceso de desarrollo voy escribiendo pruebas unitarias de todo el código que escribo.

Además, ese código es probado de manera constante mediante el servicio de integración continua de Travis, por tanto durante todo el tiempo de desarrollo se entremezclan las fases de diseño, codificación y testing.

Para ser honestos, no he llevado un control de las horas invertidas en el trabajo. El compaginar la primeriza pre-paternidad con el día a día laboral, unido a los esfuerzos para realizar este Trabajo de Fin de Máster ha sido un camino bastante duro. No obstante supongo que más de 375 horas habré sacado entre la investigación, la codificación, la documentación y el formateo de este trabajo.

4. Fundamentación teórica y estado del arte

Antes de empezar a describir el trabajo realizado para mejorar la generación de código en las aplicaciones basadas en Liferay Portal, es importante introducir los conceptos de generación de código, calidad software, y de DSL o "*Domain Specific Language*" [7], así como la descripción del propio proyecto Open Source Liferay Portal.

4.1. Generación de código

Generación de código es la técnica de construir y utilizar programas que escriben otros programas. Un ejemplo concreto sería el utilizar un generación de código para construir el código de acceso a una base de datos, o el código necesario para comunicar a través de un servicio web. La idea principal consiste en crear código consistente y de la máxima calidad, de una manera más rápida que la tradicional.

Por otro lado, es una técnica parecida al diseño de patrones, a la programación extrema o al diseño orientado a objetos que, utilizada de una manera adecuada, puede cambiar de manera radical el modo de producir código y conseguir una mayor productividad. Entre los beneficios que afectan a los desarrolladores, podemos encontrar:

- Calidad. Grandes cantidades de código escrito "a mano" tienden a tener inconsistencias en cuanto a calidad se refiere, debido a que los desarrolladores encuentran nuevas o mejores maneras de realizar las cosas a medida que avanza el proyecto. La generación de código, por ejemplo a partir de unas plantillas, crea una base de código consistente de manera instantánea, y cuando esas plantillas son modificadas y el generador ejecutado, las mejoras en el código o la corrección de errores son aplicadas de manera consistente en toda la base de código.

- Consistencia. El código que es producido por un generador de código es consistente con el diseño de las APIs, así como con los convenios de nombres definidos para nombrar variables. Esto resulta en la existencia de interfaces bien definidas fáciles de utilizar y/o entender.

- Un punto único de conocimiento. Los cambios realizados en las plantillas se filtran a través de las diferentes capas de generación de código para

implementar dichos cambios por todo el sistema. Mediante una arquitectura de generación de código adecuada es posible cambiar, por ejemplo, un nombre de tabla en un único punto para luego regenerar todas las piezas involucradas: esquema de datos, capa del modelo, documentación, tests, etc.

- Importancia de las decisiones de diseño. Muchas veces las reglas de negocio de alto nivel se diluyen una vez se avanza en la implementación del código. Los generadores de código utilizan ficheros de definición abstractos que especifican el diseño del código a generar. Estos ficheros son mucho más pequeños y específicos que el código generado, por lo que es mucho más fácil detectar en ellos pequeñas excepciones u otras consideraciones.

Además de las ventajas relacionadas con los desarrolladores, es importante destacar que la generación de código afecta de manera muy directa a las funciones de gestión o de negocio. Estas ventajas son:

- Consistencia arquitectural. El generador de código utilizado en un proyecto es el resultado de las decisiones arquitecturales realizadas antes de las diferentes fases del desarrollo. Las consecuencias directas de esta consistencia son: 1) el generador anima a los desarrolladores a trabajar dentro del marco definido por la arquitectura; 2) cuando se encuentran dificultades en que el generador realice alguna tarea, ésto indica que no se está trabajando dentro de la arquitectura actual; 3) si el generador está bien documentado proporciona una estructura y enfoque consistentes durante todo la vida del proyecto, incluso aunque exista rotación de la plantilla de desarrolladores.

- Abstracción. Las arquitecturas de los diferentes generadores de código separan las capas de la aplicación (negocio, acceso a datos, interfaz de usuario, definiciones del modelo, etc.) en ficheros de definición independientes del lenguaje de implementación. La separación de la semántica de la aplicación del código que implementa dicha semántica tiene beneficios claramente demostrados: 1) los desarrolladores podrán añadir nuevas plantillas que traducirán la lógica a otros lenguajes de programación, o incluso otras plataformas, de una manera mucho más sencilla que hacerlo de una forma manual; 2) es posible analizar y validar el diseño desde la abstracción; 3) capturar la semántica de la aplicación en un nivel abstracto puede ayudar al desarrollo de otros productos distintos al código de implementación, como por ejemplo casos de prueba o materiales de soporte.

- Mejora de la moral del equipo de desarrollo. Los proyectos muy duraderos pueden ser bastante duros para los equipos, y si además el código es tedioso

o “aburrido”, es incluso peor. La generación de código reduce los calendarios de los proyectos y mantiene a los desarrolladores enfocados en tareas interesantes, al contrario que en el manejo de grandes volúmenes de código “aburrido”. Además, al ser un código base con mayor calidad, los equipos se sienten más orgullosos del mismo.

- Desarrollo ágil. Una característica de las bases de código generado es la maleabilidad, esto es, que el software es mucho más fácil de cambiar o actualizar a lo largo del tiempo.

En cuanto a las desventajas de utilizar la generación de código como herramienta principal de desarrollo, podríamos destacar las siguientes:

- Incremento del número de pasadas de compilación. Generar código mediante lenguajes compilados implica el compilador ha de realizar una pasada más, una sobre las plantillas de generación de código, y otra sobre el código generado. Una alternativa a este enfoque podría ser utilizar otros lenguajes de programación para el generador de código, como podría ser Ruby [8] o Javascript [9]. Este enfoque es el que sigue JHipster, por ejemplo, que basa toda la generación de código en Javascript, generando aplicaciones full-stack en el que el backend está escrito en Java con el framework SpringBoot [10], y el front-end en Angular 4 [11].

- Lenta curva de aprendizaje. Cuando un nuevo desarrollador llega al equipo, es importante que aprenda a utilizar las herramientas de manera rápida. El generador de código simplifica tanto el proceso de desarrollo, que a partir de unas simples plantillas ya creadas por otros en el pasado es posible generar el código de la aplicación de una manera fácil. Puede ser fácil caer en el error de “conformarse” con una aplicación funcionando sin conocer los estados internos de la misma, estados definidos por el generador de código.

- Tentación de utilizar la generación de código para todo. Como dice A. Maslow en su famosa frase “*If all you have is a hammer, everything looks like a nail.*” [12], una vez tengamos un generador de código funcionando, intentaremos hacer todas las nuevas funcionalidades con el generador, cuando ésto podría no ser una buena idea.

- Difícil implementación de optimizaciones particulares. La generación de código es muy rápida y muy directa cuando las mejoras o correcciones de bugs son muy sencillas o cuando son muy transversales, aplicando de una manera general a la base de código. Pero cuando la solución a implementar o corregir es muy particular, el generador de código puede entorpecer una

solución más apropiada que de otra manera sería más rápida, puesto que intentaría aplicarla de una manera general. Por otro lado, si el generador tratase de aplicar la particularidad a un trozo de código únicamente, la complejidad del generador aumentaría, pues éste tendría que lidiar con bifurcaciones en la ejecución para determinar cuándo aplicar la particularidad. Por tanto su mantenibilidad disminuiría.

4.2. Calidad de un proyecto software

La calidad de un proyecto software es un valor bastante subjetivo, por lo que suele ser analizada según varias perspectivas. Sin embargo, las buenas prácticas en la industria recomiendan seguir ciertos enfoques para medir dicha calidad de una manera lo más objetiva posible. Para ello habitualmente se distingue entre criterios de calidad de producto y criterios de calidad de proceso.

Los siguientes conceptos, además de formar parte del día a día profesional del alumno, ha sido obtenidos de los conocimientos adquiridos durante el estudio de la asignatura “Gestión y Mejora de Procesos Software” de este mismo máster, donde se presentaron los trabajos “*Problemática y dificultades de la mejora de procesos software. Alternativas Agile*” e “*ITIL v3: Implantación del servicio Gestión de Versiones*” relacionados con la calidad de los procesos de desarrollo de software.

4.2.1. Calidad del producto software

En cuanto a calidad de producto, podríamos medir complejidad ciclomática [13], código duplicado (Copy&Paste), malas prácticas del lenguaje, encapsulamiento, uso de patrones, escritura de tests, cobertura del código, entre otras y no es este orden de importancia, puesto que la aplicación de cada una de ellas aporta valor a la calidad del software desarrollado.

4.2.1.1. Complejidad ciclomática

La complejidad ciclomática es una métrica del software que proporciona una medición cuantitativa de la complejidad lógica de un programa. Es una de las métricas de software de mayor aceptación, ya que ha sido concebida para ser independiente del lenguaje.

El resultado obtenido en el cálculo de la complejidad ciclomática define el número de caminos independientes dentro de un fragmento de código y determina la cota superior del número de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia al menos una vez.

La medida resultante puede ser utilizada en el desarrollo, mantenimiento y reingeniería para estimar el riesgo, coste y estabilidad.

4.2.1.2. Código duplicado

El denominado Copy&Paste consiste en la repetición de bloques significativos de código en más de un lugar de la base de código de un programa software, originado por la copia, pegado y luego edición del bloque en la nueva ubicación.

En general el código duplicado se considera una mala práctica, pues implica una falta de competencias en cuanto al diseño y codificación de aplicaciones, sin tener en cuenta la mantenibilidad del software a futuro, puesto que si fuera necesario modificar uno de los bloques, sería más que recomendable el aplicarlo a todos los lugares donde estuviera copiado o duplicado. Este proceso, si no se realiza con la ayuda de un entorno integrado de desarrollo, o IDE, es muy propenso a errores.

En alguna ocasión este código duplicado podría ser considerado como aceptable o necesario, por ejemplo en código boilerplate, o código que es necesario incluirlo en muchos lugar con poca o ninguna alteración.

El uso de herramientas como PMD + CPD [14], que automatizan la localización de este código duplicado, mejoran la salud del código.

4.2.1.3. Malas prácticas del lenguaje

Es interesante detectar los malos usos del lenguaje del software utilizado para codificar la aplicación, puesto que estos malos usos repercutirán en el rendimiento de la misma. Por ello es aconsejable utilizar las guías de buenas prácticas que los creadores del lenguaje suelen ofrecer, de modo que se exploten las capacidades del lenguaje sin penalizar en rendimiento, o mantenibilidad.

A modo de ejemplo, tanto Oracle como Microsoft ofrecen guías con las convenciones más habituales para el desarrollo utilizando sus lenguaje de programación, Java [15] y .NET [16], respectivamente.

El uso de herramientas específicas como Findbugs para Java [17], que automatizan la localización de potenciales errores por un mal uso del lenguaje, mejoran la salud del código.

4.2.1.4. Patrones de diseño

El concepto de patrones de diseño tiene sus orígenes en los trabajos del arquitecto Christopher Alexander. Alexander destaca cómo la relación entre los problemas recurrentes y sus respectivas soluciones establecen unos patrones de la siguiente manera:

“Cada patrón describe un problema, el cual ocurre una y otra vez en nuestro entorno, y por tanto describe el núcleo de la solución al problema de tal manera que es posible utilizar esta solución millones de veces, y sin hacerlo nunca de la misma manera dos veces” [18]

Veinte años más tarde, Gamma *et al.* [19] recogieron estas ideas y las adoptaron al campo del diseño de software orientado a objetos reutilizables. Los patrones de diseño ofrecen una manera conveniente de capturar, documentar, organizar y diseminar el conocimiento existente de un área del conocimiento determinada de una manera consistente y en un formato accesible.

Los patrones de diseño se diferencian de los algoritmos y de las estructuras de datos en que los conceptos que describen no pueden ser codificados y utilizados como subrutinas o como una clase objeto. Los patrones se diferencian de los frameworks en tanto que no describen una estructura de un sistema completo, sino que los patrones interrelacionados se utilizan típicamente juntos para resolver un problema de diseño general en un contexto dado.

4.2.1.5. Buenas prácticas de diseño

Es importante utilizar no sólo patrones de diseño, sino también buenas prácticas establecidas en la industria en cuanto al diseño del código. Ejemplos de estas buenas prácticas son el seguimiento de los principios SOLID [62], la Ley de Demeter [63] y el encapsulamiento en general.

El acrónimo SOLID viene representado por:

- S (Single Responsibility Principle): o Principio de responsabilidad única. Una clase sólo debería tener una única responsabilidad.

- O (Open/Closed Principle): o Principio de Abierto/Cerrado. Las entidades del software debería estar abiertas a la extensión pero cerradas para la modificación.

- L (Liskov Substitution Principle): o Principio de Sustitución de Liskov. Los objetos de un programa deberían ser reemplazables con instancias de sus subtipos win alterar la corrección del mismo.

- I (Interface Segregation Principle): o Principio de segregación de la interfaz. Es mejor tener muchas interfaces específicas a un cliente que una interfaz de propósito general.

- D (Dependency Inversion Principle): o Principio de la inversión de dependencias. Se debería depender sobre abstracciones, no sobre concreciones.

En cuanto a la Ley de Demeter [63], basta resumir sus principios, que acercan el diseño al bajo acoplamiento:

- Cada unidad debería tener un conocimiento limitado sobre otras unidades, esto es, sólo de aquéllas que están muy relacionadas.

- Cada unidad debería poder comunicarse únicamente con sus amigos, y no hablar con extraños.

- Sólo es posible hablar con los amigos más inmediatos.

En lo que se refiere al encapsulamiento, éste consiste simple y llanamente en esconder la complejidad. En el mundo real, los objetos esconden habitualmente la información sobre cómo funcionan, por lo que no es necesario conocer los *internals* del mismo para utilizarlo.

Al crear objetos en un lenguaje orientado a objetos, es posible esconder la complejidad del funcionamiento interno de estos objetos. Como desarrollador, existen dos razones por las cuales es importante esconder dicha complejidad.

La primera es proporcionar una manera simplificada y entendible de utilizar el objeto sin necesitar comprender la complejidad interna. Por ejemplo, un conductor no necesitaría conocer cómo funciona el motor de combustión interna de su vehículo.

Es suficiente con saber arrancar el coche, cambiar las marchas, repostar, parar el coche o apagar el motor. Para completar estas operaciones se sabe utilizar la llave, el acelerador, el embrague, el freno, entre otros. Estas operaciones básicas suponen la interfaz del coche, o conjunto de operaciones que es posible hacer sin conocer su funcionamiento interno.

Del mismo modo, escondiendo a un usuario la complejidad de un objeto permite a cualquiera utilizarlo y encontrar maneras de reutilizarlo en el futuro a pesar de desconocer su funcionamiento interno. Este concepto de mantener los detalles de implementación escondidos del resto del sistema es la clave del diseño orientado a objetos.

4.2.1.6. Escritura de tests

Un síntoma de que no podemos medir la calidad de un código es la ausencia de tests, o pruebas escritas que de manera automatizada se ejecutan para verificar comportamientos del software.

Podremos escribir:

- pruebas unitarias, o que no tienen dependencia con otros componentes, muy fáciles de automatizar.

- pruebas de integración, aquéllas que dependen de algún otro componente (servicio web, base de datos, sistema de archivos, otra clase o proyecto, etc.), relativamente fáciles de automatizar.

- pruebas funcionales, que se escriben en un lenguaje de alto nivel y representan la especificación funcional del sistema, en general fácil de automatizar pero extremadamente difíciles de mantener, puesto que simulan navegaciones reales de usuarios en el sistema, y ésto puede generar la aparición de errores al cambiar la interfaz de usuario más rápido que las pruebas.

- pruebas exploratorias, realizadas por seres humanos con un conocimiento avanzado del sistema, que les permite intentar romper el mismo utilizando todos los caminos posibles, no sólo el denominado “*happy path*” [20], o camino que sabemos que funciona.

Existen otros muchos tipos de pruebas, como pruebas de rendimiento, pruebas de usabilidad, pruebas de A/B, etc.

Los tests deben estar siempre actualizados respecto a las especificaciones funcionales del software, de lo contrario podríamos encontrar que los test avisan de falsos positivos. Esto es de suma importancia, puesto que los tests representan una documentación ejecutable que cambia con el tiempo, por lo que es posible obtener la trazabilidad de los requisitos observando la historia de cambios de los tests.

Si un test falla, una vez identificada la causa raíz que lo cause, será necesario:

1. actualizar el código del test para reflejar cambios en la especificación, o bien
2. actualizar el código del proyecto, si el tests demuestra que es incorrecto, o bien
3. eliminar el test, si éste ya no es necesario.

4.2.1.7. Cobertura de código

La cobertura de código es una medida utilizada para describir el porcentaje del código que es ejecutado por la ejecución de un conjunto de pruebas. Esto quiere decir que ese código tiene menos probabilidades de contener errores no detectados en comparación con un código con una baja cobertura de código.

Es posible medir esta cobertura según diferentes perspectivas:

- cobertura de paquete: porcentaje de paquetes cubiertos por los tests. Quiere decir que los tests ejecutan código de los paquetes cubiertos.

- cobertura de clase: porcentaje de clases cubiertas por los tests. Quiere decir que los tests ejecutan código de las clases cubiertas.

- cobertura de método: porcentaje de métodos cubiertos por los tests. Quiere decir que los tests ejecutan código de los métodos cubiertos.

- cobertura de línea: porcentaje de líneas cubiertas por los tests. Quiere decir que los tests ejecutan código de las líneas cubiertas. En este caso es importante destacar que las líneas que impliquen bifurcaciones (if/else/switch) y/o bucles (for/while) tienen que satisfacer todas las condiciones.

    - bifurcaciones: al menos un test debe ejecutar cada una de las ramas de la bifurcación, satisfaciendo dichas condiciones.

- bucles: al menos un tests debe ejecutar las condiciones de entrada y de salida del bucle, satisfaciendo dichas condiciones.

4.2.2. Calidad del proceso software

Si nos referimos a la calidad de proceso, podemos hablar de utilización de un repositorio de versionado del código, revisión entre pares, ejecución de pruebas automatizadas, integración continua, despliegue continuo, entrega continua, basándose todas ellas en un uso adecuado de la automatización de tareas.

4.2.2.1. Versionado de código

El no utilizar un versionado del código puede considerarse un anti-patrón en la industria de desarrollo de software. Sin él, no podemos garantizar la construcción de versiones de un proyecto, puesto que al no estar versionado, no sabemos qué se ha modificado entre cambio y cambio. Esta gestión del cambio incremental entre modificación y modificación permite a los desarrolladores tener una visión mucho más aislada y concreta del proyecto, puesto que son capaces de identificar qué hay que hacer para repetir una tarea simplemente con visualizar el cambio concreto que produjo algo parecido con anterioridad en la vida del proyecto.

4.2.2.2. Revisión entre pares

Si conseguimos que varios miembros del mismo equipo colaboren sobre un mismo desarrollo, estaremos poniendo muchos más ojos sobre una funcionalidad en concreto que si lo hiciera únicamente uno de ellos. Esta revisión, denominada habitualmente entre pares, o *peer-review* [21], si se realiza por otra persona, permite revisar el código propuesto desde el punto de vista de otra persona distinta al desarrollador original, que podría estar sesgado al haber desarrollado él o ella esa funcionalidad.

Si además la persona que revisa tiene más experiencia en el proyecto, podrá aconsejar o mentorizar al desarrollador a través de estas sesiones de revisión de código, de manera que el equipo mejore tanto en conocimiento funcional como en conocimiento técnico del proyecto en cuestión.

4.2.2.3. Pruebas automatizadas

Si hablamos de pruebas automatizadas, éstas son necesarias para demostrar de una manera repetible que un software funciona como se espera, esto es, según los requisitos funcionales. Sin ellas, podríamos afirmar que el software funciona de casualidad, puesto que las pruebas manuales nunca son suficientes para demostrar que el software funciona como se espera.

4.2.2.4. Integración continua

En cuanto a integración continua, se define como el proceso de integrar frecuentemente los cambios realizados por el equipo de desarrollo, en el repositorio central de versionado del código. De esta manera, los posibles conflictos que pudieran ocurrir debido a que se trabaje en los mismos ficheros aparecen mucho antes, en la fase de desarrollo, reduciendo notablemente la aparición en fases posteriores del desarrollo o incluso la explotación del mismo. Si además incluímos un proceso automatizado de construcción continua, posibles errores de compilación (en lenguajes compilados) no serán encontrados en sistemas en producción, puesto que lo habrá detectado en tiempo de desarrollo la integración continua.

En este proceso podría definirse una *pipeline* mínima en la que se construyera el proyecto, se ejecutaran las pruebas automatizadas, tanto unitarias como de integración, se empaquetara la aplicación, se versionara, y se archivara en un repositorio de versiones, de modo que se pudiera reproducir el estado de la aplicación en cualquier punto de la historia del proyecto. Ésto vendría garantizado por la existencia de un repositorio de código con cambios frecuentes.

El uso de servidores de integración continua como Jenkins [22], Bamboo [23], Travis CI [5], Go CD [24] facilitan la implantación de este proceso de desarrollo.

4.2.2.5. Despliegue continuo

Una vez que tengamos instaurada la buena práctica de la integración continua, si además somos capaces de coger esa versión generada por la automatización, y la desplegamos de manera controlada a una máquina interna, o a un entorno de desarrollo, estamos consiguiendo reproducir el comportamiento de la aplicación de una manera repetible: de manera automática llevamos el código desarrollado a un entorno de pruebas. Y como hemos conseguido que sea un proceso repetible, podremos conseguir añadir un paso más, que no es otro que pasar la misma versión

de la aplicación a un entorno de producción sin casi esfuerzos, idealmente pulsar un botón, de manera que cuando hayamos considerado que la versión en el entorno de desarrollo es correcta, pasemos entonces al entorno de producción.

Esta validación de corrección del entorno de desarrollo ha de hacerse también con buenas prácticas, a ser posible automatizadas, que permitan garantizar que el software se comporta como se espera. Un buen ejemplo sería la automatización de pruebas funcionales contra este entorno de desarrollo. Si las pruebas que consideremos aceptables pasan, entonces se puede pasar a producción, pulsando un botón.

4.2.2.6. Entrega continua

El enfoque de entrega continua es casi igual que el anterior de despliegue continuo, con la salvedad que el equipo está tan seguro que su código funciona como se espera, que automáticamente despliega contra producción, previo paso por las fases anteriores: desarrollo, integración, etc. Y esta seguridad se tiene gracias a tener instauradas las prácticas anteriores: pruebas automatizadas, revisión entre pares, integración continua, despliegue a desarrollo de manera continua, pruebas funcionales automatizadas, entre otras.

En este punto de madurez, el equipo es capaz de asumir que si las pruebas funcionales automatizadas pasan, la versión es válida para ser desplegada en el entorno de producción. Y simplemente añaden estos pasos al pipeline, de modo que sea posible que un código recién escrito pase del equipo del desarrollador por todos los entornos hasta llegar a producción.

4.3. DSL, o lenguajes específicos de dominio

El uso de los denominados DSLs [7], o lenguajes específicos de dominio, no es algo novedoso, puesto que aparece en la historia del desarrollo de software desde hace muchos años.

Estos lenguajes específicos de dominio se caracterizan por ser pequeños lenguajes, enfocados en un aspecto particular de un sistema software. Sin embargo, es imposible construir un programa por completo con ellos, aunque a menudo se utilizan múltiples DSLs en un sistema escrito principalmente en los lenguaje de propósito general.

Es posible encontrar los DSLs de dos maneras: externos e internos. Un DSL externo es un lenguaje que es parseado de manera independiente al lenguaje de propósito general anfitrión. Un buen ejemplo de ello serían las expresiones regulares o CSS. Los DSLs internos, por contra, son una forma particular de API en un lenguaje de propósito general, y a menudo denominados como interfaces "*fluent*", o fluidas.

En general, los desarrolladores valoran positivamente la existencia de DSLs, puesto que con un DSL bien diseñado puede ser mucho más fácil programar que con una librería tradicional, mejorando notablemente la productividad del desarrollador. Y ésto es siempre algo a valorar. En particular, además un DSL puede mejorar la comunicación con los expertos del dominio, suponiendo una herramienta importante para atajar uno de los problemas más importantes en el desarrollo de software. CSS es un ejemplo excelente de ello, puesto que no es necesario considerarse un desarrollador para poder escribir CSS. Sin embargo, a pesar de ello, no es habitual que los usuarios finales escriban directamente en el DSL. No obstante, es la mejora en la comunicación lo que realmente aporta valor a su uso.

A pesar que los DSLs no son algo moderno, llevan mucho tiempo entre nosotros, la falta de conocimiento sobre cómo desarrollar con ellos supone una barrera considerable.

Podríamos considerar un DSL como bien diseñado si se basa en los tres siguientes principios [25]:

1. Un DSL proporciona un mapeo directo a los artefactos de dominio del problema.

2. El DSL debe utilizar el vocabulario común del dominio del problema. Este vocabulario se convierte en el catalizador para una mejor comunicación entre los desarrolladores y los expertos del dominio.

3. El DSL debe abstraer la implementación subyacente. No puede contener complejidad accidental que afecte a los detalles de implementación.

Existen muchas discusiones en la web sobre las ventajas y desventajas de los DSLs. De hecho, cuanto mejor diseñados están, más fácil es el proceso de escribir programas con ellos.

Enumerando las ventajas, podemos citar que los DSL tienen mayor expresividad en comparación con los lenguajes de propósito general, pudiendo reducir notablemente el tiempo de desarrollo de una aplicación. Además, mejoran la corrección de ésta, así como la comunicación entre los expertos del dominio y los desarrolladores.

Los DSL puede ser utilizados como un mecanismo para proteger los sistemas software como propiedad intelectual, al definir construcciones propietarias que definan el comportamiento de la aplicación. Por otro lado, al usar un vocabulario específico del dominio, la documentación del código es mucho mejor, o incluso llegar al caso de no ser ésta requerida, al ser el propio lenguaje especializado una documentación de las operaciones a realizar, siempre y cuando el lenguaje utilizado sea claro y describa de manera unívoca las operaciones a realizar.

En cuanto a sus desventajas, a pesar de que se reduce el precio final del desarrollo conjunto del software, a menudo se considera como un alto coste que los desarrolladores tengan que ser expertos del dominio para poder sacar toda la potencia del DSL. En tales casos, la creación del DSL requiere además un completo conocimiento de las limitaciones y los límites del dominio, que en caso contrario, puede llevar a la creación de un DSL incompleto o incluso inusable.

Entrando con más profundidad sobre los tipos de DSL, podemos distinguir entre externos e internos, como se muestra a continuación.

4.3.1. DSL externos

En un lenguaje que es diferente del lenguaje principal (habitualmente de propósito general). Algunos ejemplos de DSL externos son SQL, CSS y HTML. Muchos de ellos están ligados a una tecnología o infraestructura en particular. Y a menudo tales DSL son interpretados o traducidos a código en lenguaje de propósito general a través de herramientas de generación de código. Otro ejemplo claro es XML. A medida que cogió popularidad, muchos DSLs externos han sido modelados con él.

Las ventajas de los DSL externos incluyen una baja especificación y un mínimo o incluso un no-cumplimiento de los estándares. De esa manera, los desarrolladores pueden expresar los artefactos del dominio de una manera útil y compacta. La calidad de tales lenguajes depende de manera muy clara de la habilidad del desarrollador para escribir un intérprete o generador de código de alta calidad.

Los DSL externos no se integran en el lenguaje principal y por tanto operaciones como la refactorización se complican o incluso se hacen imposibles de implementar de manera automática.

4.3.2. DSL internos

Los DSL internos están embebidos en el lenguaje principal y por tanto se integran completamente entre los símbolos del mismo. Suponen una manera muy específica de utilizar los lenguajes de programación de propósito general. Además, proporcionan una manera amigable de hacer syntactic sugar en el dominio con los APIs existentes, utilizando las construcciones nativas del lenguaje de programación subyacente.

Algunos lenguajes de propósito general son más fácilmente extensibles mediante DSLs internos. Sin embargo, la calidad del DSL interno dependerá de las características del lenguaje anfitrión. Algunos autores se refieren a los DSL internos como interfaces *fluent*, término que enfatiza el hecho de que un DSL interno es en realidad un tipo de API, pero un API diseñado para que su vocabulario sea adecuado para construcciones a modo de sentencias, en lugar de secuencias de invocaciones de métodos. Estas construcciones tendrían sentido incluso en contextos aislados.

Debido a que complican la sintaxis del lenguaje anfitrión, no son fácilmente legibles por no-desarrolladores, tal y como sí era posible con los DSL externos. Además, la gramática del lenguaje anfitrión impone unas restricciones sobre las posibilidades de expresividad del DSL interno: algunos lenguajes son dinámicos, mientras que en otros conseguir esa flexibilidad podría ser un gran reto.

Existen dos enfoques para implementar los DSLs internos: heterogéneos y homogéneos.

4.3.2.1. DSL internos heterogéneos

En los DSL internos y heterogéneos, el lenguaje anfitrión y el lenguaje embebido no son procesados por el mismo intérprete/compilador, por tanto se necesitarán dos intérpretes o compiladores, uno para cada lenguaje.

4.3.2.2. DSL internos homogéneos

En los DSL internos y homogéneos, el lenguaje anfitrión y el lenguaje embebido sí son procesados por el mismo intérprete/compilador, por tanto sólo será necesario uno de ellos.

4.3.3. Patrones y DSL

El patrón por excelencia para el diseño de DSLs es el patrón de tipo creacional Builder, que soluciona problemas de creación de instancias a partir de datos parciales, ayudando a encapsular y abstraer dicha creación.

4.3.3.1. Patrón Builder

El patrón Builder [26] es utilizado para permitir la creación de una variedad de objetos complejos desde un objeto fuente, o Producto. El objeto fuente se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo a través de un conjunto de llamadas a interfaces comunes de la clase Abstract Builder.

Su intención es abstraer el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.

Las clases que entran en juego en este patrón son:

- Builder, o interfaz abstracta para crear productos.
- Builder Concreto, o implementación del Builder, que construye y reúne las partes necesarias para construir los productos.
- Director, que construye un objeto usando el patrón Builder.
- Producto, el objeto complejo bajo construcción.

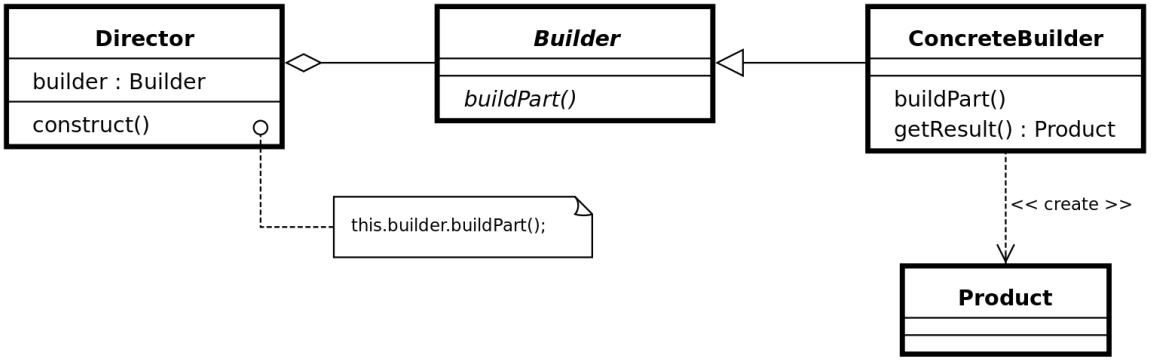


Diagrama de clases del patrón Builder.

Entre los beneficios más claros de utilizar este patrón, tenemos:

- Reduce el acoplamiento.
- Permite variar la representación interna de estructuras complejas, respetando la interfaz común de la clase Builder.
- Se independiza el código de construcción de la representación. Las clases concretas que tratan las representaciones internas no forman parte de la interfaz del Builder.
- Cada Builder concreto tiene el código específico para crear y modificar una estructura interna concreta.
- Distintos Director con distintas utilidades (visores, parsers, etc) pueden utilizar el mismo Builder concreto.
- Permite un mayor control en el proceso de creación del objeto. El Director controla la creación paso a paso, solo cuando el Builder ha terminado de construir el objeto lo recupera el Director.

De este modo, es muy importante en la generación de APIs *fluent*, pues permite la composición de operaciones parciales de creación sobre los objetos que queramos construir.

4.4. Service Builder: generador de código de aplicaciones basadas en Liferay Portal

Para poder buscar una solución a cualquier problema, o una alternativa a un escenario existente, es necesario entender tanto el problema como el escenario de una manera absoluta.

En este caso, en el que queremos mejorar la manera de desarrollar aplicaciones basadas en Liferay Portal, primero tendremos que entender cómo se escriben estas aplicaciones.

Tal y como se ha mencionado con anterioridad, Liferay Portal ofrece ciertos servicios de infraestructura para optimizar el uso de los recursos, así como para aprovechar al completo el potencial de la plataforma.

Service Builder es una herramienta de generación de código guiada por el modelo construída por Liferay, que permite la definición de modelos de objetos personalizados denominados *entidades*. Service Builder genera una capa de servicio mediante el uso de “mapeos objeto/relacionales”, en adelante ORM [29], que se refiere a la técnica de mapear una representación de datos desde un modelo de objetos a un modelo de datos relacionales con un esquema basado en SQL. De esta manera, se proporciona una clara separación entre el modelo de objetos y el código de acceso a la base de datos subyacente.

Service Builder es una aplicación OSGi [30], por tanto empaquetada en un archivo JAR, que se añade como dependencia a la aplicación que se quiera construir siguiendo este enfoque de generación de aplicaciones para Liferay Portal. Añade ciertas tareas a los scripts de construcción, que habilitan la generación del código, de modo que se puedan integrar en el ciclo de vida del desarrollo.

Si ponemos el foco en el lenguaje utilizado por Service Builder, tal y como hemos visto con anterioridad acerca de los DSL, podemos afirmar que Service Builder incorpora un DSL externo para la generación de código. Este DSL se basa en XML para construir un descriptor del modelo, definiendo en este XML las operaciones válidas mediante el anidamiento de elementos XML. A modo de ejemplo, la operación “Definir una entidad en el modelo”, vendrá dada por la operación “añadir un elemento <entity> dentro de un elemento <service-builder>”.

En cuanto a generación de código desde el punto de vista conceptual, Service Builder forma parte de los denominados generadores de clases parciales, o “Partial-Class generator”, los cuales leen un fichero de definición abstracto que contiene suficiente información para construir un conjunto de clases. A continuación, usa unas plantillas para construir una librería de clases base como salida. Estas clases son compiladas con otras clases escritas por los desarrolladores para completar el conjunto de clases de producción.

La ventaja principal de construir una clase de manera parcial es que permite sobreescribir la lógica de las clases generadas en el caso de que la lógica de negocio tenga unos requisitos personalizados que no puedan ser cubiertas por el generador.

Una vez el generador de clases parciales construye la clase base, los desarrolladores añaden personalizaciones en una clase derivada. Este enfoque de

generación parcial más derivaciones está determinado por la generación de ciertas clases base que serán redefinidas en tiempo de ejecución de Service Builder con los métodos definidos por los desarrolladores en las clases propias del desarrollo, donde éstos podrán añadir las personalizaciones y la lógica de negocio propia de la aplicación, excepto en las clases de persistencia. De hecho, las primeras son las únicas clases generadas por Service Builder que pueden ser personalizadas. Asegurando que todas las personalizaciones son realizadas en unas pocas clases, hace que las aplicaciones creadas con Service Builder sean más fáciles de mantener.

En la actualidad, Service Builder supone una de las piezas más importantes de Liferay Portal, pues la inmensa mayoría de las aplicaciones incluidas de serie en Liferay Portal están construidas encima de Service Builder.

4.4.1. Capas generadas

Service Builder toma el fichero XML descriptor del modelo como entrada y genera diferentes capas de código. Genera capas para el modelo, la persistencia y el servicio de las aplicaciones basadas en Liferay Portal, suponiendo estas capas una clara separación de conceptos que facilita el desarrollo de aplicaciones.

La capa del modelo es la responsable de definir los objetos que representan a las entidades de la aplicación; la capa de persistencia es responsable de guardar y recuperar las entidades de la base de datos; y la capa de servicio es responsable de exponer las operaciones CRUD, así como otras operaciones de interés en forma de un API.

Cada entidad generada por Service Builder dispone de:

- un conjunto de clases de implementación del modelo.
- un conjunto de clases de implementación de la persistencia.
- un conjunto de clases de implementación de un servicio local.
- un conjunto de clases de implementación de un servicio remoto (opcional).

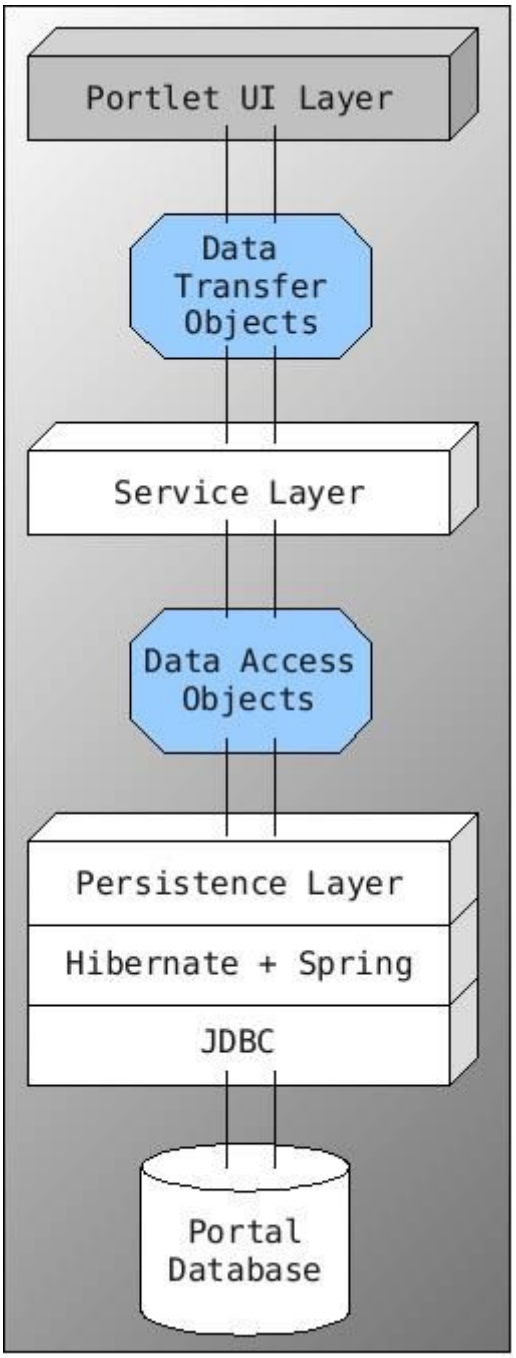


Diagrama de las capas generadas con Service Builder.

4.4.1.1. Capa de modelo

La capa del modelo recoge toda la información relativa a las entidades, como son los atributos propios de las mismas, y cualquier otra operación sobre estos datos que se requiera añadir.

Service Builder genera la siguiente jerarquía de clases para la capa de modelo:

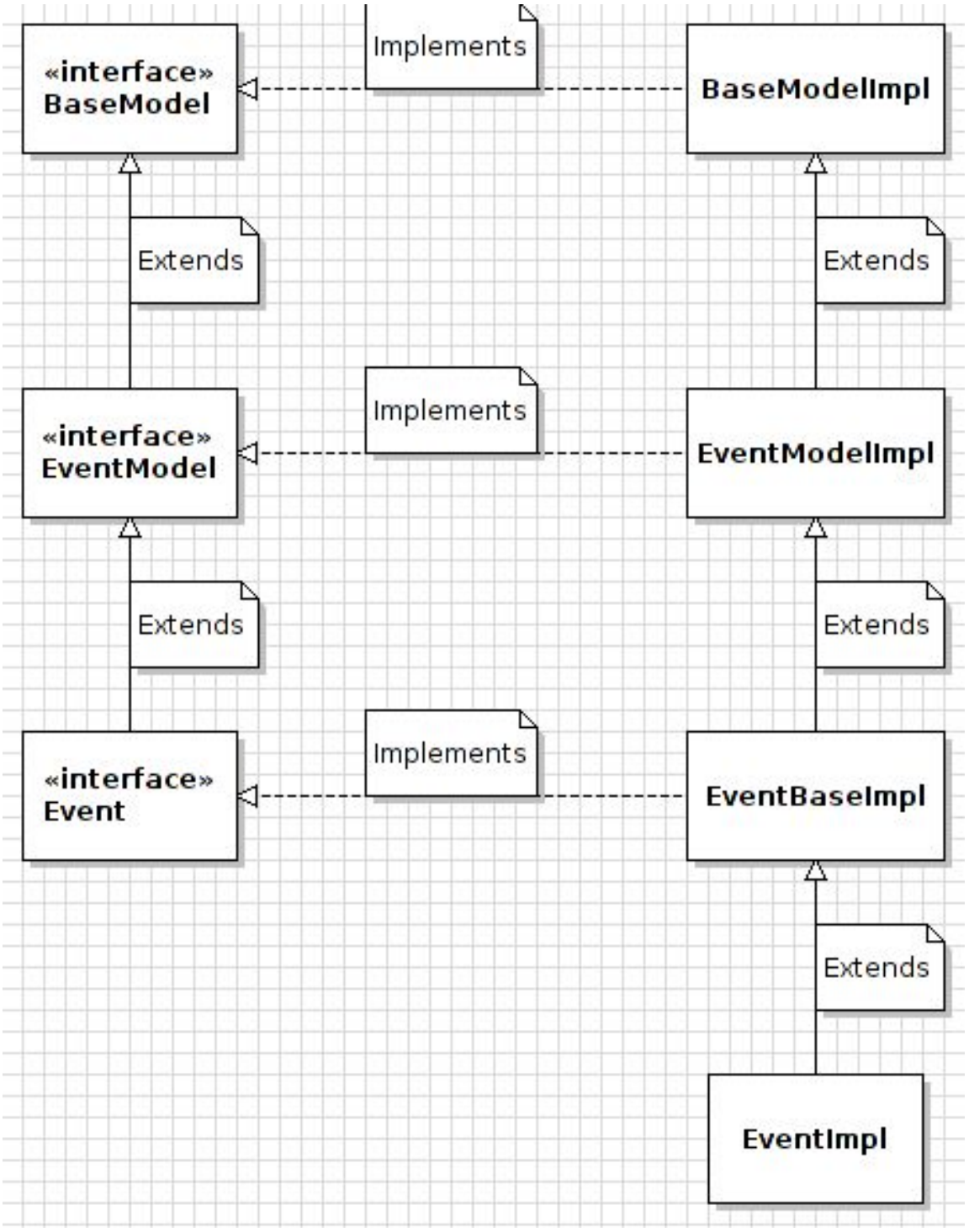
- [Entity]Model: Interfaz base del modelo. Esta interfaz, junto a su implementación [Entity]ModelImpl, sirven únicamente como contenedor de los métodos getter y setter de acceso a las propiedades del modelo, generados por Service Builder. Cualquier método de utilidad o ayuda, así como toda la lógica de aplicación aplicada en el modelo debe ser añadida a la clase [Entity]Impl.

- [Entity]ModelImpl: Clase de implementación base del modelo.

- [Entity]: Interfaz del modelo que extiende [Entity]Model.

- [Entity]Impl: Clase de implementación del modelo. En esta clase es posible añadir al modelo métodos de utilidad o ayuda y lógica de la aplicación. Si no se añaden ningún método, sólo los métodos getters y setters autogenerados estarán disponibles. Por el contrario, añadiendo métodos en esta clase hace que Service Builder añada los correspondientes métodos en la interfaz [Entity] la siguiente vez que se ejecute.

- [Entity]Wrapper: Clase de envoltura del modelo, que envuelve a la interfaz [Entity].



Clases e interfaces del modelo generadas por Service Builder para una entidad ficticia, denominada "Event". Únicamente EventImpl permite la adición de métodos personalizados

4.4.1.2. Capa de persistencia

En cuanto a la capa de persistencia, genera la mayoría del código común, a veces denominado *boilerplate*, para implementar las operaciones CRUD (Create, Read, Update, Delete), así como otras operaciones de búsqueda sobre la base de datos,

permitiendo a los desarrolladores centrarse en los aspectos de diseño de más alto nivel de los servicios. Este código generado es además agnóstico de la base de datos, como Liferay Portal, de modo que se podrá aplicar a las diferentes bases de datos soportadas, que a día de hoy son MySQL [31], MariaDB [32] y PostgreSQL [33], en la versión Community Edition de Liferay Portal, y SQL Server [34], Oracle [35], DB2 [36] y Sybase [37], en la versión Enterprise.

Service Builder proporciona la habilidad de generar métodos de búsqueda (*finder methods*), que recuperan de la base de datos objetos representando a una entidad a partir de unos parámetros especificados. Además, proporciona métodos *finder* que tienen en cuenta el sistema de permisos de Liferay Portal para detectar si se tiene acceso a los datos (autorización).

Otra funcionalidad generada por Service Builder, relacionada con la persistencia, es el uso de cachés. Liferay Portal cachea los objetos en tres niveles:

- a nivel de entidad,
- a nivel de método finder,
- a nivel de Hibernate.

Para habilitar el cacheado a nivel de entidad, basta con definirlo en el descriptor XML.

Si la consulta de recuperación de datos de la base de datos es relativamente compleja, por ejemplo recuperando datos de múltiples tablas con un JOIN de SQL, posiblemente Hibernate [38] y los métodos finder no sean suficiente para tal fin. Por esto Service Builder permite además definir consultas SQL personalizadas en un fichero XML, e implementar unos métodos finder específicos para tal fin, que ejecutarán dichas consultas.

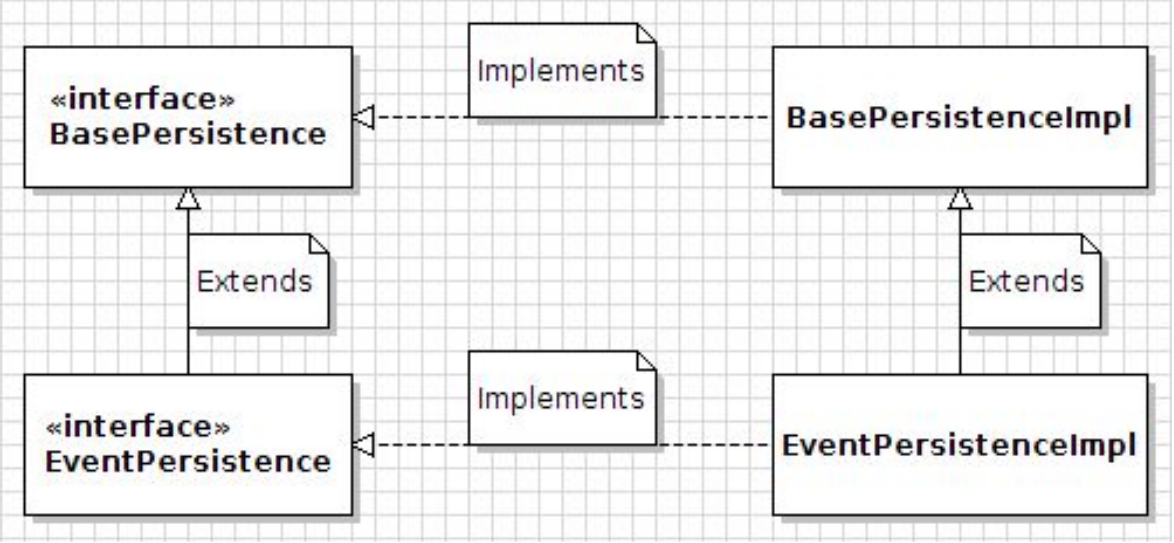
Para terminar, Service Builder genera además código para dar soporte a consultas dinámicas utilizando el API de criterios de Hibernate, que permite especificar consultas de manera programática sobre las entidades definiendo un conjunto de restricciones.

Service Builder genera la siguiente jerarquía de clases para la capa de persistencia:

- [Entity]Persistence: Interfaz que define los métodos CRUD para la entidad, tales como create, remove, countAll, find, findAll, etc.

- [Entity]PersistenceImpl: Clase de implementación, que implementa la interfaz [Entity]Persistence.

- [Entity]Util: Clase de que envuelve a [Entity]PersistenceImpl y proporciona acceso directo a la base de datos mediante operaciones CRUD. La recomendación de Liferay es que esta clase de utilidad sea utilizada únicamente desde la capa de servicio.



Clases e Interfaces de persistencia generadas por Service Builder, generadas para una entidad ficticia, denominada "Event".

4.4.1.3. Capa de servicios

La clase de implementación del servicio local es responsable de las llamadas a la capa de persistencia, para recuperar y almacenar los datos de las entidades. Estas clases de servicios locales contienen la lógica de negocio específica de la aplicación. Pueden ser invocadas únicamente por clientes ejecutándose en la misma máquina virtual Java (JVM [39]).

Service Builder genera la siguiente jerarquía de clases para los servicios locales:

- [Entity]LocalService: Interfaz del servicio local.

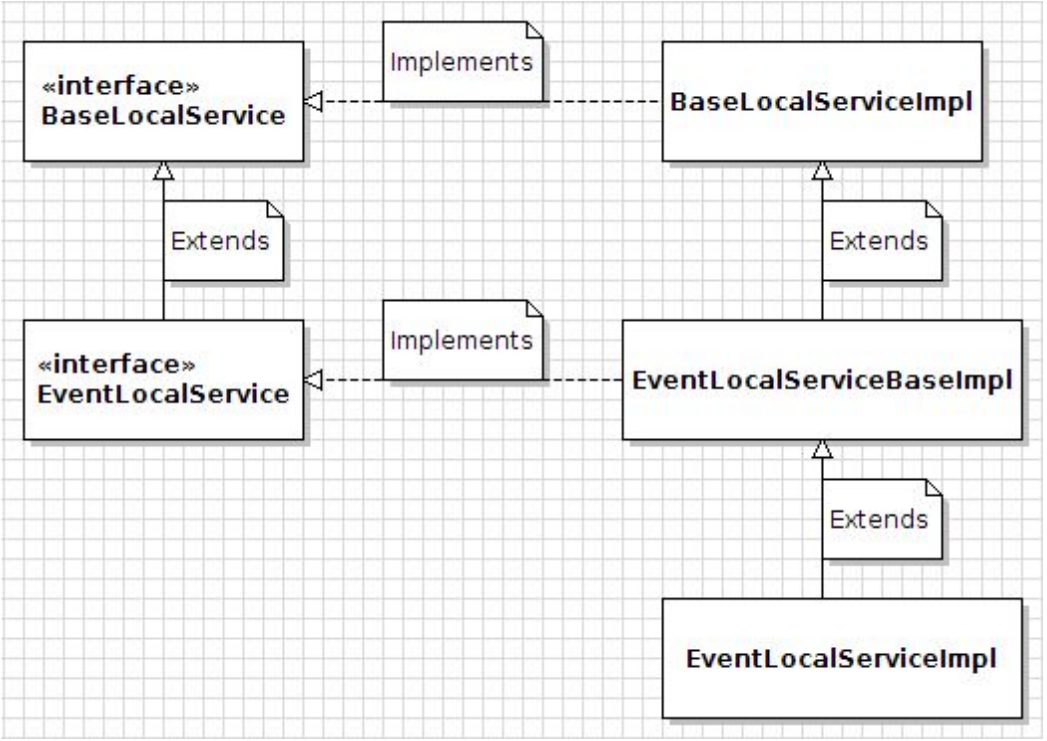
- [Entity]LocalServiceImpl: Clase de Implementación del servicio Local. Ésta es la única clase del servicio local que puede ser modificada de forma manual, por lo que será aquí donde se añadirá la lógica de negocio. Por cada método personalizado añadido en esta clase, Service Builder añadirá en la siguiente

generación de código el correspondiente método en la interfaz [Entity]LocalService, de modo que sea expuesto hacia el API.

- [Entity]LocalServiceBaseImpl: Clase de implementación base del servicio local. Esta clase es abstracta, por lo que no podrá ser instanciada. Service Builder inyecta en esta clase instancias de otros servicios, así como las clases de persistencia.

- [Entity]LocalServiceUtil: Clase de utilidad del servicio local, que envuelve [Entity]LocalServiceImpl y que, antes de OSGi, servía como de punto de entrada principal a la capa de servicio. En la actualidad los servicios son recuperados directamente con OSGi, recogiendo las propias implementaciones de las interfaces. Esta es la forma actual y preferida de hacerlo, y *se mantienen las clases de utilidad en la generación de código por compatibilidad hacia atrás*. A modo de ejemplo, se recogería una instancia de una clase que implemente la interfaz [Entity]LocalService.

- [Entity]LocalServiceWrapper: Clase de envoltura del servicio local, que implementa [Entity]LocalService. Esta clase está diseñada para ser extendida y permitir a los desarrolladores personalizar los servicios locales de la entidad.



Clases e interfaces de los servicios locales generadas por Service Builder para una entidad ficticia, denominada "Event". Únicamente EventLocalServiceImpl permite la adición de métodos personalizados.

Los servicios remotos habitualmente incluyen código adicional de comprobación de permisos de ejecución, y están pensados para ser accedidos desde cualquier ubicación, ya sea Internet o la propia red local. Service Builder genera de manera automática el código necesario para permitir el acceso a estos servicios remotos, que incluyen utilidades SOAP [40] y pueden ser accedidos vía SOAP o JSON [41].

Service Builder genera la siguiente jerarquía de clases para los servicios remotos:

- [Entity]Service: Interfaz del servicio remoto.

- [Entity]ServiceImpl: Clase de implementación del servicio remoto. Ésta es la única clase del servicio remoto que puede ser modificada de forma manual, por lo que será aquí donde se añadirán las comprobaciones de seguridad pertinentes, junto con una invocación al método correspondiente del servicio local. Por cada método personalizado añadido en esta clase, Service Builder añadirá en la siguiente generación de código el correspondiente método en la interfaz [Entity]Service, de modo que sea expuesto hacia el API.

- [Entity]ServiceBaseImpl: Clase de implementación base del servicio remoto. Es una clase abstract, por lo que no podrá ser instanciada.

- [Entity]ServiceUtil: Clase de utilidad del servicio remoto, que envuelve [Entity]ServiceImpl y que, antes de OSGi, servía como de punto de entrada principal a la capa de servicio remoto. En la actualidad los servicios son recuperados directamente con OSGi, recogiendo las propias implementaciones de las interfaces. Esta es la forma actual y preferida de hacerlo, y se mantienen las clases de utilidad en la generación de código por compatibilidad hacia atrás. A modo de ejemplo, se recogería una instancia de una clase que implemente la interfaz [Entity]lService.

- [Entity]ServiceWrapper: Clase de envoltura del servicio remoto, que implementa [Entity]Service. Esta clase está diseñada para ser extendida y permitir a los desarrolladores personalizar los servicios remotos de la entidad.

- [Entity]ServiceSoap: Clase de utilidad SOAP a la que el utilidad del servicio remoto [Entity]ServiceUtil puede acceder.

- [Entity]Soap: Clase SOAP del modelo, similar a [Entity]ModelImpl, con la particularidad que [Entity]Soap es serializable y no implementa [Entity].

4.4.2. Definición del modelo

La definición de las entidades del modelo de Liferay Portal se definen en un archivo XML denominado *service.xml*.

A modo de ejemplo de descriptor de los modelos, puede consultarse el descriptor de la aplicación de gestión documental [60].

En este documento XML se identifican varios bloques:

- Definición del espacio de nombres de la aplicación que servirá de paraguas para las entidades del modelo, identificado por el elemento XML <*service-builder/*>.
    - Metadatos del espacio de nombres, identificados como atributos XML del elemento <*service-builder/*>.
- Definición de entidades, identificadas como elementos XML <*entity/*>.
    - Metadatos de la entidad, identificados como atributos XML del elemento <*entity/*>.
    - Atributos de la entidad, identificados como elementos <*column/*>, con sus metadatos de columna.
    - Criterios de ordenación sobre la entidad, identificados como elementos <*order-column/*>.
    - Operaciones de consulta sobre la entidad, identificadas como elementos <*finder/*>, que generarán métodos Java para realizar consultas en la base de datos.
    - Referencias a otras entidades del modelo, identificadas como elementos <*reference/*>, que generarán código para acceder a otras entidades desde la entidad de interés.
- Definición de excepciones, identificadas como elementos <*exceptions/*>, que generarán clases Java específicas para el tratamiento de errores de las entidades del modelo.

Al ser un descriptor basado en XML, Liferay Portal ofrece una definición del tipo de documento, o DTD [42], en el que documenta y define los diferentes elementos XML que son válidos en el fichero, de modo que se valide la corrección del XML formado. Este DTD se encuentra disponible online para su consulta [61].

De esta manera, un desarrollador podría construir una aplicación a partir de un fichero XML representando un modelo, y Service Builder generaría toda las capas de aplicación necesarias para dar soporte a la misma.

Pasemos a detallar todos y cada uno de los elementos que es posible utilizar durante la definición del modelo.

4.4.2.1. Entidad del modelo *service-builder*

El elemento *service-builder* es el elemento raíz del descriptor de despliegue de Service Builder para generar los servicios que estarán disponibles desde una aplicación basada en Liferay. La generación de código producirá utilidades de Spring [43], de SOAP, y las clases de persistencia de Hibernate para facilitar el desarrollo de los servicios.

```
<service-builder api-package-path="com.liferay.document.library.kernel"
auto-import-default-references="false"
package-path="com.liferay.portlet.documentlibrary">
```

Ejemplo de elemento service-builder.

Los atributos de este elemento son:

- package-path: especifica el nombre del paquete donde generar el código.
- auto-import-default-references: especifica si es necesario o no aplicar las referencias por defecto. Su valor predeterminado es true.
- auto-namespace-tables: especifica si los nombres de las tablas deben comenzar por el espacio de nombres definido.
- mvcc-enabled: especifica si es necesario habilitar el control de concurrencia multiversión sobre las filas de las tablas. Su valor predeterminado es false.

Únicamente puede existir un elemento service-builder en el fichero XML descriptor del modelo.

4.4.2.2. Entidad del modelo *author*

El elemento *author* representa el nombre del usuario asociado con el código generado.

```
<author>Manuel de la Peña</author>
```

Ejemplo de elemento author.

Sólo puede existir un elemento author dentro del elemento raíz service-builder, siendo opcional su existencia.

4.4.2.3. Entidad del modelo *namespace*

El elemento *namespace* representa el espacio de nombres del modelo. Este nombre debe ser único. Los nombres de las tablas generadas estarán prefijadas con este namespace. Por otro lado el código Javascript para acceso a los servicios JSON tendrá este namespace como scope: por ejemplo, Liferay.Service.NAMESPACE.

```
<namespace>DL</namespace>
```

Ejemplo de elemento namespace.

Sólo puede existir un elemento namespace dentro del elemento raíz service-builder, siendo obligatoria su existencia.

4.4.2.4. Entidad del modelo *entity*

El elemento *entity* representa habitualmente a una tabla de la base de datos, aunque si no contiene ninguna columna representará a una fachada de acceso a servicios.

```
<entity human-name="document library content" local-service="true" name="DLContent"
remote-service="false">
```

Ejemplo de elemento entity.

Los atributos de este elemento son:

- name: define el nombre de la entidad.

- human-name: define el nombre de la entidad en formato entendible por un humano. Si no se especifica ninguno, se utilizará el valor del atributo name.

- table: especifica el nombre de la tabla de la entidad en la base de datos.

- uuid: si su valor es true generará una columna de tipo UUID [44] para el servicio. Se generarán métodos para buscar y eliminar en base a este UUID. Su valor predeterminado es false.

- uuid-accessor: si su valor es true, generará un acceso a la columna (column-accessor) de tipo UUID para el servicio. Este acceso permitirá una manera rápida y segura en cuanto a tipos de acceder al campo UUID de la entidad.

- local-service: define si se han de generar las interfaces locales (o de servicios locales) para el servicio. Su valor predeterminado es false.

- remote-service: define si se han de generar las interfaces remotas (o de servicios web) para el servicio. Su valor predeterminado es true.

- persistence-class: especifica el nombre de una clase personalizada de persistencia, que a su vez debe implementar la interfaz autogenerada, o extender la clase de persistencia. Este permite sobreescribir comportamientos sin modificar las clases autogeneradas.

- data-source: especifica la fuente de datos a la que conectar la clase de persistencia. El valor predeterminado es la fuente de datos de Liferay Portal.

- session-factory: especifica la factoría de sesiones de acceso a datos que se asignará a la clase de persistencia. El valor predeterminado es la factoría de Liferay Portal.

- tx-manager: especifica el gestor de transacciones que usará Spring. El valor por defecto es el gestor de Spring, en el que Hibernate que envuelve a la factoría y a la fuente de datos de Liferay. Marcando como “none” este atributo, se deshabilitará el gestor de transacciones.

- cache-enabled: especifica si las consultas de base de datos de esta entidad se cachearán. Si los datos de la tabla pudieran ser modificados por otros programas, marcar este valor a false.

- dynamic-update-enabled: especifica si las propiedades sin modificar son excluidas en las consultas de actualización de SQL. El valor predeterminado es el valor del atributo mvcc-enabled.

- mvcc-enabled: especifica si se habilita el control de concurrencia multiversión en la entidad para prevenir pérdidas por actualizaciones. Su valor predeterminado es el valor del elemento service-builder.

- json-enabled: especifica si una entidad debe ser anotada para ser serializada en los servicios web JSON.

- trash-enabled: especifica si Service Builder debe generar métodos relacionados con la papelera, esto es, enviar registros a un estado intermedio previo a la eliminación definitiva de la base de datos.

- deprecated: marca la entidad como deprecada.

Pueden existir tantos elementos entity como se necesiten dentro del elemento raíz service-builder.

4.4.2.5. Entidad del modelo *column*

El elemento *column* representa una columna de la base de datos.

```
<column name="contentId" primary="true" type="long" />
```

Ejemplo de elemento column.

Los atributos de este elemento son:

- name: define el nombre de la columna en la base de datos, así como los métodos get y set para recuperar y almacenar esos valores en las clases java del modelo.

- type: especifica el tipo de datos de la columna, por ejemplo String, Boolean, Integer, etc.

- db-name: especifica si el nombre físico de la columna debe ser diferente al definido en el atributo name.

- primary: define si la columna forma parte de la clave primaria de la tabla.

- mapping-key: este atributo se mantiene por compatibilidad hacia atrás, y actualmente se asume su valor con el de la clave primaria de la entidad.

- entity y mapping-table: la asignación de valores a estos dos atributos definirán el comportamiento de la columna respecto a relaciones muchos a muchos, especificando el primero el nombre del método que devolverá la contrapartida de la relación, siendo el segundo el nombre de la tabla de relación.

- id-type e id-param: la combinación de estos atributos definirán la generación de claves primarias con auto-incremento a la hora de insertar registros en las tablas. Existen cuatro combinaciones válidas, dependiendo del tipo de base de datos utilizada. En todo caso, la clave primaria del objeto del modelo recibirá un valor nulo y será Hibernate el que sepa cómo reemplazar ese nulo por el valor auto-incremental. Si no se especifica un id-type, entonces se asumirá que la primary-key será asignada de otra manera, y por tanto no se autogenerará. Las cuatro combinaciones son las siguientes:

    a. Utilizar una clase para generar la clave primaria. id-type tomará el valor "*class*" e id-param el nombre cualificado de la clase de generación.

    b. Utilizar un incremento válido cuando no existan otros procesos insertando en la misma tabla. id-type tomará el valor "*increment*". Esta implementación no está recomendada para entornos en clúster.

    c. Utilizar una columna de tipo identidad. id-type tomará el valor "*identity*". La consulta SQL de creación de tabla generará una columna de tipo identidad que autogenerará de manera nativa la clave primaria al realizar inserciones en la tabla. Esta implementación sólo está soportada en DB2, MySQL y MS SQL Server.

    d. Utilizar una secuencia. id-type tomará el valor "sequence" e id-param el nombre de la secuencia. Esta implementación sólo será válida en aquellas bases de datos que soporten secuencias, como son DB2, Oracle, PostgreSQL y Sybase.

- accessor: especifica si es necesario generar un accessor para la columna.

- filter-primary: especifica si la columna utiliza la columna de la clave primaria en los métodos de búsqueda con filtros. Una única columna puede tener este

valor a true. Si ninguna columna es marcada como tal, se utilizará la clave primaria por defecto.

- convert-null: especifica si los valores de la columna son convertido a un valor no nulo en caso de ser nulos. Únicamente aplica a los campos de tipo String. El valor predeterminado es true.

- lazy: este atributo es únicamente válido si el tipo de columna es de tipo Blob [45], y especifica si se debe hacer una recuperación de tipo *lazy* [46], únicamente al ser utilizada, de los campos Blob. El valor predeterminado es true.

- localized: especifica si los valores de la columna admiten diferentes valores en función del idioma de la aplicación. El valor por predeterminado es false.

- json-enabled: especifica si la columna debe ser anotada para serialización en los servicios web JSON. Por defecto, si el valor en la entidad es true, los valores de este atributo así lo serán también.

- container-model: especifica si la columna representa a la clave primaria de un modelo que contiene a otras entidades.

- parent-container-model: especifica si la columna representa a la clave primaria de un modelo que contiene a esta entidad.

Pueden existir tantos elementos column como se necesiten dentro de un elemento entity, siendo su presencia opcional.

4.4.2.6. Entidad del modelo *order*

El elemento *order* especifica un orden predeterminado de las entidades al ser recuperadas de la base de datos.

```
<order by="asc">
```

Ejemplo de elemento order.

El único atributo que tiene este elemento es:

- order: con valores ASC o DESC; define el orden ascendente o descendente, respectivamente, predeterminado para la recuperación de las filas de la tabla de la entidad.

Sólo puede existir un elemento order dentro de cada elemento entidad, siendo opcional su existencia.

4.4.2.7. Entidad del modelo *order-column*

El elemento *order-column* permite refinar la ordenación de los registros de una tabla en base a las filas de la misma.

```
<order-column name="folderId" />
```

Ejemplo de elemento order-column.

Los atributos que tiene este elemento son:

- name: especifica el nombre de la columna a utilizar en la ordenación.
- case-sensitive: especifica si la ordenación debe utilizar criterios sensibles a mayúsculas.
- order-by: especifica si la columna en concreto debe ser ordenada de manera ascendente o descendente.

Pueden existir tantos order-column como se necesiten dentro del elemento order, siendo su existencia opcional.

4.4.2.8. Entidad del modelo *finder*

El elemento *finder* representa un método de búsqueda autogenerado.

```
<finder name="G_F_N" return-type="DLFileEntry" unique="true">
```

Ejemplo de elemento finder.

Los atributos que tiene este elemento son:

- name: nombre del método finder. En los servicios existirá una método con signatura "*findByNombreDelFinder*".
- return-type: especifica el tipo de retorno del método, pudiendo tener como valores aceptados "Collection" o el nombre de una de las entidades del

modelo. Si el tipo de retorno es “Collection”, se retornará una lista de las entidades. Si es una entidad, se retornará como máximo una entidad.

- unique: implica que la entidad retornada es única.
- where: permite añadir un filtro personalizado al método finder de búsqueda.
- db-index: indica a Service Builder si debe generar un índice de SQL para este método de búsqueda. El valor predeterminado es true.

Pueden existir tantos finder como se necesiten dentro del elemento entity, siendo su existencia opcional.

4.4.2.9. Entidad del modelo *finder-column*

El elemento *finder-column* especifica la columna por la cual realizar las búsquedas.

```
<finder-column name="smallImageId" />
```

Ejemplo de elemento finder-column.

Los atributos que tiene este elemento son:

- name: especifica el nombre de la columna por la cual realizar la consulta.
- case-sensitive: sólo aplica en aquellas columnas de tipo String, para poder hacer búsquedas sensibles a mayúsculas.
- comparator: toma los valores =, !=, <, <=, >, >=, o LIKE para realizar las búsquedas por la columna.
- arrayable-operator: toma valores AND u OR, y genera un método finder adicional, en el que cada valor del array será comparado con el valor de la columna utilizando el criterio definido por el atributo comparator, combinando los resultados con este operador (AND u OR). A modo de ejemplo, un elemento finder-column con el comparator = y el arrayable-operator OR, actuará como una cláusula IN de SQL.

Pueden existir tantos finder-column como se necesiten dentro del elemento finder, siendo obligatoria la existencia de al menos uno.

4.4.2.10. Entidad del modelo *reference*

El elemento *reference* permite inyectar servicios de otros modelos de aplicaciones basadas en Liferay Portal dentro del mismo cargador de clases.

```
<reference entity="ClassName" package-path="com.liferay.portal" />
```

Ejemplo de elemento reference.

Los atributos que tiene este elemento son:

- entity: especifica el nombre de la entidad que queremos inyectar.
- package-path: especifica el paquete de clases en el cual se encuentra la entidad a inyectar.

Pueden existir tantos reference como se necesiten dentro del elemento entity, siendo su existencia opcional.

4.4.2.11. Entidad del modelo *tx-required*

El elemento *tx-required* indicará que el método especificado requiere una transacción. Por defecto, todos los métodos cuyo nombre comience por add, check, clear, delete, set y update requieren la propagación de transacciones. Todos los otros métodos de la persistencia generados soportan transacciones igualmente, pero se asume que en modo sólo-lectura.

```
<tx-required>remove</tx-required>
```

Ejemplo de elemento tx-required.

Pueden existir tantos tx-required como se necesiten dentro del elemento entity, siendo su existencia opcional.

4.4.2.12. Entidad del modelo *exceptions*

El elemento *exceptions* contiene una lista de clases de excepción generadas.

```
<exceptions>
    <exception>DirectoryName</exception>
</exceptions>
```

Ejemplo de elemento exceptions.

Puede existir un único elemento exceptions dentro del elemento raíz service-builder, siendo su existencia opcional.

4.4.2.13. Entidad del modelo *exception*

El elemento *exception* contiene el nombre de clase de una excepción a generar.

```
<exception>DirectoryName</exception>
```

Ejemplo de elemento exception.

Pueden existir tantos elementos exception como se necesiten dentro del elemento exceptions, siendo su existencia opcional.

4.4.2.14. Entidad del modelo *service-builder-import*

El elemento *service-builder-import* permite separar un fichero extenso en varios más pequeños, mediante la agregación de éstos en uno solo. La única condición es que todos los ficheros de servicio deben tener mismo autor y mismo espacio de nombres.

```
<service-builder-import>../players/service.xml</service-builder-import>
```

Ejemplo de elemento service-builder-import.

El único atributo que tiene este elemento es:

- file: ruta relativa al fichero que se quiera importar.

Pueden existir tantos elementos service-builder-import como se necesiten dentro del elemento raíz service-builder, siendo su existencia opcional.

4.5. Vocabulario del dominio

En el siguiente apartado se enumeran las operaciones determinadas por el descriptor del modelo en formato XML, conformando el vocabulario del dominio.

Básicamente, estas operaciones consistirán en la configuración de las relaciones entre las entidades del dominio del problema, mediante la adición de una o más de ellas en otra. Por ejemplo, la adición de una columna a una entidad.

4.5.1. Operaciones sobre el modelo Service-Builder

A continuación se describen las operaciones que es posible realizar para definir el modelo, entidad del dominio definida por la etiqueta XML <service-builder>, que determinarán la lógica de negocio que cualquier implementador debería conocer para proponer una alternativa o solución.

4.5.1.1. Añadir el Autor

Para añadir un autor al modelo es necesario añadir un único elemento XML <author> bajo el elemento raíz <service-builder>.

4.5.1.2. Añadir una Entidad

Para añadir una entidad al modelo es necesario añadir un elemento XML <entity> bajo el elemento raíz <service-builder>. Podrán añadirse tantas entidades como sean necesarias.

4.5.1.3. Añadir Excepciones

Para añadir un conjunto de excepciones al modelo es necesario añadir un único elemento XML <exceptions> bajo el elemento raíz <service-builder>.

4.5.1.4. Importar una definición de Service Builder

Para añadir una entidad al modelo es necesario añadir un elemento XML <service-builder-import> bajo el elemento raíz <service-builder>. Podrán añadirse tantas importaciones como sean necesarias.

4.5.2. Operaciones sobre las entidades

A continuación se describen las operaciones que es posible realizar para definir una entidad, entidad del dominio definida por la etiqueta XML <entity>.

4.5.2.1. Añadir una Columna

Para añadir una columna a la entidad es necesario añadir un elemento XML <column> bajo el elemento <entity> apropiado. Podrán añadirse tantas columnas como sean necesarias.

4.5.2.2. Añadir una Método de búsqueda

Para añadir un método de búsqueda a la entidad es necesario añadir un elemento XML <finder> bajo el elemento <entity> apropiado. Podrán añadirse tantos métodos de búsqueda como sean necesarios.

4.5.2.3. Añadir una Criterio de Ordenación

Para añadir un criterio de ordenación a la entidad es necesario añadir un elemento XML <order> bajo el elemento <entity> apropiado. Podrá añadirse un único criterio de ordenación, siendo ésto opcional.

4.5.2.4. Añadir una Referencia a otra entidad de Service Builder

Para añadir una referencia a la entidad es necesario añadir un elemento XML <reference> bajo el elemento <entity> apropiado. Podrán añadirse tantas referencias como sean necesarias.

4.5.2.5. Definir métodos que soportan transacciones

Para añadir un conjunto de métodos que requieran transacciones a la entidad es necesario añadir un elemento XML <tx-required> bajo el elemento <entity> apropiado. Podrán añadirse tantos conjuntos de métodos como sean necesarios.

4.5.3. Operaciones sobre las Excepciones

A continuación se describen las operaciones que es posible realizar para definir las excepciones, entidad del dominio definida por la etiqueta XML <exceptions>.

4.5.3.1. Añadir una Excepción

Para añadir una excepción a las excepciones es necesario añadir un elemento XML <exception> bajo el elemento <exceptions>, si existe. Podrán añadirse tantas excepciones como sean necesarias.

4.5.4. Operaciones sobre los métodos de búsqueda

A continuación se describen las operaciones que es posible realizar para definir los métodos de búsqueda, entidad del modelo definida por la etiqueta XML <finder>.

4.5.4.1. Añadir una Columna de búsqueda

Para añadir una columna de búsqueda al método de búsqueda es necesario añadir un elemento XML <finder-column> bajo el elemento <finder> adecuado. Podrán añadirse tantas columnas de búsqueda como sean necesarias.

4.5.5. Operaciones sobre los criterios de ordenación

A continuación se describen las operaciones que es posible realizar para definir los métodos de ordenación, entidad del modelo definida por la etiqueta XML <order>.

4.5.5.1. Añadir una Columna de ordenación

Para añadir una columna de ordenación al método de búsqueda es necesario añadir un elemento XML <order-column> bajo el elemento <order> adecuado. Podrán añadirse tantas columnas de ordenación como sean necesarias.

5. Desarrollo y resultados obtenidos

Tras examinar la manera de definir las entidades del modelo, realizado a través de un fichero XML, se observa que *es necesario replantear dicha manera* y buscar un enfoque en el que el desarrollador tenga un apoyo basado en algo más robusto que la sencilla validación realizada por DTD, puesto que a pesar de que se validan los elementos XML del documento, no se valida la semántica de los mismos. A modo de ejemplo, podríamos añadir dos veces una misma columna, o dos columnas con el atributo *filter-primary* a *true* (caso no permitido), y DTD no lo detectaría en ningún caso. De esta manera, el error lo encontraríamos al compilar el código una vez generado.

El enfoque en el que basar la manera de definir el modelo debería apoyarse en el uso de un DSL, puesto que ofrece las ventajas anteriormente mencionadas sobre productividad y conocimiento del dominio del problema. Este DSL a generar, será de tipo interno y homogéneo. Interno debido a que la idea es aprovechar las construcciones de un lenguaje de programación de propósito general para apoyar en ellas las operaciones definidas por el dominio del problema. Y homogéneo por integrar el DSL en las herramientas de construcción ya existentes en Liferay Portal para generar código.

Por tanto, el DSL a desarrollar se escribirá sobre un lenguaje compilado, en lugar de un simple descriptor XML como hasta la actualidad, de modo que pudiera aprovecharse toda la potencia del compilador, y fuera éste el que determinara las operaciones válidas a realizar, o lo que es lo mismo, impidiera las operaciones no permitidas.

Para ello, al ser la definición del modelo de entidades de Liferay Portal un caso de uso muy específico, la creación de un lenguaje especializado para tal fin, en forma de DSL, ayudaría en gran medida a no cometer errores.

Durante el desarrollo de este trabajo, se ha tenido en cuenta el enfoque de generación de un lenguaje específico, o DSL, para construir el modelo del dominio necesario para construir aplicaciones basadas en Liferay Portal.

Para ello se ha desarrollado un lenguaje específico de dominio interno (*fluent*) que mapea las operaciones a realizar durante la definición del modelo de una aplicación basada en Liferay Portal a operaciones embebidas en el propio lenguaje. Estas operaciones de definición del modelo, que pasaremos a detallar más adelante, se traducen a métodos de un API fluent [47] escrito en Java.

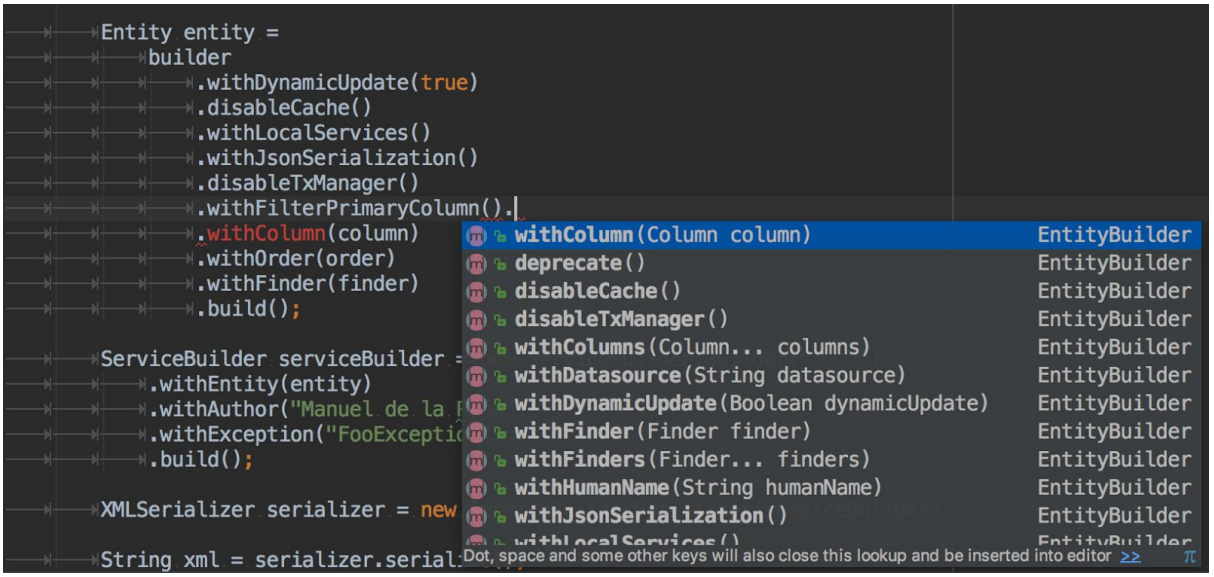
En una primera fase, el DSL sería capaz de generar el mismo XML que un desarrollador escribiría a mano. Esta opción ofrece las ventajas proporcionadas por un compilador, como podrían ser la adición de restricciones que un fichero XML no sería capaz de detectar, aún teniendo un DTD o incluso un XMLSchema [48]. A modo de ejemplo, con XML podríamos añadir dos columnas con el mismo nombre a una entidad, y no tendríamos noticias de esta duplicidad hasta que no ejecutáramos el generador de código y compiláramos el código resultante. Sin embargo, con el enfoque del DSL específico, podemos crear restricciones en cuanto a la duplicidad de columnas, o cualquier otro elemento del modelo, de modo que no se añadan duplicados.

Otra posible restricción sería en cuanto a la adición de ciertos tipos de datos. En el caso de las entidades de Liferay Portal, únicamente es posible añadir una columna primaria por la que realizar filtros, y con XML, esta restricción no sería posible realizarla, puesto que nada nos impide de definir un elemento XML con un atributo con un valor específico:

```
<column
    accessor="false" container-model="false" convert-null="false"
    filter-primary="true"
    json-enabled="false" lazy="true" localized="false"
    name="companyId"
    parent-container-model="false" primary="true" type="long"
/>

<column
    accessor="false" container-model="false" convert-null="false"
    filter-primary="true"
    json-enabled="false" lazy="true" localized="false"
    name="groupId"
    parent-container-model="false" primary="true" type="long"
/>
```

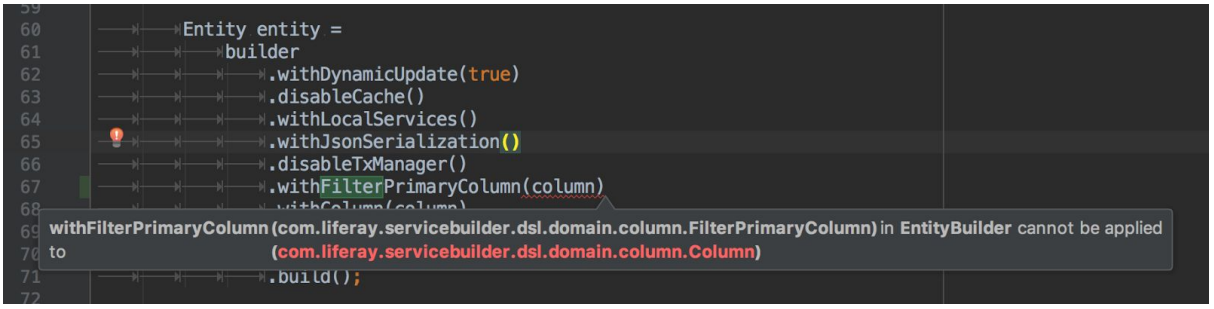
Ejemplo de columnas con valores incompatibles, al no poder existir dos columnas filter-primary al mismo tiempo.

Con un DSL interno, o *fluent*, podríamos crear una restricción de modo que el propio lenguaje de dominio no permitiera esa operación. A modo de ejemplo se muestran a continuación dos capturas del entorno de desarrollo integrado (IDE) IntelliJ [2], en el que el compilador notifica de las operaciones aceptadas, así como de los tipos válidos en cada operación.



El compilador no permite la adición de otra columna del tipo FilterPrimary, puesto que el lenguaje ha sido diseñado con estas restricciones del dominio.

Por último, basándonos en un sistema de tipos fuerte, podríamos conseguir que el DSL incluso nos impidiese añadir elementos del tipo incorrecto:



El sistema de tipos obliga a que las columnas que vayan a ser filtradas sean del tipo FilterPrimaryColumn, y el desarrollador no pueda equivocarse.

De esta manera, las reglas de negocio son satisfechas e implementadas de manera automática mediante el lenguaje de dominio, haciéndolas disponibles al equipo de desarrollo de una manera directa desde el negocio. Puede observarse que la comunicación entre los analistas del negocio y los desarrolladores se ve incrementada por el hecho de utilizar un DSL.

5.1. Implementación del lenguaje interno de dominio

En el siguiente apartado se enumeran las operaciones que se desean implementar en el nuevo DSL interno, que en una primera fase supondrán la generación del descriptor del modelo en formato XML. En futuras revisiones, el propio DSL realizará

la propia generación de código, omitiendo el paso de generación intermedia del descriptor XML, quedando esta parte fuera del alcance de este trabajo.

Al ser un DSL interno y homogéneo, el lenguaje de implementación será Java, que es el lenguaje de ejecución actual de Service Builder.

En cuanto al convenio de nombres de clases y métodos Java, se ha buscado utilizar unos nombres completamente descriptivos, que determinen de manera unívoca y sin conflictos la operación a realizar, de manera que un desarrollador pueda simplemente leer el API de métodos disponibles y sepa qué hacer con cada uno de ellos, de una manera natural y transparente.

En general, se ha preferido el uso del prefijo "*with*" para identificar aquellas operaciones que implican la composición de elementos sobre el objeto a construir, como añadir una columna o un criterio de ordenación, a un objeto de tipo Entity o a un objeto de tipo Finder, respectivamente.

Estos dos ejemplos propuestos quedarían así:

- para añadir una columna a una entidad, se utilizará el método "*withColumn(column)*".

- para definir el tipo de comparación en los métodos de búsqueda, se utilizará "*withComparator(comparatorType)*".

En los apartados siguientes se describen las operaciones que se han implementado para definir el modelo en el nuevo DSL. Estas operaciones sobre las diferentes entidades (ServiceBuilder, Entity, Finder, FinderColumn, Order, OrderColumn, Reference y TxRequiredMethod) determinan la lógica de negocio que cualquier implementador debe conocer para proponer una nueva aplicación basada en Liferay Portal, suponiendo por tanto el conocimiento funcional o especificación del DSL.

A modo de resumen, estas operaciones se basarán en la configuración de los metadatos de cada entidad del dominio, y en la adición a una entidad de una o más de una de las entidades restantes, con las consiguientes restricciones que pudieran darse. A modo de ejemplo, una operación a implementar sería la adición de una columna de búsqueda a un método de búsqueda.

5.1.1. Patrones

Tal y como hemos visto con anterioridad, se ha considerado el patrón Builder como el más recomendable para realizar la implementación de un DSL. Por ello, se han definido clases de tipo Builder concreto sobre las entidades del dominio, de modo que para construir dichas entidades sea necesario hacerlo a través de estos Builder concretos.

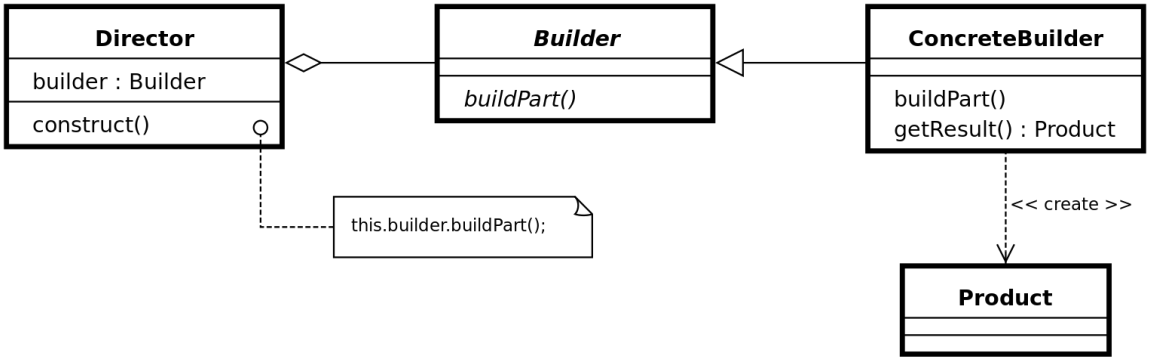


Diagrama de clases del patrón Builder.

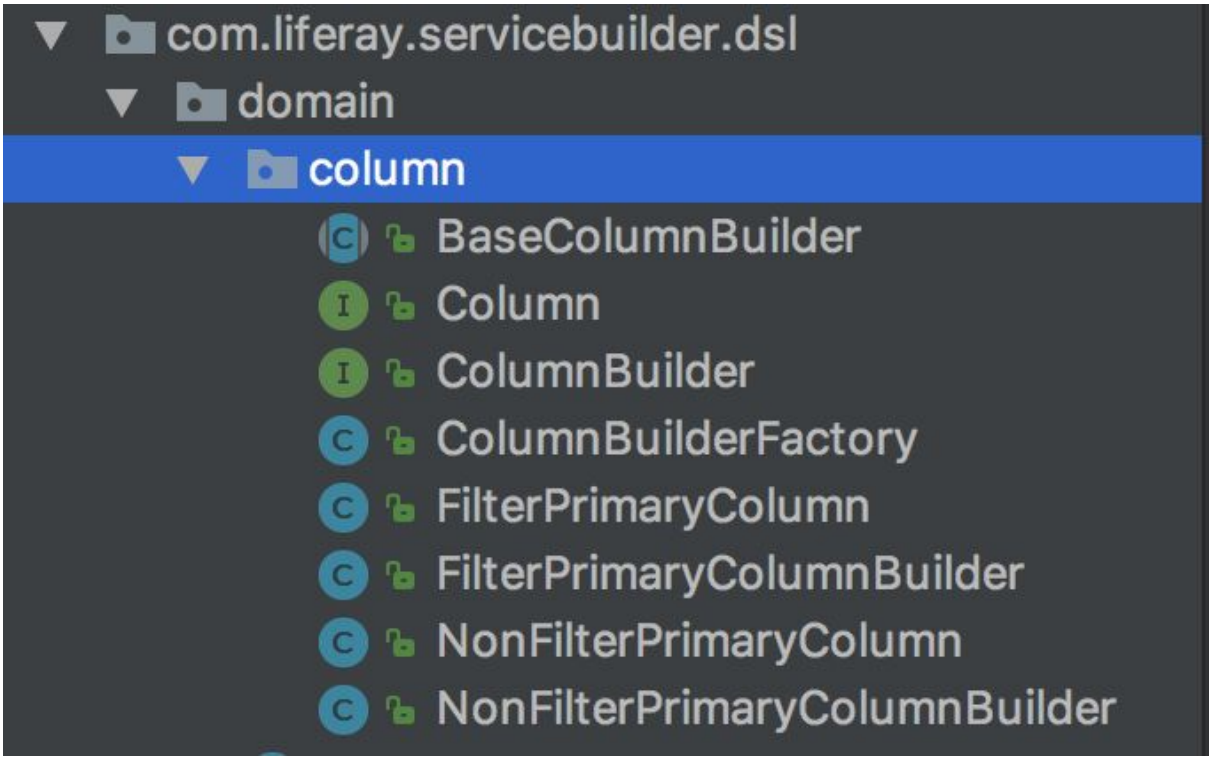
Siguiendo el diagrama de clases propuesto, las entidades del dominio serán los Directores, y las clases Builder concretos del patrón serán las propias Builder de la implementación.

Cada clase del tipo Builder se ha implementado como una clase interna a la clase del modelo, de manera que por ejemplo la clase Reference contiene a su Builder asociado como clase interna.

```
public class Reference implements ServiceBuilderElement {
  ...
   public static class Builder {
     ...
     public Builder(String entity, String packagePath) {
       ...
     }

     ...
  }
  ...
}
```

Detalle de implementación de un Builder concreto en un Director. En este caso el Director está representado por la entidad Reference.

Esta manera de implementación de Builder concretos se repite en todas las clases del dominio, excepto para el modelo de la entidad Column, en la que debido a su mayor complejidad, con la existencia de dos tipos diferentes de columnas, se ha optado por extraer las clases Builder concretos a clases separadas.



Detalle del paquete com.liferay.servicebuilder.dsl.domain.column, donde se muestran los Builder concretos como clases separadas.

5.1.2. Operaciones sobre el modelo ServiceBuilder

A continuación se describen las operaciones que se desean para definir el modelo de una aplicación basada en Liferay Portal.

5.1.2.1. Construcción del Builder

Para inicializar el objeto Builder el DSL pasará en su constructor aquellos valores que sean requeridos por el XML anterior:

```
public Builder(String packagePath, String namespace);
```

De esta manera, todo objeto del dominio con tipo ServiceBuilder deberá ser creado con un packagePath y un namespace, puesto que el patrón Builder determinará este constructor como único punto de acceso a la construcción de elementos de tipo ServiceBuilder.

5.1.2.2. Añadir el Autor

Para añadir un autor al modelo es conveniente añadir un método al objeto Builder de la entidad ServiceBuilder que así lo hiciera:

```
.withAuthor("Manuel de la Peña").build();
```

En el ejemplo anterior, "Manuel de la Peña" representa una cadena de caracteres que quedará fijada como valor del autor de la entidad del dominio ServiceBuilder.

Este método podrá ser invocado tantas veces como se quiera, siendo aplicado el valor de la última invocación.

5.1.2.3. Añadir una Entidad

Para añadir una entidad al modelo es conveniente añadir un método al objeto Builder de la entidad ServiceBuilder que así lo hiciera:

```
.withEntity(entity).build();
```

En el ejemplo anterior, "entity" representa una instancia de la clase Entity, que habrá sido creada y configurada mediante el mismo patrón Builder, como se mostrará más adelante.

Este método *withEntity* podrá ser invocado tantas veces como se quiera, añadiendo la entidad a la lista de entidades participantes en el descriptor del modelo, con la salvedad que si la entidad a utilizar a existe en la lista de entidades, el DSL no la añadirá, impidiendo duplicados. Esta validación la realizará a nivel del valor del atributo "*name*" del objeto *entity*, utilizando las capacidades de la interfaz *Comparable* de Java con su método *compareTo*.

5.1.2.4. Añadir Excepciones

Para añadir una excepción al modelo es conveniente añadir un método al objeto Builder de la entidad ServiceBuilder que así lo hiciera:

```
.withException("MyCustomException").build();
```

En el ejemplo anterior, "MyCustomException" representa el nombre en formato cadena de la excepción a añadir, y será añadida a la lista de excepciones del modelo.

Este método *withException* podrá ser invocado tantas veces como se quiera, añadiendo la excepción a la lista de excepciones participantes en el descriptor del modelo, con la salvedad que si la excepción a utilizar ya existe en la lista de excepciones, el DSL no la añadirá, impidiendo duplicados. Esta validación la realizará a nivel del valor de la cadena.

Podríamos también añadir varias excepciones a la vez:

```
// añade varias excepciones a la vez
.withExceptions(exception1, exception2).build();

//añade un array de excepciones
.withReferences(exceptionsArray).build();
```

5.1.2.5. Importar una definición de Service Builder

Para añadir una excepción al modelo es conveniente añadir un método al objeto Builder de la entidad ServiceBuilder que así lo hiciera:

```
.importServiceBuilderFile("../foo/service.xml").build();
```

En el ejemplo anterior, "../foo/service.xml" representa el path en formato cadena del fichero a importar, y será añadida a la lista de importaciones del modelo.

Este método *importServiceBuilderFile* podrá ser invocado tantas veces como se quiera, añadiendo el path a la lista de importaciones participantes en el descriptor

del modelo, con la salvedad que si el path a utilizar ya existe en la lista de importaciones, el DSL no la añadirá, impidiendo duplicados. Esta validación la realizará a nivel del valor de la cadena.

5.1.2.6. Definir si las entidades predefinen las referencias importadas

Para predeterminar el comportamiento de las referencias importadas, es conveniente añadir un método al objeto Builder de la entidad ServiceBuilder que así lo hiciera:

```
.autoImportDefaultReferences().build();
```

Al estar desactivado por defecto la importación automática de las referencias, el método del API en el DSL indicará el caso contrario, esto es, activar dicha importación.

Este método podrán ser invocado tantas veces como se quiera, siendo el valor de la última invocación el utilizado por la generación de código.

5.1.2.7. Definir si las entidades prefijan las tablas con el espacio de nombres

Para definir si las tablas de base de datos prefijan su nombre con el espacio de nombres, es conveniente añadir un método al objeto Builder de la entidad ServiceBuilder que así lo hiciera:

```
.autoNamespaceTables().build();
```

Al estar desactivado por defecto el prefijado de las tablas de base de datos con el espacio de nombres, el método del API en el DSL indicará el caso contrario, esto es, activar dicho prefijado.

Este método podrán ser invocado tantas veces como se quiera, siendo el valor de la última invocación el utilizado por la generación de código.

5.1.2.8. Definir si las entidades utilizan el control de concurrencia multiversión

Para predeterminar el comportamiento de las entidades en cuanto al control de concurrencia multiversión, es conveniente añadir un método al objeto Builder de la entidad ServiceBuilder que así lo hiciera:

```
.enableMvcc().build();
```

Al estar desactivado por defecto el control de concurrencia multiversión, el método del API en el DSL indicará el caso contrario, esto es, activar dicho control.

Este método podrán ser invocado tantas veces como se quiera, siendo el valor de la última invocación el utilizado por la generación de código.

5.1.3. Operaciones sobre Entity

A continuación se describen las operaciones que se han implementado para definir una entidad.

5.1.3.1. Construcción del Builder

Para inicializar el objeto Builder el DSL pasará en su constructor aquellos valores que sean requeridos por el XML anterior:

```
public BuilderImpl(String name);
```

De esta manera, todo objeto del dominio con tipo Entity deberá ser creado con un name para la entidad, puesto que el patrón Builder determinará este constructor como único punto de acceso a la construcción de elementos de tipo Entity.

En el caso de Entity, podremos construir dos tipo de columnas: FilterPrimaryColumn y NonFilterPrimaryColumn , como se mostrará más adelante. Por ello, se ha tenido que refinar el patrón Builder de manera que se puedan contribuir esos dos tipos y únicamente se pueda añadir una columna del tipo FilterPrimary. De este modo, una vez invocado este método, no se podrá volver a invocar, y será el compilador el que impida un uso indebido del API.

5.1.3.2. Deprecar

Para deprecar una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.deprecate().build();
```

Al estar desactivada por defecto la deprecación de entidades, el método del API en el DSL indicará el caso contrario, esto es, activar dicha deprecación.

5.1.3.3. Deshabilitar la caché

Para deshabilitar el uso de caché en las consultas de bases de datos de una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.disableCache().build();
```

Al estar activado por defecto el uso de cachés, el método del API en el DSL indicará el caso contrario, esto es, desactivar dicho uso.

5.1.3.4. Deshabilitar el mánager de transacciones

Para deshabilitar el mánager de transacciones de una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.disableTxManager().build();
```

Al estar activado por defecto el manager de transacciones, el método del API en el DSL indicará el caso contrario, esto es, desactivar dicho manager.

5.1.3.5. Añadir una Columna

Para añadir una columna a una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withColumn(column).build();
```

En el ejemplo anterior, “column” representa una instancia de la clase Column, que habrá sido creada y configurada mediante el mismo patrón Builder, como se mostrará más adelante.

Este método *withColumn* podrá ser invocado tantas veces como se quiera, añadiendo la columna a la lista de entidades participantes en el descriptor del modelo, con la salvedad que si la columna a utilizar a existe en la lista de columnas de la entidad, el DSL no la añadirá, impidiendo duplicados. Esta validación la realizará a nivel del valor del atributo “*name*” del objeto *column*, utilizando las capacidades de la interfaz *Comparable* de Java con su método *compareTo*.

Podríamos también añadir varias columnas a la vez:

```
// añade varias columnas a la vez
.withColumns(column1, column2).build();

//añade un array de columnas
.withColumns(columnsArray).build();
```

Además, tal y como se ha mencionado en el punto anterior, sólo podrá añadirse una única columna con el tipo FilterPrimary, de modo que existirá un método para tal fin.

```
.withFilterPrimaryColumn(filterPrimaryColumn).build();
```

Gracias al diseño realizar en el patrón Builder de esta clase, no es posible invocar de nuevo al método *withFilterPrimaryColumn* si éste ha sido invocado con anterioridad.

5.1.3.6. Especificar una fuente de datos (datasource)

Para especificar la fuente de datos de una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withDatasource(“externalDatasource”).build();
```

En el ejemplo anterior, “externalDatasource” representa el nombre del datasource al que se conectará la entidad.

5.1.3.7. Actualizar de manera dinámica las consultas de base de datos

Para especificar la fuente de datos de una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withDynamicUpdate(false).build();
```

El valor dynamicUpdate de la entidad vendrá definido por el valor del atributo MVCC, de modo que si no se ha aplicado ningún valor, se tomará éste último en su lugar.

Esta característica la implementa el DSL mediante siguiente el método getter:

```
public Boolean hasDynamicUpdate() {
	if (_dynamicUpdateEnabled == null) {
		return hasMvccEnabled();
	}

	return _dynamicUpdateEnabled;
}
```

5.1.3.8. Añadir un Método de búsqueda

Para añadir un método de búsqueda a una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withFinder(finder).build();
```

En el ejemplo anterior, “finder” representa una instancia de la clase Finder, que habrá sido creada y configurada mediante el mismo patrón Builder, como se mostrará más adelante.

Este método *withFinder* podrá ser invocado tantas veces como se quiera, añadiendo un método de búsqueda a la lista de métodos de búsqueda de la entidad, con la salvedad que si el método de búsqueda a utilizar a existe en la lista de métodos de búsqueda de la entidad, el DSL no la añadirá, impidiendo duplicados. Esta

validación la realizará a nivel del valor del atributo "*name*" del objeto *finder*, utilizando las capacidades de la interfaz *Comparable* de Java con su método *compareTo*.

Podríamos también añadir varios métodos de búsqueda a la vez:

```
// añade varios métodos a la vez
.withFinders(finder1, finder2).build();

//añade un array de métodos
.withFinders(findersArray).build();
```

5.1.3.9. Definir el nombre entendible por un humano

Para especificar el nombre entendible por un humano de una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withHumanName("humanName").build();
```

En el ejemplo anterior, "humanName" representa el nombre humano que se utilizará para describir a la entidad.

5.1.3.10. Utilizar la serialización en formato JSON

Para especificar si una entidad será serializada en formato JSON es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withJSONSerialization().build();
```

El valor jsonEnabled de la entidad vendrá definido por el valor del atributo remoteService, de modo que si no se ha aplicado ningún valor, se tomará éste último en su lugar.

Esta característica la implementa el DSL mediante siguiente el método getter:

```
public boolean hasJsonSerialization() {
```

```
        if (_jsonEnabled == null) {
            return _remoteService;
        }

        return _jsonEnabled;
    }
```

5.1.3.11. Habilitar los servicios locales

Para especificar si una entidad tiene servicios locales es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withLocalservices().build();
```

Al estar desactivada por defecto la creación de servicios locales, el método del API en el DSL indicará el caso contrario, esto es, activar dicha creación.

5.1.3.12. Habilitar el control de concurrencia multivalor

Para especificar si una entidad tiene habilitado el control de concurrencia multivalor (MVCC) es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withMVCC(true).build();
```

En el ejemplo anterior, es posible pasar un valor boolean para habilitar o deshabilitar el MVCC en la entidad.

5.1.3.13. Añadir un Criterio de Ordenación

Para añadir un método de ordenación a una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withOrder(order).build();
```

En el ejemplo anterior, “order” representa una instancia de la clase Order, que habrá sido creada y configurada mediante el mismo patrón Builder, como se mostrará más adelante.

Este método *withOrder* podrá ser invocado tantas veces como se quiera, aplicando el valor de la última invocación.

5.1.3.14. Definir la clase de persistencia

Para especificar la clase de persistencia de una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withPersistenceClass(“persistenceClass”).build();
```

En el ejemplo anterior “persistenceClass” representa el valor de la clase de persistencia a utilizar para la entidad.

5.1.3.15. Añadir una Referencia a otra entidad de Service Builder

Para añadir una referencia a otra entidad de ServiceBuilder en una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withReference(reference).build();
```

En el ejemplo anterior, “reference” representa una instancia de la clase Reference, que habrá sido creada y configurada mediante el mismo patrón Builder, como se mostrará más adelante.

Este método *withReference* podrá ser invocado tantas veces como se quiera, añadiendo una referencia a la lista de referencias de la entidad, con la salvedad que si la referencia a utilizar a existe en la lista de referencias de la entidad, el DSL no la añadirá, impidiendo duplicados. Esta validación la realizará a nivel del valor del atributo “*entity*” del objeto *reference*, utilizando las capacidades de la interfaz *Comparable* de Java con su método *compareTo*.

Podríamos también añadir varias referencias a la vez:

```
// añade varias referencias a la vez
.withReferences(reference1, reference2).build();

//añade un array de referencias
.withReferences(referencesArray).build();
```

5.1.3.16. Habilitar los servicios remotos

Para especificar si una entidad tiene servicios remotos es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withRemoteservices().build();
```

Al estar desactivada por defecto la creación de servicios remotos, el método del API en el DSL indicará el caso contrario, esto es, activar dicha creación.

5.1.3.17. Definir factoría de sesiones

Para especificar la factoría de sesiones de una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withSessionFactory("sessionFactory").build();
```

En el ejemplo anterior, "sessionfactory" representa el nombre de la factoría de sesiones que se utilizará para las clases de persistencia de la entidad.

5.1.3.18. Definir la tabla de base de datos

Para especificar el nombre de tabla de base de datos de una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withTable("table").build();
```

En el ejemplo anterior, "table" representa el nombre de la tabla de base de datos que se utilizará para la entidad.

5.1.3.19. Habilitar la papelera de reciclaje

Para especificar si una entidad habilita o no la papelera de reciclaje es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withTrashEnabled().build();
```

Al estar desactivada por defecto la papelera de reciclaje para la entidad, el método del API en el DSL indicará el caso contrario, esto es, activar la papelera de reciclaje.

5.1.3.20. Definir el manager de transacciones

Para especificar el nombre del manager de transacciones de una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withTxManager("txManager").build();
```

En el ejemplo anterior, "txManager" representa el nombre del manager de transacciones que se utilizará para la entidad.

5.1.3.21. Añadir una Método que requiera transacciones

Para añadir un método que requiera transacciones en una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withTxRequiredMethod(txRequiredMethod).build();
```

En el ejemplo anterior, "txRequired" representa una instancia de la clase TxRequiredMethod, que habrá sido creada y configurada mediante el mismo patrón Builder, como se mostrará más adelante.

Este método *withTxRequiredMethod* podrá ser invocado tantas veces como se quiera, añadiendo un método que requiera transacciones a la lista de métodos que requieran transacciones de la entidad, con la salvedad que si el método que requiera transacciones a utilizar a existe en la lista de métodos que requieran transacciones de la entidad, el DSL no la añadirá, impidiendo duplicados. Esta

validación la realizará a nivel del valor del atributo "*methodName*" del objeto *txRequiredMethod*, utilizando las capacidades de la interfaz *Comparable* de Java con su método *compareTo*.

Podríamos también añadir varios métodos que requieran transacciones a la vez:

```
// añade varias métodos a la vez
.withTxRequiredMethods(txRequiredMethod1, txRequiredMethod2).build();

//añade un array de métodos
.withTxRequiredMethods(txRequiredMethodsArray).build();
```

5.1.3.22. Definir el UUID

Para especificar si es necesario generar un campo UUID en una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withUuid().build();
```

Al estar desactivada por defecto la creación de un campo UUID, el método del API en el DSL indicará el caso contrario, esto es, activar dicha creación.

5.1.3.23. Definir el UUID accessor

Para especificar si es necesario generar un accesor para el campo UUID de una entidad es conveniente añadir un método al objeto Builder de la entidad Entity que así lo hiciera:

```
.withUuidAccessor().build();
```

Al estar desactivada por defecto la creación de un accesor para el campo UUID, el método del API en el DSL indicará el caso contrario, esto es, activar dicha creación.

5.1.4. Operaciones sobre Finder

A continuación se describen las operaciones que se desean implementar para definir los métodos de búsqueda.

5.1.4.1. Construcción del Builder

Para inicializar el objeto Builder el DSL pasará en su constructor aquellos valores que sean requeridos por el XML anterior:

```
public Builder(String name, String returnType,
               FinderColumn finderColumn);
```

De esta manera, todo objeto del dominio con tipo Finder deberá ser creado con un name para el método de búsqueda, un tipo de retorno, y una columna por la que realizar la búsqueda, puesto que el patrón Builder determinará este constructor como único punto de acceso a la construcción de elementos de tipo Finder.

5.1.4.2. Definir si el método retorna una única entidad

Para especificar si el método devuelve una única entidad es conveniente añadir un método al objeto Builder de la entidad Finder que así lo hiciera:

```
.unique().build();
```

Al estar desactivado por defecto el retorno de entidades de manera única el método del API en el DSL indicará el caso contrario, esto es, activar dicho retorno único.

5.1.4.3. Definir si es necesario generar un índice SQL

Para especificar si la generación del método generará la creación de un índice de SQL es conveniente añadir un método al objeto Builder de la entidad Finder que así lo hiciera:

```
.withoutSQLIndex().build();
```

Si no se invoca este método, se creará un índice que incluya la columnas que componen la búsqueda. De lo contrario, no se creará dicho índice.

Al estar activada por defecto la creación del índice, el método del API en el DSL indicará el caso contrario, esto es, desactivar dicha creación.

5.1.4.4. Definir un criterio de búsqueda predeterminado

Para especificar un criterio de búsqueda adicional que será siempre aplicado en el método de búsqueda es conveniente añadir un método al objeto Builder de la entidad Finder que así lo hiciera:

```
.where("ID <> 23").build();
```

En el ejemplo anterior, el criterio "ID <> 23" será añadido siempre a los criterios de búsqueda determinados por el método de búsqueda.

5.1.4.5. Añadir una Columna de búsqueda

Para añadir una columna por la que buscar en el método de búsqueda es conveniente añadir un método al objeto Builder de la entidad Finder que así lo hiciera:

```
.withFinderColumn(finderColumn).build();
```

En el ejemplo anterior, "finderColumn" representa una instancia de la clase FinderColumn, que habrá sido creada y configurada mediante el mismo patrón Builder, como se mostrará más adelante.

Este método *withFinderColumn* podrá ser invocado tantas veces como se quiera, añadiendo una columna por la que buscar a la lista de columnas del método de búsqueda, con la salvedad que si la columna a añadir ya existe en la lista de columnas del método de búsqueda, el DSL no la añadirá, impidiendo duplicados. Esta validación la realizará a nivel del valor del atributo "*name*" del objeto *finderColumn*, utilizando las capacidades de la interfaz *Comparable* de Java con su método *compareTo*.

Podríamos también añadir varias columnas a la vez:

```
// añade varias columnas a la vez
.withFinderColumns(finderColumn1, finderColumn2).build();
```

```
//añade un array de columnas
.withFinderColumns(finderColumnsArray).build();
```

5.1.5. Operaciones sobre FinderColumn

A continuación se describen las operaciones que se desean para definir las columnas a añadir a los métodos de búsqueda.

5.1.5.1. Construcción del Builder

Para inicializar el objeto Builder el DSL pasará en su constructor aquellos valores que sean requeridos por el XML anterior:

```
public Builder(String name);
```

De esta manera, todo objeto del dominio con tipo FinderColumn deberá ser creado con un name para la columna, puesto que el patrón Builder determinará este constructor como único punto de acceso a la construcción de elementos de tipo FinderColumn.

5.1.5.2. Definir si la búsqueda es sensible a mayúsculas

Para especificar si la búsqueda se realizará de manera sensible a mayúsculas es conveniente añadir un método al objeto Builder de la entidad FinderColumn que así lo hiciera:

```
.caseInsensitive().build();
```

Al utilizar búsquedas sensibles a mayúsculas por defecto, el método del API en el DSL indicará el caso contrario, esto es, especificar búsquedas no sensibles a mayúsculas.

5.1.5.3. Definir el operador de búsqueda para consultas con valores múltiples

Para especificar el tipo de operador a utilizar en búsquedas con valores múltiples es conveniente añadir un método al objeto Builder de la entidad FinderColumn que así lo hiciera:

```
.withArrayableOperator(arrayableOperator).build();
```

En el ejemplo anterior, “arrayableOperator” representa un valor de la enumeración ArrayableOperator, que tomará como posibles valores AND y OR.

- AND: se utilizará para consultar sobre varios valores utilizando un criterio aditivo: se añadirán al resultado si se cumple la búsqueda para todos los valores.

- OR: se utilizará para consultar sobre varios valores utilizando un criterio selectivo: se añadirán al resultado si se cumple la búsqueda para alguno de los valores.

5.1.5.4. Definir el tipo de comparación a realizar en la búsqueda

Para especificar el tipo de comparación a utilizar en las búsquedas es conveniente añadir un método al objeto Builder de la entidad FinderColumn que así lo hiciera:

```
.withComparator(comparator).build();
```

En el ejemplo anterior, “comparator” representa un valor de la enumeración FinderComparator, que tomará como posibles valores DISTINCT, EQUALS, GREATER, GREATER_EQUALS, LESS, LESS_EQUALS y LIKE.

- DISTINCT: se utilizará para definir que la consulta debe ser distinta al valor seleccionado.

- EQUALS: se utilizará para definir que la consulta debe ser igual al valor seleccionado.

- GREATER: se utilizará para definir que la consulta debe ser mayor al valor seleccionado.

- GREATER_EQUALS: se utilizará para definir que la consulta debe ser mayor o igual al valor seleccionado.
- LESS: se utilizará para definir que la consulta debe ser menor al valor seleccionado.
- LESS_EQUALS: se utilizará para definir que la consulta debe ser menor o igual al valor seleccionado.
- LIKE: se utilizará para definir que la consulta debe utilizar un criterio LIKE de SQL para realizar búsquedas parciales sobre un campo de texto.

5.1.6. Operaciones sobre Order

A continuación se describen las operaciones que se desean implementar para definir los métodos de ordenación.

5.1.6.1. Construcción del Builder

Para inicializar el objeto Builder el DSL no necesitará pasar en su constructor ningún valor, puesto que sean en el XML anterior no se definía ninguno como requerido:

```
public Builder();
```

Del mismo modo que con anterioridad, el patrón Builder determinará este constructor como único punto de acceso a la construcción de elementos de tipo Order.

5.1.6.2. Definir el tipo de ordenación

Para especificar el tipo de ordenación sobre la ordenación es conveniente añadir un método al objeto Builder de la entidad Order que así lo hiciera:

```
.by(orderBy).build();
```

En el ejemplo anterior, “orderBy” representa un valor de la enumeración OrderBy, que únicamente recoge los valores ASC y DESC, representando respectivamente los criterios de ordenación ascendente y descendente.

5.1.6.3. Añadir una Columna de ordenación

Para añadir una columna por la que ordenar en el criterio de ordenación es conveniente añadir un método al objeto Builder de la entidad Order que así lo hiciera:

```
.withOrderColumn(orderColumn).build();
```

En el ejemplo anterior, “orderColumn” representa una instancia de la clase OrderColumn, que habrá sido creada y configurada mediante el mismo patrón Builder, como se mostrará más adelante.

Este método *withOrderColumn* podrá ser invocado tantas veces como se quiera, añadiendo una columna por la que ordenar a la lista de columnas del método de ordenación, con la salvedad que si la columna a añadir ya existe en la lista de columnas del método de ordenación, el DSL no la añadirá, impidiendo duplicados. Esta validación la realizará a nivel del valor del atributo “*name*” del objeto *orderColumn*, utilizando las capacidades de la interfaz *Comparable* de Java con su método *compareTo*.

Podríamos también añadir varias columnas a la vez:

```
// añade varias columnas a la vez
.withOrderColumns(orderColumn1, orderColumn2).build();

//añade un array de columnas
.withOrderColumns(orderColumnsArray).build();
```

5.1.7. Operaciones sobre OrderColumn

A continuación se describen las operaciones que se desean para definir las columnas a añadir a los métodos de ordenación.

5.1.7.1. Construcción del Builder

Para inicializar el objeto Builder el DSL pasará en su constructor aquellos valores que sean requeridos por el XML anterior:

```
public Builder(String name);
```

De esta manera, todo objeto del dominio con tipo OrderColumn deberá ser creado con un name para la columna, puesto que el patrón Builder determinará este constructor como único punto de acceso a la construcción de elementos de tipo OrderColumn.

5.1.7.2. Definir si la ordenación es sensible a mayúsculas

Para especificar si la ordenación se realizará de manera sensible a mayúsculas es conveniente añadir un método al objeto Builder de la entidad OrderColumn que así lo hiciera:

```
.caseInsensitive().build();
```

Al utilizar ordenaciones sensibles a mayúsculas por defecto, el método del API en el DSL indicará el caso contrario, esto es, especificar ordenaciones no sensibles a mayúsculas.

5.1.7.3. Definir el tipo de ordenación

Para especificar el tipo de operador a utilizar en ordenaciones es conveniente añadir un método al objeto Builder de la entidad OrderColumn que así lo hiciera:

```
.descending().build();
```

El valor por defecto del criterio de ordenación es OrderBy.ASC, por tanto el método del API en el DSL indica la operación contraria, que utilizará un valor OrderBy.DESC en las ordenaciones.

5.1.8. Operaciones sobre Reference

A continuación se describen las operaciones que se desean implementar para definir las referencias a otras entidades.

5.1.8.1. Construcción del Builder

Para inicializar el objeto Builder el DSL pasará en su constructor aquellos valores que sean requeridos por el XML anterior:

```
public Builder(String entity, String packagePath);
```

De esta manera, todo objeto del dominio con tipo Reference deberá ser creado con el nombre de la entidad a la que hacer referencia, así como el paquete en el que se encuentra dicha entidad, puesto que el patrón Builder determinará este constructor como único punto de acceso a la construcción de elementos de tipo Reference.

5.1.9. Operaciones sobre TxRequiredMethod

A continuación se describen las operaciones que se desean implementar para definir las referencias a otras entidades.

5.1.9.1. Construcción del Builder

Para inicializar el objeto Builder el DSL pasará en su constructor aquellos valores que sean requeridos por el XML anterior:

```
public Builder(String methodName);
```

De esta manera, todo objeto del dominio con tipo Reference deberá ser creado con el nombre del método que requiere una transacción, puesto que el patrón Builder determinará este constructor como único punto de acceso a la construcción de elementos de tipo TxRequiredMethod.

5.2. Validación de la solución

El utilizar un lenguaje DSL nos ofrece bastante más seguridad y robustez a la hora de componer objetos del dominio en las aplicaciones basadas en Liferay Portal, puesto con el sistema anterior, el DSL externo basado en XML, no teníamos manera alguna ni para validar las operaciones posibles, ni para controlar la duplicidad de elementos, o incluso la existencia de valores dependientes.

Con el trabajo desarrollado en este proyecto, más los trabajos futuros que se pudieran realizar, será posible extender el lenguaje hasta satisfacer todas las reglas del negocio impuestas para la creación del modelo de las aplicaciones. De esta manera, utilizando el DSL, los procesos de creación del modelo serán siempre repetibles y seguros, pues nos estaremos apoyando tanto en el sistema de tipos de Java, como en la máquina virtual java (JVM) para detectar operaciones no válidas en tiempo de compilación.

Por otro lado, gracias a la expresividad del DSL, el programa resultante será mucho más conciso, puesto que describir todo el modelo en un descriptor XML resulta en ficheros XML demasiado grandes, propensos a errores, y por tanto poco mantenibles. Por contra, el resultado de utilizar el DSL será un pequeño programa Java mucho más legible que el XML.

Por ejemplo, si quisiéramos añadir un conjunto de excepciones al modelo, en formato XML tendríamos:

```
<exceptions>
  <exception>ArticleContent</exception>
  <exception>ArticleDisplayDate</exception>
  <exception>ArticleExpirationDate</exception>
  <exception>ArticleId</exception>
  <exception>ArticleReviewDate</exception>
  <exception>ArticleSmallImageName</exception>
  <exception>ArticleSmallImageSize</exception>
  <exception>ArticleTitle</exception>
  <exception>ArticleVersion</exception>
  <exception>DuplicateArticleId</exception>
  <exception>DuplicateFeedId</exception>
  <exception>DuplicateFolderName</exception>
  <exception>FeedContentField</exception>
  <exception>FeedId</exception>
  <exception>FeedName</exception>
  <exception>FeedTargetLayoutFriendlyUrl</exception>
  <exception>FeedTargetPortletId</exception>
  <exception>FolderName</exception>
```

```
  <exception>InvalidDDMStructure</exception>
</exceptions>
```

Adición de excepciones al modelo utilizando el descriptor XML.

Con el DSL, siguiendo el estilo de API fluent, tendríamos la siguiente forma simplificada:

```
ServiceBuilder serviceBuilder =
  new ServiceBuilder.Builder("com.liferay.journal", "Journal").withExceptions(
  "ArticleContent", "ArticleDisplayDate",
  "ArticleExpirationDate", "ArticleId", "ArticleReviewDate",
  "ArticleSmallImageName", "ArticleSmallImageSize",
  "ArticleTitle", "ArticleVersion", "DuplicateArticleId",
  "DuplicateFeedId", "DuplicateFolderName",
  "FeedContentField", "FeedId", "FeedName",
  "FeedTargetLayoutFriendlyUrl", "FeedTargetPortletId",
  "FolderName", "InvalidDDMStructure")
.build();
```

Adición de múltiples excepciones utilizando el DSL implementado.

La ventaja en cuanto a líneas de código escritas es significativa, pero además es importante la ganancia en cuanto a evitar código redundante. Cada elemento XML tiene su etiqueta de apertura y de cierre, mientras que en el API *fluent* únicamente tenemos la invocación del método para añadir varias excepciones al mismo tiempo.

Esta ganancia en elementos a escribir dependerá mucho de cómo se haya diseñado el API *fluent* respecto a su versión anterior en XML. Por ejemplo, no observamos ganancia en cuanto a espacio y número de líneas escritas al aplicar el DSL a la creación de una columna frente a utilizar el descriptor XML:

```
<column name="id" primary="true" type="long" />
```

Definición de una columna utilizando el XML.

```
Column journalArticleColumnId =
  ColumnBuilderFactory.getColumnBuilder("id", ServiceBuilderType.LONG)
    .asPrimaryKey()
    .build();
```

Definición de una columna utilizando el DSL implementado.

Sin embargo, el tipado en el DSL nos impide cometer un error a la hora de definir el tipo de la columna. Por ejemplo:

```
<column name="id" primary="true" type="lon" />
```

Definición de una columna utilizando el XML con un tipo incorrecto.

Este error no sería detectado hasta que se ejecutase la generación de código y se compilase el código autogenerado, puesto que el XML no es compilado y no permite la validación de valores de atributos. Con el DSL, sin embargo, el parámetro de la factoría únicamente permite utilizar un tipo enumerado, ganando en robustez y mantenibilidad, lo cual impacta en la calidad del proceso de desarrollo.

6. Proyecto desarrollado

A continuación se detalla el proyecto desarrollado que implementa el DSL especificado en este documento.

Se describirán los pasos para construirlo, para ejecutar las pruebas automatizadas, y para ejecutar el proyecto.

6.1. Código fuente del DSL interno propuesto

El código fuente de este DSL queda recogida en un proyecto Java, denominado Liferay DSL Builder. Su código puede encontrarse en un repositorio de código abierto en Github [49], bajo una licencia de código abierto GPL v3 [50], por lo que cualquiera puede descargarlo y modificarlo sin cambiar las atribuciones al desarrollador original.

6.1.1. Estructura de paquetes

En los paquetes definidos para el proyecto se implementarán los métodos del API *fluent* que describen las operaciones descritas en los apartados anteriores. Algunos ejemplos de estas operaciones podrían ser la no adición de entidades duplicadas, o la asignación valores dependientes unos de otros, entre otros.

A continuación se detallan las clases del proyecto, organizadas por paquetes.

6.1.1.1. com.liferay.servicebuilder.dsl.domain

Bajo este paquete se encuentran las clases Java de los objetos del dominio, esto es, las clases que contienen el DSL objetivo de este trabajo.

El diagrama UML de clases de este paquete es el siguiente:

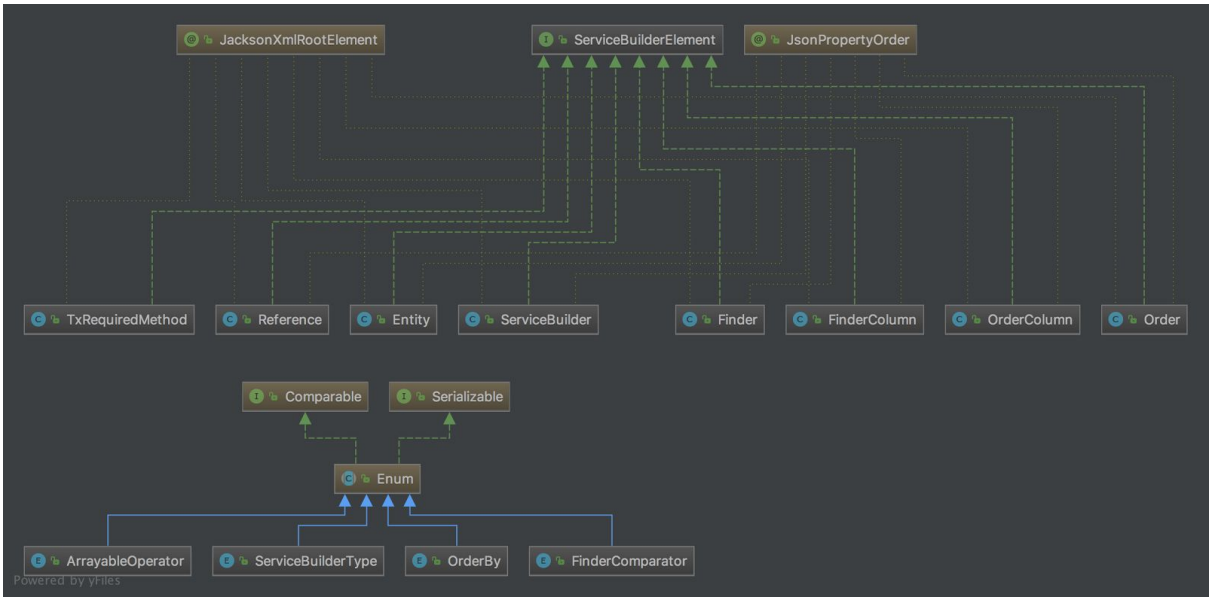


Diagrama de clases del paquete com.liferay.servicebuilder.dsl.domain

Las clases que aquí se encuentran son:

- ArrayableOperator: clase Java de tipo enumeración que recoge los valores constantes para los criterios de adición de múltiples valores en las búsquedas, AND Y OR.

- Entity: clase Java que representa las operaciones del DSL sobre las entidades del dominio.

- Finder: clase Java que representa las operaciones del DSL sobre los métodos de búsqueda de entidades del dominio.

- FinderColumn: clase Java que representa las operaciones del DSL sobre las columnas de búsqueda en los métodos de búsqueda de entidades del dominio.

- FinderComparator: clase Java de tipo enumeración que recoge los valores constantes para los criterios de comparación en las búsquedas, DISTINCT, EQUALS, GREATER, GREATER_EQUALS, LESS, LESS_EQUALS, LIKE.

- Order: : clase Java que representa las operaciones del DSL sobre el criterio de ordenación en las entidades del dominio.

- OrderBy: clase Java de tipo enumeración que recoge los valores constantes para los criterios de ordenación, ASC y DESC.

- OrderColumn: clase Java que representa las operaciones del DSL sobre las columnas de ordenación de los criterios de ordenación en las entidades del dominio.

- Reference: clase Java que representa las operaciones del DSL sobre las referencias a otras entidades del dominio.

- ServiceBuilder: clase Java que representa las operaciones del DSL sobre el modelo completo del dominio. Será el hilo conductor de todas las operaciones, pues es necesario una instancia de esta clase para poder componer el modelo con el resto de entidades del dominio.

- ServiceBuilderElement: interfaz que representa cualquier elemento del dominio. Por tanto, una entidad, una columna, un método de búsqueda, un método de ordenación, etc, serán un ServiceBuilderElement. De esta manera se facilita el trabajo con código genérico o abstracto, en lugar del uso de clases concretas.

- TxRequiredMethod: clase Java que representa las operaciones del DSL sobre los métodos que requieren transacciones en las entidades del dominio.

6.1.1.2. com.liferay.servicebuilder.dsl.domain.column

Este paquete contiene las clases Java de los objetos del dominio relacionados con las columnas de las entidades del modelo.

El diagrama UML de clases de este paquete es el siguiente:

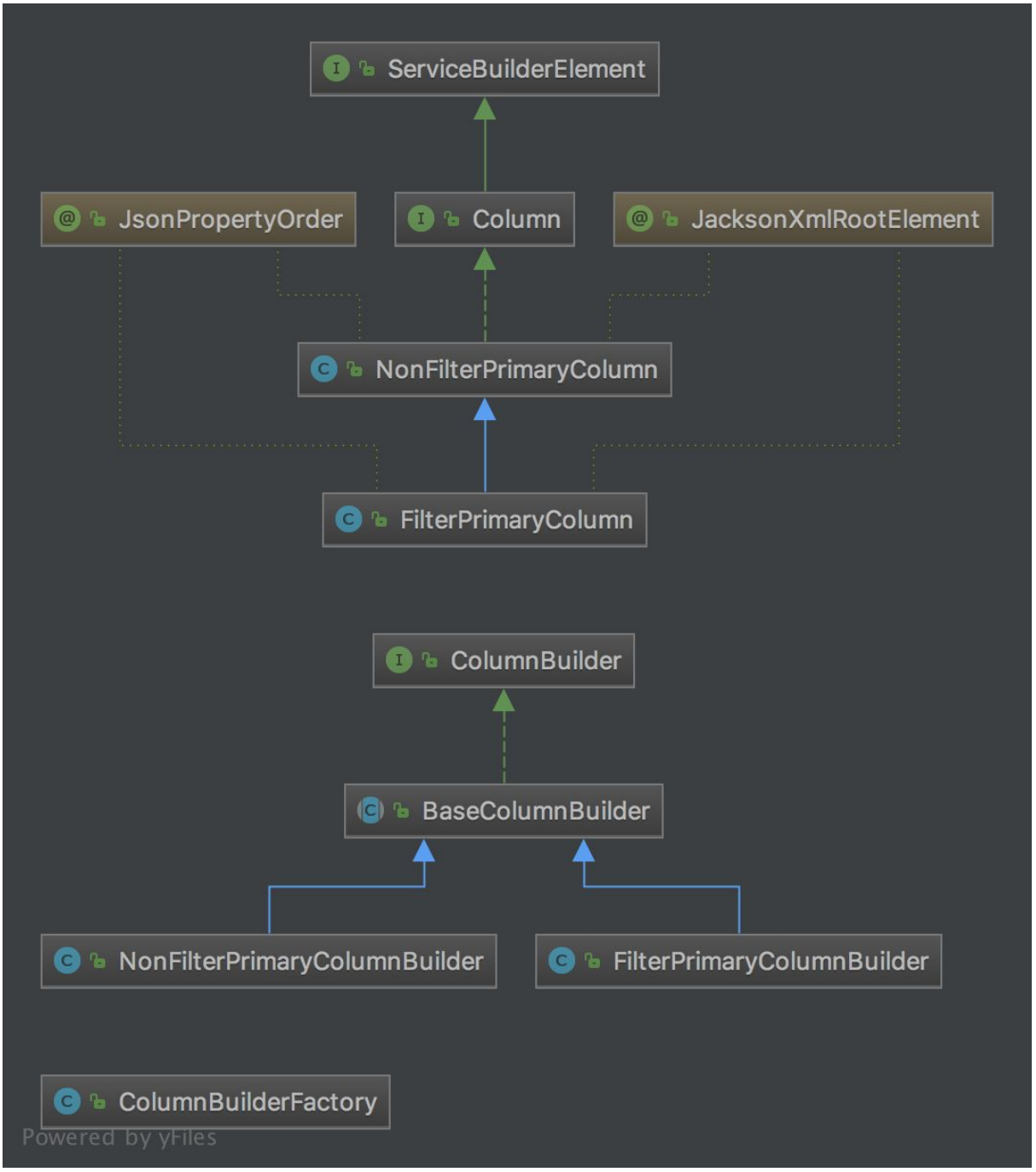


Diagrama de clases del paquete com.liferay.servicebuilder.dsl.domain.column

Las clases que aquí se encuentran son:

- BaseColumnBuilder, clase que implementa la interfaz ColumnBuilder, que representa el patrón Builder para la construcción de objetos del tipo Column. Tendrá todos los métodos comunes a los dos tipos de columna que existen: FilterPrimary y NonFilterPrimary, definiendo métodos abstractos que implementarán las subclases. De esta manera, se podrá aplicar el patrón TemplateMethod en la clase abstracta.

- Column: interfaz con los métodos getter/setter que representan el acceso a las diferentes características de un atributo de una entidad.

- ColumnBuilder: interfaz con los métodos del API fluent que determinarán las operaciones del DSL que construye y compone columnas.

- ColumnBuilderFactory, clase de utilidad para facilitar la creación de objetos concretos del Builder de columna, en este caso permite crear columnas de tipo FilterPrimary o columnas NonFilterPrimary.

- FilterPrimaryColumn: clase concreta que implementa una columna con la característica filter-primary a true. Esta clase es necesaria para aislar las operaciones que con ella se pueden hacer en el DSL, puesto que sólo es posible añadir una columna de este tipo.

- FilterPrimaryBuilder: clase concreta que implementa el patrón Builder para una columna con la característica filter-primary a true. Esta clase es necesaria para aislar las operaciones que con ella se pueden hacer en el DSL, puesto que sólo es posible añadir una columna de este tipo.

- NonFilterPrimaryColumn: clase concreta que implementa una columna con la característica filter-primary a false. Esta clase representa a las columnas regulares.

- NonFilterPrimaryBuilder: clase concreta que implementa el patrón Builder para una columna con la característica filter-primary a false. Esta clase representa a las operaciones que se pueden hacer con las columnas regulares.

6.1.1.3. com.liferay.servicebuilder.dsl.io

Este paquete contiene las clases Java de los procesos de lectura/escritura de ficheros, necesarios para la generación del fichero XML descriptor del modelo.

En ellas se producirá la generación de un fichero XML.

El diagrama UML de clases de este paquete es el siguiente:

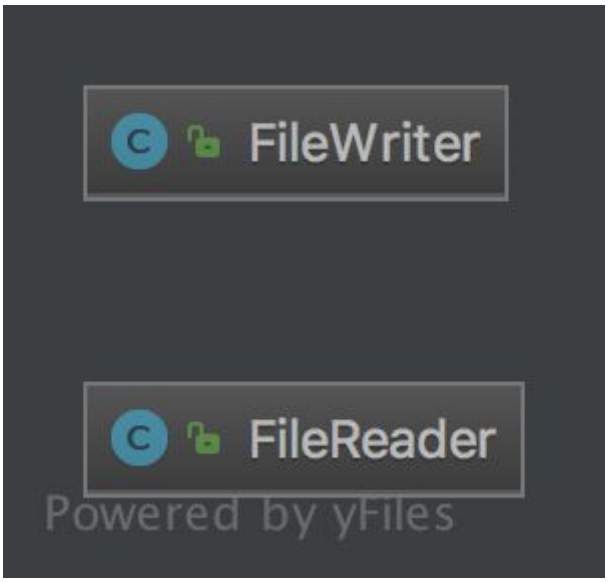


Diagrama de clases del paquete com.liferay.servicebuilder.dsl.io

Las clases que aquí se encuentran son:

- FileReader: en esta clase permite leer de un fichero XML, pasado por parámetro.
- FileWriter: en esta clase permite escribir un objeto ServiceBuilder del dominio del problema a un fichero XML, pasados ambos por parámetro.

6.1.1.4. com.liferay.servicebuilder.dsl.xml

Este paquete contiene las clases Java de los procesos de serialización de las entidades del dominio del problema a un formato XML.

El diagrama UML de clases de este paquete es el siguiente:

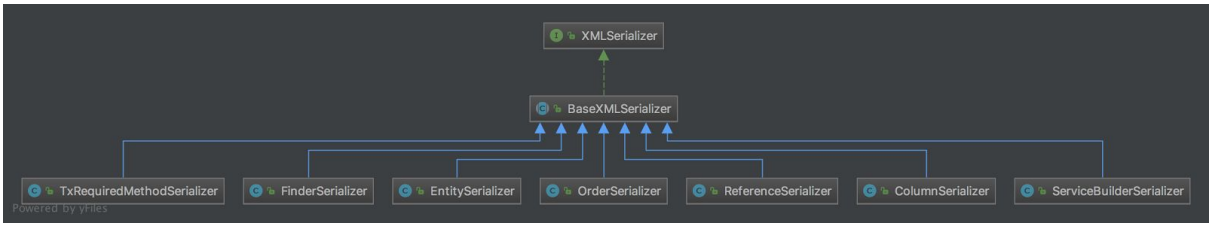


Diagrama de clases del paquete com.liferay.servicebuilder.dsl.io

Las clases que aquí se encuentran son:

- BaseXMLSerializer: en esta clase abstract, que implementa la interfaz XMLSerializer, únicamente tiene un método de serialización, se implementará la serialización de un objeto Entity a un formato XML válido según el DTD de Liferay Portal. Este método seguirá el patrón TemplateMethod, por el cual las clases que extiendan esta clase abstracta compondrán las partes que el algoritmo defina como reemplazables. En este caso, la construcción del serializer mediante un objeto de tipo genérico (o interfaz) ServiceBuilderElement. Por ello, cualquier objeto del modelo podrá ser serializado, ya que todos implementan dicha interfaz. La serialización se realizará utilizando las librerías de Jackson.

- ColumnSerializer: contribuye al algoritmo proporcionando un constructor a partir de un objeto Column.

- EntitySerializer: contribuye al algoritmo proporcionando un constructor a partir de un objeto Entity.

- FinderSerializer: contribuye al algoritmo proporcionando un constructor a partir de un objeto Finder.

- OrderSerializer: contribuye al algoritmo proporcionando un constructor a partir de un objeto Order.

- ReferenceSerializer: contribuye al algoritmo proporcionando un constructor a partir de un objeto Reference.

- ServiceBuilderSerializer: contribuye al algoritmo proporcionando un constructor a partir de un objeto ServiceBuilder.

- TxRequiredMethodSerializer: contribuye al algoritmo proporcionando un constructor a partir de un objeto TxRequiredMethod.

- XMLSerializer: interfaz que define el contrato para serializar un objeto.

6.1.2. Estructura de paquetes de Test

A continuación se detallan las clases de tests, organizadas por paquetes.

6.1.2.1. com.liferay.servicebuilder.dsl.domain

Este paquete contiene las clases Java de test de los objetos del dominio, esto es, las clases que contienen el DSL objetivo de este trabajo.

En ellos se verificará que los métodos del API *fluent* tienen el comportamiento esperado, como podría ser la no adición de entidades duplicadas, o la asignación valores dependientes unos de otros, entre otros.

Las clases de test que aquí se encuentran son:

- EntityTest: en esta clase se probarán de manera unitaria las siguientes operaciones de construcción de una entidad:
    - no invocar ningún método del API *fluent* asigna los valores predeterminados
    - deshabilitar la cache
    - añadir una columna
    - añadir columnas con el mismo nombre no añade la segunda
    - añadir una columna del tipo filter-primary permite añadir más columnas de tipo non-filter-primary.
    - añadir una columna del tipo filter-primary no permite añadir una columna de tipo non-filter-primary con el mismo nombre.
    - añadir una columna del tipo filter-primary no permite añadir otra columna de tipo filter-primary.
    - añadir varias columnas a la vez
    - añadir un datasource
    - deprecar una entidad
    - habilitar la actualización dinámica
    - habilitar el control de concurrencia multiversión no afecta a la actualización dinámica
    - verificar que si no se habilita la actualización dinámica este valor se toma del control de concurrencia multiversión.
    - añadir un método de búsqueda
    - añadir métodos de búsqueda con el mismo nombre no añade el segundo
    - añadir varios métodos de búsqueda a la vez
    - definir un nombre legible por un humano
    - habilitar la serialización en JSON
    - habilitar los servicios remotos habilita la serialización en JSON
    - habilitar los servicios locales

    - deshabilitar el control de concurrencia multiversión
    - habilitar el control de concurrencia multiversión
    - añadir un orden
    - definir la clase de persistencia
    - añadir una referencia
    - añadir referencias con el mismo nombre no añade la segunda
    - añadir varias referencias a la vez
    - habilitar los servicios remotos
    - definir la factoría de sesiones
    - definir la tabla de base de datos
    - habilitar la papelera de reciclaje
    - habilitar el mánager de transacciones
    - deshabilitar el mánager de transacciones
    - añadir un método que requiera transacciones
    - añadir métodos que requieran transacciones con el mismo nombre no añade el segundo
    - añadir varios métodos que requieran transacciones a la vez
    - habilitar la creación de un campo UUID
    - habilitar la creación de un accesor al UUID
    - comprobar que la igualdad de dos entidades se define a partir de tener el mismo nombre

- FinderColumnTest: en esta clase se probarán de manera unitaria las siguientes operaciones de construcción de una columna de búsqueda:

    - no invocar ningún método del API *fluent* asigna los valores predeterminados
    - definir la búsqueda como no sensible a mayúsculas
    - definir el operador sobre arrays
    - definir el criterio de comparación
    - comprobar que la igualdad de dos columnas de búsqueda se define a partir de tener el mismo nombre

- FinderTest: en esta clase se probarán de manera unitaria las siguientes operaciones de construcción de un método de búsqueda:

    - no invocar ningún método del API *fluent* asigna los valores predeterminados
    - añadir un método que devuelve una colección de entidades
    - añadir un método que retorna un único elemento
    - añadir un criterio de búsqueda que siempre se añadirá

    - añadir dos columnas de búsqueda con el mismo nombre no añadirá la segunda
    - añadir una columna de búsqueda
    - añadir varias columnas de búsqueda a la vez
    - deshabilitar la creación de índices de SQL
    - comprobar que la igualdad de dos métodos de búsqueda se define a partir de tener el mismo nombre

- OrderColumnTest: en esta clase se probarán de manera unitaria las siguientes operaciones de construcción de una columnas de ordenación:

    - no invocar ningún método del API *fluent* asigna los valores predeterminados
    - definir que la ordenación no será sensible a mayúsculas
    - definir que la ordenación será descendente
    - comprobar que la igualdad de dos columnas de ordenación se define a partir de tener el mismo nombre

- OrderTest: en esta clase se probarán de manera unitaria las siguientes operaciones de construcción de un método de ordenación:

    - no invocar ningún método del API *fluent* asigna los valores predeterminados
    - definir que la ordenación será ascendente
    - definir que la ordenación será descendente
    - añadir una columna de ordenación
    - añadir dos columnas de ordenación con el mismo nombre no añadirá la segunda
    - añadir varias columnas de ordenación a la vez

- ReferenceTest: en esta clase se probarán de manera unitaria las siguientes operaciones de construcción de una referencia:

    - no invocar ningún método del API *fluent* asigna los valores predeterminados
    - inyectar un servicio
    - comprobar que la igualdad de dos referencias se define a partir de tener la misma entidad y el mismo paquete

- ServiceBuilderTest: en esta clase se probarán de manera unitaria las siguientes operaciones de construcción de un descriptor del modelo de Service Builder:

    - no invocar ningún método del API *fluent* asigna los valores predeterminados
    - importar de manera automática las referencias por defecto
    - prefijar las tablas con el espacio de nombres
    - añadir el autor
    - añadir varios autores únicamente asigna el último
    - importar un descriptor del modelo de Service Builder
    - importar dos descriptores del modelo de Service Builder con el mismo path sólo importa uno
    - importar varios descriptores del modelo de Service Builder
    - añadir varias entidades a la vez
    - añadir una entidad
    - añadir dos entidades con el mismo nombre no añade la segunda
    - añadir una excepción
    - añadir dos excepciones con el mismo nombre no añade la segunda
    - añadir varias excepciones a la vez
    - habilitar el control de concurrencia multiversión

- TxRequiredMethodTest: en esta clase se probarán de manera unitaria las siguientes operaciones de construcción de un método que requiera transacciones:

    - no invocar ningún método del API *fluent* asigna los valores predeterminados
    - añadir un método
    - comprobar que la igualdad de dos métodos que requieran transacciones se define a partir de tener el mismo nombre de método

6.1.2.2. com.liferay.servicebuilder.dsl.domain.column

Este paquete contiene las clases Java de test de los objetos del dominio relacionados con las columnas de las entidades del modelo.

Las clases de test que aquí se encuentran son:

- BaseColumnTest: en esta clase abstract, que mantiene los métodos comunes a los dos tipos de columna (FilterPrimary y NonFilterPrimary), se probarán de manera unitaria las siguientes operaciones de construcción de una columna:

    - no invocar ningún método del API *fluent* asigna los valores predeterminados
    - definir como clave primaria
    - autogenerar la clave primaria desde una clase
    - autogenerar la clave primaria desde una identidad
    - autogenerar la clave primaria desde un incremento
    - autogenerar la clave primaria desde una secuencia
    - definir si se refiere a un contenedor
    - definir si se convierten los valores nulos
    - definir como filter-primary
    - definir si es localizable (soporta multi-idioma)
    - definir si se refiere al contenedor padre
    - definir si tiene un accessor
    - definir el nombre de base de datos
    - habilitar la serialización en JSON
    - definir una relación muchos a muchos
    - definir una relación muchos a muchos con otro descriptor del modelo de Service Builder
    - deshabilitar la lectura lazy en campos Blob
    - deshabilitar la lectura lazy en campos no Blob
    - comprobar que la igualdad de dos columnas se define a partir de tener el mismo nombre

- ColumnBuilderFactoryTest: en esta clase se probarán de manera unitaria las siguientes operaciones de obtención de columnas:

    - obtener una columna de tipo non-filter-primary, esto es, regular.
    - obtener una columna de tipo filter-primary

- FilteredPrimaryColumnTest: en esta clase se implementarán los métodos abstractos definidos en la clase base BaseColumnTest.

- NonFilteredPrimaryColumnTest: en esta clase se implementarán los métodos abstractos definidos en la clase base BaseColumnTest.

6.1.2.3. com.liferay.servicebuilder.dsl.io

Este paquete contiene las clases Java de test de los procesos de lectura/escritura de ficheros, necesarios para la generación del fichero XML descriptor del modelo.

En ellos se verificará que la invocación genera un fichero XML.

Las clases de test que aquí se encuentran son:

- FileIOTest: en esta clase se probarán de manera unitaria las siguientes operaciones en cuanto a la entrada/salida:
    - crear un fichero XML
    - leer de un fichero XML

6.1.2.4. com.liferay.servicebuilder.dsl.xml

Este paquete contiene las clases Java de test de los procesos de serialización de las entidades del dominio del problema a un formato XML.

Las clases de test que aquí se encuentran son:

- EntitySerializerTest: en esta clase se probará de manera unitaria que un objeto Entity es serializado en un formato XML válido según el DTD de Liferay Portal:
    - serializar un objeto Entity
- FilterPrimaryColumnSerializerTest: en esta clase se probará de manera unitaria que un objeto Column del tipo filter-primary es serializado en un formato XML válido según el DTD de Liferay Portal:
    - serializar un objeto FilterPrimaryColumn
- FinderSerializerTest: en esta clase se probará de manera unitaria que un objeto Finder es serializado en un formato XML válido según el DTD de Liferay Portal:
    - serializar un objeto Finder
- NonFilterPrimaryColumnSerializerTest: en esta clase se probará de manera unitaria que un objeto Column regular (no filter-primary) es serializado en un formato XML válido según el DTD de Liferay Portal:
    - serializar un objeto NonFilterPrimaryColumn

- OrderSerializerTest: en esta clase se probará de manera unitaria que un objeto Order es serializado en un formato XML válido según el DTD de Liferay Portal:
    - serializar un objeto Order
- ReferenceSerializerTest: en esta clase se probará de manera unitaria que un objeto Reference es serializado en un formato XML válido según el DTD de Liferay Portal:
    - serializar un objeto Reference
- ServiceBuilderSerializerTest: en esta clase se probará de manera unitaria que un objeto ServiceBuilder, construido a partir del DSL con las demás entidades del modelo, es serializado en un formato XML válido según el DTD de Liferay Portal:
    - serializar un objeto ServiceBuilder
- TxRequiredMethodSerializerTest: en esta clase se probará de manera unitaria que un objeto TxRequiredMethod es serializado en un formato XML válido según el DTD de Liferay Portal:
    - serializar un objeto TxRequiredMethod

6.2. Construcción

Para construir el proyecto es necesario tener instalado Java en su versión 8 en el equipo local.

No es necesario instalar Gradle [3] en el sistema, pues el proyecto incluye un wrapper [51] de Gradle para utilizar siempre la versión necesaria para su construcción, sin delegar en la del sistema.

Una vez instalado, basta con ejecutar el comando "*./gradlew assemble*" para construir el proyecto:

```
~/proyectos/liferay-service-builder • ./gradlew assemble
Starting a Gradle Daemon (subsequent builds will be faster)
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:jar
:startScripts
:distTar
:distZip
:assemble

BUILD SUCCESSFUL

Total time: 4.545 secs
```

Detalle del proceso de construcción.

6.3. Análisis de la calidad de código

Tal y como hemos descrito en el apartado relativo a la calidad del código, vamos a enumerar aquí las buenas prácticas que se han implementado para garantizar la calidad del código, y que además se garantice que el software hace lo que los requisitos funcionales dicten.

En el caso concreto de este proyecto, los requisitos funcionales consisten en validar que las operaciones existentes en el DSL anterior, el formado por el descriptor en XML, existen en el nuevo DSL, y además generan el mismo formato de salida que éste, puesto que el generar el mismo XML era uno de los requisitos de partida del proyecto.

En cuanto a la calidad del proceso:

1. se han escrito pruebas unitarias
2. se ha implantado integración continua
3. se ha analizado la cobertura de los tests sobre el código

6.3.1. Ejecución de las pruebas automatizadas

Para ejecutar los tests automatizados, basta con ejecutar el comando "*./gradlew test*":

```
com.liferay.servicebuilder.dsl.xml.FinderSerializerTest > testSerialize PASSED

com.liferay.servicebuilder.dsl.xml.NonFilterPrimaryColumnSerializerTest > testSerialize PASSED

com.liferay.servicebuilder.dsl.xml.OrderSerializerTest > testSerialize PASSED

com.liferay.servicebuilder.dsl.xml.ReferenceSerializerTest > testSerialize PASSED

com.liferay.servicebuilder.dsl.xml.ServiceBuilderSerializerTest > testSerialize PASSED

com.liferay.servicebuilder.dsl.xml.TxRequiredMethodSerializerTest > testSerialize PASSED

-------------------------------------------------------------------------
|  Results: SUCCESS (156 tests, 156 successes, 0 failures, 0 skipped)  |
-------------------------------------------------------------------------

BUILD SUCCESSFUL

Total time: 3.548 secs
```

Detalle del proceso de ejecución de tests automatizados.

Esta tarea ejecutará toda la batería de pruebas descrita con anterioridad. Esta ejecución nos garantiza que no se producen regresiones sobre la base de código actual en base a unos tests actualizados.

6.3.2. Análisis estático de código

Para ejecutar el análisis estático de código, identificado por las herramientas PMD [14] y FindBugs [17], basta con ejecutar los siguientes comandos:

- *./gradlew pmdMain*: ejecuta PMD para el código de producción.
- *./gradlew pmdTest*: ejecuta PMD para el código de test.
- *./gradlew findBugsMain*: ejecuta FindBugs para el código de producción.
- *./gradlew findBugsTest*: ejecuta FindBugs para el código de test.

Gradle dispone de una fase en su ciclo de vida para realizar comprobaciones, denominada *check*. SI ejecutamos "*./gradlew check*" se ejecutarían todas las comprobaciones, que en este caso incluiría PMD, FindBugs y la ejecución de tests.

6.3.3. Integración continua

Para implementar integración continua, tal y como se ha descrito anteriormente, se ha utilizado el servicio Travis CI [5], que es un servicio de integración continua (CI as a Service) gratuito para proyectos Open Source, como es el caso del proyecto aquí desarrollado.

Travis CI basa su servicio en definir un ciclo de vida del proyecto, compuesto por fases, de modo que permite "engancharse" a ciertas fases de ese ciclo de vida. De esta manera, el servicio consigue ser agnóstico al lenguaje y a la plataforma, puesto que cada desarrollador de cada proyecto indicará en cada fase qué es lo que quiere realizar.

Para ello, es posible indicarle a Travis CI el ciclo de vida concreto que queremos ejecutar, utilizando los puntos de extensión que ofrece. A través de un descriptor denominado *.travis.yml* [52], ubicado en el directorio raíz del proyecto, es posible indicar a Travis CI que realice tareas en cada punto del ciclo de vida definido.

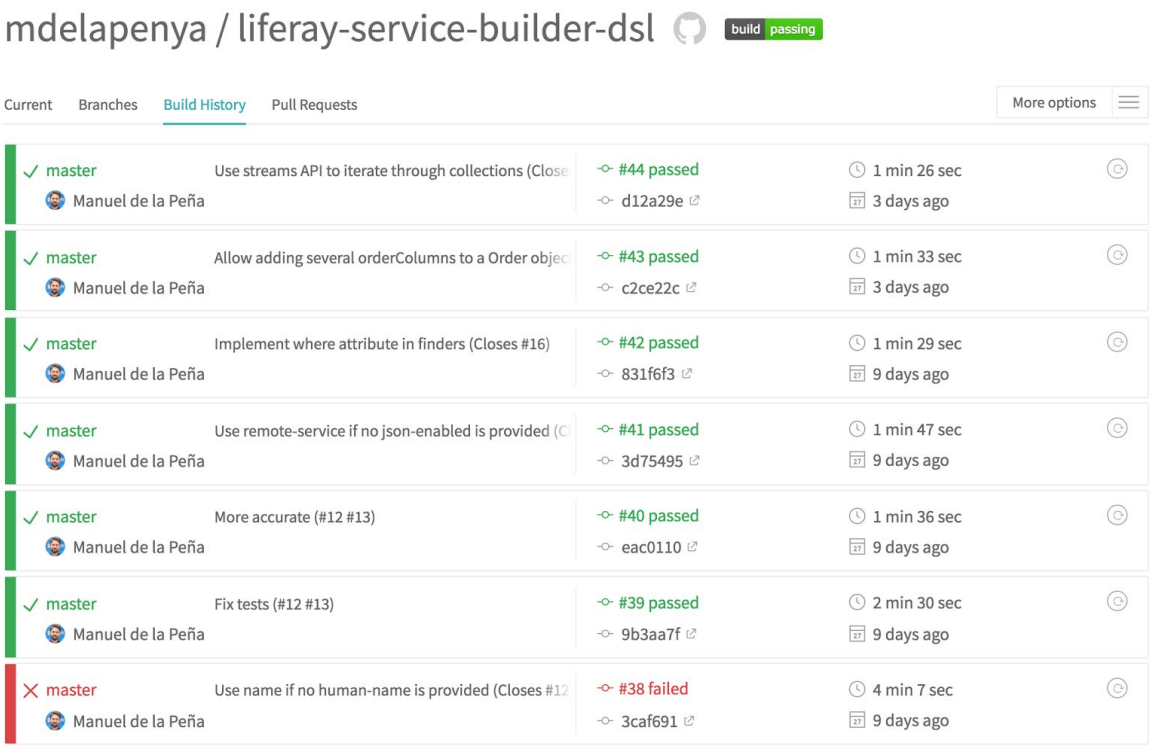
```
jdk:
 - oraclejdk8
language: java

install:
 - ./gradlew --info assemble
script:
 - ./gradlew check
 - ./gradlew jacocoTestReport

before_cache:
 - rm -f  $HOME/.gradle/caches/modules-2/modules-2.lock
 - rm -fr $HOME/.gradle/caches/*/plugin-resolution/

cache:
 directories:
   - $HOME/.gradle/caches/
   - $HOME/.gradle/wrapper/
after_success:
 - bash <(curl -s https://codecov.io/bash)
```

Detalle del descriptor de Travis CI

En Travis CI, la fase principal viene definida por el punto de extensión **script**. En ella es donde debemos ejecutar las tareas que nos interesen. En el caso de nuestro DSL, se ejecutan las comprobaciones descritas anteriormente (*./gradlew check*) y una medición de cobertura de código (*./gradlew jacocoTestReport*).

En la fase **after_success**, que se ejecuta únicamente si no existieron errores en la fase de **script**, enviamos al servicio Codecov.io los datos de cobertura, para obtener un análisis pormenorizado de la cobertura del código, como veremos más adelante.

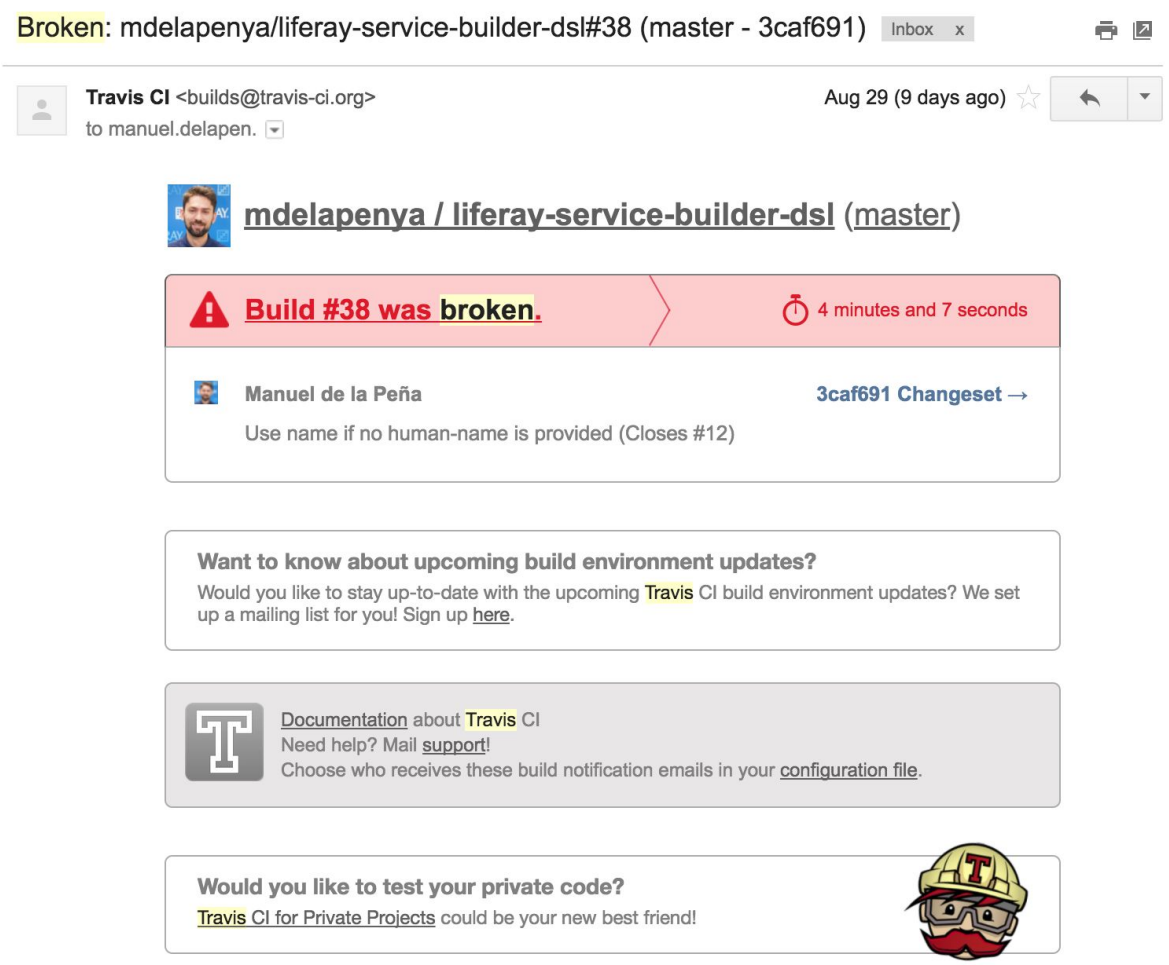
El proyecto de este trabajo es analizado continuamente en Travis CI [53] donde, al ser un proyecto Open Source, es posible observar la evolución del mismo de manera gratuita:



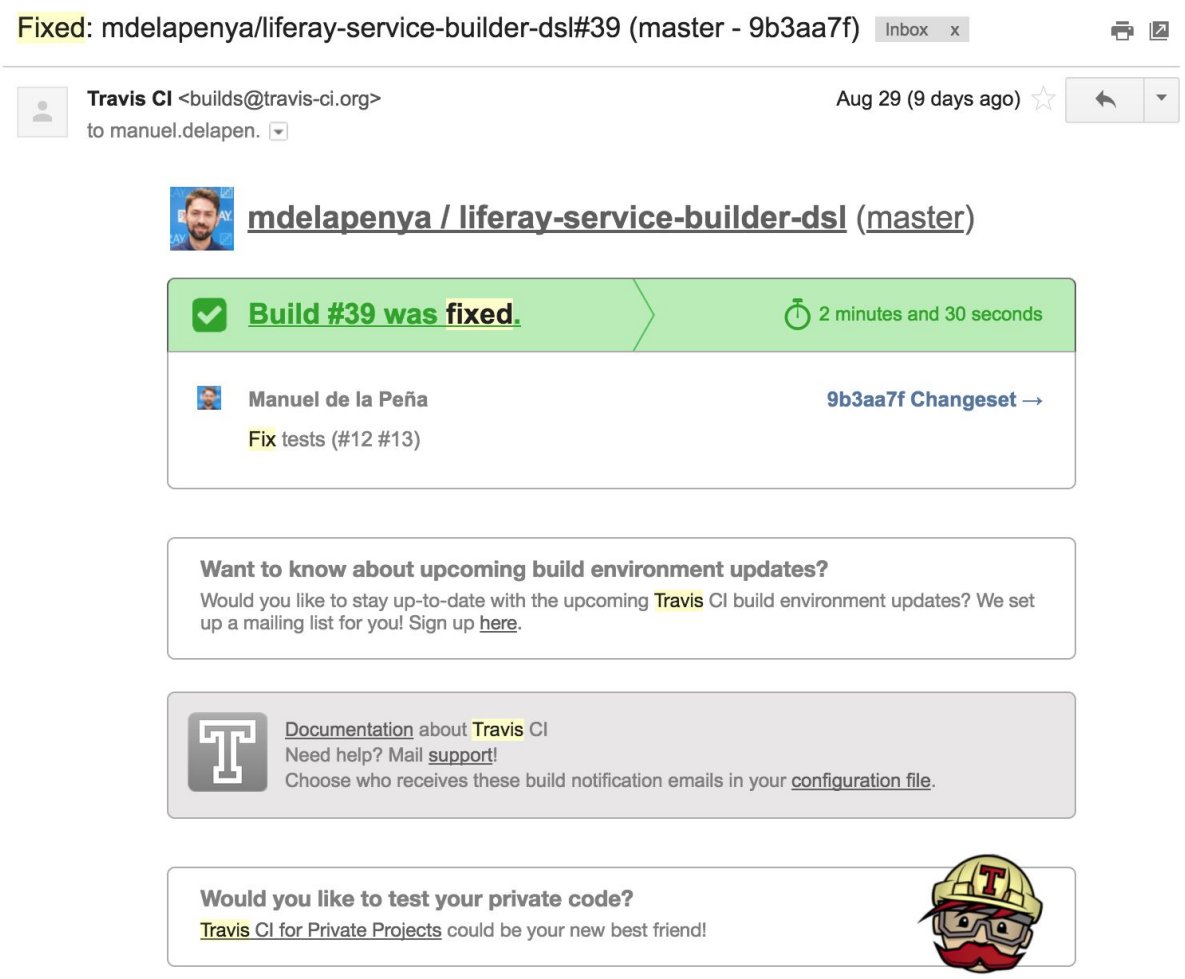
Detalle del estado del proyecto en TravisCI.

Cada ejecución, denominada habitualmente *build*, ejecuta el ciclo de vida completo definido, y pasará (verde) o se romperá (rojo) si existe algún fallo en las diferentes fases del ciclo de vida o *pipeline*, notificando al desarrollador mediante un email del fallo.

En la imagen se observa que hubo una ejecución fallida, la #38 [54], en este caso debido a que un test falló tras unos cambios. La siguiente ejecución, #39 [55], arregla los tests, dejando el proyecto en verde de nuevo, con la consiguiente notificación por email del arreglo.



Detalle de notificación por email de proyecto fallido en TravisCI.



Detalle de notificación por email de proyecto corregido en TravisCI.

6.3.4. Medición de la cobertura de código

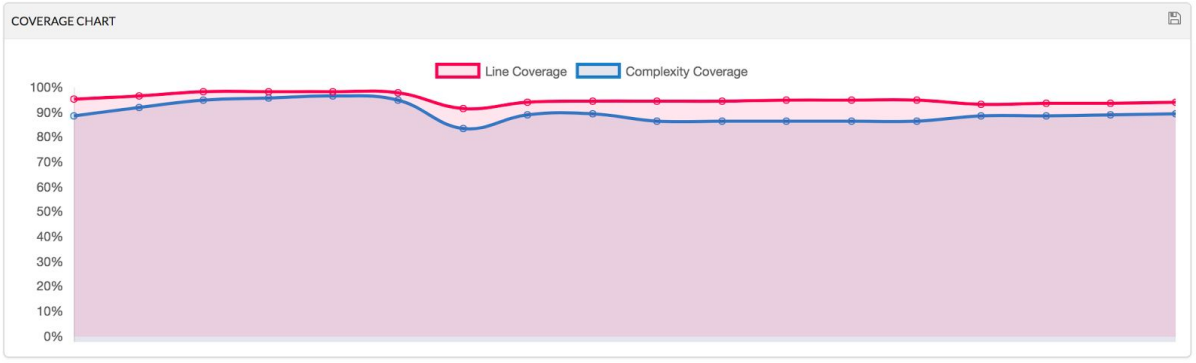
Para analizar la cobertura de código el proyecto utiliza JaCoCo [56], proyecto de medición de cobertura para proyectos Java con integración muy sencilla en proyectos Gradle. Para ello aporta al ciclo de vida de Gradle una tarea "*jacocoTestReport*" para realizar la medición en base a los tests escritos.

Como vimos anteriormente en el descriptor de Travis CI, invocaremos esta tarea en la fase de **script**, de modo que si no ha habido ningún fallo, en la fase de **after_sucess** invocaremos el servicio de Codecov.io [6], servicio cloud de métricas de cobertura, igualmente gratuito para proyectos Open Source, mediante una llamada descrita en su documentación oficial [57].

Una vez recopilados los informes de cobertura de código, éstos son enviados a Codecov.io. En él podremos ver de una manera muy sencilla y visual el estado del proyecto en cuanto a cobertura se refiere.

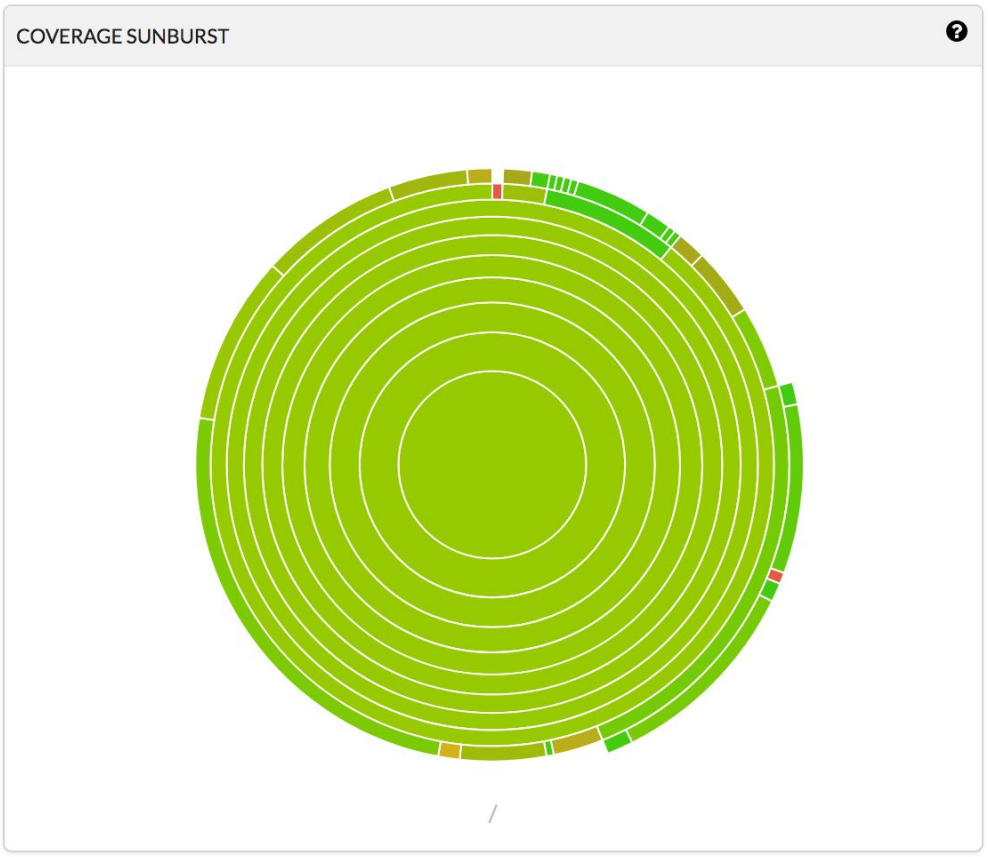
Además, si la cobertura descendiese, seríamos notificados de ello, haciendo que fallara el proceso de construcción de Travis CI comentado con anterioridad, por tanto recibiríamos un feedback muy valioso en tiempo de desarrollo.

La cobertura de código del proyecto se encuentra alojada en Codecov.io [58], y de ellas se han obtenido las siguientes gráficas:



Detalle de la tendencia en cuanto a porcentaje de código cubierto por los tests en Codecov.

El gráfico anterior muestra tanto la cobertura de código como la cobertura sobre el código en relación a la complejidad ciclomática.



Detalle del estado de cada clase y paquete del proyecto en cuanto a cobertura de código en Codecov.

El gráfico anterior es navegable, por tanto es posible clicar sobre las zonas para llegar a las clases de interés.

Files					Complexity	Coverage
src/main/java/com/liferay/servicebuilder/dsl	512	481	17	14	89.59%	93.94%
Project Totals (29 files)	512	481	17	14	89.59%	93.94%

Tabla resumen del estado de la cobertura del código del proyecto en Codecov.

En esta tabla resumen aparecen los paquetes de código del proyecto, pudiendo navegar por ellos mostrando los datos particulares para cada elemento presentado.

/ ... / java / com / liferay / servicebuilder / dsl / domain

Files					Complexity	Coverage
column	119	115	2	2	95.23%	96.63%
ArrayableOperator.java	6	5	0	1	66.66%	83.33%
Entity.java	126	121	4	1	96.66%	96.03%
Finder.java	39	36	2	1	90.00%	92.30%
FinderColumn.java	24	22	1	1	85.71%	91.66%
FinderComparator.java	7	6	0	1	66.66%	85.71%
Order.java	23	22	1	0	100.00%	95.65%
OrderBy.java	2	2	0	0	100.00%	100.00%
OrderColumn.java	22	20	1	1	83.33%	90.90%
Reference.java	19	17	1	1	85.71%	89.47%
ServiceBuilder.java	47	44	3	0	100.00%	93.61%
ServiceBuilderType.java	8	7	0	1	75.00%	87.50%
TxRequiredMethod.java	14	12	1	1	75.00%	85.71%
Folder Totals (13 files)	456	429	16	11	92.15%	94.07%
Project Totals (29 files)	512	481	17	14	89.59%	93.94%

Tabla resumen del estado de la cobertura del código del proyecto para un paquete en particular en Codecov..

7. Conclusiones y Trabajos futuros

7.1. Conclusiones

Este trabajo ha servido para acercarnos por un lado al mundo de la calidad del software, mediante la definición de procesos que ayuden a los equipos de desarrollo a instaurar buenas prácticas que les ayuden a escribir software "*que funcione*", como podría ser la integración continua o la escritura de tests.

Además nos hemos aprovechado de la flexibilidad de los lenguajes DSL para construir un idioma específico para el dominio de interés, que no es otro que desarrollar aplicaciones basadas en Liferay Portal, acercando el conocimiento de los analistas de negocio a los desarrolladores.

Y por último hemos puesto el esfuerzo en explicar el descriptor en XML del modelo de las aplicaciones basadas en Liferay Portal para construir éstas de una manera mucho más eficaz que utilizar dicho descriptor, como se realiza hasta el momento, y utilizar el DSL implementado.

7.2. Trabajos futuros

En el camino por delante, nos encontramos diferentes alternativas según la perspectiva con que miremos el trabajo realizado.

En lo que al DSL se refiere, podríamos seguir extendiendo el lenguaje añadiendo más reglas de negocio, buscando que el DSL fuera completo respecto al dominio del problema, y todas las operaciones con sus respectivas validaciones fueran implementadas desde el DSL, y no dejadas a la voluntad del desarrollador.

Respecto a la herramienta al completo, tras el desarrollo actual aún es necesario invocar a la versión anterior de Service Builder para, a partir del descriptor XML generado con esta herramienta, realizar la generación de código. Una mejora significativa sería la inclusión de esta herramienta como librería en Service Builder de modo que no fuera necesario el XML intermedio, y simplemente se utilizara el conjunto de operaciones definidas por el DSL para generar el código de manera directa.

Otro objetivo fundamental de los trabajos subsiguientes sería el integrar la herramienta en el proyecto Open Source Liferay Portal [59], de manera que se convirtiera en la herramienta de referencia para la construcción de aplicaciones basadas en Liferay Portal, tanto para la comunidad de usuarios (empresas y desarrolladores), como para los propios desarrolladores del proyecto.

8. Referencias y lecturas recomendadas

[1] Liferay Portal https://www.liferay.com

[2] IntelliJ Idea https://www.jetbrains.com/idea/

[3] Gradle https://gradle.org

[4] jUnit http://junit.org/junit4

[5] Travis CI https://travis-ci.org

[6] Codecov.io https://codecov.io

[7] DSL https://es.wikipedia.org/wiki/Lenguaje_de_dominio_espec%C3%ADfico

[8] Ruby https://www.ruby-lang.org/es

[9] Javascript https://developer.mozilla.org/es/docs/Web/JavaScript

[10] Spring Boot https://projects.spring.io/spring-boot

[11] Angular https://angular.io

[12] A. Maslow, ley del instrumento https://en.wikipedia.org/wiki/Law_of_the_instrument

[13] Cyclomatic Complexity http://www.chambers.com.au/glossary/mc_cabe_cyclomatic_complexity.php

[14] PMD CPD (Copy Paste Detector) http://pmd.sourceforge.net/pmd-4.3.0/cpd.html

[15] Java code conventions http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

[16] .NET code conventions https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/inside-a-program/coding-conventions

[17] Findbugs http://findbugs.sourceforge.net

[18] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, 1977.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[20] Happy path https://en.wikipedia.org/wiki/Happy_path

[21] Peer Reviews https://en.wikipedia.org/wiki/Peer_review

[22] Jenkins https://jenkins.io

[23] Atlassian Bamboo https://es.atlassian.com/software/bamboo

[24] ThoughtWorks GoCD https://www.gocd.org/

[25] Debasish, G. 2011 DSLs in Action, Manning Publications

[26] Builder Pattern https://mdelapenya.github.io/java/patrones-diseno-jvm/#builder

[27] Portlet 1.0 JSR-168 https://jcp.org/en/jsr/detail?id=168

[28] Portlet 2.0 JSR-286 https://jcp.org/en/jsr/detail?id=286

[29] ORM https://es.wikipedia.org/wiki/Mapeo_objeto-relacional

[30] OSGi https://www.osgi.org

[31] MySQL https://www.mysql.com

[32] MariaDB https://mariadb.org

[33] PostgreSQL https://www.postgresql.org

[34] SQL Server https://www.microsoft.com/es-es/sql-server/sql-server-2016

[35] Oracle https://www.oracle.com/es/database/index.html

[36] DB2 https://www.ibm.com/analytics/es/es/technology/db2/

[37] Sybase https://www.sap.com/products/sybase-ase.html

[38] Hibernate http://hibernate.org

[39] JVM Specification https://docs.oracle.com/javase/specs/jvms/se8/html/index.html

[40] SOAP https://es.wikipedia.org/wiki/Simple_Object_Access_Protocol

[41] JSON http://www.json.org/json-es.html

[42] DTD https://es.wikipedia.org/wiki/Definici%C3%B3n_de_tipo_de_documento

[43] Spring https://spring.io

[44] UUID https://es.wikipedia.org/wiki/Identificador_%C3%BAnico_universal

[45] Binay Large Object (Blob) https://es.wikipedia.org/wiki/Binary_large_object

[46] Inicialización Lazy https://msdn.microsoft.com/es-es/library/dd997286(v=vs.110).aspx

[47] Fluent API https://en.wikipedia.org/wiki/Fluent_interface

[48] XMLSchema https://es.wikipedia.org/wiki/XML_Schema

[49] Repositorio del proyecto Open Source en Github https://github.com/mdelapenya/liferay-service-builder-dsl

[50] GPL v3 https://github.com/mdelapenya/liferay-service-builder-dsl/blob/master/LICENSE.md

[51] Gradle Wrapper https://docs.gradle.org/current/userguide/gradle_wrapper.html

[52] Descriptor .travis.yml https://docs.travis-ci.com/user/customizing-the-build

[53] Integración continua del proyecto en Travis CI https://travis-ci.org/mdelapenya/liferay-service-builder-dsl

[54] Ejemplo de ejecución fallida en Travis CI https://travis-ci.org/mdelapenya/liferay-service-builder-dsl/builds/269471329

[55] Ejemplo de ejecución con éxito en Travis CI https://travis-ci.org/mdelapenya/liferay-service-builder-dsl/builds/269474487

[56] JaCoCo http://www.eclemma.org/jacoco

[57] Documentación de Codecov.io para enviar los datos de cobertura de código. https://github.com/codecov/example-java

[58] Cobertura de código del proyecto en Codecov https://codecov.io/gh/mdelapenya/liferay-service-builder-dsl

[59] Liferay Portal en Github https://github.com/liferay/liferay-portal

[60] Descriptor XML de Service Builder https://github.com/liferay/liferay-portal/blob/master/portal-impl/src/com/liferay/portlet/documentlibrary/service.xml

[61] DTD de Liferay 7 http://www.liferay.com/dtd/liferay-service-builder_7_0_0.dtd

[62] Principios SOLID https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

[63] Ley de Demeter https://en.wikipedia.org/wiki/Law_of_Demeter