

Clasificación de Refactorizaciones

Dynamic Refactoring Plugin 3.0

Universidad de Burgos
ESCUELA POLITÉCNICA SUPERIOR
INGENIERÍA INFORMÁTICA



SISTEMAS INFORMÁTICOS

Clasificación de Refactorizaciones
Dynamic Refactoring Plugin 3.0

Alumnos:
Míryam Gómez San Martín
Íñigo Mediavilla Saiz

Tutor:
Raúl Marticorena Sánchez

Burgos, Julio de 2011

AGRADECIMIENTOS

Nos gustaría aprovechar estas líneas para dar las gracias a todas las personas que nos han ayudado a lo largo del proyecto, de una u otra manera.

En primer lugar, nos gustaría agradecer a nuestro tutor del proyecto Raúl Marticorena Sánchez, por su tiempo, colaboración y ayuda. Sus conocimientos han facilitado en gran medida la realización del proyecto. Su dedicación y su motivación han sido un impulso que ha llevado al proyecto a cotas de otra manera inalcanzables.

A todos los profesores que nos han ayudado a resolver dudas acerca de diversos aspectos del proyecto, en especial a Lourdes Saiz Bárcena por las soluciones y consejos que nos ha aportado.

A todos los profesores que han estado presentes en las distintas asignaturas de estos años de carrera y que han servido para aportar los conocimientos básicos para la realización del proyecto y que servirán en un futuro para un buen desempeño en el mundo laboral.

Sobre todo a familiares, amigos y a nuestros padres por apoyarnos y comprendernos tanto en los buenos momentos como en los malos. Por soportar la falta de atención recibida en este período tan importante para nosotros. Por proporcionarnos esos momentos de abstracción y de liberación del trabajo tan necesarios.

Este proyecto es en gran parte vuestro, GRACIAS.

CONVENCIONES TIPOGRÁFICAS

A continuación se indican las convenciones tipográficas utilizadas a lo largo de la presente memoria:

Fuente	Qué representa	Ejemplo/s
<i>Italic</i>	Referencia a títulos de manuales o documentos.	Véase <i>Anexo 3 – Especificación de Diseño</i> .
	Permite dar énfasis.	<i>Se deben seguir los siguientes pasos.</i>
	Representa conceptos complejos, los cuales son detallados en el glosario.	<i>Bad smell</i> <i>JAR</i> <i>OSGI</i>
	Referencia a componentes gráficos; como son: nombres de ventanas, diálogos, botones, etc La selección de elementos de menús se indica mediante >	Diríjase a <i>Mi PC>Propiedades</i> , o lo que es lo mismo, escoja <i>Propiedades</i> del menú contextual que aparece al pulsar el botón derecho de su ratón sobre el ícono <i>Mi PC</i> .
Computer	Referencia a ficheros o directorios.	<code>javamoon-2.6.1.jar</code> <code>dynamicrefactoring.plugin</code>
	Referencia a nombres de clases o métodos o variables.	<code>run()</code> <code>M2_HOME</code>
	Referencia a comandos.	<code>install</code>
Herramientas y Marcas	Utilizado para referirse a herramientas software, productos, lenguajes de programación, etc	<code>Apache Maven</code> <code>Github</code> <code>Microsoft</code> <code>Elipse</code> <code>Java</code>
Ventana Comandos	Referencia a fragmentos de código.	<code>mvn install</code>

ÍNDICE GENERAL DEL PROYECTO

Parte I

Descripción del Proyecto.....	3
Lista de cambios.....	9
1. INTRODUCCIÓN.....	11
2. OBJETIVOS DEL PROYECTO.....	13
2.1. Requisitos del software.....	13
2.2. Objetivos técnicos.....	14
3. CONCEPTOS TEÓRICOS.....	17
3.1. Modelo conceptual MOON.....	17
3.2. Refactorización de código.....	18
3.3. Programación orientada a objetos.....	20
3.4. Patrones de diseño.....	20
3.5. Metodología de desarrollo.....	22
3.6. Integración continua.....	24
4. TÉCNICAS Y HERRAMIENTAS.....	28
4.1. Automatización de la construcción - Maven.....	28
4.2. Control de versiones - Git.....	31
4.3. Gestión de tareas - Fogbugz.....	45
4.4. Prototipado de interfaces - Balsamiq Mockup.....	48
4.5. Pruebas de interfaz - SWTBot.....	50
4.6. Integración continua - Hudson.....	50
4.7. Control de calidad del código - Sonar.....	52
5. ASPECTOS RELEVANTES DEL PROYECTO.....	54
5.1. Ciclo de vida.....	54
5.2. Proceso de integración continua.....	55
5.3. Repositorio único de refactorizaciones y clasificaciones.....	60
5.4. Versionado de refactorizaciones.....	63
5.5. Carga de clases.....	65
5.6. Buscador del asistente de refactorizaciones.....	66
5.7. Bug encontrado en Eclipse Forms.....	67
6. TRABAJOS RELACIONADOS.....	68
6.1. Mejoras incorporadas.....	68
6.2. Desaparición de alternativas	72
6.3. JDeodorant.....	73
6.4. Refactory.....	74
6.5. RefactoringNG.....	76
6.6. API de refactorizaciones de Eclipse.....	83
7. CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURO.....	87
7.1. Conclusiones.....	87
7.2. Líneas de trabajo futuro.....	90

Anexo I	
Plan del Proyecto Software.....	95
Lista de cambios.....	101
1. INTRODUCCIÓN.....	103
2. GESTIÓN DE RIESGOS.....	104
2.1. Clasificación de la gestión de riesgos.....	104
2.2. Elementos de la gestión de riesgos.....	105
2.3. Estimación de riesgos asociados al proyecto.....	108
2.4. Control de riesgos asociados al proyecto.....	111
3. PLANIFICACIÓN TEMPORAL.....	113
3.1. Planificación inicial del proyecto.....	113
3.2. Planificación final del proyecto.....	119
3.3. Desviaciones en la planificación inicial.....	123
3.4. Descripción de las tareas.....	124
4. ESTUDIO DE VIABILIDAD.....	130
4.1. Viabilidad técnica.....	131
4.2. Viabilidad legal.....	131
4.3. Viabilidad económica.....	133
4.4. Viabilidad temporal.....	144
Anexo II	
Especificación de Requisitos.....	151
Lista de cambios.....	157
1. INTRODUCCIÓN.....	159
2. OBJETIVOS DEL PROYECTO.....	160
2.1. Lista de Cambios.....	163
3. LISTA DE USUARIOS PARTICIPANTES.....	164
4. CATÁLOGO DE REQUISITOS DEL SISTEMA.....	164
4.1. Actores.....	164
4.2. Requisitos funcionales.....	165
4.3. Requisitos no funcionales.....	170
4.4. Requisitos de información.....	173
5. ESPECIFICACIÓN DE REQUISITOS.....	175
5.1. Diagramas de casos de uso.....	175
5.2. Plantillas de casos de uso.....	180
6. INTERFAZ DE USUARIO.....	202
6.1. Vista del catálogo de refactorizaciones.....	202
6.2. Primera página del asistente de refactorizaciones.....	204
6.3. Editor de clasificaciones.....	205
6.4. Página del asistente de entradas, predicados y acciones	207
Anexo III	
Especificación del Diseño.....	213
Lista de cambios.....	221
1. INTRODUCCIÓN.....	223
2. DISEÑO DE DATOS.....	224

2.1. DTD de refactorizaciones.....	224
2.2. DTD de clasificaciones.....	227
3. DISEÑO ARQUITECTÓNICO.....	230
3.1. Arquitectura lógica.....	231
3.2. Diagramas de clases.....	232
4. DISEÑO DE LA INTERFAZ.....	245
4.1. Asistente para creación y edición de refactorizaciones.....	245
4.2. Interfaz para la exportación de refactorizaciones.....	255
4.3. Vista Available Refactorings.....	257
4.4. Vista Refactoring Catalog Browser.....	259
4.5. Editor de Clasificaciones.....	260
5. DISEÑO PROCEDIMENTAL.....	262
5.1. Diagramas de secuencia.....	262
6. REPRESENTACIÓN CTTE DE DYNAMIC REFACTORING.....	280
6.1. Modificaciones en la primera página del wizard.....	280
6.2. Visualizar catálogo de Refactorizaciones.....	281
6.3. Editar clasificaciones.....	282
6.4. Añadir una nueva clasificación.....	283
6.5. Eliminar una clasificación.....	283
7. PATRONES DE DISEÑO.....	284
7.1. Patrón Objeto Constructor - Builder.....	284
7.2. Patrón Fachada.....	286
7.3. Patrón Singleton.....	288
7.4. Patrón Comando	289
8. REFERENCIA CRUZADA CON LOS REQUISITOS.....	292
9. DISEÑO DE PRUEBAS.....	294
9.1. RF 1: Visualizar refactorizaciones según clasificación	294
9.2. RF 2: Refrescar visualización de refactorizaciones	295
9.3. RF 3: Añadir filtro de refactorizaciones	296
9.4. RF 4: Seleccionar opción aplicar filtro	297
9.5. RF 5: Eliminar filtro de refactorizaciones	297
9.6. RF 6: Eliminar todos los filtros de refactorizaciones	298
9.7. RF 7: Seleccionar opción ver refactorizaciones filtradas	298
9.8. RF 8: Visualizar detalle refactorización	299
9.9. RF 9: Añadir clasificación	299
9.10. RF 10: Editar clasificación	300
9.11. RF 11: Eliminar clasificación	301
9.12. RF 12: Añadir categoría a una clasificación	301
9.13. RF 13: Renombrar categoría de una clasificación	302
9.14. RF 14: Eliminar categoría de una clasificación.....	302
9.15. RF 15: Mostrar resumen elemento seleccionado	303
9.16. RF 16: Realizar búsqueda de elementos	303
10. SELECCIÓN DE MÉTRICAS.....	304
11. ENTORNO TECNOLÓGICO DE LA APLICACIÓN.....	308
12. PLAN DE DESARROLLO E IMPLANTACIÓN.....	308

<u>12.1. Diagrama de componentes.....</u>	309
<u>12.2. Diagrama de despliegue.....</u>	313
Anexo IV	
Documentación Técnica del Programador.....	317
<u>Lista de cambios.....</u>	323
<u>1. INTRODUCCIÓN.....</u>	325
<u>2. MANUAL DEL PROGRAMADOR.....</u>	325
<u>2.1. Contenido del CD.....</u>	325
<u>2.2. Documentación de bibliotecas.....</u>	328
<u>2.3. Instalación de herramientas.....</u>	344
<u>2.4. Desarrollo plugin Eclipse.....</u>	355
<u>2.5. Maven.....</u>	361
<u>2.6. Integración bibliotecas MOON y JavaMOON.....</u>	369
<u>3. ASIGNAR INFORMACIÓN DE MARCA AL PLUGIN.....</u>	373
<u>4. FIRMA PLUGIN.....</u>	376
<u>4.1. Pasos.....</u>	376
<u>5. PRUEBAS.....</u>	379
<u>5.1. Pruebas Unitarias.....</u>	379
<u>5.2. Pruebas de interfaz.....</u>	383
<u>5.3. Pruebas de Cobertura.....</u>	384
<u>6. MÉTRICAS.....</u>	389
<u>6.1. Evaluación de métricas.....</u>	389
<u>6.2. Resultados obtenidos.....</u>	390
<u>7. INTERNACIONALIZACIÓN.....</u>	401
<u>8. AYUDA EN ECLIPSE.....</u>	402

Anexo V	
Manual de Instalación.....	407
<u>Lista de cambios.....</u>	413
<u>1. INTRODUCCIÓN.....</u>	415
<u>2. REQUISITOS SOFTWARE.....</u>	415
<u>3. REQUISITOS HARDWARE.....</u>	416
<u>4. INSTALACIÓN DE JAVA DEVELOPMENT KIT.....</u>	417
<u>4.1. Proceso de instalación.....</u>	418
<u>4.2. Configuración de variables de entorno.....</u>	423
<u>5. INSTALACIÓN DE ECLIPSE.....</u>	426
<u>6. INSTALACIÓN DE ANT.....</u>	428
<u>6.1. Proceso de instalación.....</u>	429
<u>6.2. Configuración de variables de entorno.....</u>	430
<u>7. INSTALACIÓN DE LA APLICACIÓN.....</u>	431
<u>7.1. Instalación de Dynamic Refactoring Plugin.....</u>	431
<u>7.2. Instalación de la tarea Ant RefactoringPlan.....</u>	445
<u>8. EJECUCIÓN DE LA APLICACIÓN.....</u>	445
<u>8.1. Ejecución de Dynamic Refactoring Plugin.....</u>	445
<u>8.2. Ejecución de la tarea Ant RefactoringPlan.....</u>	448

Anexo VI	
Manual de Usuario.....	455
Lista de cambios.....	463
1. INTRODUCCIÓN.....	465
1.1. Primeros pasos.....	465
2. MENÚ DYNAMIC REFACTORING.....	472
2.1. New Refactoring	473
2.2. Edit Refactoring.....	486
2.3. Delete Refactoring.....	491
2.4. Export Refactoring.....	493
3. VISTA: AVAILABLE REFACTORINGS.....	496
4. VISTA: REFACTORING CATALOG BROWSER.....	499
4.1. Refactoring Catalog Browser Toolbar.....	500
4.2. Classification Panel.....	503
4.3. Summary Panel.....	517
5. EDITOR: CLASSIFICATIONS EDITOR.....	527
5.1. Classifications.....	529
5.2. Selected Classification.....	532
5.3. Categories.....	534
6. CONSULTAR AYUDA.....	538
7. CONFIGURACIÓN ADICIONAL.....	540
7.1. Catálogo de clasificaciones.....	540
7.2. El fichero XML de clasificaciones.....	540
7.3. Exportación de refactorizaciones.....	542
Bibliografía.....	543
Glosario.....	549

Universidad de Burgos
ESCUELA POLITÉCNICA SUPERIOR
INGENIERÍA INFORMÁTICA



SISTEMAS INFORMÁTICOS

Clasificación de Refactorizaciones
Dynamic Refactoring Plugin 3.0

Alumnos:
Míryam Gómez San Martín
Íñigo Mediavilla Saiz

Tutor:
Raúl Marticorena Sánchez

Burgos, Julio de 2011

PARTE I
DESCRIPCIÓN DEL PROYECTO

ÍNDICE DE CONTENIDO

<i>Parte I</i>	
<i>Descripción del Proyecto.....</i>	3
<i>Lista de cambios.....</i>	9
<i>1. INTRODUCCIÓN.....</i>	11
<i>2. OBJETIVOS DEL PROYECTO.....</i>	13
<i> 2.1. Requisitos del software.....</i>	13
<i> 2.2. Objetivos técnicos.....</i>	14
<i> 2.2.1. Mejora del proceso de desarrollo.....</i>	15
<i> 2.2.2. Mejora de la calidad del código.....</i>	15
<i>3. CONCEPTOS TEÓRICOS.....</i>	17
<i> 3.1. Modelo conceptual MOON.....</i>	17
<i> 3.2. Refactorización de código.....</i>	18
<i> 3.2.1. Pasos al aplicar una refactorización.....</i>	18
<i> 3.2.2. Construcción dinámica de refactorizaciones.....</i>	19
<i> 3.3. Programación orientada a objetos.....</i>	20
<i> 3.4. Patrones de diseño.....</i>	20
<i> 3.5. Metodología de desarrollo.....</i>	22
<i> 3.6. Integración continua.....</i>	24
<i> 3.6.1. Teoría.....</i>	24
<i> 3.6.2. Prácticas recomendadas.....</i>	25
<i>4. TÉCNICAS Y HERRAMIENTAS.....</i>	28
<i> 4.1. Automatización de la construcción - Maven.....</i>	28
<i> 4.1.1. POM: Project Object Model.....</i>	28
<i> 4.1.2. Plugins.....</i>	29
<i> 4.1.3. Ciclos de vida de la fase de construcción.....</i>	30
<i> 4.1.4. Dependencias.....</i>	31
<i> 4.2. Control de versiones - Git.....</i>	31
<i> 4.2.1. Ventajas del control de versiones.....</i>	31
<i> 4.2.2. Introducción y breve historia de Git.....</i>	32
<i> 4.2.3. Características de Git.....</i>	33
<i> 4.2.4. El modelo de objetos de Git.....</i>	34
<i> 4.2.5. Elección de Git como sistema de control de versiones.....</i>	42
<i> 4.3. Gestión de tareas - Fogbugz.....</i>	45
<i> 4.3.1. Ventajas de la gestión de tareas.....</i>	45
<i> 4.3.2. FogBugz.....</i>	46
<i> 4.3.3. Foglyn.....</i>	48
<i> 4.4. Prototipado de interfaces - Balsamiq Mockup.....</i>	48
<i> 4.5. Pruebas de interfaz - SWTBot.....</i>	50
<i> 4.6. Integración continua - Hudson.....</i>	50

<u>4.7. Control de calidad del código - Sonar</u>	52
5. ASPECTOS RELEVANTES DEL PROYECTO.....	54
<u>5.1. Ciclo de vida</u>	54
<u>5.2. Proceso de integración continua</u>	55
<u>5.2.1. Tareas incorporadas al proceso de construcción</u>	57
<u>5.2.2. Configuración avanzada</u>	58
<u>5.3. Repositorio único de refactorizaciones y clasificaciones</u>	60
<u>5.3.1. Solución adoptada</u>	62
<u>5.4. Versionado de refactorizaciones</u>	63
<u>5.5. Carga de clases</u>	65
<u>5.6. Buscador del asistente de refactorizaciones</u>	66
<u>5.7. Bug encontrado en Eclipse Forms</u>	67
6. TRABAJOS RELACIONADOS.....	68
<u>6.1. Mejoras incorporadas</u>	68
<u>6.2. Desaparición de alternativas</u>	72
<u>6.3. JDeodorant</u>	73
<u>6.3.1. Ventajas</u>	73
<u>6.3.2. Desventajas</u>	74
<u>6.3.3. Conclusión</u>	74
<u>6.4. Refactory</u>	74
<u>6.4.1. Ventajas</u>	75
<u>6.4.2. Desventajas</u>	76
<u>6.5. RefactoringNG</u>	76
<u>6.5.1. Árbol, atributos y contenido</u>	77
<u>6.5.2. Listas</u>	80
<u>6.5.3. NoneOf</u>	81
<u>6.5.4. Conclusiones</u>	82
<u>6.6. API de refactorizaciones de Eclipse</u>	83
<u>6.6.1. Pasos del proceso de refactorización</u>	83
<u>6.6.2. Modificaciones necesarias más importantes</u>	84
<u>6.6.3. Conclusiones</u>	85
7. CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURO.....	87
<u>7.1. Conclusiones</u>	87
<u>7.1.1. Requisitos funcionales</u>	87
<u>7.1.2. Objetivos técnicos</u>	88
<u>7.2. Líneas de trabajo futuro</u>	90

ÍNDICE DE ILUSTRACIONES

Ilustración 1: Representación de un objeto blob.....	36
Ilustración 2: Representación de un objeto árbol.....	37
Ilustración 3: Representación de un objeto commit.....	38
Ilustración 4: Modelo completo de objetos de Git.....	40
Ilustración 5: Representación de un objeto tag.....	41
Ilustración 6: Prototipo creado con Balsamiq Mockup.....	50
Ilustración 7: Pantalla de entrada con la lista de proyectos de Hudson.....	52
Ilustración 8: Pantalla principal de Sonar.....	53
Ilustración 9: Ciclo de vida utilizado en el proyecto.....	54
Ilustración 10: Refactory aplicando una refactorización a un metamodelo.....	75
Ilustración 11: Previsualización de resultados con RefactoringNG.....	82

ÍNDICE DE TABLAS

Tabla 1: Clasificación de patrones de diseño.....	22
---	----

LISTA DE CAMBIOS

Número	Fecha	Descripción	Autor/es
0	24/01/11	Creadas secciones de Técnicas y Herramientas.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
1	07/02/11	Correcciones de formato y contenido a la versión anterior y agregada comparativa de Git con otros SCV. Además se agrega la comparación con otro software alternativo.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
2	22/02/11	Añadidos RefactoringNG y API de Eclipse al apartado referente a trabajos relacionados.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
3	28/02/11	Se empiezan a escribir los objetivos del proyecto.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
4	11/03/11	Sustituido modelo de citas.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
5	08/04/11	Comienzo de la redacción de las líneas de trabajo futuro y los aspectos relevantes del desarrollo.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
6	25/04/11	Ampliación del apartado de aspectos relevantes del desarrollo y adaptación al nuevo estilo de plantilla.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
7	31/05/11	Agregadas nuevas herramientas, sección de mejoras agregadas al proyecto y ampliados los aspectos relevantes.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
8	5/06/11	Versión antes de lectura para corrección de defectos.	Míryam Gómez San Martín Íñigo Mediavilla Saiz

1. INTRODUCCIÓN

La presente memoria tiene como objetivo fundamental la especificación y desarrollo de las diferentes etapas por las que ha ido evolucionando la realización del proyecto, el cual consiste en extender la funcionalidad de un plugin para Eclipse que permite la construcción y ejecución de refactorizaciones dinámicas y que ha sido desarrollado previamente en dos proyectos anteriores.

Se trabaja sobre un modelo de objetos que soporta los conceptos comunes a todos los lenguajes orientados a objetos pero que aporta la suficiente flexibilidad como para poder adaptarlo a cada lenguaje en particular, en este caso Java, extendiendo las clases del modelo mediante herencia para añadir la nueva funcionalidad e información requerida por cada lenguaje. Esta particularidad es la que permite que las refactorizaciones que utilizan únicamente clases de este modelo, sean independientes del lenguaje. Este modelo ha sido creado en trabajos anteriores y recibe el nombre de modelo MOON, la correspondiente extensión para lenguaje Java es el denominado JavaMOON. Toda la información se almacena como instancias de un grafo. El grafo puede ser recorrido para comprobar el estado inicial y realizar las transformaciones sobre dichas instancias, transformando el estado del código. Al recorrer de nuevo el grafo, el código refactorizado puede ser regenerado.

Al comenzar este proyecto se partía de un plugin que ya ofrecía la funcionalidad por la que había sido creado, es decir, ya permitía definir un conjunto de refactorizaciones de forma dinámica. A este en una segunda versión del plugin se habían incorporado numerosas funcionalidades consiguiendo como resultado una herramienta mucho más completa, por tanto teníamos la responsabilidad de tratar de mejorar un producto ya de gran valor.

Se podía haber optado por continuar el proyecto por diversas vías diferentes, pero finalmente se decidió hacerlo principalmente por tres, que son: aumentar las posibilidades de aplicación de las propias refactorizaciones, facilitar el uso del asistente de creación de refactorizaciones y sobre todo ayudar al usuario en el conocimiento del amplio catálogo de refactorizaciones. Por supuesto, como en todo proyecto que es una continuación o evolución de previos los cambios introducidos no sólo se han limitado a las nuevas características implementadas, sino que también ha supuesto numerosas mejoras necesarias en el diseño y corrección de defectos de las versiones previas.

En esta tercera versión del proyecto se ha hecho especial hincapié en la calidad del código desarrollado. Este fue uno de los objetivos que se marcó al arrancar el proyecto. Este

objetivo se estableció al tener en cuenta el hecho de que el proyecto alcanzaba su tercera versión y el tamaño del mismo empezaba a ser considerable. La meta que se marcó fue la de incrementar la calidad del código para poder así favorecer su mantenimiento. Para cumplir con esta meta los esfuerzos se han centrado fundamentalmente en el control del diseño del código y la incorporación de pruebas.

Otro aspecto en el que se ha querido mejorar el proyecto ha sido en su proceso de construcción. En cualquier otro proyecto la construcción de un producto software que se pueda entregar al cliente a partir de los fuentes es un proceso costoso, formado por muchos pequeños pasos repetitivos que requieren mucho tiempo de los desarrolladores. Cuando el proyecto se trata de un plugin de Eclipse el número de pasos es especialmente elevado. Al conseguir integrar todo el proceso de construcción de los entregables bajo un sólo comando se ha conseguido reducir el tiempo que los desarrolladores tienen que dedicar a estas tareas repetitivas así como incorporar el control de la calidad del producto en dicho proceso.

Llegado ya el final del proyecto, los que hemos sido partícipes del mismo estamos satisfechos del esfuerzo y del trabajo dedicados. Nuestra esperanza es que aquellos que tengan la posibilidad de probar el resultado, puedan valorar el producto con la misma impresión positiva con que nosotros lo hicimos cuando probamos la versión anterior; que este trabajo realizado haya sumado valor, en la mayor medida de lo posible, a lo aportado por nuestros predecesores.

2. OBJETIVOS DEL PROYECTO

A continuación se detallarán, a modo de introducción, los objetivos fundamentales del proyecto, tanto los relacionados con el funcionamiento de la aplicación como los de carácter más técnico.

2.1. Requisitos del software

Otros proyectos de plugin de refactorización han caído en desuso por la supremacía de las refactorizaciones proporcionadas por los propios *IDEs*, ejemplo de estos casos son: ContraCT y RefactorIt. Una mirada al Eclipse Market Place [Eclipse Market Place n.d.] muestra que los plugins de refactorización que tienen más éxito son los que proporcionan valores añadidos. Algunos de ellos han sido objeto de estudio en la sección de trabajos relacionados, de la presente memoria, destinada a tal fin.

La versión anterior del proyecto dispone de ese factor diferenciador de valor añadido, muestra de ello son las siguientes características con las que cuenta:

- Planes de refactorizaciones.
- Importación/Exportación de refactorizaciones y planes de las mismas.
- Refactorizaciones adicionales de interés: JUnit3 a JUnit4, Genéricas, etc.
- Posibilidad de definir refactorizaciones para otros lenguajes.
- Historial de refactorizaciones realizadas, posibilidad de deshacer.

Pero en ocasiones esto suponía una complejidad para el usuario que encontraba dificultad para poder comprender el modelo de JavaMOON con las entradas, precondiciones, postcondiciones y acciones a la hora de crear una refactorización. En otras ocasiones al usuario le resultaba difícil encontrar la refactorización que necesitaba debido a que la opción disponible para mostrar refactorizaciones en base al contexto no es suficiente. Era necesaria una respuesta para hacer al usuario partícipe de estas interesantes posibilidades que el plugin ofrece y además debía ser una solución que se presentara de forma natural para el usuario.

Por tanto se consideró que la solución que se aportara debería contar con:

- Una ayuda en forma de catálogo con toda la información disponible sobre las refactorizaciones, pero clasificada en base a criterios lógicos.
- La posibilidad en el estudio del catálogo de personalizar la forma en que las refactorizaciones aparecen organizadas. Para la personalización del catálogo se debería permitir al usuario escoger su clasificación preferida.
- La opción de definir clasificaciones personales con sus correspondientes categorías y asignar estas a las refactorizaciones dentro de cada clasificación.
- El catálogo también debería permitir hacer búsquedas de refactorizaciones en base a criterios, pudiendo tratarse de filtros por nombre, por categoría o por palabra clave. Los criterios se aplicarían como filtros en el catálogo que se pudieran acumular formando reglas compuestas.

Dado que la definición de refactorizaciones es uno de los aspectos más complejos de utilizar de entre toda la funcionalidad con la que el plugin cuenta, se decidió que había que implementar una serie de mejoras al asistente para facilitar su uso:

- Un aspecto que se debería mejorar en el asistente, era la información que se mostraba de las entradas, precondiciones, acciones y postcondiciones.
- Se debía mejorar el buscador de cada tipo de elemento debido al elevado número de elementos disponibles, ya que la búsqueda existente no otorgaba la flexibilidad esperada.

Otra mejora debería venir a la hora de facilitar la instalación del plugin al usuario:

- Se debía mejorar el sistema de copia manual de ficheros por un sistema automatizado que utilizara las ventajas del gestor de instalaciones de Eclipse y permitiera actualizaciones automáticas.

2.2. *Objetivos técnicos*

Los objetivos no funcionales del proyecto del plugin de refactorización se han dividido en dos: la mejora del proceso de desarrollo del plugin y el incremento de la calidad del código del producto.

2.2.1. Mejora del proceso de desarrollo

El proceso de construcción de un plugin de Eclipse es un proceso complejo. Como parte de la estructura de Eclipse el plugin tiene que definirse como un paquete de *OSGI* y tiene que tener definida una plataforma objetivo desde la que el plugin puede obtener sus dependencias. En definitiva, es un proceso complejo que aparece explicado de una forma mucho más extensa en el *Anexo 4 - Documentación Técnica del Programador* y que en versiones previas del plugin se venía realizando de forma manual o utilizando asistentes proporcionados por el propio Eclipse, pero que necesitaba introducir mejoras para facilitar el desarrollo a los programadores.

Las mejoras introducidas en el proceso de desarrollo deberían aportar las siguientes ventajas:

- Desarrollo basado en un modelo colaborativo soportado por un sistema de control de versiones y un control de tareas y bugs.
- Generación de todos los artefactos necesarios con un sólo comando: ficheros *JAR*, características, repositorios *P2* y firmado de los *JAR*.
- Ejecución de los tests como parte del proceso de construcción del producto, si los test no pasan el resto de artefactos no es generado. De este modo las pruebas se convierten en un aspecto central del desarrollo.
- Generación de informes sobre ejecución, cobertura de los tests y métricas del código como parte del proceso de construcción.
- Ejecución del proceso de construcción completo de forma periódica sobre la última versión del proyecto disponible, la cual se encuentra almacenada en el repositorio de control de versiones.

2.2.2. Mejora de la calidad del código

La calidad del código es un aspecto muy importante de todo proyecto software, pero lo es más a medida que el tamaño del proyecto va aumentando y se van incorporando nuevas versiones del producto. Este proyecto en concreto es una tercera versión, lo que da una idea de la complejidad del proyecto debido a su tamaño y al hecho de que haya sido desarrollado por tres grupos de desarrolladores distintos.

Como objetivo importante de este proyecto se decidió incrementar la calidad del código del proyecto con idea de facilitar el propio desarrollo y mantenimiento y el de posibles futuras versiones del producto. Se tenía además la ventaja de poder aprovechar los objetivos definidos para la mejora del proceso de desarrollo. Se decidió por tanto que se debía mejorar la calidad a través de:

- Inclusión de test gráficos.
- Un incremento del número de tests y de la cobertura del código.
- Una mejora en la documentación del código.
- Mejora generalizada en las métricas especialmente en las de complejidad de los módulos y en las de números de defectos del código.

3. CONCEPTOS TEÓRICOS

En este apartado quedan recogidos aquellos conceptos teóricos, relacionados con las distintas partes del proyecto, que han sido considerados necesarios para la comprensión del presente proyecto.

3.1. *Modelo conceptual MOON*

Los componentes clave que deben poseer las herramientas capaces de asistir el proceso de ejecución de refactorizaciones son: un modelo donde se acumule una base del conocimiento sobre el sistema software candidato a ser refactorizado, y un motor de refactorizaciones para la ejecución de las mismas.

Este modelo debe establecer la estructura e identificar los elementos del sistema software de los que se van a calcular las métricas. En el caso de un sistema software orientado a objetos estos elementos se corresponden con: clases, métodos, atributos, etc.

En este sentido, en trabajos realizados previamente se ha propuesto el lenguaje *modelo minimal MOON*. Se puede entender la idea de *modelo minimal* como un metamodelo que permita obtener un conjunto de instancias que representen un sistema software implementado en algún lenguaje orientado a objetos estáticamente tipados. Así, por ejemplo, cada uno de los métodos de una determinada clase pasará a ser una instancia de una metaclass que representa métodos, y cuyos atributos podrían ser el nombre del método, la lista de parámetros, etc. También se intenta que MOON sea lo más general posible para dar cabida a una amplia familia de lenguajes.

Para extender la aplicación a otros lenguajes se requiere obtener una herramienta que parsee el código fuente original o el código binario para obtener el *modelo minimal*. Este *parser* se tendrá que adecuar a las diferentes características del lenguaje de programación a parsear. MOON, se trata de un lenguaje extensible, lo que permite adaptarlo a las características particulares de cada uno de los lenguajes orientados a objetos.

Se puede obtener una descripción más detallada de la definición del metalenguaje MOON en [Crespo 2000] y [Crespo, López, & Marticorena n.d.] .

3.2. Refactorización de código

Según define Martin Fowler, la refactorización de código es una técnica para la reestructuración de un fragmento de código, modificando su estructura interna sin cambiar su comportamiento externo.

Debido a que el código generado por los desarrolladores nunca es perfecto, es necesaria esta técnica de reestructuración, que permitirá obtener código mejor diseñado, que pueda entenderse sin dificultad y, por tanto, que sea más fácil de depurar y modificar. También puede ocurrir que ya se disponga de un código óptimo, pero que sea necesaria su modificación para añadir nuevos requerimientos al sistema.

La técnica de la refactorización de código se basa en la aplicación de una serie de pequeñas transformaciones que no varían el comportamiento del código al ser aplicadas. Se parte del código existente y de su diseño subyacente. Cada transformación se aplica sucesivamente junto a otras para obtener una secuencia de transformaciones, que acaban produciendo una reestructuración sobre el código. El resultado es la mejora del código y su diseño. Este proceso de refactorización debe ser continuo y paralelo al propio desarrollo software.

Es necesario que tras aplicar cada refactorización el sistema siga funcionando correctamente. Para verificarlo se necesita una batería de pruebas adecuada al sistema con la cual comprobar el correcto funcionamiento del código, es decir, que su comportamiento no ha sido alterado. En la práctica las refactorizaciones ayudan a:

- Hacer el software más fácil de entender.
- Estructurar el código de forma que se minimice el coste de los cambios.
- Programar de una manera más rápida y eficaz, ayudando a encontrar errores.
- Recuperar la estructura del código tras haber efectuado una serie de cambios acumulativos sobre el mismo.

3.2.1. Pasos al aplicar una refactorización

Toda refactorización ya sea realizada de forma manual o automática mediante alguna aplicación, se debería llevar a cabo a través de una serie de etapas básicas que se detallan a continuación:

1. Detectar el punto del código en que puede ser necesario llevar a cabo una refactorización. Debido a que no existe una manera clara que permita identificar dónde es necesario efectuar una refactorización, serán indicativas algunas situaciones que se nos pueden presentar, como pueden ser: código duplicado, listas de parámetros demasiado largas, abuso de sentencias condicionales, etc.
2. A continuación se debe buscar la refactorización adecuada que podría resolverlo, para lo que el programador puede valerse de los catálogos de refactorizaciones que los distintos autores han definido.
3. Asegurarse de que se pueden llevar a cabo de forma correcta los cambios necesarios en el sistema software, definiendo una serie de comprobaciones que puedan verificarse de forma automática en cada paso.
4. Si así se considera oportuno se procederá a la refactorización del código. La manera más recomendable de hacerlo es a través de pequeños pasos, que permitan volver atrás con facilidad si se detectara un error durante el proceso. Lo ideal y lo más seguro es disponer de una batería de pruebas que pueda ejecutarse después de cada cambio para verificar que el sistema sigue funcionando de manera adecuada.
5. El último paso es realizar la comprobación de que los cambios llevados a cabo no han supuesto un cambio en el comportamiento externo.

3.2.2. Construcción dinámica de refactorizaciones

Existen diversos catálogos de refactorizaciones disponibles que han sido creados por diferentes autores, como por ejemplo Martin Fowler uno de los pioneros en el campo de la refactorización de código. Sin embargo, es un campo que está todavía madurando, no existe excesiva documentación y existen pocas herramientas, o bien son mejorables.

Los diseñadores y programadores pueden encontrarse en algún desarrollo con que las refactorizaciones previamente definidas no se adaptan a lo que requieren en ese momento. Por ello, es necesaria alguna herramienta que permita la construcción de nuevas refactorizaciones que se adapten completamente al problema particular.

El objetivo por tanto es proporcionar una técnica que permita realizar un cambio de diseño sin cambiar el comportamiento externo de la aplicación. En ese caso se pueden

seguir los siguientes pasos para la construcción de refactorizaciones:

1. Desglosar el cambio en pequeños pasos, localizando los puntos donde sea necesario compilar y/o probar.
2. Realizar el cambio aplicando pequeños pasos de tal forma que si se produce algún problema se podrá considerar cómo eliminarlo de la refactorización.
3. Realizar el cambio de nuevo, comprobando el resultado y redefiniendo los pasos si fuese necesario.
4. Tras ejecutar la refactorización unas cuantas veces dentro de este proceso de depuración, se obtendrá la definición de la nueva refactorización.

Como ya se ha comentado, el objetivo final es la construcción de refactorizaciones que modifiquen la estructura interna sin cambiar el comportamiento externo del sistema. Por tanto, es fundamental disponer de una buena batería de pruebas con la que poder confirmar el correcto funcionamiento de la aplicación.

3.3. Programación orientada a objetos

En este apartado simplemente se dará una breve explicación de esta técnica, ya que se supone de sobra conocida. De todos modos se recomienda la consulta de [Meyer 1999] a aquellos interesados en ampliar conocimientos.

La Programación Orientada a Objetos es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones y programas de computador. Es una evolución de la programación procedural basada en funciones. Su uso se popularizó a principios de la década de 1990.

Permite agrupar secciones de código con funcionalidades comunes y está basado en varias técnicas, incluyendo herencia, modularidad, polimorfismo, y encapsulamiento. En la actualidad, son muchos los lenguajes de programación que soportan la orientación a objetos.

3.4. Patrones de diseño

Un patrón de diseño describe un problema que ocurre una y otra vez en nuestro entorno, luego se describe el núcleo de la solución a dicho problema de tal forma que se

puede usar esta solución un millón de veces sin hacerlo dos veces de la misma forma [Alexander 1980].

En general podemos decir que un patrón de diseño es una solución general, fruto de la experiencia, a un problema general que puede adaptarse a un problema concreto.

Los elementos que constituyen un patrón de diseño son:

- Nombre: describe un problema de diseño.
- Problema: describe cuándo aplicar el patrón, es decir, el contexto.
- Solución: describe los elementos que conforman el diseño, sus relaciones, sus responsabilidades y colaboraciones. Proporciona una visión abstracta de un problema y cómo se organizan los elementos del mismo para resolverlo.
- Consecuencias: los resultados de aplicar un patrón de diseño. Incluyen el impacto de propiedades del sistema como flexibilidad, portabilidad y extensibilidad.

Los patrones se pueden clasificar según los siguientes criterios de catalogación:

- Propósito del patrón:
 - Patrones creacionales: abstraen el proceso de instanciación de los objetos.
 - Patrones estructurales: expresan cómo las clases y objetos se componen para formar estructuras mayores.
 - Patrones de comportamiento: están relacionados con los algoritmos y con la asignación de responsabilidades.
- Ámbito, indica si el patrón se aplica principalmente a clases o a objetos:
 - Patrones de clases: Indica relaciones entre clases y subclases. Estas relaciones se realizan en tiempo de compilación, son estáticas.
 - Patrones de objetos: Indica relaciones entre objetos. Estas relaciones pueden cambiar en tiempo de ejecución, son dinámicas.

En la siguiente tabla podemos observar la clasificación de los patrones de diseño en función de los criterios anteriormente descritos:

		Propósito		
		Creacional	Estructural	Comportamiento
Alcance	Clase	<i>Factory Method</i>	<i>Adapter(Class)</i>	<i>Interpreter</i> <i>Template Method</i>
	Objeto	<i>Abstract Factory</i> <i>Builder</i> <i>Prototype</i> <i>Singleton</i>	<i>Adapter(Object)</i> <i>Bridge</i> <i>Composite</i> <i>Decorator</i> <i>Facade</i> <i>Flyweight</i> <i>Proxy</i>	<i>Chain of Responsibility</i> <i>Command</i> <i>Iterator</i> <i>Mediator</i> <i>Memento</i> <i>Observer</i> <i>State</i> <i>Strategy</i> <i>Visitor</i>

Tabla 1: Clasificación de patrones de diseño

3.5. Metodología de desarrollo

Durante el desarrollo del proyecto se ha utilizado una metodología de desarrollo ágil con un ciclo de desarrollo iterativo e incremental en el que los requisitos se han ido definiendo a medida que el proyecto ha ido evolucionando.

Los principios fundamentales de las metodologías de desarrollo quedaron definidos en el manifiesto ágil [Agile Manifesto n.d.].

- Individuos e interacciones sobre procesos y herramientas.
- Software funcionando sobre documentación extensiva.
- Colaboración con el cliente sobre negociación contractual.
- Respuesta ante el cambio sobre seguir un plan.

El mismo manifiesto define un conjunto de principios que subyacen a los cuatro anteriores:

1. Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.

2. Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos *Ágiles* aprovechan el cambio para proporcionar ventaja competitiva al cliente.
3. Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
4. Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.
5. Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
6. El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
7. El software funcionando es la medida principal de progreso.
8. Los procesos *Ágiles* promueven el desarrollo sostenible. Los promotores, desarrolladores y los propios usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
9. La atención continua a la excelencia técnica y al buen diseño mejora la *Agilidad*.
10. La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
11. Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.
12. A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

La mayoría de estos principios se han adoptado como parte del desarrollo del proyecto. El desarrollo se ha centrado en la entrega temprana y continua de valor, lo que se ha visto favorecida por la automatización de la construcción del producto. Los requisitos se han ido adaptando a lo largo del desarrollo. Se han mantenido reuniones frecuentes con el tutor que ha hecho la labor de responsable del proyecto. Se ha favorecido la comunicación continua y la colaboración entre los miembros del equipo. Además todo el proceso de

implementación se ha guiado por la búsqueda de la simplicidad y el mejor diseño posible.

Algunos principios que se han utilizado en el proyecto se han extraído de ciertas metodologías concretas que se adscriben al grupo de las metodologías ágiles. Por ejemplo, se ha utilizado *TDD* tal y como *XP* [Beck 2000] promueve, además de otras técnicas, y las tareas a implementar se han priorizado para cada iteración del mismo modo que defiende *SCRUM* [Schwaber 2004].

3.6. Integración continua

La integración continua implementa un proceso continuo de control de la calidad basado en pequeños esfuerzos de control muy frecuentes. El objetivo de la integración continua es aumentar la calidad del software y reducir el tiempo necesario para desarrollarlo reemplazando la práctica tradicional de realizar el control de calidad al finalizar el desarrollo.

La integración continua tiene por uno de sus principios la automatización, todo el proceso en que se basa la integración continua sería imposible sin disponer de una herramienta capaz de automatizar cada uno de los pasos. Por esa razón se hará una introducción teórica a la integración continua, seguido por una breve explicación de los principios más importantes. En la sección dedicada a *Técnicas y Herramientas* se dará una explicación de la herramienta utilizada en el proyecto para la automatización del proceso, esta es Apache Maven.

3.6.1. Teoría

En la mayoría de los proyectos actuales el equipo comparte el código del proyecto en un repositorio común. Cuando alguno de los desarrolladores va a realizar algún cambio en primer lugar obtiene una copia del código desde la que trabajará. A medida que otros desarrolladores suben cambios al repositorio de código la copia del desarrollador inicial va gradualmente perdiendo similitud con el código del repositorio. Si el desarrollador decide subir algunos de los cambios, primero debe actualizar su código con los cambios guardados en el repositorio desde el que se hizo la copia. Cuantos más cambios contenga el repositorio más trabajo tendrá que hacer el desarrollador antes de subir los suyos propios.

Si el desarrollador pospone la integración de su código al repositorio demasiado tiempo, el repositorio puede llegar a acabar siendo tan diferente de las líneas de desarrollo que se entra en lo que se conoce como *integration hell* [Integration Hell n.d.], donde el

tiempo que se tarda en integrar excede el tiempo que se tardó en hacer los cambios originales. En el peor caso los desarrolladores tienen que descartar completamente sus cambios y rehacer el trabajo.

La integración continua supone integrar con la suficiente frecuencia para evitar las desventajas de una integración caótica. La práctica pretende reducir el trabajo duplicado y por tanto reducir costes y tiempos.

3.6.2. Prácticas recomendadas

A continuación se expondrán una serie de buenas prácticas recomendadas para la implantación de un proceso basado en la integración continua.

Mantener un repositorio de código

Esta práctica defiende el uso de un sistema de control de versiones para el código del proyecto. Todos los artefactos necesarios para construir el proyecto deben ser accesibles desde el repositorio. Según esta práctica la convención es que el sistema deberá poder ser construido a partir del contenido del repositorio sin requerir dependencias adicionales.

Respecto a las recomendaciones de uso del repositorio de código existen posturas enfrentadas. Los defensores del *eXtreme Programming* [Beck 2000] abogan por rechazar el uso de las ramas y defienden que todos los cambios deben ser integrados a la rama principal y rechazan la creación de múltiples versiones del software mantenidas de forma simultánea. Sin embargo este principio es muy discutido por quienes defienden un uso razonable de las ramas [Código Software n.d.].

Automatizar el proceso de construcción

El sistema completo debería de poder construirse a partir de un solo comando. La mecanización del proceso debe incluir la automatización de la integración, que a menudo incluye el despliegue a un entorno similar al de producción. En muchos casos, el *script* encargado del ensamblado no sólo compila binarios, también genera la documentación, páginas web, estadísticas y ficheros de distribución.

Además una de las fases a incluir debe ser la de pruebas. Una vez el código es compilado, todas las pruebas deben ejecutarse para confirmar que el sistema se comporta de la manera esperada.

Subir cambios a la rama principal todos los días

Subiendo los cambios al repositorio regularmente, cada desarrollador con ello reduce el número de cambios conflictivos. Prolongar la frecuencia de actualización supone el riesgo de entrar en conflictos con otros cambios que pueden ser muy difíciles de resolver.

Muchos programadores recomiendan guardar todos los cambios al menos una vez al día, una vez por cada característica añadida, y además construir el proyecto una vez al día durante la noche.

Cada commit a la rama principal debe disparar la construcción

El sistema debe construirse cada vez que se actualiza la rama principal para verificar que los cambios agregados se han integrado correctamente. Una práctica habitual es utilizar integración continua automatizada, aunque se puede realizar de forma manual. En general, la integración continua es sinónimo de una automatización en la que un servidor de integración continua o un demonio monitor de los cambios en el control de versiones disparan el proceso de construcción de forma automática.

La construcción debe ser rápida

El proceso de construcción debe ser rápido de modo que si hay un problema de integración debe ser rápidamente identificado.

Ejecutar las pruebas en una copia del entorno de producción

Tener un entorno de prueba puede llevar a fallos en los sistemas a prueba cuando se despliegan en un entorno de producción, debido a que el entorno de producción difiere del entorno de prueba de forma significativa. Sin embargo, construir una réplica del entorno de producción es demasiado costoso. En su lugar, el entorno de preproducción debe ser construido para ser una versión escalable del entorno de producción real, para reducir costes, pero manteniendo la composición y los matices de la pila de tecnologías.

Facilitar el acceso a los últimos entregables

Hacer los artefactos generados accesibles a los probadores tras construir el sistema puede reducir la cantidad de trabajo duplicado. El hecho de realizar pruebas tempranas reduce las posibilidades de defectos en las versiones disponibles para los usuarios. Además encontrar los errores antes reduce en muchos casos el trabajo necesario para resolverlos.

Resultados del último proceso accesibles

Debe ser sencillo descubrir si el proceso de construcción falló, dónde y quién hizo el cambio que provocó ese fallo.

Automatizar el despliegue

La mayoría de los sistemas de integración continua permiten ejecutar *scripts* después de que el proceso de construcción termine. En la mayoría de las situaciones es posible ejecutar un proceso que despliegue la aplicación a un servidor de test al que cualquiera pueda acceder. Un paso más en esta manera de pensar es el despliegue continuo, que defiende que el software debe ser desplegado directamente a producción, a menudo con una automatización adicional para evitar defectos [Ries 2009].

4. TÉCNICAS Y HERRAMIENTAS

En este apartado se recogen las técnicas metodológicas utilizadas en la realización del proyecto, así como las herramientas de desarrollo utilizadas, detallándose en mayor o menor medida, según su relevancia, las características y funciones más importantes dentro del ámbito de nuestra aplicación. Del mismo modo, podrán ser analizadas otras como alternativa a considerar frente a las primeras.

4.1. Automatización de la construcción - Maven

Apache Maven ha sido la herramienta utilizada en el proyecto para la automatización del proceso de integración continua. Maven es una herramienta software para la gestión y la automatización del proceso de construcción de software. Es principalmente utilizada para la programación en Java pero también puede ser utilizada para gestionar proyectos escritos en C#, Ruby, Scala y otros lenguajes. Maven proporciona la misma funcionalidad que Apache Ant pero está basado en conceptos diferentes y funciona de una manera completamente distinta.

Maven utiliza un modelo conocido como *Project Object Model (POM)* para describir el software del proyecto a construir, sus dependencias con otros módulos externos y con otros componentes y el orden de construcción. El proceso de construcción con la herramienta viene definido con una serie de objetivos predefinidos, llamados *targets*, los cuales permiten ejecutar tareas habituales como la compilación del código o su empaquetado.

Además la herramienta es capaz de descargar dinámicamente bibliotecas Java y extensiones a su propio núcleo, plugins, desde uno o varios repositorios. La herramienta proporciona soporte para la descarga de paquetes desde el repositorio central sin necesitar configuración previa y permite la configuración de un proyecto para la descarga de librerías de otros repositorios o la subida de bibliotecas a repositorios específicos después de que un proceso de construcción se ejecute con éxito.

4.1.1. POM: Project Object Model

El *POM* proporciona toda la configuración que necesita un proyecto. La configuración general incluye el nombre del propio proyecto, su propietario y sus dependencias con otros proyectos. También permite configurar fases individuales del proceso de construcción que

son implementadas como plugins. Por ejemplo, se puede configurar la extensión encargada de la compilación para que compile utilizando la versión 1.5 de Java o especificar que un proyecto puede ser empaquetado incluso si algún test unitario fallara.

Los proyectos grandes deben ser divididos en varios módulos o subproyectos, cada uno con su propio *POM*. Uno de ellos puede ser el *POM* raíz a través del cual se compilan el resto de módulos con un solo comando. Los *POM* pueden heredar configuración de otros ficheros *POM*, de hecho todos los *POM* heredan del que el propio Maven define [Apache Maven n.d.].

4.1.2. Plugins

La mayor parte de la funcionalidad con la que cuenta Maven está en sus plugins. Un plugin proporciona una serie de objetivos, *goals*, que pueden ser ejecutados utilizando la siguiente sintaxis:

```
mvn [nombre-plugin] : [nombre-objetivo]
```

Por ejemplo un proyecto Java puede ser compilado con el objetivo `compile` de la extensión `compiler` ejecutando:

```
mvn compiler:compile
```

Existen plugins de Maven para compilar, empaquetar, testar, interactuar con el control de versiones, ejecutar un servidor web, generar ficheros de proyecto de Eclipse y un largo etcétera. Los plugins son configurados en la sección `<plugins>` del fichero `pom.xml`. Algunas extensiones básicas vienen incluidos por defecto en todos los proyectos y además están preconfiguradas con una serie de valores adecuados.

Sin embargo sería demasiado trabajoso si se tuvieran que ejecutar varios objetivos de forma manual por ejemplo para compilar, ejecutar pruebas y empaquetar un proyecto:

```
mvn compiler:compile  
mvn surefire:test  
mvn jar:jar
```

El concepto de ciclo de vida de Maven hace frente a este problema.

4.1.3. Ciclos de vida de la fase de construcción

El ciclo de vida de Maven es una lista de fases las cuales pueden ser utilizadas para establecer un orden de ejecución de los objetivos. Uno de los ciclos de vida estándar es el ciclo de vida por defecto que incluye las siguientes fases en el orden indicado [Maven - Introduction to the Build Lifecycle n.d.]:

- process-resources
- compile
- process-test-resources
- test-compile
- test
- package
- install
- deploy

Los objetivos proporcionados por los plugins pueden estar asociados con diferentes fases del ciclo de vida. Por ejemplo, por defecto, el objetivo `compiler:compile` está asociado con la fase de compilación, mientras que el objetivo `surefire:test` está asociado con la fase de test.

```
mvn test
```

Cuando el comando anterior se ejecuta, Maven lanzará todos los objetivos asociados con cada una de las fases previas a la fase de test. Por lo tanto, lanzará el objetivo `resources:resources` que se encuentra asociado a la fase de procesamiento de recursos, después `compiler:compile` y así hasta que finalmente lance `surefire:test`.

Maven también tiene ciclos de vida estándar asociados con la limpieza del proyecto y con la generación de una web para el proyecto. Si la limpieza fuera una parte del ciclo de vida por defecto el proyecto sería limpiado cada vez que se construya el proyecto. Este no es el comportamiento que se deseaba, así que a la limpieza se le asignó su propio ciclo de vida.

Gracias a los ciclos de vida estándar, cualquiera puede construir, testar e instalar cada proyecto Maven utilizando simplemente el comando `mvn install`.

4.1.4. Dependencias

Un proyecto que depende de la librería Hibernate simplemente tiene que declarar su dependencia con el proyecto de Hibernate en su *POM*. Maven automáticamente descargará la librería Hibernate y aquellas librerías de las que Hibernate depende y las guardará en el repositorio local de usuario. El repositorio central de Maven 2 [Maven Central Repo n.d.] es utilizado por defecto en la búsqueda de librerías, pero también da la oportunidad de poder configurar repositorios adicionales.

Los proyectos desarrollados en una máquina pueden depender unos de otros a través del repositorio local. El repositorio local es simplemente una estructura de directorios que actúa tanto como una caché para dependencias como de almacenamiento centralizado para bibliotecas construidas localmente. El comando `mvn install` construye un proyecto y coloca sus binarios en el repositorio local. De este modo otros proyectos pueden utilizar este proyecto especificando las coordenadas de localización en su *POM*.

4.2. Control de versiones - Git

El control de versiones aparecía como uno de los principios básicos necesarios para la adopción del proceso de integración continua. A continuación se comentarán las ventajas adicionales aportadas por el uso de un sistema de control de versiones y posteriormente se describirán la historia y principios del sistema de versiones escogido para el desarrollo del proyecto Git.

4.2.1. Ventajas del control de versiones

Las ventajas del uso del control de versiones son las siguientes:

- Permite el trabajo simultáneo e independiente de los desarrolladores.
- El repositorio sirve como botón de deshacer a corto, medio y largo plazo.
- El repositorio permite marcar hitos en el desarrollo del código. La mayoría de sistemas permiten *etiquetar* hitos dándoles nombres significativos fácilmente

reconocibles por los usuarios, por ejemplo los nombres de las versiones liberadas.

- El repositorio funciona como histórico de cambios realizados permitiendo ver la evolución del proyecto.
- Posibilita la integración continua pues el repositorio se convierte en la fuente desde la que el software encargado de la construcción automática recoge la última versión del proyecto. A partir de esta versión del código dicho software compila el proyecto y lo empaqueta, ejecuta las pruebas y el resto de tareas definidas en el proceso de construcción.
- Integración con el sistema de tareas. Los actualizaciones marcan finalización de tareas y las tareas apuntan a los actualizaciones que las completan.
- Servidor de versiones del producto. Las últimas versiones de interés del producto pueden ser almacenadas y hacerse accesibles desde el control de versiones, lo que se facilita especialmente con las etiquetas.
- Integración con herramientas y procesos de revisión del código como Gerrit [Gerrit n.d.].
- Los últimos servicios de repositorio web, como son Github o Gitorious por ejemplo, facilitan la dinamización del desarrollo gracias a las herramientas públicas de acceso a los fuentes y a los cambios realizados en los commits, comentarios sobre los cambios, etc. Todo ello accesible desde el navegador. Además incorporan utilidades para la promoción del producto como páginas web o *wikis* con manuales que son fácilmente accesibles desde la misma web en la que se puede acceder a los ficheros fuente.

4.2.2. Introducción y breve historia de Git

El desarrollo de Git [Git - Fast Version Control System n.d.] comenzó cuando varios de los desarrolladores del *kernel* de Linux decidieron dejar de utilizar BitKeeper como sistema de control de versiones para el desarrollo del núcleo. Previamente el propietario de los derechos de *copyright* sobre Bitkeeper había anulado el privilegio de uso del sistema de forma gratuita a estos tras acusar a uno de ellos de haber utilizado ingeniería inversa para descifrar los protocolos del sistema.

Linus Torvalds quería un sistema distribuido que pudiera utilizar del mismo modo que BitKeeper, pero ninguno de los sistemas gratuitos disponibles cumplía los requisitos, especialmente de rendimiento que Torvalds exigía.

Torvalds definió como criterios para el próximo sistema:

- Tomar CVS como un ejemplo de lo que no hacer. En caso de duda, hacer lo contrario que hacía CVS.
- El sistema debía soportar un flujo distribuido similar al utilizado previamente con BitKeeper.
- Debían existir medidas de seguridad muy fuertes contra la corrupción, tanto accidental como intencionada.
- El nuevo sistema debía cumplir con unos requisitos de rendimiento muy altos para ser capaz de no ralentizar el proceso de desarrollo del kernel.

Los primeros tres criterios eliminaban todos los sistemas de control preexistentes excepto Monotone y el cuarto criterio excluía todos. Es por ello que, inmediatamente tras el desarrollo de la versión 2.6.12-rc2, Torvalds decidió escribir su propio sistema.

El desarrollo de Git comenzó el 3 de abril de 2005. El proyecto fue anunciado el 6 de abril y sus fuentes ya empezaron a estar bajo el control de la propia herramienta el 7 de abril. Torvalds alcanzó de forma inmediata sus objetivos de rendimiento: el 29 de abril el recién nacido Git ya incorporaba parches al árbol del kernel de Linux con una frecuencia de 6,7 por segundo. Ya el 16 de junio el kernel 2.6.12 era completamente gestionado por Git.

Como curiosidad cabe decir que cuando Torvalds fue preguntado por el motivo del nombre *-git* significa estúpido o persona desagradable en inglés- este contestó: "*Soy un bastardo egocéntrico que nombra todos los proyectos en base a sí mismo. Primero Linux y ahora Git.*"

4.2.3. Características de Git

Git es un sistema de control de versiones distribuido, enfocado a la velocidad, la eficiencia y la usabilidad en grandes proyectos. Sus características principales incluyen:

- Desarrollo distribuido. Como ocurre en la mayoría de los sistemas de control

de versiones modernos, Git otorga a cada desarrollador una copia local del historial de desarrollo completo y los cambios son copiados de dicho repositorio a otro. Esos cambios son importados como ramas de desarrollo adicionales y puede hacerse fusiones de ellas del mismo modo que se haría sobre una rama local. Los repositorios pueden ser accedidos bien a través del eficiente protocolo Git, opcionalmente envuelto en `ssh` para autentificación y seguridad, o simplemente utilizando HTTP para publicar el repositorio en cualquier sitio sin ninguna configuración especial sobre el servidor web.

- Fuerte soporte para el desarrollo no lineal. Git soporta la creación y el *merge* de ramas de forma muy rápida y además incluye herramientas potentes para visualizar y navegar un historial no lineal.
- Manejo eficiente de grandes proyectos. Git es muy rápido y escala muy bien incluso cuando se trabaja con proyectos grandes e históricos de cambios profundos. Es normalmente un orden de magnitud más rápido que otros sistemas de control de versiones e incluso varias órdenes de magnitud en algunas operaciones.
- El historial de Git se guarda de tal manera que el nombre de una revisión en particular, un *commit* en términos de Git, depende del historial completo de desarrollo que precedió a dicho commit. Además las etiquetas con las que cuenta también pueden ser firmadas criptográficamente.
- Diseño en *toolkit*. Siguiendo la tradición de Unix, Git es una colección de muchas pequeñas herramientas escritas en C, y un conjunto de *scripts* que proporcionan envoltorios. Git proporciona por lo tanto, uso sencillo para los usuarios y facilidades para poder desarrollar *scripts* que proporcionen nuevas funcionalidades.

4.2.4. *El modelo de objetos de Git*

A continuación se detalla el modelo de objetos de Git:

El SHA

Toda la información necesaria para representar el historial de un proyecto es guardado en ficheros referenciados por un nombre de objeto formado por 40 dígitos que

tiene un aspecto como el siguiente:

```
6ff87c4664981e4397625791c8ea3bbb5f2279a3
```

Estas cadenas de cuarenta caracteres son utilizadas en muchos sitios en Git. En cada caso el nombre es calculado tomando el *hash SHA1* del contenido de un objeto. SHA1 es una función *hash* criptográfica. Lo que eso significa es que es virtualmente imposible encontrar dos objetos diferentes con el mismo nombre. Esto presenta un conjunto de ventajas entre las que se encuentran:

- Git puede determinar de forma sencilla si dos objetos son idénticos o no simplemente comparando sus nombres.
- Debido a que los nombres se calculan de forma idéntica en cada repositorio, el mismo contenido guardado en dos repositorios siempre será guardado bajo el mismo nombre.
- Git puede detectar errores cuando lee un objeto comprobando que el nombre de un objeto se corresponde todavía con el *hash SHA1* de su contenido.

Los objetos

Cada objeto está formado por tres propiedades: un tipo, un tamaño y un contenido. Existen cuatro tipos de objetos: *blob*, *tree*, *commit* y *tag*. Los contenidos de un objeto dependen de qué tipo de objeto se trate:

- Un *blob* es utilizado para almacenar datos. Es generalmente un fichero.
- Un *tree* es básicamente como un directorio. Referencia a un conjunto de otros árboles o blobs, es decir, ficheros y subdirectorios.
- Un *commit* apunta a un árbol único, marcándolo como una instantánea del contenido del proyecto en un determinado momento en el tiempo. Contiene metainformación sobre ese momento: como la fecha, el autor de los cambios desde el último commit, un puntero al commit previo, etc.
- Una etiqueta es la forma de especificar un commit como especial de alguna manera. Habitualmente es utilizado para poder etiquetar ciertos commits como *releases* específicas.

Casi todo Git está construido sobre la manipulación de esta simple estructura de cuatro tipos diferentes de objetos. Es algo así como su propio sistema de ficheros que se asienta sobre el sistema de ficheros de la máquina.

Diferente de otros SCM

Es importante señalar que esta manera de trabajar de Git es muy diferente de la que tienen la mayoría de los sistemas *SCM* con los que los desarrolladores están familiarizados. Subversion, CVS, Perforce y Mercurial, por ejemplo, todos ellos utilizan sistemas de almacenamiento Delta, es decir, almacenan las diferencias entre un commit y el siguiente. Git no hace esto, por contra Git captura una instantánea del contenido de los ficheros y directorios de un proyecto cada vez que se hace un commit. Este es un concepto muy importante a comprender cuando se utiliza Git.

Objeto blob

Un blob generalmente contiene el contenido de un fichero.

5b1d3..	
blob	size
#ifndef REVISION_H #define REVISION_H #include "parse-options.h" #define SEEN (1u<<0) #define UNINTERESTING (1u #define TREESAME (1u<<2)	

Ilustración 1: Representación de un objeto blob

Se puede utilizar `git show` para examinar los contenidos de cualquier blob.

Asumiendo que conocemos el `SHA` de un blob, se puede mostrar su contenido de la siguiente manera:

```
$ git show 6ff87c4664  
  
Note that the only valid version of the GPL as far as this project  
is concerned is _this_ particular version of the license (ie v2, not  
v2.2 or v3.x or whatever), unless explicitly otherwise stated.
```

Un objeto blob no es nada más que datos binarios. No hace referencia a ningún atributo adicional de ningún tipo, ni siquiera al nombre de un fichero.

Debido a que el blob se define exclusivamente en base a los datos de su contenido, si dos ficheros en un árbol de directorios, o en varias versiones diferentes del repositorio, tienen el mismo contenido compartirán el mismo objeto blob. El objeto es totalmente independiente de su localización en el árbol de directorios y renombrar un fichero no cambia el objeto con el que dicho fichero está asociado.

Objeto tree

c36d4..		
tree	size	
blob	5b1d3	README
tree	03e78	lib
tree	cdc8b	test
blob	cba0a	test.rb
blob	911e7	xdiff

Ilustración 2: Representación de un objeto árbol

Un árbol es un objeto que simplemente contiene un grupo de punteros a blobs y a otros árboles. Generalmente representa los contenidos de un directorio o subdirectorio.

El comando git show también es capaz de mostrar el contenido de árboles, pero git ls-tree proporciona más detalle. Asumiendo que conocemos el SHA de un árbol lo podemos examinar con:

```
$ git ls-tree fb3a8bdd0ce
100644 blob 63c918c667fa005ff12ad89437f2fdc80926e21c      .gitignore
100644 blob 5529b198e8d14decbe4ad99db3f7fb632de0439d      .mailmap
100644 blob 6ff87c4664981e4397625791c8ea3bbb5f2279a3      COPYING
040000 tree 2fb783e477100ce076f6bf57e4a6f026013dc745      Documentation
100755 blob 3c0032cec592a765692234f1cba47dfdcc3a9200      GIT-VERSION-GEN
100644 blob 289b046a443c0647624607d471289b2c7cd470b      INSTALL
100644 blob 4eb463797adc693dc168b926b6932ff53f17d0b1      Makefile
100644 blob 548142c327a6790ff8821d67c2ee1eff7a656b52      README
```

Como podemos ver un árbol contiene una lista de entradas, cada una con un modo, un tipo de objeto, un nombre SHA1 y un nombre de fichero.

Un objeto referenciado por un árbol puede ser un blob, representando los contenidos de un fichero u otro árbol, representando el contenido de un subdirectorio. Debido a que los árboles y los blobs, igual que el resto de objetos, son nombrados en base al hash SHA1 de

sus contenidos, dos árboles tienen el mismo SHA1 si y sólo si sus contenidos, incluyendo los contenidos de sus subdirectorios de manera recursiva son idénticos. Esto permite a Git determinar de forma rápida las diferencias entre dos objetos árboles relacionados, debido a que puede ignorar entradas con nombres de objeto idénticos.

Objeto commit

ae668..	
commit	size
tree	c4ec5
parent	a149e
author	Scott
committer	Scott
my commit message goes here and it is really, really cool	

Ilustración 3: Representación de un objeto commit

El objeto *commit* enlaza el estado concreto de un árbol en un momento determinado con una descripción y con cómo se llegó a ese estado y porqué.

Se puede utilizar la opción `-pretty=raw` junto con `git show` o `git log` para examinar un commit cualquiera:

```
$ git show -s --pretty=raw 2be7fcb476
commit 2be7fcb4764f2dbcee52635b91fdb1b3dcf7ab4
tree fb3a8bdd0ceddd019615af4d57a53f43d8cee2bf
parent 257a84d9d02e90447b149af58b271c19405edb6a
author Dave Watson <dwatson@mimvista.com> 1187576872 -0400
committer Junio C Hamano <gitster@pobox.com> 1187591163 -0700
    Fix misspelling of 'suppress' in docs
    Signed-off-by: Junio C Hamano <gitster@pobox.com>
```

Como se ha podido ver un commit se define por:

- Un árbol: el nombre SHA1 de un objeto árbol, representando el contenido de un directorio en un momento determinado.

- Su padre o sus padres: El nombre SHA1 de uno o varios commits que representan los pasos inmediatamente previos en el historial. En el ejemplo anterior, el commit tiene sólo un parent; los commits tras un *merge* pueden tener varios. Un commit sin padres es conocido como commit *root* y representa la revisión inicial de un proyecto. Cada proyecto debe tener al menos un *root* y puede tener más aunque no es común ni habitualmente una buena idea.
- Un autor: persona responsable del cambio, junto con su fecha.
- Responsable del commit: el nombre de la persona que hizo efectivo dicho commit, junto con la fecha en la que se efectuó. Esto puede ser distinto del autor, por ejemplo si el autor escribió un parche y otra persona se encargó de utilizar el parche para crear el commit.
- Un comentario describiendo el commit.

Hay que tener en cuenta que el commit por sí mismo no lleva ninguna información sobre los cambios, todos los cambios son calculados comparando los contenidos del árbol al que el commit se refiere, con los árboles asociados con los commits parent. En particular, Git no registra el renombrado de ficheros explícitamente, aunque puede identificar casos en los que la existencia de ficheros con los mismos datos en distintas rutas sugiere un renombrado.

Un commit es creado habitualmente con `git commit` que crea un commit cuyo parent es la última revisión disponible en el repositorio, *HEAD*, y cuyo árbol es el contenido actual almacenado en el índice.

El modelo de objetos

Ahora que ya se conocen los tres tipos de objetos principales echaremos un vistazo a como encajan unos con otros.

Si se tiene una estructura de proyecto simple como la siguiente:

```
$>tree
.
| -- README
`-- lib
```

```

|-- inc
|   '-- tricks.rb
`-- mylib.rb

2 directories, 3 files

```

Y queremos realizar un commit a un repositorio de Git este se representaría de la siguiente manera:

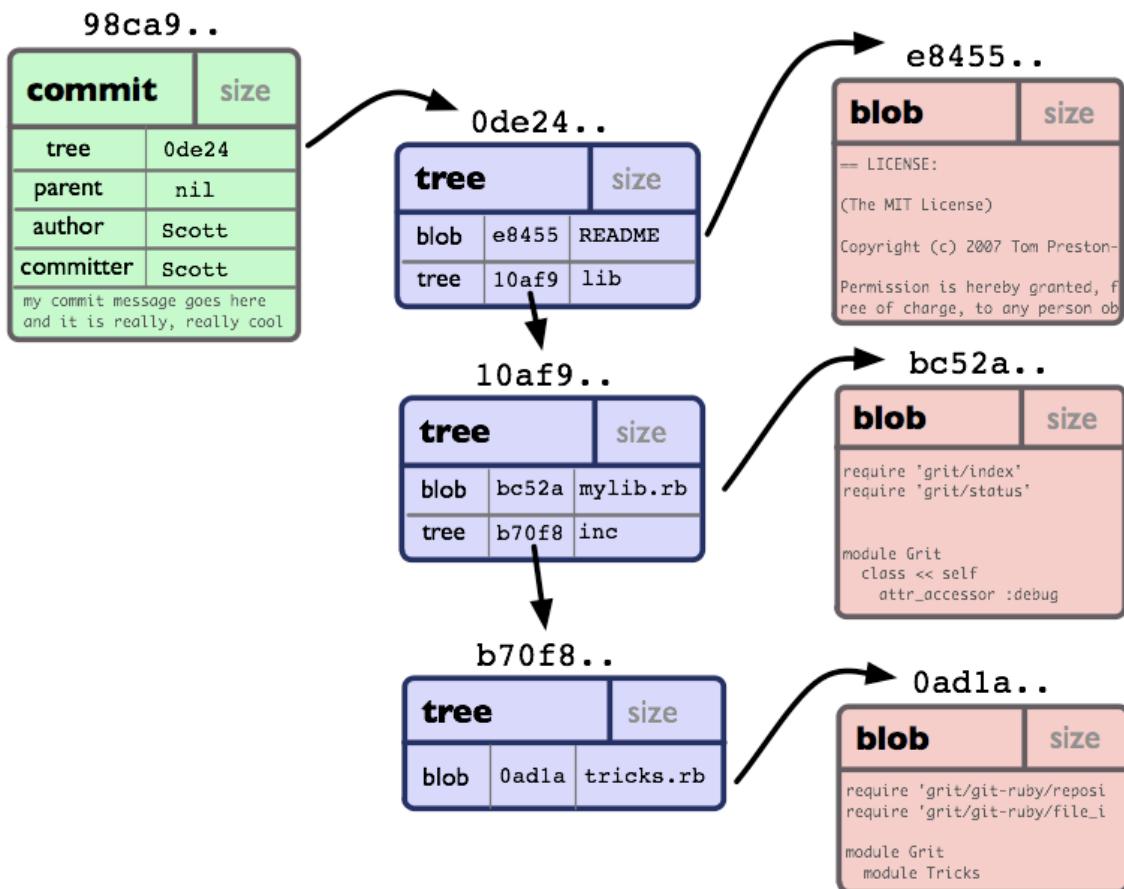


Ilustración 4: Modelo completo de objetos de Git

Se puede comprobar que se ha creado un árbol de objetos para cada directorio, incluido el raíz, y un blob para cada fichero. Así tenemos un objeto commit que apunta al directorio raíz lo que nos permite ver el aspecto del proyecto cuando el commit fue realizado.

Etiquetas (tags)

49e11..

tag	size
object	ae668
type	commit
tagger	Scott
my tag message that explains this tag	

Ilustración 5: Representación de un objeto tag

Un *tag* contiene un nombre de objeto, conocido simplemente como objeto, un tipo de objeto, un nombre de etiqueta, el nombre de la persona que creó el tag y un mensaje que puede contener una firma, como puede se puede ver ejecutando el comando `git cat-file`:

```
$ git cat-file tag v1.5.0
object 437b1b20df4b356c9342dac8d38849f24ef44f27
type commit
tag v1.5.0
tagger Junio C Hamano <junkio@cox.net> 1171411200 +0000

GIT 1.5.0
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.6 (GNU/Linux)

iD8DBQBF01GqwMbZpPMRm5oRAuRiAJ9ohBLd7s2kqjkKlqqC57SbnmzQCdG4ui
nLE/L9aUXdWeTFPron96DLA=
=2E+0
-----END PGP SIGNATURE-----
```

4.2.5. Elección de Git como sistema de control de versiones

Cuando se planteó la elección de la herramienta de control de versiones a utilizar se tuvo claro que había una serie de requisitos que la herramienta que se escogiera debía de cumplir, estos eran:

- En primer lugar debía ser una herramienta que hiciera muy sencillo el trabajo con ramas. Desde el principio del proyecto se planteó que se seguiría el patrón rama por tarea [Plastic SCM blog: Branch per task workflow explained n.d.] para la integración del código fuente en el repositorio y por tanto el seguimiento de este proceso se complicaría notablemente si la herramienta a utilizar no facilitaba la creación y fusión de las ramas.
- Debía de ser una herramienta que ofreciera soporte para trabajar de forma distribuida. Se quería disponer de todo el conjunto de ventajas que ofrece este tipo de sistemas entre las que se incluyen la posibilidad de trabajar sin conexión a red y guardando los cambios en el repositorio, la posibilidad de disponer un repositorio local en el que probar los cambios sin miedo a corromper el repositorio público, el incremento de velocidad derivado de reducir el número de operaciones con conexión a red. Eso sí, contando siempre con una copia completa del historial en el repositorio local.
- Debía contar con un servicio de alojamiento de nuestro repositorio de forma gratuita. Esto se veía facilitado por el hecho de que nuestro proyecto iba a ser liberado con licencia GPL, por tanto existían diversas opciones de repositorios gratuitos para proyectos de código libre.
- Debería de ser rápido también ya que un sistema cuyos comandos no fueran suficientemente rápidos podía provocar que decidiéramos limitar su uso. Disminuyendo de este modo las ventajas asociadas al trabajo con el control de versiones y probablemente la frecuencia de integración.
- Si era posible debía existir un plugin que permitiera acceder a los comandos más habituales y al estado del repositorio sin abandonar el entorno de programación de Eclipse.

Bajo estos requisitos iniciales quedaban descartados sistemas como CVS y Subversion y se presentaba como candidato firme Git. Se decidió hacer un estudio de sus

características para contrastar sus ventajas y sus inconvenientes y en base a ellas tomar la decisión definitiva sobre su adopción. A continuación se expone un resumen de las conclusiones obtenidas:

Ventajas

- Gran rendimiento muy por encima de los requisitos demandados para el proyecto y superando en las comparativas en la mayoría de los apartados a los sistemas de control de versiones más avanzados [GitBenchmarks - Git SCM Wiki n.d.].
- Amplia comunidad de usuarios. Solo el servicio de alojamiento de repositorios Github contaba según estadísticas propias con más de 300.000 usuarios y alojaba en torno a 85.000 repositorios a dos de diciembre del año pasado. Ver estadísticas más actuales en [ezgraphs n.d.].
- Amplio abanico de documentación en Internet con una gran variedad de libros de introducción y perfeccionamiento en su uso disponibles en [Pro Git - Book n.d.] y [Git Community Book n.d.], con ejemplos para cada uno de los comandos con los que la herramienta cuenta, un número enorme de tutoriales y blogs haciendo referencia al sistema.
- Disponibilidad de todo tipo de comandos para dar respuesta a cada una de las necesidades, con un mayor rango de opciones que el resto de sistemas. Por poner un ejemplo Git ofrece un comando `stash` para guardar los cambios no actualizados al repositorio. Este comando permite almacenar esos cambios, obtener una versión del repositorio y luego recuperar esos cambios realizados y aplicarlos al código obtenido. Además, por su diseño tipo *framework* similar a Linux permite combinar esos comandos para crear tus propios comandos personalizados de forma muy sencilla.
- Manejo de ramas sencillo y flexible. La creación de ramas no supone ningún tipo de sobrecarga, ni aumento de espacio en el repositorio y además tanto la creación, como la fusión, como el cambio de rama de trabajo son sencillos y rápidos.
- Soporte completo para trabajo distribuido. Cada usuario cuenta con su repositorio local en el que guarda sus cambios sin necesidad de acceso a red y con un rendimiento muy bueno. Cuando el usuario lo considera necesario

puede subir dicho historial a un repositorio público. Los conflictos que se producen al publicar desde varios repositorios distintos al repositorio local se solucionan con la misma sencillez con la que se fusionan ramas en local.

- Dispone de un plugin para desarrollo en Eclipse [EGit n.d.], asentado y fiable, situado entre los 10 plugins más descargados y con un manual bastante completo que incorpora los comandos necesarios para un ciclo de control de versiones habitual de un proyecto.
- Dispone de un sorprendente rango de servicios de hosting de repositorios de alta calidad. Servicios como [GitHub n.d.] o [Gitorious n.d.] no sólo ofrecen repositorios gratuitos para proyectos de código libre y con funcionalidades de gestión del repositorio desde el navegador, también incorporan características de valor añadido como sistemas de gestión de tareas, la posibilidad de creación de *wikis* o revisión de código por poner un ejemplo.

Desventajas

- Git no es excesivamente intuitivo y con una curva de aprendizaje bastante profunda incluso para usuarios que provienen de otros sistemas de control de versiones como Subversion, dado que Git funciona con conceptos distintos lo que provoca que se da la paradoja de que incluso comandos similares realizan funciones distintas en ambos sistemas. Esto se ve incrementado por la gran variedad de comandos de los que el usuario dispone que pueden abrumar a un usuario novato. Es muy recomendable iniciarse poco a poco con Git y los libros de iniciación ya citados deben ser una parada indispensable.
- Hace algo difícil trabajar con él desde Windows. La obligación de firmar las subidas de cambios a los servidores públicos en Internet como Github obliga a generar un conjunto de claves `ssh` para empezar a trabajar con repositorios remotos. Además si se quiere disponer de toda la flexibilidad de los distintos comandos de Git se debe utilizar un entorno similar a Linux sobre Windows como `cygwin`.
- No se dispone de un gran número de herramientas de interfaz gráfica para la interacción con los repositorios, especialmente en Windows. Esta pega es menor porque el plugin EGit nos proporciona la funcionalidad necesaria desde Eclipse y por tanto nos permite trabajar también en Windows. Sin embargo si

quisiéramos utilizar otras opciones que EGit no ofrece y seguir trabajando con una interfaz gráfica las alternativas disponibles que se nos ofrecen no son excesivas.

- El trabajo de forma distribuida con Git hace que a veces sea imposible seguir el rastro de todos los clones del repositorio principal en casos con cientos de usuarios. Este no es mayor problema para un proceso con sólo dos usuarios como el de este proyecto pero si es una característica a tener en cuenta y a gestionar con ciertas políticas de uso en el caso de proyectos con muchos usuarios.

Conclusión

Considerando todos estos aspectos positivos y negativos de la herramienta se consideró que Git se adaptaba perfectamente a los requisitos iniciales. Se consiguió superar el problema principal inicial de la difícil adaptación al uso de la herramienta con la consulta de la información disponible en la web y especialmente con los libros de introducción y el manual de EGit. Una vez superado ese escollo, Git ha cumplido con todos los requisitos inicialmente planteados y nos ha permitido trabajar sin dificultades de forma distribuida, haciendo uso intensivo de las ramas, con un servicio de repositorio público gratuito como Github y ejecutando las operaciones habituales sin necesidad de abandonar el entorno de desarrollo de Eclipse gracias a Egit.

4.3. Gestión de tareas - Fogbugz

En el proyecto se ha utilizado una metodología basada en tareas para el desarrollo. La idea ha sido en todo momento llevar un registro de las tareas y bugs con la intención de agruparlas y priorizarlas para llevar siempre a cabo aquellas que proporcionaran mayor valor añadido. A continuación se explicarán las ventajas de este enfoque adoptado para después revisar las herramientas que se han utilizado para dar este enfoque.

4.3.1. Ventajas de la gestión de tareas

Las ventajas de la gestión de tareas son las siguientes:

- El proceso se basa en priorizar las tareas e implementar las tareas de mayor prioridad, es decir, las que han sido consideradas como capaces de ofrecer mayor valor añadido de forma similar a lo que proponen las metodologías

ágiles.

- Facilita la comunicación entre los miembros del equipo y el reporte de bugs por parte de los clientes.
- Facilita el seguimiento de las tareas realizadas en el proyecto y de las mejoras añadidas respecto a la versión previa del plugin.
- Integración con el sistema de control de versiones.
 - Permite elaborar planes para crear iteraciones formadas por grupos de tareas que se definen para ser incluidas en próximas entregas del producto.
 - Permite hacer referencias a commits en los que ciertas tareas son solucionadas o en el sentido contrario marcar en los commits que fallos han sido resueltos o qué características han sido implementadas.
 - Permite utilizar el patrón de desarrollo *branch per task*, es decir, una rama por tarea [Plastic SCM blog: Branch per task workflow explained n.d.] que se basa en que un desarrollador cuando decide solucionar un error o implementar una característica nueva del software crea la tarea en el gestor de tareas y en el control de versiones crea una rama para desarrollar esa tarea. La rama se reintegra en la rama principal del repositorio únicamente cuando esta tarea ha sido completada y se ha comprobado que el proceso de construcción funciona y las pruebas no fallan. De este modo, se consigue que la rama principal del repositorio siempre esté libre de fallos, que no haya problemas de integración porque las tareas son pequeños cambios bien definidos y que siempre se pueda conocer a la perfección qué tareas han sido implementadas en qué revisión del control de versiones.

4.3.2. *FogBugz*

Fogbugz [Spolsky n.d.] es una herramienta web de gestión de proyectos, desarrollada por Fog Creek Software, cuya función principal es el control de tareas y de errores, bugs. Otros aspectos adicionales que permite como añadido son los foros de discusión y *wikis*.

Características:

Gestión de proyectos:

- Permite administrar múltiples proyectos, los cuales se encuentran compuestos por áreas, que a su vez se dividen en hitos.
- Organización de las tareas y errores en forma de esquema con estructura de árbol para su mejor visualización.
- Mantiene un histórico de cada una de las tareas, incluyendo las modificaciones y actualizaciones realizadas.
- Ofrece la posibilidad de adjuntar a la tarea cualquier archivo que se considere de relevancia para la misma.
- Búsquedas para filtrar la lista de tareas basadas en palabras clave sobre cualquier campo que conforma la tarea, como son: título, descripción, etc. Además permite crear filtros para almacenar las búsquedas sobre tareas.

Gestión del tiempo:

- Proporciona la posibilidad de introducir estimaciones de las tareas con el objetivo de que nos sirva de guía para la planificación del proyecto. Además se puede realizar la programación de hitos para predecir la finalización de las tareas asignadas.
- Predicción de posibles fechas de finalización de un hito y probabilidad de las mismas por parte de la propia herramienta basándose en la información recogida en los históricos almacenados, así como en el rendimiento observado de los desarrolladores.
- Muestra partes de horas y la historia, por día, de un usuario a partir del trabajo realizado en las tareas.

Gestión general:

- Disponibilidad de generar diagramas de barras y de sectores que representen cualquier lista de tareas, se encuentren filtradas o no. Además se podrán ver

los gráficos de datos actuales o históricos.

- Permite analizar en detalle la información jerárquica dentro de una sección del gráfico generado.
- Obtención de informes de tareas, de usuarios, de proyectos, así como los parámetros de los mismos.

4.3.3. Foglyn

Foglyn es un plugin para Eclipse que permite directamente desde este *IDE* crear, ver, modificar, asignar, resolver o cerrar los casos de FogBugz. Técnicamente se trata de un conector FogBugz para Mylyn.

Foglyn trabaja con Mylyn, que es una interfaz centrada en tareas para Eclipse. Foglyn integra casos FogBugz en Mylyn y le permite hacer un seguimiento del contexto, es decir de los ficheros utilizados, en relación con las tareas asignadas.

Características:

- Foglyn muestra en Eclipse todos los casos de FogBugz como una lista de tareas, permitiendo gestionar cada uno de estos casos.
- Foglyn puede trabajar en modo fuera de línea posponiendo la sincronización, con FogBugz, de las modificaciones de los casos para cuando estemos en línea.
- Insertar hipervínculos a los casos FogBugz directamente en el código fuente.
- Foglyn trabaja con las siguientes versiones de Eclipse:
 - Eclipse 3.4, Ganímedes.
 - Eclipse 3.5, Galileo.
 - Eclipse 3.6, Helios.

4.4. Prototipado de interfaces - Balsamiq Mockup

En los comienzos de un proyecto software se debe pensar en el diseño de la interfaz

gráfica. Por lo tanto, es conveniente ir realizando bocetos de aquello que se deberá mostrar y la forma en la que realizar su presentación. A medida que se avanza en el proyecto se hacen cambios, bien porque se añade nueva funcionalidad o bien porque se mejora la presentación de las ya existentes, y esos bocetos se convierten en prototipos que darán paso a la versión definitiva. Estos son los llamados *mockups*.

Para ello, podemos utilizar papel y lápiz o bien una herramienta que nos ayude en esta tarea, con la cual conseguir una mejor visualización del prototipo. El uso de una herramienta para tal fin nos aporta numerosas ventajas, entre ellas se encuentra la utilización de un formato digital.

En nuestro caso, hemos decidido utilizar una herramienta, en concreto Balsamiq Mockups [Balsamiq Studios, LLC n.d.]. Lo interesante de este programa es que, si bien los gráficos que utiliza son simples, logran que el diseñador pueda mostrar apropiadamente su idea de la estructura del diseño dejando en un segundo plano el diseño gráfico y poniendo máxima atención a la propia interfaz de usuario, es decir, a la interacción del usuario con la aplicación.

Balsamiq Mockups es un programa de escritorio, programado en Flex y Adobe AIR, que al ser creado en AIR es multiplataforma, y por tanto, instalable en Windows, Linux y Mac OS. Su interfaz es sencilla y muy intuitiva, cuenta con una colección muy grande de controles con los que crear cualquier prototipo, los cuales son altamente personalizables.

Asimismo, permite incorporar opciones de comportamiento así como enlaces a otras pantallas. También permite realizar la exportación del propio prototipo como una imagen para poder enviarlo por correo electrónico o parar imprimirla directamente.

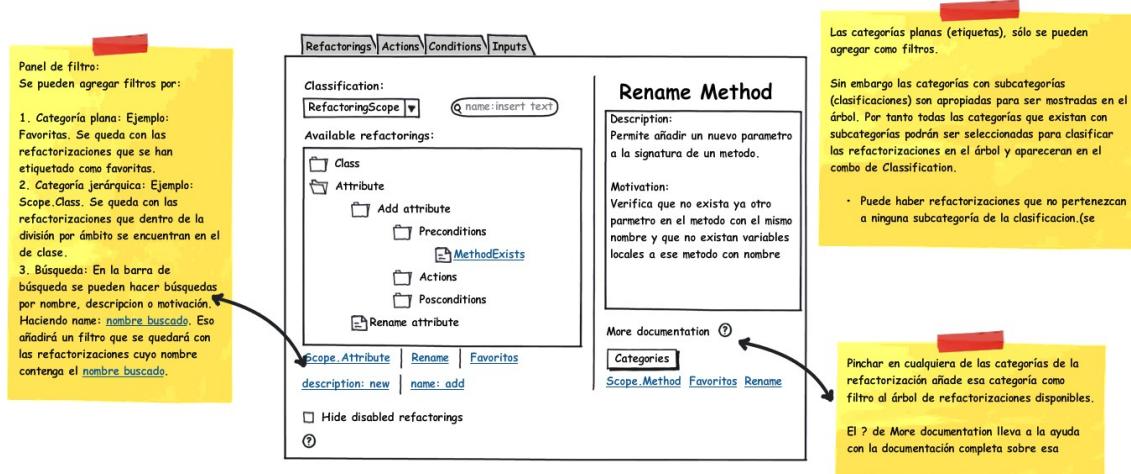


Ilustración 6: Prototipo creado con Balsamiq Mockup

4.5. Pruebas de interfaz - SWTBot

SWTBot [SWTBot - User Guide n.d.] es una herramienta que permite definir de forma sencilla pruebas de interfaz para aplicaciones basadas en SWT y en Eclipse.

La ventaja que SWTBot proporciona es que ofrece una *API* sencilla de leer gracias a que oculta la complejidad inherente a SWT y Eclipse. Además, define su propio conjunto de aserciones para facilitar la comprobación de requisitos sobre la interfaz.

SWTBot puede ejecutarse en cualquier plataforma en la que se pueda ejecutar SWT y además proporciona un conjunto de tareas de ANT que permiten ejecutar las pruebas definidas con SWTBot en cualquier sistema de integración continua. Esto ha permitido en el caso concreto del proyecto que las pruebas de interfaz creadas hayan sido parte del conjunto de pruebas ejecutadas como parte del proceso de construcción diario lanzado por Hudson. Para más información sobre como se han configurado Maven y Hudson para la ejecución de las pruebas de interfaz para el plugin ver:

http://wiki.eclipse.org/SWTBot/CI_Server [SWTBot/CI Server - Eclipsepedia n.d.] y [How to run SWTBot tests with Tycho - Tycho - Confluence n.d.].

4.6. Integración continua - Hudson

De forma general Hudson [Hudson Continuous Integration n.d.] proporciona funcionalidad para monitorizar la ejecución de trabajos repetitivos. Sin embargo, en la

mayoría de los casos es utilizado para la construcción y ejecución de las pruebas de un proyecto software de forma continua, del mismo modo como lo pueden hacer otros sistemas como CruiseControl o DamageControl.

Hudson hace sencillo para los desarrolladores integrar sus cambios a un proyecto, a la vez que facilita a los usuarios obtener la última versión del software que se construyó. La consecuencia de utilizar este proceso de construcción automatizado es en la mayoría de los casos un incremento de la productividad.

Las características que Hudson ofrece se pueden resumir:

- Instalación sencilla: sólo es necesario ejecutar un único comando, `java -jar hudson.war`, para ejecutarlo. No es necesaria ninguna instalación adicional o la configuración de ningún tipo de base de datos.
- Configuración sencilla: Hudson puede ser configurado enteramente desde su interfaz de usuario. No es necesario modificar de forma manual ningún fichero XML.
- Historial de cambios: Hudson puede generar una lista de cambios realizados tras el último proceso de construcción a partir de sistemas de control de versiones como CVS, Subversion o Git entre otros.
- Integración con RSS o correo: los resultados de la monitorización de los procesos de construcción son notificados en tiempo real mediante RSS o por correo electrónico.
- Informes sobre los tests de JUnit: la herramienta genera resúmenes de los informes de pruebas con JUnit que incluyen información sobre el historial de los tests.
- Soporte de plugins: Hudson puede extenderse gracias a plugins. Un usuario puede definir sus propios plugins para automatizar procesos que su equipo necesite.

The screenshot shows the Hudson web interface. At the top, there's a blue header bar with the Hudson logo on the left and a search bar with a magnifying glass icon on the right. Below the header, there's a navigation menu with links like 'Crear nueva Tarea', 'Administrar Hudson', 'Actividad', 'Historia de ejecuciones', 'Dependencia entre proyectos', and 'Comprobar firma de ficheros'. To the right of the menu is a button labeled 'ACTIVAR AUTO REFRESCO' with a refresh symbol. On the far right of the header is a help icon (a question mark inside a circle).

The main content area has a title 'Todo' with a '+' button. Below it is a table with columns: S, W, Tarea (sorted by name), Último éxito, Último fallo, and Última duración. There are two rows of data:

S	W	Tarea	Último éxito	Último fallo	Última duración
		dynamicrefactoring.plugin	48 Min (#106)	8 días 13 Hor (#103)	8 Min 59 Seg
		dynamicrefactoring.plugin(local)	3 Mes 2 días (#10)	1 Mes 0 días (#29)	9 Min 0 Seg

Below the table, there's a link 'Icono: S M L' and three RSS feed links: 'Suscribirse a RSS de: todos los trabajos', 'sólo los fallidos', and 'los más recientes'.

On the left side, there are two sections: 'Trabajos en la cola' (No hay tareas pendientes) and 'Estado de los nodos' (with a table showing two available nodes). A large, semi-transparent illustration of a smiling face is visible on the left side of the main content area.

Ilustración 7: Pantalla de entrada con la lista de proyectos de Hudson

4.7. Control de calidad del código - Sonar

Sonar [Sonar n.d.] es una herramienta que permite controlar la evolución de la calidad del código de cada proyecto de forma continuada. Se basa en tres aspectos fundamentales: definiciones de defectos del código, mediciones de métricas y ejecución de pruebas y cobertura del código.

En el apartado de defectos del código Sonar contiene unas 600 reglas que van desde patrones de nombrado hasta antipatrones de código complejo. La aplicación proporciona distintos perfiles configurables que permiten definir al usuario que conjunto de defectos se deben comprobar sobre el código. En el caso del plugin de refactorización el perfil utilizado ha sido uno de los perfiles más críticos basado en las reglas definidas por la herramienta Findbugs [FindBugs™ n.d.] .

Para las métricas Sonar mide aspectos como el número de líneas de código, la complejidad ciclomática, las líneas de código duplicado o el porcentaje de elementos y clases con comentarios. Además Sonar define sus métricas a partir de módulos que se van componiendo hasta formar el proyecto. En la pantalla principal de la aplicación se muestran las métricas de todo el proyecto y a partir de ahí se puede ir descendiendo para pasar por las métricas de los subproyectos, las de los paquetes y terminar con las de las clases.

En el caso de los tests también se puede realizar la misma navegación y cuando finalmente se llega al nivel de las clases, Sonar muestra el código fuente de las mismas con las líneas que han sido ejecutadas como parte de los test coloreadas en verde.

Además la aplicación también contiene herramientas como la *máquina del tiempo* que muestra en una tabla y en un gráfico la evolución durante la vida del proyecto de cualquier métrica de entre la gran variedad que Sonar ofrece.

Finalmente la gran ventaja que Sonar ofrece es que tanto su instalación como su ejecución son muy sencillas y viene perfectamente integrado con Maven. Esto ha permitido que su incorporación al proceso de construcción del proyecto haya sido muy accesible.

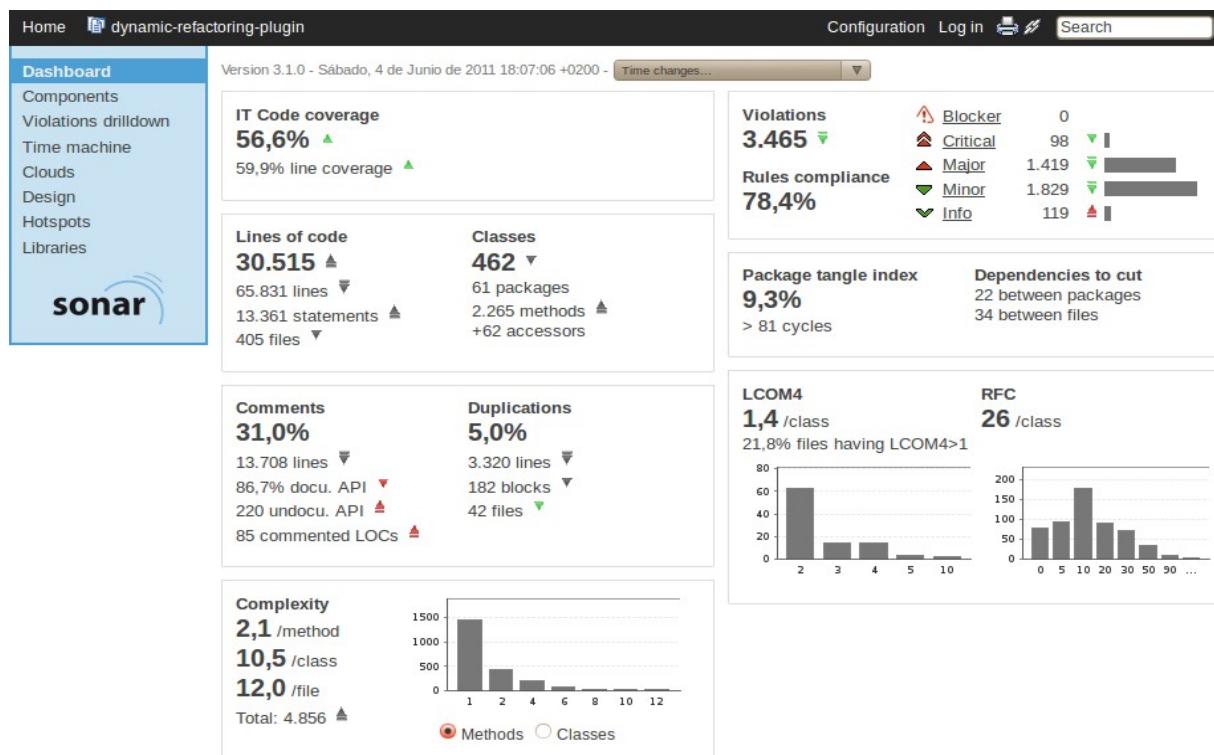


Ilustración 8: Pantalla principal de Sonar

5. ASPECTOS RELEVANTES DEL PROYECTO

En este apartado se describe el proceso seguido en el desarrollo del proyecto, así como las soluciones adoptadas para resolver los aspectos más importantes del mismo.

5.1. Ciclo de vida

Como ciclo de vida se ha querido prescindir de los modelos *clásicos* [Pressman 2006] basados en las tres fases; análisis, diseño e implementación. En su lugar se ha seguido un modelo más similar al promulgado por *SCRUM* [Schwaber 2004] y las metodologías *ágiles*, basado en iteraciones y en requisitos que cambian con la evolución propia de un proyecto.

Al comienzo del proyecto se partió de un primer estudio del producto y una definición de los requisitos iniciales realizado de manera conjunta con el tutor. Los requisitos se transformaron en tareas que se priorizaron.

Con estas tareas definidas se conformaban las iteraciones, al final de cada iteración se tenía una reunión con el tutor. En dicha reunión se empezaba con una demostración al tutor de las características implementadas en dicha iteración. Posteriormente se realizaba un planteamiento de los problemas y los nuevos requisitos que habían surgido como resultado del trabajo completado. Esos requisitos se priorizaban nuevamente y basándose en una estimación del tiempo de desarrollo se definía la lista de tareas a implementar en la siguiente iteración, así sucesivamente dando una entrega continua del producto.

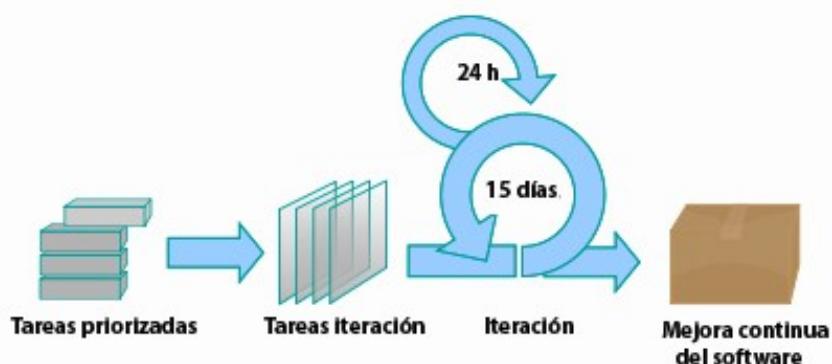


Ilustración 9: Ciclo de vida utilizado en el proyecto

Para la implementación de todas las tareas de cada iteración se definieron una serie de requisitos de calidad que se debían de cumplir para que una tarea se considerara como completada. La tarea debía estar testada y debía cumplir con los estándares del código definidos al principio del proyecto.

Antes de cada reunión y de realizar la demostración del proyecto se realizaba el proceso de construcción del producto. Este permitía comprobar que todas las pruebas se ejecutaban correctamente y si se había cumplido con los estándares de calidad definidos utilizando la herramienta Sonar.

Gracias a la utilización de este modelo se conseguía que al final de cada iteración se dispusiera de un producto entregable y acorde con los estándares de calidad apropiados para ser entregado al cliente, cuyo rol era llevado a cabo por el tutor. Además la redefinición de los requisitos a lo largo del proyecto y la priorización han permitido un desarrollo flexible y basado en la implementación de las características más importantes en cada momento.

5.2. Proceso de integración continua

Desde el comienzo del desarrollo se consideró que podría ser de gran valor para el proyecto la adopción de un proceso basado en el principio conocido como integración continua [Fowler 2000]. La teoría y algunas de las prácticas recomendadas por este principio ya han sido descritas en el apartado 3.6 Integración continua. A pesar de ello a continuación describiremos a modo de resumen las principales ventajas de este enfoque:

- Detección temprana de fallos. El control de la calidad no se pospone para el final del desarrollo sino que forma parte de cada pequeña evolución del proyecto. Lo que significa que los fallos son detectados pronto, ya que tienen que superar los test. Fallos que se detectan pronto significa menores costes dado que éstos no se extienden y por tanto son más fáciles de corregir para el desarrollador.

Ese control tan exhaustivo también lleva a un mejor software debido a que hay menos errores no percibidos que perduran hasta la liberación del software y son sufridos por el usuario. Otra ventaja colateral es que los tests actúan como red protectora del proceso de desarrollo. El programador puede realizar refactorizaciones y mejoras en el diseño del código con la confianza de que los tests le avisarán si alguna de las mejoras ha introducido un bug en el propio producto.

- Simplificación de procesos con la automatización. Con la automatización se consigue que procesos manuales sean llevados a cabo por la herramienta de construcción. En proyectos en los que se carece de cualquier herramienta de automatización de la construcción cada proceso debe ser realizado de forma más o menos manual y repetitiva por el programador. Esto lleva a la perdida de tiempo del programador en el corto plazo y al descuido de los procesos o el abandono de ciertos pasos en el medio largo plazo e incluso introduce errores.

Por poner un ejemplo en el proyecto sin la automatización incorporada gracias a Maven si en un momento dado se quisiera llevar a cabo el mismo proceso que ahora se ejecuta con un simple comando, `mvn clean install`, de forma manual, se tendría que: compilar los binarios del proyecto del plugin, generar el *JAR* y firmarlo. Crear un espacio de trabajo y copiar los binarios del repositorio en él para luego compilar los binarios del proyecto de los tests y empaquetar los tests. Ejecutar los tests con *SWTBot* activado para los tests de interfaz gráfica y con *JaCoCo* para controlar la cobertura de los mismos, etc. El número de pasos es tan elevado que alguno se ha dejado sin describir. Todos estos pasos son descritos de forma más detallada en el *Anexo 4 - Documentación Técnica del Programador* pero en definitiva se ha mostrado de forma obvia que pedir al programador que ejecute todos estos pasos a mano de forma regular es relegar su jornada de trabajo a una productividad cercana a cero.

Sin embargo si se dispone de una herramienta de automatización incorporar nuevos procesos útiles sólo supone al desarrollador el tiempo necesario para configurar la herramienta para que lleve a cabo dichos procesos. A partir de entonces el usuario dispondrá del valor añadido aportado por dichos procesos simplemente con ejecutar un comando. Esto permite en definitiva incorporar pasos que sin disponer de la automatización ni serían considerados por ser imposibles o muy tediosos de ejecutar a mano.

- Reducción de riesgos y liberación de versiones del producto más frecuente. Con la integración continua dado que no hay una fase de integración, sino que se va dando en cada pequeño avance, se reducen los riesgos derivados de una fase de integración cuya duración es muy difícil de estimar. Como añadido puesto que todos los pasos para liberar una nueva versión de un producto están automatizados, es más sencillo sacar nuevas versiones o *releases* y hacerlas disponibles al usuario. Esto permite al usuario ser más partícipe de

las evoluciones del proyecto lo que tiene como ventaja derivada que éste puede aportar su opinión sobre nuevas características de forma más rápida. Así las características en desarrollo pueden ser probadas de forma ágil y descartadas si no son valoradas por el usuario.

5.2.1. Tareas incorporadas al proceso de construcción

En anteriores versiones del proyecto el proceso de construcción había sido manual. La construcción se basaba en utilizar los asistentes de exportación proporcionados por Eclipse. En esta nueva versión del plugin se decidió sustituir el sistema anterior por un sistema de construcción automatizado debido a las ventajas ya enumeradas anteriormente.

Los pasos necesarios para introducir esta mejora fueron los siguientes:

- El primer paso consistió en automatizar la compilación del proyecto y además la ejecución de los tests. Es necesario resaltar que configurar Maven para ejecutar los tests de proyectos de plugins de Eclipse es algo más complejo de lo habitual por el hecho de que estos proyectos son paquetes *OSGI*.

Los paquetes *OSGI* necesitan de un entorno de ejecución especial, no es suficiente con ejecutarlos sobre una máquina virtual de Java. Además, definen sus dependencias con otros paquetes *OSGI* de una manera distinta a como definen sus dependencias los ficheros *JAR* habituales. Esto hizo que fuera necesario utilizar un plugin de Maven conocido como Tycho [Tycho n.d.] para realizar el proceso de construcción del proyecto. Tycho permite ejecutar los tests dentro del entorno de ejecución específico que los paquetes *OSGI* requieren y también permite definir repositorios web desde los que descargar dependencias de manera similar a como Maven lo hace para dependencias entre bibliotecas normales de Java.

- El segundo paso vino dado por la decisión de hacer posible instalar el plugin desde Internet. Como consecuencia de esta decisión fue necesario crear nuevos proyectos para generar la característica o *feature* del plugin y el *repositorio P2*.

La *característica* del plugin es una construcción del sistema de paquetes de Eclipse. Según Eclipse, una característica es un conjunto de plugins instalables. Si se quiere hacer uno o varios paquetes instalables a través de

Internet es necesario definir una característica que los agrupe a estos. La característica permite definir cierta metainformación del propio instalable: su descripción, su licencia, su logo. A su vez, permite fijar parámetros de instalación como sobre qué sistemas operativos o arquitecturas de PC los plugins pueden ejecutarse.

El *repositorio P2* es una agrupación de características instalables agrupadas por categorías. Cuando se hace uso del gestor de instalaciones de Eclipse, por debajo este recurre a repositorios P2 para obtener los plugins que se pueden instalar y cuáles se pueden actualizar de los que ya están instalados.

De nuevo el plugin Tycho fue el que hizo posible el generar la característica del plugin y el repositorio P2 como parte del proceso de construcción. Estas tareas se encuentran entre las fases finales de este proceso. Éste está configurado para que dichos artefactos sólo se generen si el proceso de compilación y la ejecución de los tests han sido exitosos.

- Además de los dos pasos que anteriormente han sido comentados, el proceso de construcción se ocupa de realizar otras tareas que aunque de menor peso, aportan valor al proyecto. Entre estas tareas se encuentran:
 - Generación de la documentación en formato JavaDoc del proyecto.
 - Generación del informe de cobertura de los tests.
 - Firmado del plugin con el certificado digital.
 - Ejecución de los tests de interfaz gráfica.
 - Generación de informe completo de calidad del código del proyecto mediante la herramienta Sonar [Sonar n.d.]. Para información sobre las características de la herramienta éstas son enumeradas en *Anexo 4 - Documentación Técnica del Programador*.
 - Cambio de versión automatizado del plugin.

5.2.2. Configuración avanzada

Con la intención de llevar un paso más adelante las ventajas de la automatización del proceso de construcción se decidió configurar un servidor encargado de ejecutar la

construcción del proyecto de forma diaria. Esto tendría una serie de ventajas, la cuales son las siguientes:

- La construcción del proyecto de forma diaria permitiría levantar alarmas cuando hubiera algún fallo en el código del proyecto que impidiera que el proceso de construcción se ejecutara de forma correcta. De esta manera no sería necesaria la intervención de los desarrolladores para que todas las comprobaciones asociadas al proceso de construcción se llevaran a cabo diariamente.
- El contar con el servidor accesible desde Internet permitiría que cualquier componente del equipo pudiera acceder a la información del proceso de construcción en cualquier momento. Si como añadido se configuraba la herramienta Sonar en el propio servidor, la información proporcionada por esta también se haría accesible.
- El servidor también podía ser configurado como repositorio de instalación del plugin desde Internet. La configuración no sería muy compleja. Bastaba con configurar un servidor web que hiciera accesible el fichero del repositorio P2 generado tras el último proceso de construcción.

Finalmente la posibilidad de configurar este servidor nos la brindó el servicio Amazon Elastic Compute Cloud (Amazon EC2) [Amazon Elastic Compute Cloud (Amazon EC2) n.d.]. Este servicio permite crear una instancia de una máquina virtual en la que se escoge el sistema operativo que se quiere ejecutar. Una vez creada la máquina se puede configurar y personalizar al gusto del usuario. Actualmente este servicio dispone de una oferta para ejecutar máquinas virtuales de bajas prestaciones de forma gratuita.

Configuración inicial

Sobre el servicio de Amazon EC2 se configuró Hudson (<http://hudson-ci.org/>) [Hudson Continuous Integration n.d.] como servidor de integración continua. Dentro de Hudson se configuró un proyecto encargado de ejecutar el proceso de construcción del proyecto de forma diaria.

Este proceso se encargaba de descargarse la última versión disponible del proyecto en la rama principal del repositorio de Github. A partir de esa versión se ejecutaban todos los pasos de la construcción del proyecto: compilación, empaquetado, ejecución de los tests y generación del repositorio P2. Una vez el repositorio se generaba, el fichero en formato

ZIP que contenía dicho repositorio se hacía disponible a través de un servidor web de Apache, lo que permitía instalar el plugin desde Internet. Tras ese proceso se ejecutaba el de generación de métricas del proyecto con Sonar.

Con dicha configuración se cerraba el ciclo completo del proceso de construcción y se generaba toda la información necesaria. Sin embargo, esta configuración contó desde un principio con ciertos problemas de estabilidad. Estos problemas de estabilidad se acrecentaron cuando se introdujeron las pruebas de interfaz gráfica que obligaron a una configuración especial de Hudson. El proceso de construcción provocaba reinicios de la máquina virtual del servidor, debido a la baja disponibilidad de memoria. Además la instalación del plugin desde el servidor se convirtió en un proceso excesivamente lento y se dio la circunstancia inesperada de que tras cada reinicio del servidor debido al problema de memoria su *DNS* cambiaba, con lo que también lo hacía la *URL* del repositorio de instalación del plugin.

Configuración definitiva

Finalmente se decidió tomar un enfoque alternativo para afrontar estos problemas. El proceso de construcción diario se mantuvo en el servidor pero la obtención de métricas del código se pasó a realizar de forma local en los equipos de los desarrolladores. Fue necesario dado que gran parte de los errores del servidor se debían al gran consumo de memoria de Sonar. Para afrontar el problema de los cambios de *DNS* ante el reinicio del servidor y de la lentitud de la instalación del plugin se optó por el hosting proporcionado por Google code (<http://code.google.com/>) [Google Code n.d.]. Google Code proporcionaba una web con una URL constante y una velocidad de acceso para la instalación del plugin razonable.

5.3. Repositorio único de refactorizaciones y clasificaciones

En la versión anterior del plugin al arrancar la herramienta se generaba un fichero temporal con los nombres de todas las refactorizaciones existentes y las rutas a sus ficheros de definición XML. Cuando una sección del código necesitaba obtener los datos de una refactorización tenía que:

1. Acceder al fichero XML con la lista de las refactorizaciones existentes para obtener la ruta al fichero de definición de la refactorización.
2. Pasarle la ruta obtenida al lector de refactorizaciones para que este leyera el fichero y devolviera un objeto con los datos de la refactorización.

Este proceso obligaba a todas las secciones que necesitaban realizar consultas sobre refactorizaciones a conocer demasiados detalles sobre la forma en que el repositorio de refactorizaciones estaba implementado de forma interna.

El caso de las modificaciones sobre refactorizaciones era aún más complejo dado que en el proceso de actualización era necesario:

1. Acceder al fichero `XML` con la lista de las refactorizaciones existentes para obtener la ruta al fichero de definición de la refactorización.
2. Pasar la ruta obtenida al lector de refactorizaciones para que este escribiera el fichero con las modificaciones.
3. Tras la modificación se debía actualizar el fichero temporal con la lista de todas las refactorizaciones.

Todos estos pasos los tenía que llevar a cabo de forma manual cualquier clase que quisiera modificar una refactorización. Este diseño tenía una serie de desventajas:

1. Cada vez que se quería leer una refactorización era necesario hacer dos accesos a disco; uno para buscar la ruta de la definición de la refactorización y otro para leer la propia refactorización. En el caso de las modificaciones eran necesarios tres.
2. Los accesos a clases tan dependientes de una implementación concreta como los lectores de refactorizaciones en `XML` estaban extendidos a lo largo de todo el código del plugin. Esto provocaba que esas clases fueran muy difíciles de modificar dado que había muchas clases que dependían de ellas. También hacía impensable mejoras que incluyeran por ejemplo cambiar el formato o el soporte en que las refactorizaciones se almacenaban.
3. Si cada clase podía leer los ficheros `XML`, cada una creaba sus propias instancias repetidas de la misma refactorización, con el problema aún mayor de que si se producían cambios en las refactorizaciones dichas instancias quedaban obsoletas. Con este enfoque se hacía además imposible controlar desde un punto común qué refactorizaciones habían quedado obsoletas y que por tanto había que actualizar.
4. Si alguna clase realizaba una modificación sobre una refactorización tenía que tener en cuenta siempre actualizar el `XML` con la lista de refactorizaciones

disponibles porque si no lo hacía dicho fichero quedaba obsoleto, lo que podía dar lugar a errores posteriores en las lecturas.

5.3.1. Solución adoptada

Se optó por la solución de eliminar el fichero temporal anteriormente mencionado , el cual contenía la lista de refactorizaciones. En su lugar se decidió utilizar una clase que iba a centralizar todas las operaciones de consulta y modificación de refactorizaciones.

Esta clase al arrancar el plugin lee todas las refactorizaciones de los ficheros XML. Para la modificación de una refactorización la clase proporciona un método muy sencillo en el que simplemente se pasa el nombre de la refactorización a modificar y los nuevos atributos de la refactorización y ella se ocupa de la actualización del fichero XML aislando al resto de clases de la necesidad de conocer los detalles internos del almacenamiento.

Como ventaja añadida dado que el resto de clases no tienen acceso a la modificación de las refactorizaciones excepto a través de la clase centralizadora, dicha clase está siempre en la situación de ofrecer la información más actualizada sobre las refactorizaciones. No sólo eso si no que este enfoque permite definir una serie de contratos sobre las modificaciones a las refactorizaciones. Es así como se ha conseguido obligar a que no haya en ningún momento dos refactorizaciones con el mismo nombre o a que con el cambio en el sistema de versionado ninguna clase intente modificar una refactorización que sea del plugin y por tanto no editable.

En definitiva a pesar del esfuerzo notable de modificación del código que ha sido necesario se considera que el nuevo enfoque supone una notable mejora en el diseño del plugin dado que afronta todos los principales problemas que el anterior enfoque tenía:

1. Sólo es necesario una lectura de los XML de las refactorizaciones durante todo el ciclo de vida del plugin, es decir, en su arranque. A partir de entonces todas las lecturas se hacen a partir de memoria a través del repositorio central.
2. Las clases que necesitan hacer consultas o modificaciones sobre las propias refactorizaciones no necesitan conocer los detalles específicos referentes al almacenamiento de las mismas. Éstas sólo interaccionan con el repositorio central que actúa a forma de fachada proporcionando un API sencilla para las operaciones básicas.

El resto de clases es totalmente independiente del funcionamiento interno del

almacenamiento de las propias refactorizaciones que la fachada podría decidir almacenar las refactorizaciones en una base de datos y esto sería indiferente para el resto de clases.

3. Dado que las refactorizaciones son inmutables el catálogo se puede permitir devolver las mismas instancias de las refactorizaciones por cada consulta con lo que no pueden existir instancias repetidas de las refactorizaciones. Además cuando una clase necesita obtener los datos más actualizados de una refactorización sabe que sólo necesita consultar al repositorio, con lo que se evitan los problemas de refactorizaciones obsoletas cuya validez no se puede comprobar.
4. Las clases clientes son más sencillas y ahora no pueden cometer errores en la actualización de refactorizaciones dado que el repositorio se ocupa de los detalles internos.

Finalmente tras comprobar el resultado tan positivo del cambio se decidió utilizar el mismo enfoque para la manipulación de las clasificaciones.

5.4. Versionado de refactorizaciones

Cuando se introdujo la instalación del plugin a través de Internet se hizo de nuevo visible un problema que ya se había presentado en versiones previas del plugin, este no era otro que, cómo afrontar la actualización de refactorizaciones o clasificaciones sin descartar las modificaciones realizadas por los propios usuarios pero incorporando las modificaciones aportadas por la nueva versión del plugin.

La solución debía incluir dos requisitos principalmente: no se debían perder las modificaciones realizadas por el usuario pero se debían incorporar las modificaciones del plugin.

Para dar solución a este problema se plantearon tres aproximaciones:

1. Con cada nueva actualización del plugin se incorporarían todas las nuevas modificaciones del plugin y se haría una copia de seguridad de los cambios del usuario. El usuario podría optar por mantener las nuevas versiones del plugin o recuperar las versiones con sus cambios de la copia de seguridad realizada mediante la funcionalidad de importación de refactorizaciones con la que el plugin cuenta.

2. En la segunda opción se otorgaba una solución mejorada de la primera pero más compleja. En lugar de sobrescribir y dejar exclusivamente en manos del usuario la recuperación de sus versiones, tras la actualización se le mostraría una pantalla con los conflictos encontrados entre sus modificaciones y las modificaciones incorporadas con la nueva versión del plugin. El usuario sería el encargado de resolver estos conflictos especificando para cada uno de ellos la versión con la que desearía quedarse.
3. En la última opción se optaba por una solución completamente distinta en la que se trataba de forma separada las refactorizaciones del propio usuario y las del plugin. Las refactorizaciones del plugin no eran editables para el usuario y serían renovadas por cada actualización del plugin que se realizase. Las refactorizaciones del usuario eran independientes y no se verían afectadas por las actualizaciones.

Finalmente se optó por la tercera opción por los siguientes motivos:

- La primera opción era relativamente sencilla. Sin embargo, no hacía fácil para el usuario mantener sus refactorizaciones dado que por defecto iban a ser descartadas y tendría que copiarlas explícitamente para recuperarlas. Además era necesario introducir un sistema para hacer la copia de seguridad de las refactorizaciones antiguas cada vez que el plugin se actualizara.
- La segunda opción también contaba con la desventaja del sistema de control de la actualización del plugin y además se hacía compleja la implementación de la pantalla de selección de versiones.

Pero sobre todo se consideró que seguía siendo una opción incómoda desde el punto de vista del usuario dado que tras cada actualización este debería escoger con qué versión quedarse para cada conflicto que se presentara. En muchos casos, el usuario no sabría en qué apoyarse para tomar la decisión. Se tuvo miedo de que ante las dudas que se plantearían al usuario, este se viera tentado a evitar las actualizaciones por miedo a no saber qué decisión tomar ante los conflictos.

- La última opción era la más sencilla y además nos permitía incorporar las actualizaciones del plugin al usuario sin obligar a este a decidir sobre qué hacer con sus propias modificaciones. En el caso de que el usuario quisiera crear sus propias versiones a partir de las del plugin se le proporcionaría la

opción de realizar una copia modificable.

5.5. Carga de clases

Parte del proceso de mejora de las refactorizaciones ha consistido en la incorporación de las nuevas versiones de las bibliotecas de MOON y JavaMOON y la modificación de la carga de clases. En versiones previas del plugin sólo se podían ejecutar clases que importaran clases exclusivamente de los paquetes del *runtime* de Java: `java.lang`, `java.util`, `java.io` y `java.lang.annotation`. En la última versión del plugin se pueden importar clases de cualquier paquete de la biblioteca `rt.jar` del *runtime*.

Sin embargo este cambio que suponía una mejora importante para el plugin trajo como consecuencia que era necesario hacer cargas en memoria de gran tamaño y sacó a la luz problemas en el código que habían permanecido ocultos debido al consumo de recursos tan escaso que se hacía.

El primer problema que salió a la luz se presentó cuando tras implementar los cambios en la carga de clases se comprobó que no se podía ejecutar ninguna refactorización porque al ejecutarlas saltaba un error de desbordamiento de la pila. Al estudiar el problema se comprobó que este desbordamiento se producía debido a que el sistema de deshacer las refactorizaciones guardaba una copia en disco del modelo completo por cada refactorización que se ejecutaba. A partir de ese modelo se recuperaban los ficheros cuando se quería deshacer una refactorización. Para realizar la persistencia en disco se utilizaba serialización pero, al haberse incrementado hasta tal punto el tamaño del modelo con el nuevo enfoque en la carga de clases, la serialización fallaba desbordando la pila.

Cuando finalmente se consiguió solucionar el anterior problema apareció uno nuevo también derivado del sistema de deshacer las refactorizaciones. Para permitir deshacer, el sistema que se utilizaba guardaba una lista de las refactorizaciones aplicadas hasta el momento en un historial. El problema es que estos objetos que representaban las refactorizaciones en el historial contenían una referencia al modelo previo a la ejecución de la refactorización. Este problema no se había presentado en versiones previas por los tamaños tan pequeños del modelo, pero con tamaños del modelo en torno a los 150MB, a partir de unas pocas refactorizaciones aplicadas se superaba el gigabyte de memoria consumida lo que hacía imposible continuar con la ejecución. Utilizando el monitor del *JDK* de Java `jvisualvm` es como se pudo identificar las refactorizaciones aplicadas con el incremento de memoria.

La solución que se aportó en este caso supuso sustituir el sistema encargado de deshacer las refactorizaciones basado en almacenar el modelo tras cada refactorización, por un sistema en el que lo único que se guarda es la fecha en la que se había realizado la refactorización. A partir de esa fecha, se utiliza el historial de Eclipse para deshacer las refactorizaciones. Para los que desconocen este historial de Eclipse, comentar que dispone de un sistema de historial que guarda cada modificación que se aplica sobre cada fichero. Este sistema permite volver atrás por ejemplo los cambios realizados en un día en un fichero si se comprueban que han introducido un error. Ese mismo sistema que Eclipse pone a disposición del usuario es el que se utilizó en el código para restaurar los ficheros al deshacer una refactorización. La implementación adoptada tiene las ventajas de además de tratarse de una solución sencilla también es rápida y eficiente. Tras implementar la mejora se pudo comprobar que el consumo de la memoria se estabilizaba a partir de la primera refactorización aplicada.

5.6. Buscador del asistente de refactorizaciones

El plugin proporciona un número elevado de opciones entre las que poder escoger las entradas, las precondiciones, las acciones y las postcondiciones. Esto podía hacer al usuario difícil saber cual de todos ellos escoger debido a la gran variedad que disponía. Versiones anteriores del plugin ya proporcionaban un sistema de búsqueda sobre los elementos en base a su nombre que permitía filtrar mediante expresiones regulares.

Sin embargo, este sistema se consideró insuficiente debido a que en primer lugar el nombre no proporcionaba suficiente información como para descartar elementos y a que la búsqueda por expresiones regulares no era tan flexible como se deseaba.

Para solucionar esas carencias se pensó que se podía utilizar la información de los elementos, la cual se almacenaba en su documentación en formato JavaDoc, para permitir hacer búsquedas más útiles para el usuario. En la implementación definitiva del proyecto cuando se introduce un término en la búsqueda no sólo se considera si coincide con el nombre del elemento, sino que también se considerá si ese término existe dentro de la descripción del elemento en su documentación.

Además se reemplazó la búsqueda por expresiones regulares exclusivamente por una búsqueda que permitiera más opciones. Para ello se recurrió a la biblioteca Lucene [Apache Lucene n.d.] de Apache. Gracias al trabajo de integración de esta biblioteca en el buscador el sistema de búsqueda previo se ha sustituido completamente, lo que ha posibilitado

incorporar una serie de importantes mejoras. Ahora las búsquedas permiten operadores lógicos, se pueden hacer en base a varios campos de entre los metadatos de cada elemento, los resultados se muestran de forma priorizada según relevancia y se permiten operadores de proximidad entre otras cosas. Para más información sobre el conjunto de operadores disponibles para las búsquedas que permite Lucene estos están publicados en [Query Parser Syntax n.d.].

Todas estas posibilidades se han incorporado sin perder la velocidad y la posibilidad, con la que ya se contaba, de realizar las búsquedas mediante expresiones regulares pero la infraestructura interna nada tiene que ver con la anterior, ya que se ha visto reemplazada totalmente. Las búsquedas anteriormente se basaban en comprobaciones sobre expresiones regulares de los nombres de los elementos. Actualmente, la búsqueda consta de un proceso previo en el que se procesa la documentación en formato JavaDoc y a partir de ella se extrae la descripción de cada uno de los elementos. Con esa información obtenida se genera un índice para cada tipo de elemento que permitirá hacer las búsquedas posteriormente. Finalmente todo ese proceso que esta oculto para el usuario se refleja en búsquedas más completas y significativas para el usuario, sin perder velocidad de respuesta debido al modo en que los índices están organizados.

5.7. Bug encontrado en Eclipse Forms

Durante el desarrollo de la interfaz gráfica correspondiente a la vista del catálogo de refactorizaciones del plugin, en concreto a la hora de visualizar las categorías y las palabras clave de la refactorización que es seleccionada en la vista, notamos un comportamiento extraño en el *layout* o gestor de distribución de los componentes gráficos que estábamos utilizando. Se trataba del *layout ColumnLayout* de Eclipse Forms, su funcionalidad es la de presentar los componentes en columnas según el orden en el que se le indiquen estos. El número de columnas presentado lo decide dinámicamente conforme al tamaño disponible y al número de componentes a mostrar. Cuando lo utilizábamos en el plugin para presentar tanto las categorías como las palabras clave el resultado obtenido no era el esperado ya que los componentes no eran visualizados en orden alfabético que es como se le estaban dando. Al intuir que podía tratarse de un *bug* decidimos exponer el problema encontrado en Eclipse Community Forums donde nos indicaron que ciertamente se trataba de un error y la existencia de un reporte donde quedaba recogido. A día de hoy dicho *bug* se encuentra solucionado e irá incluido en el *Target Milestone 3.7 M7*. Para visualizar este se puede consultar la siguiente dirección: https://bugs.eclipse.org/bugs/show_bug.cgi?id=308095

6. TRABAJOS RELACIONADOS

En este apartado se mostrarán algunas comparativas entre el plugin obtenido como producto final del presente proyecto y otras herramientas alternativas o equivalentes que pretendan proporcionar funcionalidades similares o relacionadas.

6.1. Mejoras incorporadas

En la versión anterior del proyecto ya estaban implementadas las características del plugin siguientes:

- Definición simplificada de refactorizaciones mediante el asistente.
- Importación/Exportación de refactorizaciones.
- Historial de refactorizaciones aplicadas y posibilidad de deshacerlas.
- Definición de planes de refactorizaciones.

En la última versión del plugin las características adicionales que hemos incluido han sido las siguientes:

- **Sistema de clasificación de las refactorizaciones:** Esto incluye un editor de clasificaciones que permite crear clasificaciones personalizadas y un editor dentro del asistente de edición de refactorizaciones que permite asignar categorías a una refactorización por cada clasificación existente. Además se ha añadido también un campo más para permitir definir las palabras clave que se deseen asociar a la refactorización.
- **Vista del catálogo de refactorizaciones:** en la zona principal de esta vista se hace uso del sistema de clasificación de refactorizaciones comentado en el punto anterior. Se muestra un árbol en el que las refactorizaciones se agrupan por las categorías de una clasificación. Dicha clasificación seleccionada se puede cambiar por el usuario para mostrar la agrupación en base a otras categorías. Esta vista además presenta un sistema de filtrado y un panel de resumen para las refactorizaciones seleccionadas.

El **sistema de filtrado** consiste en aprovechar la metainformación de la que

las refactorizaciones constan, como es: su descripción, las categorías a las que pertenecen, su nombre o sus palabras clave para permitir resaltar ciertas refactorizaciones en la vista ocultando otras. De esta manera, se puede seleccionar mostrar sólo las refactorizaciones que tengan como palabra clave por ejemplo *renombrar*. Esto puede resultar muy útil dado el elevado número de refactorizaciones con las que el plugin parte. Los criterios por los que se puede filtrar son: categoría, palabra clave y descripción.

Además, la vista proporciona al usuario ayuda integrada para aplicar estos filtros. También permite aplicar filtros acumulativos, desactivar ciertos filtros o eliminarlos. Por otra parte también se da la posibilidad al usuario de cambiar la forma en que los elementos filtrados se tratan, permitiendo bien ocultarlos o simplemente atenuar su visibilidad para marcarlos como filtrados.

El **panel de resumen** se muestra cuando una refactorización es seleccionada y se hace doble clic sobre ella en el árbol de la vista. Este panel tiene como mínimo tres pestañas: la de vista general, la correspondiente a las entradas y la de mecanismos.

La de entradas y la de mecanismos contienen las entradas y los elementos, es decir, las precondiciones, las acciones y las postcondiciones, que componen una refactorización respectivamente.

La vista general contiene la descripción, la motivación, las categorías y las palabras claves de una refactorización. En el caso de los dos últimos atributos, el panel los representa como un enlace que al ser pinchado aplica un filtro a la vista del catálogo. Por ejemplo, si en una refactorización que contiene la palabra clave *renombrar* se pulsa sobre el enlace de dicha palabra, sería como aplicar en la lista el filtro por palabra clave *renombrar* sólo que la aplicación del filtro es más práctica, ya que el usuario no se tiene que encargar de su creación.

Si la definición de la refactorización contiene una imagen o un ejemplo estos se muestran en pestañas adicionales del panel. La pestaña de ejemplos está especialmente tratada para que los ejemplos aparezcan como código fuente formateado para lenguaje Java.

- **Mejoras en el asistente de creación/edición de refactorizaciones:** se han mejorado las páginas del asistente dedicadas a la selección de entradas,

de precondiciones, de acciones y postcondiciones. Cada una de ellas posee un sistema de búsqueda de elementos que permite realizar búsquedas basadas tanto en el nombre como en la descripción que aparece en la documentación en formato JavaDoc. Las búsquedas permiten utilizar comodines, operadores binarios, búsquedas por proximidad o cualquiera de los distintos elementos de la sintaxis definida en [Query Parser Syntax n.d.] .

Además se han reorganizado las pantallas de tal modo que el navegador con la documentación en formato JavaDoc aparece en un formato más pequeño por debajo de un nuevo panel con la descripción y otros parámetros de mayor utilidad sobre el elemento. Estos parámetros son:

- ◆ Para entradas: el conjunto de refactorizaciones en las que la entrada que ha sido seleccionada participa y por otro lado el conjunto de refactorizaciones en las que la entrada es principal.
- ◆ Para precondiciones y postcondiciones: el conjunto de refactorizaciones en las que el predicado seleccionado participa.
- ◆ Para acciones: el conjunto de refactorizaciones en las que la acción participa.

Cada una de las refactorizaciones viene representada por un nombre pero cuando se pasa el ratón por encima de ese nombre se muestra una etiqueta con la descripción y la motivación de la propia refactorización.

- **Incorporación de las nuevas bibliotecas de MOON y JavaMOON:** Con la incorporación de las nuevas bibliotecas de MOON y JavaMOON se ha ampliado el rango de proyectos a los que se pueden aplicar las refactorizaciones, de proyectos pequeños con pocas clases a proyectos medianos-grandes con un gran número de clases. Esto además ha supuesto a su vez la necesidad de modificar en el plugin la carga de clases para poder realizar refactorizaciones sobre tipos que utilicen cualquier biblioteca del JDK. La modificación ha incrementado el consumo de memoria del plugin, lo que ha obligado a su vez a realizar otros ajustes que eran necesarios sobre ciertas partes del plugin. Por ejemplo, hemos tenido que realizar el cambio completo del sistema del historial de refactorizaciones y de deshacer refactorizaciones aplicadas que es explicado más detenidamente en el apartado 5.

- **Construcción del proyecto automatizada:** gracias a la introducción de Maven ahora el proyecto se construye con un sólo comando `mvn install`. Este proceso también ha permitido que el liberar nuevas versiones sea muy sencillo y que el siguiente punto sea posible: el plugin se puede instalar a través de Internet.
- **Instalación del plugin a través de Internet:** en esta última versión no es necesario instalar el plugin de forma manual a base de copiar carpetas. Sólo hace falta utilizar el instalador de plugins de Eclipse y conocer la dirección de descarga desde Internet del plugin. Esta nueva forma de instalación permite desinstalaciones más sencillas, la vuelta a versiones anteriores del plugin o *actualizaciones automáticas* cuando haya una nueva versión disponible del mismo. Por último se ha ampliado la ayuda con las nuevas funcionalidades del plugin y se ha incorporado una pantalla de bienvenida para que la primera vez que los usuarios se instalen el plugin tenga un acceso directo a la explicación de las características de la herramienta.
- **Versionado de las refactorizaciones:** con la incorporación de la posibilidad de obtener las actualizaciones que ofrecía el punto anterior se presentó el problema de cómo facilitar al usuario actualizaciones sobre las clasificaciones y las refactorizaciones. Esas no debían colisionar con las modificaciones que el usuario hubiera realizado sobre las clasificaciones y refactorizaciones a modo personal.

Para resolver este problema se tomó la decisión de definir dos tipos diferentes de clasificaciones y refactorizaciones: las de usuario y las del plugin.

Las últimas son propiedad exclusiva del plugin y no pueden ser modificadas bajo ningún motivo por el propio usuario. Cuando se obtiene una actualización del plugin esta podría traer nuevas versiones de los elementos, pero estos no podrían colisionar con las modificaciones del usuario porque los dos tipos son independientes. Por otro lado el usuario puede realizar cualquier tipo de modificación de sus propios elementos. Si se quiere modificar un elemento similar a uno proporcionado por el plugin se puede realizar una copia de éste y a partir de ahí realizar las modificaciones que se consideren necesarias.

De forma visual, se pueden identificar los elementos del plugin debido a que aparecen representados en la interfaz con un candado para indicar que no son

editables.

6.2. Desaparición de alternativas

Una de las estrategias que se siguió para decidir las características a implementar en el proyecto fue buscar puntos de extensión al plugin partiendo de las comparaciones realizadas en el proyecto anterior con otros ejemplos de software de refactorización. Las conclusiones que se obtuvieron no fueron muy alentadoras dado que las dos aplicaciones comparadas habían dejado de disponer de soporte. Las últimas versiones de RefactorIt [Aqrис n.d.] y ConTRaCT [Kniesel n.d.] son ambas previas a 2008.

No es necesario un estudio concienzudo de la situación para comprender que la principal razón de este declive de las alternativas independientes de plugins de refactorización se debe a que todos los grandes entornos de desarrollo disponen de un catálogo de refactorizaciones suficiente. Los programadores de los entornos de desarrollo no han sido ajenos a la conveniencia de las refactorizaciones en los procesos de desarrollo y sus soluciones han acabado ganando la partida. Probablemente la razón principal proviene del hecho de que cualquier usuario de su producto disponía de la refactorizaciones que iba a necesitar en la mayoría de los casos sin necesidad de instalar ningún plugin adicional. El usuario dispone de la confianza de que el soporte de las refactorizaciones va a mantenerse con el desarrollo del *IDE* y de que los menús de refactorización van a estar plenamente integrados con la interfaz del entorno. Por tanto las refactorizaciones van a estar siempre disponibles al usuario en el contexto más adecuado para su uso.

La alternativa a este monopolio de las soluciones proporcionadas por los entornos de desarrollo, en nuestro caso Eclipse, deben ser opciones que cubran deficiencias de los entornos de desarrollo u ofrezcan características diferenciadoras. Debido a que este parece el único camino viable, a continuación se van a estudiar una serie de desarrollos que han comprendido esta necesidad y ofrecen variantes funcionales dentro del ámbito del software de refactorización. Para cada uno de ellos la metodología consistirá en describir el enfoque adoptado, para luego comparar en qué aspectos confluye o diverge con nuestro plugin de refactorización valorando las ventajas y los inconvenientes de cada uno. Finalmente se va a comparar el plugin de refactorización con la solución *estándar* que pone a disposición Eclipse y se van a tratar de explicar las ventajas de que dispone por las que se considera que el plugin es una solución viable.

6.3. JDeodorant

JDeodorant [JDeodorant n.d.] es un plugin de Eclipse que identifica problemas de diseño en el software, los llamados *bad smells*, y los resuelve aplicando las refactorizaciones apropiadas.

Actualmente la herramienta identifica cuatro tipos de problemas de diseño: *envidia de características* o *Feature Envy*, comprobación de tipos o *Type Checking*, método excesivamente largo o *Long Method* y clase omnipotente o *God Class*.

- Los problemas causados por *envidia de características* son resueltos utilizando la refactorización *mover método*.
- Los problemas referentes a *comprobación de tipos* son resueltos con las refactorizaciones *reemplazar sentencia condicional por polimorfismo* y *reemplazar código de tipo por estado/estrategia*.
- Los problemas de *método excesivamente largo* son resueltos mediante refactorizaciones *extraer método*.
- Los problemas causados por *clase omnipotente* son resueltos a través de refactorizaciones *extraer clase*.

La herramienta es el resultado de la investigación en *Sistemas computacionales e ingeniería del software* del departamento de informática aplicada de la Universidad de Macedonia en Thessaloniki, Grecia.

6.3.1. Ventajas

Reconoce potenciales defectos en el software que podían ser desconocidos por el desarrollador sin necesidad de investigación previa por parte de éste.

Las refactorizaciones son propuestas en base a los problemas descubiertos, por tanto son refactorizaciones que en principio aseguran una mejora en el diseño del código.

Proporciona una vista de las métricas del código que puede sugerir el camino a seguir para otras posibles mejoras en el diseño.

6.3.2. Desventajas

La realidad es que el diseño y el código tienen sus propias especificidades que no se ven reflejadas a veces por las métricas. Ciertas métricas pueden indicar un punto del programa como propenso a errores cuando la lógica del código puede confirmar que ese diseño es apropiado.

En los casos apuntados anteriormente una solución en la que se permite al usuario, ya que es él quien comprende mejor el problema abordado por el software y las razones que han llevado al diseño actual, ser el motor de las acciones correctoras es más adecuado. Un usuario experto tendrá en cuenta más variables a la hora de tomar la decisión de llevar a cabo las refactorizaciones y por tanto la solución será mejor.

6.3.3. Conclusión

Por lo tanto se considera que lo que el usuario necesitará será un rango amplio de refactorizaciones disponibles, con información que permita comprender cada una de ellas, qué función desempeñan y a qué problemas responden. Esta información, junto con el conocimiento del usuario de su código, conforman la situación perfecta para que se tome la decisión más adecuada. El plugin de refactorizaciones resuelve estas dos variables con el conjunto de refactorizaciones de que dispone y la vista del catálogo de refactorizaciones y elementos de una refactorización. Además el plugin permite ampliar el catálogo de cambios definiendo refactorizaciones personalizadas.

6.4. Refactory

Refactory [Reimman n.d.] es un plugin para Eclipse que permite definir tus propias refactorizaciones independientes del lenguaje que llaman *refactorizaciones genéricas*. Este plugin surge como una solución de refactorización dentro de la programación dirigida por modelos, *Model Driven Software Development*, y permite por tanto refactorizaciones para cualquier tipo de lenguaje incluidos los lenguajes específicos de dominio, *DSL*.

En el desarrollo dirigido por modelos el cambio del código como artefacto principal a los modelos exige técnicas de refactorización genéricas. Las refactorizaciones deben poder ser reutilizadas para distintos metamodelos ya que a menudo éstas expresan las mismas acciones sólo que en contextos diferentes.

En Refactory se ha implementado una aproximación basada en roles que permite al

diseñador especificar un contexto para cada refactorización. Esto permite definir unos pocos modelos de refactorizaciones independientes del lenguaje que se pueden reutilizar en tantos metamodelos como se quiera.

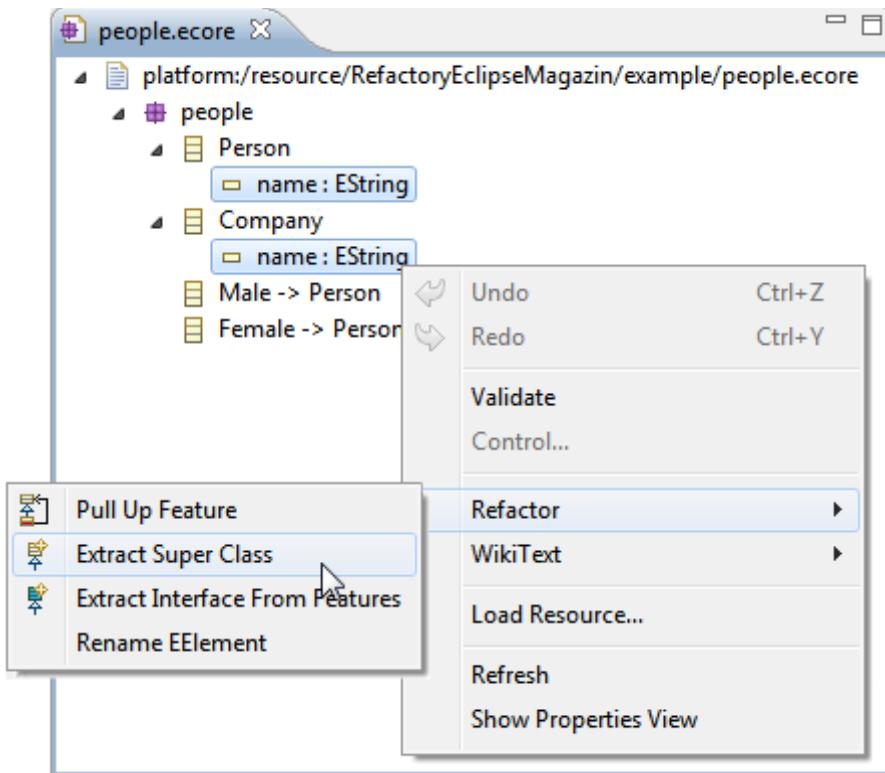


Ilustración 10: Refactory aplicando una refactorización a un metamodelo

Las refactorizaciones de los modelos se pueden integrar en el editor. En la imagen se muestra la representación de un modelo al que se va a aplicar la refactorización *extraer superclase* simplemente seleccionando los elementos a extraer y haciendo clic con el botón derecho del ratón.

6.4.1. Ventajas

La principal ventaja que dispone este plugin es que ofrece un enfoque totalmente independiente del lenguaje y por tanto permite definir refactorizaciones y luego adaptarlas para cualquier lenguaje que se desee. Esto permite utilizar esta herramienta a aquellos que deciden seguir el proceso de desarrollo basado en modelos y definen sus propios lenguajes específicos para su dominio (DSL). Con esta herramienta la creación de refactorizaciones para este tipo de lenguajes es más sencilla si se la compara con los pasos necesarios para definir una refactorización para un nuevo lenguaje en MOON.

6.4.2. Desventajas

Respecto al plugin de refactorización la principal desventaja de Refactory es que el plugin de refactorización está mucho más preparado para el trabajo con Java que es al fin y al cabo el lenguaje de trabajo de la mayoría de los usuarios de Eclipse. El plugin ya trae incorporadas las refactorizaciones de mayor interés para Java definidas y define elementos en su modelo como el parámetro formal de tipo, que posibilitan las refactorizaciones sobre clases y métodos genéricos. Refactory permite definir refactorizaciones exclusivamente sobre modelos, no sobre código, lo que imposibilita definir cualquiera de las refactorizaciones que en el plugin de refactorización se han definido bajo el ámbito de *code fragment*.

Además la definición de nuevas refactorizaciones para Java es más simple e intuitiva con el plugin de refactorización. Si un usuario quiere crear una refactorización un asistente va guiando paso a paso el proceso para introducir las entradas, las precondiciones, las poscondiciones y las acciones. Mientras que con Refactory es necesario conocer el modelo de refactorizaciones basado en roles definido en [Reimann, Seifert, & Aßmann 2010] .

6.5. RefactoringNG

RefactoringNG [RefactoringNG — Project Kenai n.d.] es una herramienta de refactorización general para Java. Las refactorizaciones se definen en un fichero que se compone de un conjunto de reglas de refactorización. Cada una de estas reglas describe una transformación de un árbol de sintaxis abstracta, *AST*, a otro y se compone de dos elementos principales: Patrón -> Transformación.

El patrón es un árbol *AST* del código fuente original y la transformación define los cambios que se aplicarán al patrón original. Por ejemplo, la siguiente regla transforma `p = null` en `p = 0`:

```
Assignment {
    Identifier [name: "p"],
    Literal [kind: NULL_LITERAL]
} ->
Assignment {
    Identifier [name: "p"],
```

```

    Literal [kind: INT_LITERAL, value: 0]
}

```

El patrón y la transformación cumplen con la misma estructura pues ambos se componen de: árbol, atributos y contenido.

6.5.1. Árbol, atributos y contenido

Los árboles toman el nombre de los AST y los atributos los de las propiedades del compilador para Java de Sun. Sólo el árbol es obligatorio, tanto los atributos como el contenido son descriptores opcionales. Los atributos están encerrados entre corchetes [] y están separados por comas. Éstos especifican información adicional del árbol. Por ejemplo, en:

Identifier

el atributo kind indica que el literal es el literal null. Dentro del patrón, si el atributo no es especificado, puede tomar cualquier valor. Por ejemplo:

```

Literal [kind:Identifier]

```

indica cualquier identificador y :

```

Literal [kind: INT_LITERAL]

```

indica cualquier literal int. En la transformación el árbol debe ser descrito completamente para que un nuevo árbol pueda ser creado. Por ejemplo, cada identificador en la transformación debe poseer el atributo name.

El contenido estará encerrado entre llaves {} y es una lista separada por comas de los hijos del nodo del árbol dado. Por ejemplo:

```

Binary [kind: PLUS] {
    Literal [kind: INT_LITERAL],
    Literal [kind: INT_LITERAL]
}

```

representa la suma de dos literales enteros. Los hijos de un árbol deben pertenecer a un tipo de los definidos por el árbol y todos ellos deben ser especificados si el árbol tiene contenido. Por ejemplo, si se define contenido en un operador binario debe tener siempre dos hijos (operandos) y ambos deben ser del tipo expresión. Si no se define el contenido del

operador binario, entonces los operandos pueden tener cualquier valor. Así:

```
Binary [kind: MINUS]
```

significa cualquier sustracción. El mismo árbol podría ser definido de la siguiente manera:

```
Binary [kind: MINUS] {  
    Expression,  
    Expression  
}
```

Como no se especifica ningún atributo de `Expression`, entonces `Expression` representa cualquier tipo de expresión. En aquellas posiciones en las que corresponde un árbol, cualquier subclase de un árbol puede ser utilizado. Por ejemplo, los operandos de `Binary` pueden ser cualquiera de las subclases de `Expression`.

```
Binary [kind: MULTIPLY] {  
    Identifier,  
    Literal [kind: INT_LITERAL, value: 0]  
}
```

La jerarquía de los árboles es la misma que en el compilador para Java de Sun.

Algunos atributos pueden tener más de un valor. En dichos casos, la lista de valores se define separando los elementos con el carácter '|'. Por ejemplo:

```
Binary [kind: PLUS | MINUS]
```

indica bien una suma o una sustracción.

Cada árbol en el patrón puede tener el atributo `id`. El valor de este atributo debe ser único en una regla dada y se utiliza para referirse a un árbol en la transformación. Por ejemplo:

```
Assignment {  
    Identifier [id: p],  
    Literal [kind: NULL_LITERAL]  
} ->  
Assignment {
```

```

Identifier [ref: p],
Literal [kind: INT_LITERAL, value: 0]
}

```

reescribe `p = null` a `p = 0` donde `p` se refiere a cualquier identificador.

Las referencias a los atributos se marcan con la almohadilla `#`. Por ejemplo, `b#kind` apunta al atributo `kind` de `b`. La referencia a un atributo se puede utilizar en las transformaciones como nuevo valor del atributo. Por ejemplo, para cambiar el orden de los operandos en una división se definiría de la siguiente manera:

```

Binary [id: b, kind: DIVIDE | REMAINDER] {

    Identifier [id: x],
    Identifier [id: y]

} ->

Binary [kind: b#kind] {

    Identifier [ref: y],
    Identifier [ref: x]

}

```

El valor especial `null` indica que el árbol no debe existir. Por ejemplo, la regla siguiente añade un valor inicial a las declaraciones de variable:

```

Variable [id: v] {

    Modifiers [id: m],
    PrimitiveType [primitiveTypeKind: INT],
    null

} ->

Variable [name: v#name] {

    Modifiers [ref: m],
    PrimitiveType [primitiveTypeKind: INT],
    Literal [kind: INT_LITERAL, value: 42]

}

```

6.5.2. Listas

Las listas utilizan la misma sintaxis que las listas genéricas en Java. `List<T>` es una lista de elementos del tipo `T`.

```
List<Expression>
```

Es un ejemplo de lista de expresiones. Una lista puede ser utilizada como parte de otro árbol o como nivel raíz. Para transformar un bloque vacío en un bloque con una sentencia vacía se haría de la siguiente manera:

```
Block {  
    List<Statement> { }  
}  
->  
  
Block {  
    List<Statement> {  
        EmptyStatement  
    }  
}
```

Y la regla que transforma una lista de cadenas en otra lista de cadenas distintas se definiría de la siguiente manera:

```
List<Expression> {  
    Literal [kind: STRING_LITERAL, value: "London"],  
    Literal [kind: STRING_LITERAL, value: "Paris"]  
}  
->  
  
List<Expression> {  
    Literal [kind: STRING_LITERAL, value: "Prague"]  
}
```

El atributo `size` especifica el número de elementos.

```
List<Literal> [size: 2]  
->  
List<Literal> {  
    Literal [kind: CHAR_LITERAL, value: '@']
```

```
}
```

Mientras que `minSize` y `maxSize` permiten especificar un rango de elementos:

```
List<Expression> [minSize: 2, maxSize: 3]
```

El valor * para el atributo `maxSize` indica que no existe un límite máximo.

```
List<Catch> [minSize: 2, maxSize: *]
->
List<Catch> {
    Catch {
        Variable [name: "e"] {
            Modifiers {
                List<Annotation> { },
                Set<Modifier> { }
            },
            Identifier [name: "Exception"],
            null
        },
        Block {
            List<Statement> { }
        }
    }
}
```

6.5.3. *NoneOf*

`NoneOf` indica que el árbol puede ser cualquier cosa excepto los árboles señalados. La regla a continuación reescribe la asignación sólo si el nombre de la variable no es `x` ni `y`.

```
Assignment {
    NoneOf<Expression> [id: i] {
        Identifier [name: "x"],
        Identifier [name: "y"]
```

```

    } ,

    Literal [kind: DOUBLE_LITERAL, value: 3.14]

} ->

Assignment {

    Expression [ref: i],

    MemberSelect [identifier: "PI"] {

        Identifier [name: "Math"]

    }

}

```

6.5.4. Conclusiones

La solución presentada por RefactoringNG es una solución flexible y potente en cuanto a que permite definir con mucho detalle los patrones a buscar y las transformaciones a realizar. Permite a los usuarios de Netbeans previsualizar los resultados de aplicar las transformaciones, otro aspecto también muy positivo.

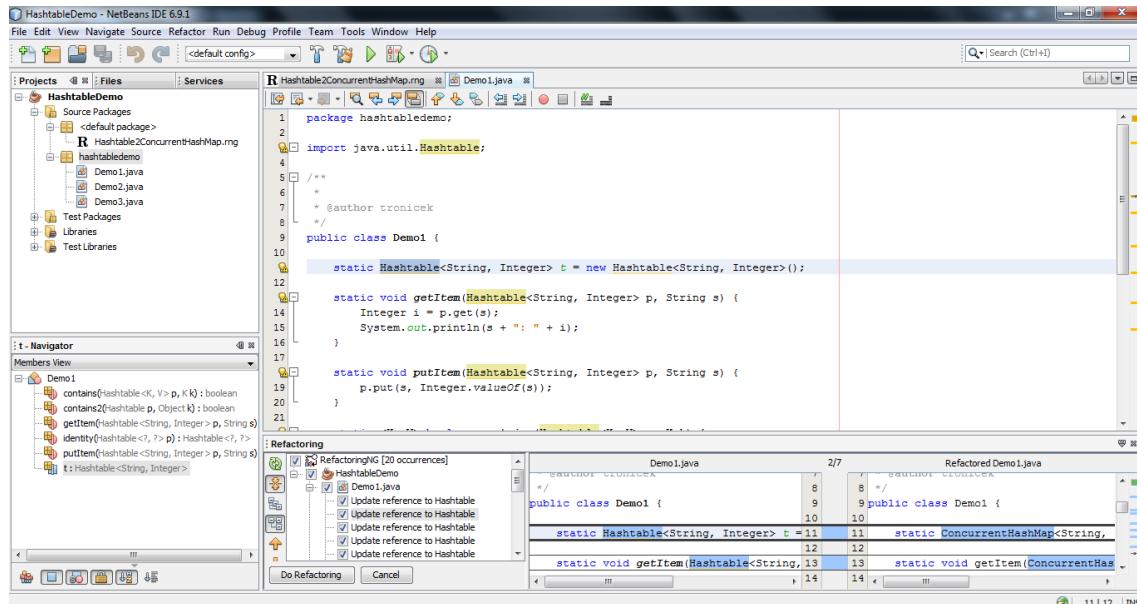


Ilustración 11: Previsualización de resultados con RefactoringNG

Sin embargo, siendo interesante la solución que esta herramienta promueve tiene ciertos defectos de fondo. Las refactorizaciones que se pueden definir carecen de la inteligencia que se suele exigir a las refactorizaciones definidas con la mayoría de herramientas. Esto es debido a que todo el modelo en que se basan es la transformación en

árbol AST del código fuente de una clase. Así estas refactorizaciones carecen de una visión global del proyecto completo, lo que hace muy complejo definir refactorizaciones que afecten a más de un fichero de código fuente.

Por ejemplo si se quiere renombrar un método se tiene que tener en cuenta todos las llamadas que a ese método se hacen desde otras clases, además de la posible existencia de subclases que sobrescribieran dicho método. De este modo se hace muy difícil crear una refactorización de este tipo sin provocar errores de compilación en los fuentes.

En definitiva, se puede considerar este plugin con una versión avanzada de una búsqueda y reemplazo basada en expresiones regulares basada en árboles AST. Al igual que ésta es potente y flexible pues permite definir transformaciones muy precisas, pero carece de la semántica necesaria para definir precondiciones y postcondiciones o para tener en cuenta las relaciones entre las clases de un modelo. Además su uso no es sencillo, pues la curva de aprendizaje del lenguaje en que se basan es prolongada.

6.6. API de refactorizaciones de Eclipse

Para comprender el funcionamiento del *API* de refactorizaciones de Eclipse primero se van a definir los pasos necesarios para ejecutar una refactorización. Esta visión general del proceso nos permitirá posteriormente comprender cuáles son las modificaciones más importantes a llevar a cabo para implementar una refactorización propia.

6.6.1. Pasos del proceso de refactorización

A continuación se presentan los pasos del proceso de refactorización en Eclipse:

- Paso 1: El usuario inicia la refactorización habitualmente lanzando la ejecución de una *acción* de Eclipse en la que su método principal `run()` se encarga de instanciar las clases principales del proceso. Las clases principales del proceso deben extender de `Refactoring`, la clase que definirá la refactorización, y `RefactoringWizard` que corresponde con la clase que define el wizard.

La información sobre lo que había seleccionado cuando el usuario disparó la ejecución de la refactorización es recuperada. Con esa selección actual, es posible determinar sobre qué elementos se deberá ejecutar la refactorización. Ese elemento seleccionado es almacenado y será posteriormente utilizado durante el proceso.

- Paso 2: El *LTK*, *Language Tool Kit*, de *Eclipse* asume el control. Primero se invoca el método llamado `checkInitialConditions()` en la propia instancia de la refactorización. Este método que ha debido de ser definido por la clase de la refactorización personalizada debe de comprobar que se cumplen las condiciones básicas para llevar a cabo dicha refactorización. Si este método encuentra algún problema, genera un objeto del tipo `RefactoringStatus` que contiene más información sobre los problemas encontrados. En dicho caso el *LTK* no continúa con la refactorización sino que aborta el proceso e informa al usuario de los problemas detectados a través de un cuadro de diálogo.
- Paso 3: Una vez determinado que no existen obstáculos fundamentales que impidan llevar a cabo la refactorización, se muestra el asistente que pide al usuario que introduzca la información adicional necesaria. La interfaz de usuario requerida se implementa como subclases de `UserInputWizardPage`. Los datos introducidos aquí son puestos a disposición de la instancia de la refactorización con la ayuda de un objeto `info` que juega el rol de modelo.
- Paso 4: Antes de que el asistente muestre la última página, que ofrece una vista detallada de los cambios a llevar a cabo ocurren dos cosas: El método `checkFinalConditions()` es invocado sobre la refactorización y los cambios a realizar sobre todos los ficheros involucrados son calculados desde el método `createChange()`. Los cambios a ejecutar se definen en una estructura de árbol cuya raíz es devuelta por el método `createChange()`. La construcción de este tipo de objetos es relativamente compleja y está fuera del ámbito de esta explicación. Para una introducción a la creación de cambios ver [Mätzel 2005] .
- Paso 5: Si el usuario no pulsa el botón *Finish* de forma inmediata, se muestra la última página del asistente; en ella el usuario puede ver los cambios que existen pendientes de ser ejecutados en detalle y descartar los que considere innecesarios. En este punto, ninguno de los cambios han sido efectuados todavía, estos se harán efectivos cuando sean confirmados definitivamente por el usuario.

6.6.2. Modificaciones necesarias más importantes

Por tanto los cambios necesarios más importantes para definir una refactorización

son:

- Definir una acción en Eclipse que se encargue de iniciar la refactorización.
- Definir extensión con la personalización de la clase `Refactoring`. La extensión personalizada debe definir el método `checkInitialConditions()` en el que se deberá comprobar si se cumplen las condiciones básicas para llevar a cabo la refactorización y además realizar la notificación en caso contrario. Se define el método `checkFinalConditions()` que es el encargado de comprobar que ciertas condiciones se cumplirán tras llevar a cabo el proceso y también se debe definir el método `createChange()` en el que se devuelven los cambios a ejecutar.
- Definir una extensión personalizada para la clase `RefactoringWizard`. Se encarga principalmente de gestionar las propias páginas del asistente, de la navegación del usuario por el mismo y también de invocar la acción específica a realizar cuando el proceso termina. Todas estas tareas ya son gestionadas por la superclase `RefactoringWizard`. Sin embargo es necesario proporcionar al asistente una referencia a la refactorización y añadir páginas adicionales, que derivarán de `UserInputWizardPage`, si la refactorización las exige.

6.6.3. Conclusiones

La definición de refactorizaciones con Eclipse como se puede ver es un proceso excesivamente complejo. Consta de una serie de pasos poco intuitivos que requieren que el usuario sea un programador con cierta experiencia en el desarrollo de plugins dentro del entorno de Eclipse. Esto limita mucho el rango de usuarios que pueden hacer uso del *API*.

Como aspectos positivos a destacar se encuentran la posibilidad de definir cierta inteligencia en las refactorizaciones con condiciones anteriores y posteriores a la ejecución de la refactorización y la muy interesante vista previa de los cambios que la refactorización aplica. La potencia del *API* tampoco debe ser ignorada, encontrándose la prueba de la misma en las refactorizaciones definidas por el propio entorno de Eclipse. Sin embargo, para sacar ventaja de estas virtudes el usuario necesita unos conocimientos que la mayoría de los programadores no dispone.

Es aquí donde se considera que nuestro plugin de refactorización tiene su nicho de mercado, en la gran mayoría de usuarios que no disponen de conocimientos tan avanzados

como el *API* de Eclipse exige, pero que también tienen interés de definir sus propias refactorizaciones y aplicarlas en sus proyectos. El plugin de refactorización permite a todos estos usuarios personalizar su catálogo de refactorizaciones de forma sencilla y con un mínimo aprendizaje.

7. CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURO

En este apartado se incluirán las conclusiones derivadas del desarrollo del presente proyecto. Asimismo, se comentarán aquellos aspectos que hayan podido quedar abiertos a posibles mejoras o extensiones futuras, a modo de líneas de trabajo futuro en el ámbito de este trabajo.

7.1. Conclusiones

Para obtener las conclusiones sobre el desarrollo del proyecto se va a partir de los objetivos que se plantearon para poder comprobar si estos han sido cumplidos, a partir de las características implementadas. Esta sección no pretende ser una lista detallada de todas las funcionalidades implementadas, estas están disponibles en el apartado 6.1.Mejoras incorporadas.

7.1.1. Requisitos funcionales

Entre los objetivos funcionales que se plantearon referentes a la navegación y la visualización de las refactorizaciones se ha implementado la siguiente funcionalidad:

- Sistema de clasificación de las refactorizaciones:
 - ◆ Un editor de clasificaciones que permite editar y crear clasificaciones propias.
 - ◆ Un árbol de selección de categorías de una refactorización en el asistente de edición de refactorizaciones. Además también se incluye un conjunto de palabras claves que identifican a una refactorización.
- Vista para facilitar la navegación: Se ha incluido una nueva vista donde las refactorizaciones se muestran clasificadas por categorías y se pueden filtrar, tal y como se había marcado en los objetivos, por múltiples criterios.

Además se puede seleccionar una refactorización para poder visualizar toda la información disponible sobre ella en un panel organizado por pestañas.

Dentro del objetivo de mejorar el asistente de refactorizaciones para facilitar su uso

se han incorporado las siguientes mejoras en cada una de las pantallas de asignación de entradas, precondiciones, acciones y postcondiciones:

- Se ha agregado un nuevo panel en el que se muestra la descripción del elemento seleccionado y una lista del conjunto de refactorizaciones en las que el elemento toma parte.
- Se ha sustituido el sistema de búsqueda de elementos por un sistema en el que se utiliza información de la documentación de los elementos en las búsquedas y que permite búsquedas avanzadas.

En el apartado de las mejoras en la instalación del plugin se ha conseguido generar el repositorio de instalación del plugin como parte del proceso de construcción automatizado. Finalmente se ha conseguido mediante un repositorio web hacer posible la instalación y actualización del plugin desde Internet utilizando la funcionalidad del instalador de Eclipse.

7.1.2. *Objetivos técnicos*

En el apartado de mejoras del *proceso de desarrollo* se han conseguido incorporar las siguientes mejoras:

- Todo el proceso de desarrollo se ha llevado a cabo utilizando como soporte el sistema de control de versiones Git y el repositorio web Github. Posteriores versiones del producto sólo tendrían que recurrir a dicho repositorio para retomar el desarrollo.
- Utilizando Maven se ha conseguido que todo el proceso de construcción del producto se lleve a cabo ejecutando el comando `mvn install`. Este proceso incluye: la descarga de las dependencias necesarias del plugin, la compilación, el empaquetado y el firmado de los ficheros *JAR*, la ejecución de las pruebas y la generación del repositorio de instalación.
- Como se citaba en el apartado anterior las pruebas se ejecutan como parte del proceso de construcción por tanto se ha conseguido convertirlas en requisitos que reducen el riesgo de que se entregue al usuario un producto con defectos.
- Parte del proceso de construcción del producto es la generación del informe de cobertura del código y los informes de métricas y de defectos del código.

Añadiendo al comando de construcción del plugin un sólo parámetro `mvn install sonar:sonar` se consigue que todas esas métricas queden reflejadas en una aplicación web que facilita su visualización.

- Se ha incorporado el servidor de integración continua Hudson y este ha sido configurado para que ejecute la construcción del producto a partir del contenido del repositorio del control de versiones en Github de forma diaria. Esto permite que incluso si los desarrolladores han olvidado ejecutar los tests sobre su última versión del código, Hudson lo haga y envíe un informe de error por correo electrónico en caso de que ocurra algún problema.

Dentro del objetivo de la mejora de la *calidad del código* lo positivo es que las variaciones se pueden reflejar de forma numérica. Estas han sido las variaciones en las principales mediciones:

- El número de tests se han incrementado de 173 a 297. Por tanto el número de test ha crecido en un 70% respecto a los tests creados en las dos versiones anteriores del producto.
- La cobertura de código de los tests ha crecido en un 20% dado que se ha pasado de un 38% inicial a un 59% final.
- La conformidad a los estándares del código, una métrica generada por Sonar que se basa en el número de defectos por línea de código y que se incrementa cuanto más conforme es un proyecto a las convenciones se ha incrementado del 57,3% al 78,4%. En términos absolutos el número de defectos del código ha descendido de 4.400 a 3.400 a pesar del incremento del tamaño del proyecto.
- Gracias al estudio de métricas se ha podido cerciorar que el incremento del 46% del tamaño de la parte correspondiente al plugin no a perjudicado al diseño, ya que este se mantiene respecto a la versión anterior. Además las métricas referentes a los valores medios se encuentran en los intervalos deseados. También se ha conseguido mejorar la profundidad media de bloque y un decremento en el numero medio de sentencias por método en la parte operativa. Además son significativos los valores obtenidos para la complejidad ya que esta medida en el histórico de proyectos de la asignatura es la que más a menudo presenta valores fuera del intervalo esperado.

7.2. Líneas de trabajo futuro

Un proyecto software nunca está terminado de forma absoluta. “El único estado estable de un proyecto es rigor mortis.” [Hunt 2000] Se puede considerar, por tanto, una señal positiva el que llegado el final del proyecto existan perspectivas de evolución del proyecto en forma de una lista de tareas pendientes de realizar. La lista de nuestras recomendaciones de mejora del plugin para versiones posteriores es la siguiente.

- El desarrollo de un repositorio *online* de refactorizaciones. Ahora mismo los usuarios pueden crear sus refactorizaciones pero no existe un mecanismo automatizado desde el que los usuarios tengan la posibilidad de proponer refactorizaciones a incluir en versiones posteriores del plugin. Este repositorio podría desarrollarse en forma de una página web. En ella, los usuarios podrían proponer sus refactorizaciones a añadir a la lista de refactorizaciones oficiales del plugin o crear sus propios grupos de refactorizaciones que otros usuarios pudieran descargar. En una evolución de esta idea el sitio podría ser una comunidad en la que los usuarios votaran por sus refactorizaciones favoritas y aportaran su conocimiento a la mejora del producto. Dos ventajas principales se derivarían de este sitio web: en primer lugar sería una forma sencilla y barata de ampliar el repositorio de refactorizaciones disponibles y en segundo lugar además se convertiría en una fuente inestimable de *feedback* a los desarrolladores a la hora de decidir las mejoras a implementar en versiones posteriores de la aplicación.
- Refrescado y actualización automática de la vista correspondiente al catálogo de refactorizaciones ante modificaciones en refactorizaciones. En la versión actual del plugin cuando un usuario edita una refactorización, la elimina o añade una refactorización nueva la vista `RefactoringCatalogBrowser` sigue mostrando el catálogo de refactorizaciones que había cuando se abrió. Si el usuario quiere que se muestre el catálogo de refactorizaciones actualizado necesita pulsar en el botón de refrescar. La mejora sugerida consistiría en que el propio catálogo centralizado de refactorizaciones incorporado en esta última versión del plugin, comunicara a la vista cuando hay un cambio para que esta se actualice automáticamente.
- Internacionalización del mecanismo de búsqueda de entradas, predicados y acciones. Como ya se ha visto en varios apartados del proyecto una de las mejoras incorporadas en esta versión del plugin ha sido la incorporación en el

asistente de creación de refactorizaciones de un panel informativo y además un avanzado sistema de búsqueda basado en la biblioteca Lucene [Apache Lucene n.d.]. Sin embargo debido a que el *API* de las bibliotecas MOON y JavaMOON estaba escrito en inglés, tanto la descripción como la búsqueda se basan en dicho idioma, una mejora a considerar para siguientes versiones de la aplicación es la de internacionalizar la documentación tanto de MOON como de JavaMOON como de los elementos del repositorio y las refactorizaciones pasándola al español para permitir a usuarios sin conocimientos de inglés sacar el máximo provecho a esta funcionalidad.

- Perfeccionamiento del mecanismo de búsqueda. A pesar de que la búsqueda en el wizard ya incorpora elementos avanzados como el análisis de las raíces de los términos indexados, *stemming*, y la eliminación de palabras sin valor, *stop words*, hay muchas posibilidades de mejora del mecanismo para hacerlo más inteligente. Una de estas mejoras a incorporar podrían ser los sinónimos de términos.
- Incorporación del ámbito de paquete al plugin: ya que la incorporación del ámbito de paquete permitiría definir refactorizaciones que afectaran a todas las clases de un paquete y de sus subpaquetes. Así se podría por ejemplo pedir al plugin que agregaría la anotación `Override` a todos los métodos de un paquete que lo necesitaran con un sólo clic.
- Ahora mismo cuando se exporta una refactorización, se exporta con las categorías a las que pertenece. Sin embargo cuando la refactorización se va a importar en otra instalación de Eclipse es necesario que esa instalación tenga definida todas las clasificaciones de las categorías a las que la refactorización a importar pertenece. En futuras versiones del plugin se debería automatizar ese proceso para facilitar las exportaciones al usuario.
- La mejora en la usabilidad del plugin también podría proponerse como una mejora del plugin. Si bien se ha tratado de tener en cuenta al usuario a la hora de diseñar todos los aspectos del plugin, no es lo mismo hacerlo desde la perspectiva de alguien que conoce el producto desde dentro que hacer pruebas con usuarios reales. Siendo así se podría plantear para futuros proyectos un estudio de usabilidad del producto realizado en colaboración con los profesores. Estos harían llegar el producto a sus alumnos de distintos niveles para que lo probarán y después contestaran a una serie de preguntas.

- Previsualización de los resultados de aplicar una refactorización. El usuario se vería muy beneficiado si tuviera la oportunidad de previsualizar los cambios que va a efectuar una refactorización al ser aplicada.

Universidad de Burgos
ESCUELA POLITÉCNICA SUPERIOR
INGENIERÍA INFORMÁTICA



Anexo 1. Plan del Proyecto Software

Clasificación de Refactorizaciones
Dynamic Refactoring Plugin 3.0

Alumnos:
Míryam Gómez San Martín
Íñigo Mediavilla Saiz

Tutor:
Raúl Marticorena Sánchez

Burgos, Julio de 2011

ANEXO I
PLAN DEL PROYECTO SOFTWARE

ÍNDICE DE CONTENIDO

Anexo I	
Plan del Proyecto Software.....	95
Lista de cambios.....	101
1. INTRODUCCIÓN.....	103
2. GESTIÓN DE RIESGOS.....	104
2.1. Clasificación de la gestión de riesgos.....	104
2.2. Elementos de la gestión de riesgos.....	105
2.3. Estimación de riesgos asociados al proyecto.....	108
2.3.1. Identificación de riesgos.....	108
2.3.2. Análisis de riesgos.....	109
2.3.3. Evaluación de riesgos.....	110
2.4. Control de riesgos asociados al proyecto.....	111
2.4.1. Planificación de la gestión de riesgos.....	111
2.4.2. Supervisión de riesgos.....	112
3. PLANIFICACIÓN TEMPORAL.....	113
3.1. Planificación inicial del proyecto.....	113
3.2. Planificación final del proyecto.....	119
3.3. Desviaciones en la planificación inicial.....	123
3.4. Descripción de las tareas.....	124
3.4.1. Estudio de la documentación previa.....	124
3.4.2. Enumeración de requisitos.....	124
3.4.3. Planificación temporal.....	124
3.4.4. Estudio preliminar de costes.....	125
3.4.5. Análisis de riesgos.....	125
3.4.6. Estudio de herramientas a utilizar.....	125
3.4.7. Requisitos tecnológicos.....	125
3.4.8. Traspaso del plugin a versión 3.5 de Eclipse.....	126
3.4.9. Incorporación de última versión de bibliotecas.....	126
3.4.10. Iteración 1.....	126
3.4.11. Iteración 2.....	127
3.4.12. Iteración 3.....	127
3.4.13. Iteración 4.....	128
3.4.14. Iteración 5.....	128
3.4.15. Iteración 6.....	129
3.4.16. Iteración 7.....	129
3.4.17. Documentación.....	130
4. ESTUDIO DE VIABILIDAD.....	130
4.1. Viabilidad técnica.....	131
4.2. Viabilidad legal.....	131

<u>4.3. Viabilidad económica.....</u>	<u>133</u>
<u>4.3.1. Análisis de costes.....</u>	<u>133</u>
<u>4.3.2. Análisis coste-beneficio.....</u>	<u>138</u>
<u>4.4. Viabilidad temporal.....</u>	<u>144</u>
<u>4.4.1. Estimación de Casos de Uso.....</u>	<u>144</u>

ÍNDICE DE ILUSTRACIONES

Ilustración 12: Diagrama de Gantt de la planificación inicial del proyecto.....	117
Ilustración 13: Diagrama de Gantt de la planificación final del proyecto.....	122

ÍNDICE DE TABLAS

Tabla 2: Categorías definidas para la clasificación de riesgos.....	109
Tabla 3: Escalas de medida definidas para la evaluación del análisis de riesgos.....	110
Tabla 4: Tabla de riesgos del proyecto.....	110
Tabla 5: Tabla de riesgos del proyecto priorizada.....	111
Tabla 6: Planificación inicial del proyecto.....	116
Tabla 7: Planificación final del proyecto.....	121
Tabla 8: Relación de licencias de las herramientas utilizadas.....	132
Tabla 9: Coste de las herramientas utilizadas.....	136
Tabla 10: Costes del proyecto.....	137
Tabla 11: Precios de las licencias comerciales de IntelliJIDEA.....	140
Tabla 12: Relación entre el precio y el número de licencias de la aplicación.....	141
Tabla 13: Valores utilizados para el cálculo del VAN.....	143
Tabla 14: Peso de los casos de uso.....	145
Tabla 15: Peso de los factores técnicos.....	146
Tabla 16: Peso de los factores de entorno.....	147

LISTA DE CAMBIOS

Número	Fecha	Descripción	Autor/es
0	11/04/11	Primera versión con introducción y comienzo de la planificación temporal.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
1	23/04/11	Planificación temporal definitiva, desviaciones y primera versión del estudio de viabilidad.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
2	08/05/11	Incorporada introducción y gestión de riesgos.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
3	09/05/11	Descripción de tareas y viabilidad legal y técnica.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
4	25/05/11	Incluida viabilidad temporal y económica.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
5	06/06/11	Revisión final e incluidos pequeños detalles.	Míryam Gómez San Martín Íñigo Mediavilla Saiz

1. INTRODUCCIÓN

El Plan de Proyecto Software tiene como objetivo inicial establecer la identificación y el análisis de riesgos asociados al desarrollo software, los cuales pueden comprometer el futuro del proyecto y su desarrollo en unos plazos aceptables, es por esta razón que se han de tener siempre presentes. Finalmente, su propósito es el de proporcionar la información necesaria para llevar un control sobre el desarrollo del proyecto software.

Durante el proceso de desarrollo de un producto *software* una de las tareas más importantes que debe llevarse a cabo es la planificación, ya que esta se puede tomar como base para la toma de decisiones. El objetivo final de la planificación es determinar el tiempo necesario para la realización del proyecto con el fin de poder ajustarse lo máximo posible a este. Además ayuda a determinar los recursos necesarios y los costes que la elaboración del mismo lleva asociados consigo.

Realizaremos el estudio de la planificación teniendo en cuenta los aspectos que se van a indicar a continuación:

- Planificación temporal del proyecto: determina el calendario con el que vamos a poder representar de forma gráfica todas las tareas a realizar para la obtención del producto final. Se mostrará tanto la planificación inicial como el resultado final, con el objetivo de poder observar la evolución que esta ha sufrido. Asimismo se analizarán los aspectos más relevantes que hayan llevado a una alteración de dicha planificación.
- Estudio de la viabilidad del proyecto desde tres puntos de vista principalmente: el punto de vista temporal, el punto de vista económico y el punto de vista legal. El estudio de la viabilidad económica estará fundamentado en la técnica de análisis coste/beneficio. En el estudio correspondiente a la viabilidad legal se explicará y justificará la licencia elegida para la distribución del producto software.

2. GESTIÓN DE RIESGOS

Los objetivos de la gestión de riesgos son identificar, controlar y eliminar las fuentes de riesgo, antes de que empiecen a afectar al cumplimiento de los objetivos del proyecto. Es por ello que es muy importante el análisis de riesgos con el fin de identificar y poder evaluar la probabilidad de que estos ocurran para estimar el impacto, clasificarlos según la importancia de los mismos y establecer con ello un plan de contingencia en caso de que el problema se presente.

Los riesgos involucran siempre dos características, estas son las siguientes:

Incertidumbre: el acontecimiento que caracteriza al riesgo puede o no ocurrir.

Pérdida potencial: si el riesgo termina por convertirse en una realidad, ocurrirán las consecuencias no deseadas o pérdidas.

Para poder cuantificar el nivel de incertidumbre y el grado de pérdidas asociado con cada riesgo se consideran diferentes categorías de riesgos, a continuación serán detalladas.

2.1. Clasificación de la gestión de riesgos

En función del nivel de incertidumbre y el grado de pérdidas asociado los riesgos se pueden clasificar en las siguientes categorías:

Riesgos del Proyecto

- Amenazan al Plan del Proyecto, es decir, afectan a la planificación temporal, al coste y a la calidad del proyecto.
- Identifican problemas potenciales de presupuesto, de calendario, de personal, de recursos, de cliente, etc.

Riesgos Técnicos

- Amenazan la calidad y la planificación temporal del producto software que se producirá.
- Identifican posibles problemas de incertidumbre técnica, ambigüedad en la especificación, en diseño, implementación, obsolescencia técnica o tecnología

puntera, interfaz, verificación y mantenimiento, etc.

Riesgos del Negocio

- Amenaza la viabilidad del software que se construirá.
- Los principales riesgos de negocio son:
 - riesgo de mercado: cuando el producto es demasiado bueno.
 - riesgo estratégico: en el caso de productos que no encajan.
 - riesgo de ventas: si el producto es poco vendible.
 - riesgo de presupuesto: cuando el producto esté fuera de presupuesto.

Se puede hacer otra categorización de los riesgos diferente en función de la facilidad que tienen para ser detectados. En este caso existen los siguientes:

Riesgos conocidos: son aquellos que se pueden predecir después de realizar una evaluación del plan del proyecto, del entorno técnico y de otras fuentes fiables de información.

Riesgos predecibles: cuando es posible realizar la extrapolación de la experiencia de proyectos anteriores.

Riesgos impredecibles: son aquellos que pueden ocurrir, pero es extremadamente difícil identificarlos por adelantado.

2.2. Elementos de la gestión de riesgos

La gestión continuada de los riesgos permite aumentar su eficiencia, para que esto sea posible tendremos que evaluar continuamente lo que pueda ir mal, determinar qué riesgos son importantes para implementar estrategias para resolverlos en caso de que se produjesen y asegurarnos de la eficacia de dichas estrategias.

Los elementos de la gestión de riesgos son los siguientes:

Estimación de riesgos

- *Identificación de riesgos:* lista de riesgos potenciales que pueden afectar a la planificación del proyecto.

Las incertidumbres sobre diferentes características del proyecto que puedan

existir se transforman en riesgos que pueden ser descritos y medidos. Un método para identificar los riesgos es crear una lista de comprobación de elementos de riesgo que podría contener dos categorías de riesgos:

Riesgos específicos del producto: se los puede identificar con un buen conocimiento de la tecnología, el personal y el entorno específico del software, para ello se debe examinar el plan del proyecto y la declaración del ámbito del software.

Riesgos genéricos: son comunes a todos los proyectos de software. Para identificarlos se crean las siguientes subcategorías:

- Tamaño del producto.
 - Impacto en el negocio.
 - Características del cliente.
 - Definición del proceso.
 - Entorno de desarrollo.
 - Tecnología a construir.
 - Tamaño y experiencia de la plantilla.
- *Análisis de riesgos:* medición de la probabilidad y el impacto de cada riesgo, y los niveles de riesgo de los métodos alternativos.

Es el proceso de examinar los riesgos en detalle para determinar su extensión, sus interrelaciones y su importancia.

Las actividades básicas son:

Evaluación: mejor comprensión del riesgo.

Se cuantifican, en lo posible, los siguientes conceptos:

Impacto: pérdida que ocasiona el riesgo. Consecuencias de los problemas asociados con el riesgo. Los factores que afectan al impacto son:

- naturaleza: problemas potenciales que se pueden producir en caso de ocurrir.

- alcance: combina la severidad con su distribución global.
- duración: combina el momento en el que se sentirá su impacto y la duración del mismo.

Probabilidad: probabilidad de que ocurra el riesgo.

Marco de tiempo: periodo de tiempo en el que es posible mitigar el riesgo.

Clasificación: se clasifican los riesgos para entender su naturaleza y elaborar planes de mitigación.

- *Evaluación de riesgos:* lista de riesgos ordenados por su impacto y por su probabilidad de ocurrencia para determinar cuáles se deben solucionar antes y a cuáles hay que asignarles más recursos.

Los riesgos pueden ordenarse según la magnitud de la exposición al riesgo:

$$[r_i, l_i, x_i]$$

donde;

r_i : riesgo.

l_i : probabilidad del riesgo.

x_i : magnitud del impacto del riesgo.

Las condiciones y prioridades pueden cambiar a lo largo del proyecto por lo que es importante destacar que el análisis y la asignación de prioridades debe realizarse de manera continua aprovechando la información disponible en cada momento.

Control de riesgos

- *Planificación de la gestión de riesgos:* plan para tratar cada uno de los riesgos significativos.
- *Supervisión de riesgos:* comprobación del progreso del control de un riesgo e identificación de la aparición de nuevos riesgos.

2.3. Estimación de riesgos asociados al proyecto

Este apartado esta dedicado a la estimación de los riesgos asociados a nuestro proyecto, en él se identificarán dichos riesgos para posteriormente pasar a ser analizados y poder así evaluarlos y clasificarlos con el objetivo final de poder establecer un orden de prioridades sobre los mismos.

2.3.1. Identificación de riesgos

A continuación se enumera una serie de riesgos que han surgido al evaluar futuros problemas y contingencias del proyecto a tratar. Dichas dificultades, surgen de las distintas reuniones con el tutor del proyecto y de una propia autocrítica de las personas involucradas, considerando sobre todo qué aspectos eran más negativos y en qué proporción afectaban al desarrollo satisfactorio del proyecto.

- Riesgo 1: El equipo de desarrollo tiene escasa experiencia en el proyecto a desarrollar (desarrollo de plugin, utilización de nuevas herramientas, bibliotecas de Eclipse y bibliotecas externas necesarias) esto hace que se requiera una inversión significativa en la curva de aprendizaje y en la propia formación de los desarrolladores.
- Riesgo 2: La planificación temporal estimada al principio del proyecto puede ser irreal o la fecha de entrega estar muy ajustada, pudiendo acarrear retrasos considerables en el desarrollo del proyecto.
- Riesgo 3: Se podría producir cambios en los requisitos iniciales por ser estos inadecuados (características que funcionan de forma distinta a como se esperaba, características necesarias sobre las que nadie pensó) teniendo un efecto muy negativo en etapas avanzadas de desarrollo.
- Riesgo 4: Se produce un solapamiento temporal con otras actividades que los desarrolladores implicados realizan, como son: trabajo, cursos, actividades personales, etc.
- Riesgo 5: Limitaciones en cuanto a disponibilidad de documentación referente al desarrollo de plugins para Eclipse (herramientas, bibliotecas, etc).
- Riesgo 6: La versión del plugin sobre la que se parte para realizar el desarrollo de este proyecto puede presentar fallos y características que no funcionan

como se esperan y que van a ser localizadas durante el desarrollo del mismo y tendrán que ser corregidas.

- Riesgo 7: Necesidad de realizar modificaciones internas durante el desarrollo del proyecto en la aplicación que afecten al propio diseño de la misma y que destaque por su complejidad.

2.3.2. Análisis de riesgos

Una sencilla técnica para llevar a cabo el análisis de riesgos será la utilización de una *tabla de riesgos*. En la primera columna de esta tabla se listarán todos los riesgos del proyecto que han sido identificados en el apartado anterior. La segunda columna servirá para clasificar cada uno de estos riesgos según la categoría a la que pertenecen. Por último, las siguientes dos columnas se utilizarán para valorar la probabilidad y el impacto de los diferentes riesgos registrados en la primera columna de la tabla. Opcionalmente se podría añadir una última columna para determinar el marco de tiempo asociado al riesgo, es decir, el periodo de tiempo en el que es posible mitigar el riesgo.

Una vez definida la estructura de la *tabla de riesgos* daremos paso a indicar las categorías a utilizar para la clasificación de los riesgos y a definir la escala de medidas a utilizar.

Para la clasificación de los riesgos vamos a utilizar las categorías que a continuación se muestran y que hemos definido de la siguiente forma:

Categoría	Descripción
PS	Tamaño del producto
BU	Impacto en el negocio
TE	Tecnología
ST	Personal
DE	Entorno de desarrollo

Tabla 2: Categorías definidas para la clasificación de riesgos

Para la especificación de estas medidas hemos elegido los términos cualitativos que a continuación se indican, junto con su correspondiente interpretación cuantitativa, todo ello se recoge en la tabla siguiente.

Atributo	Valor	Descripción
Probabilidad	Muy Probable	>70%
	Probable	entre 30% y 70%
	Improbable	<30%
Impacto	Catastrófico	Pérdida del sistema. Coste >50%
	Crítico	Recuperación de la capacidad operativa. Coste >10% (Coste <50%)
	Marginal	Coste <10%
	Despreciable	Coste ≈ 0
Marco de Tiempo	Corto Plazo	15 días
	Medio Plazo	de 1 a 2 meses.
	Largo Plazo	más de 2 meses.

Tabla 3: Escalas de medida definidas para la evaluación del análisis de riesgos

A partir de los riesgos identificados en el apartado anterior y en base a los criterios que se acaban de definir, a continuación se presenta la *tabla de riesgos* relativa a nuestro proyecto:

Riesgos	Categoría	Probabilidad	Impacto
Riesgo 1	DE y ST	Muy Probable (80%)	Crítico
Riesgo 2	BU	Improbable (20%)	Marginal
Riesgo 3	PS	Muy Probable (70%)	Marginal
Riesgo 4	ST	Probable (30%)	Despreciable
Riesgo 5	TE	Muy Probable (80%)	Crítico
Riesgo 6	DE y PS	Probable (40%)	Marginal
Riesgo 7	DE y PS	Probable (50%)	Marginal

Tabla 4: Tabla de riesgos del proyecto

2.3.3. Evaluación de riesgos

Una vez que ya se dispone de la *tabla de riesgos* deberemos de ordenarla por probabilidad y por impacto, de tal forma que los riesgos de elevada probabilidad y alto impacto pasen a las primera filas de la tabla, mientras que los riesgos de baja probabilidad caerán a la parte inferior de la misma. Esto nos permitirá establecer un orden de prioridades sobre los riesgos asociados al proyecto, lo que facilitará la definición de una linea de corte

para determinar qué riesgos deberán seguir gestionándose y cuáles no.

A continuación se muestra la *tabla de riesgos* relativa a nuestro proyecto ordenada en función de las prioridades sobre los riesgos asociados al proyecto:

Riesgos	Categoría	Probabilidad	Impacto
Riesgo 1	DE y ST	Muy Probable (80%)	Crítico
Riesgo 5	TE	Muy Probable (80%)	Crítico
Riesgo 3	PS	Muy Probable (70%)	Marginal
Riesgo 7	DE y PS	Probable (50%)	Marginal
Riesgo 6	DE y PS	Probable (40%)	Marginal
Riesgo 4	ST	Probable (30%)	Despreciable
Riesgo 2	BU	Improbable (20%)	Marginal

Tabla 5: Tabla de riesgos del proyecto priorizada

2.4. Control de riesgos asociados al proyecto

Las actividades que conforman la gestión de riesgos del proyecto descritas hasta ahora tienen un único objetivo final y este no es otro que el de ayudarnos a desarrollar una estrategia para tratar eficazmente los riesgos identificados. Esto conlleva la consideración de tres aspectos fundamentales: evitar el riesgo siempre que resulte posible, supervisar el riesgo, y por último gestionar el riesgo y establecer unos buenos planes de contingencia. A continuación detallaremos cada uno de estos aspectos.

2.4.1. Planificación de la gestión de riesgos

En nuestro caso hemos adoptado un enfoque proactivo frente al riesgo, evitando el riesgo como la mejor estrategia a seguir. Esto se consigue definiendo un plan de reducción del riesgo. Este plan contará con los siguientes aspectos:

- Utilizar una herramienta para el control de tareas con el fin de identificar cada tarea, la persona a la que está asociada en cada momento, así como reflejar en ella el tiempo estimado para su finalización y también el tiempo empleado, permitiendo la trazabilidad y medición de la misma. Con ella además de realizar la gestión de tareas también se realizará lo propio con la gestión de errores.
- Aumentar la batería de test disponibles para la aplicación con el fin de cubrir

el mayor número de casos que se puedan dar, con el objetivo de garantizar la consistencia y compatibilidad funcional del programa. También sería adecuado realizar test gráficos y pruebas de estrés.

- Definir un método de estimación de tareas, por ejemplo para cada tarea se estimará el mejor caso, el peor y el más probable.
- También se deberán recoger datos históricos para ver la evolución del trabajo (por ejemplo, mediante el estado de las tareas: abierta, en progreso, en stand by, terminada) así como también de los errores registrados (si se han detectado por los programadores y en qué momento –revisión, compilación, pruebas-, o si lo ha detectado el usuario). Gracias a ellos podemos estimar mucho mejor ya que nos podemos basar en experiencias pasadas, de igual modo nos permitirá justificar planificaciones.
- Definir plantillas de documentación y establecer mecanismos para asegurar que los documentos se vayan generando puntualmente.
- Realizar una buena documentación técnica para que aquel que desee realizar cambios futuros en la aplicación encuentre en ella una buena fuente de información, siendo de gran ayuda debida a la escasa documentación que existe sobre el desarrollo de plugins de Eclipse.
- Llevar a cabo revisiones en equipo del proyecto con el fin de que todas las personas, alumnos y tutor, vayan siguiendo el proceso y la evolución del mismo. Con ello conseguiremos poner al día la planificación y determinar la velocidad del proyecto. Además también será importante que cada uno de ellos deje sus impresiones del proyecto con el propósito de realizar una mejora continua del mismo.

2.4.2. Supervisión de riesgos

La supervisión de riesgos [Pressman 2006] asociados a nuestro proyecto supone:

- Detectar la ocurrencia de un riesgo que haya sido previsto.
- Asegurar que los pasos de reducción definidos para cada riesgo se estén aplicando correctamente.

- Identificar la aparición de nuevos riesgos que no hayan sido previstos.
- Recopilar información con el fin de que esta pueda ser de utilidad de cara al análisis de nuevos riesgos que se detecten o de riesgos de futuros proyectos.

3. PLANIFICACIÓN TEMPORAL

En este apartado se recoge la planificación temporal del proyecto, es decir, la elaboración del calendario o programa de tiempos. El calendario del proyecto es una representación gráfica de todas las actividades del proyecto, necesarias para producir el resultado final, permitiendo al desarrollador del proyecto coordinar de forma efectiva el transcurso del mismo.

De forma general se puede ver la planificación temporal como la división del proyecto en actividades, cada una de las cuales va acompañada de una estimación del tiempo destinado a su realización. El objetivo de la planificación temporal es coordinar estas tareas.

En este apartado se muestra la planificación temporal que hemos empleado para el desarrollo del presente proyecto. Para ello hemos utilizado la técnica gráfica de *Diagramas de Gantt*, que se trata de una representación gráfica de las tareas sobre una escala de tiempos. Las tareas se representan en forma de barra sobre dicha escala manteniendo la relación de proporcionalidad entre sus duraciones, su representación gráfica y su posición respecto del punto origen del proyecto.

A continuación veremos tanto la planificación inicial, estimada al comienzo del proyecto, como la evolución temporal del proyecto, es decir, el resultado final.

3.1. Planificación inicial del proyecto

La primera planificación temporal se realizó después de las primeras reuniones con el tutor en las que se especificaron los objetivos del proyecto en forma de requisitos, tanto funcionales como no funcionales.

La intención al elaborar esta primera planificación temporal fue la de no dejar olvidada ninguna de las tareas principales. En cuanto a la granularidad se trató de que las tareas se especificaran de una manera suficientemente exhaustiva como para no cometer desviaciones muy graves en su acotación temporal.

El proceso de desarrolló se decidió que se conformara en base a ciclos conocidos como iteraciones. Al final de cada ciclo se realizaría una reunión con el tutor en el que se revisarían las tareas implementadas y se corregirían aspectos que el tutor junto con los alumnos consideraran mejorables o erróneos. Este modelo de iteraciones similar al postulado por algunas metodologías ágiles [Schwaber 2004] se consideró el más adecuado para el desarrollo. Es un modelo que facilita estar centrado exclusivamente en un conjunto de tareas lo que mejora el desarrollo de las mismas. Las tareas a desarrollar deben ser las de más valor para el proyecto así se facilita la priorización y visibilidad del mismo de cara al usuario. Finalmente, la revisión periódica de la evolución del proyecto evita que el desarrollo se desvíe de los intereses principales del usuario y permite al usuario aportar nuevas sugerencias de forma temprana.

En la primera planificación se optó por posponer la elaboración del grueso de la documentación para el final del proyecto. Esta decisión se tomó debido a que se consideró que la mayoría de los aspectos a documentar no estarían definidos hasta el final del proyecto y escribirlos desde el principio supondría trabajo repetido.

Tarea	Duración	Fecha Inicio	Fecha Fin
Estudio de documentación previa	3 días	07/01/2011	12/01/2011
Enumeración de requisitos funcionales	2 días	12/01/2011	14/01/2011
Planificación temporal	1 días	14/01/2011	15/01/2011
Estudio preliminar de costes y de viabilidad	1 días	17/01/2011	18/01/2011
Estudio de herramientas a utilizar	1 días	18/01/2011	19/01/2011
Traspaso del plugin a versión 3.5 de Eclipse	2 días	19/01/2011	21/01/2011
Incorporación de última versión de bibliotecas	2 días	21/01/2011	25/01/2011
Iteración 1	9 días	26/01/2011	08/02/2011
Separación del proyecto de tests	3 días	26/01/2011	29/01/2011
Automatización de la construcción del proyecto	5 días	31/01/2011	05/02/2011
Configuración de proceso de build diario	1 días	07/02/2011	08/02/2011
Prototipado de vista de catálogo de refactorizaciones	1 días	07/02/2011	08/02/2011
Iteración 2	9 días	09/02/2011	22/02/2011
Implementación del modelo del dominio de la clasificación de refactorizaciones	4 días	09/02/2011	15/02/2011
Filtrado de Clasificaciones	5 días	15/02/2011	22/02/2011
Actualización del XML de las refactorizaciones	3 días	15/02/2011	18/02/2011
Iteración 3	9 días	23/02/2011	08/03/2011
Versión de línea de comandos del catálogo de clasificaciones	2 días	23/02/2011	25/02/2011
Primera versión de la vista del catálogo de refactorizaciones en Eclipse	4 días	25/02/2011	03/03/2011
XML de las clasificaciones	3 días	03/03/2011	08/03/2011
Iteración 4	9 días	09/03/2011	22/03/2011
Instalación de plugin desde Internet	2 días	09/03/2011	11/03/2011
Editor de clasificaciones desde Eclipse	3 días	11/03/2011	16/03/2011
Pestañas con información adicional en vista del catálogo	3 días	16/03/2011	19/03/2011
Prototipado de la ayuda a la creación de refactorizaciones	1 días	21/03/2011	22/03/2011

Iteración 5	9 días	22/03/2011	02/04/2011
Debate y prototipado sobre versiones de las refactorizaciones	2 días	22/03/2011	24/03/2011
Implementación de la obtención de la metainformación de entradas, predicados y acciones	4 días	24/03/2011	30/03/2011
Implementación de la búsqueda avanzada en la ayuda	3 días	30/03/2011	02/04/2011
Iteración 6	9 días	05/04/2011	16/04/2011
Implementación de versionado de refactorizaciones	4 días	05/04/2011	09/04/2011
Implementación de versionado de clasificaciones	3 días	11/04/2011	14/04/2011
Implementación de previsualización de resultados de aplicar refactorización	2 días	14/04/2011	16/04/2011
Documentación	28 días	18/04/2011	26/05/2011
Manual de usuario	5 días	18/04/2011	23/04/2011
Documentación técnica	6 días	25/04/2011	03/05/2011
Memoria principal del proyecto	12 días	03/05/2011	19/05/2011
Anexos	5 días	19/05/2011	26/05/2011

Tabla 6: Planificación inicial del proyecto

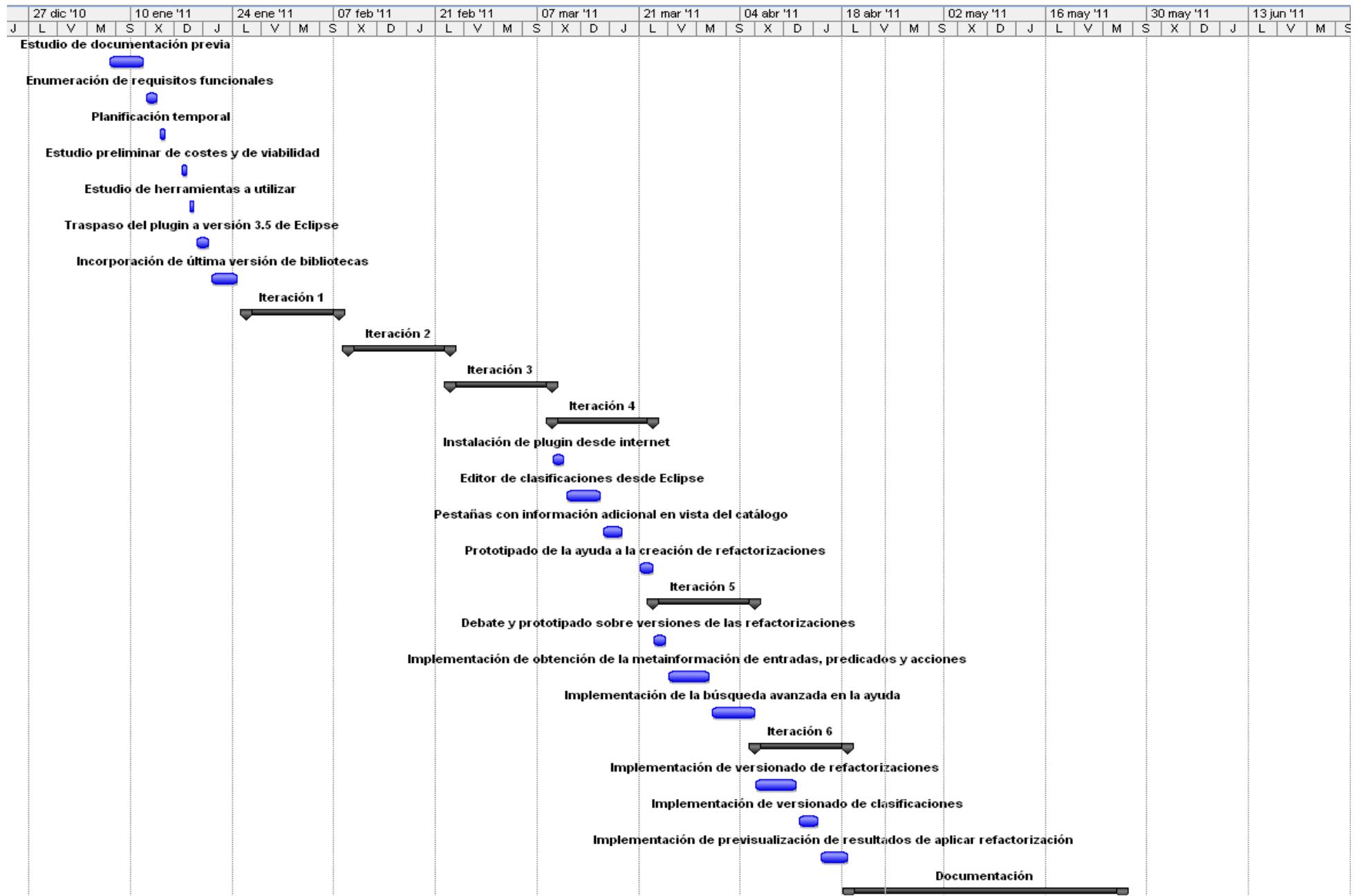


Ilustración 12: Diagrama de Gantt de la planificación inicial del proyecto

3.2. Planificación final del proyecto

En la planificación definitiva ya aparecen reflejados todas las tareas desarrolladas durante el proyecto y la fecha en la que se realizaron.

Tarea	Duración	Fecha Inicio	Fecha Fin
Estudio de documentación previa	3 días	07/01/2011	12/01/2011
Enumeración de requisitos funcionales	2 días	12/01/2011	14/01/2011
Planificación temporal	1 días	14/01/2011	15/01/2011
Estudio preliminar de costes y de viabilidad	1 días	17/01/2011	18/01/2011
Análisis de riesgos	1 días	18/01/2011	19/01/2011
Estudio de herramientas a utilizar	1 días	18/01/2011	19/01/2011
Requisitos tecnológicos	1 días	19/01/2010	20/01/2010
Traspaso del plugin a versión 3.5 de Eclipse	2 días	19/01/2011	21/01/2011
Incorporación de última versión de bibliotecas	2 días	21/01/2011	25/01/2011
Iteración 1	9 días	26/01/2011	08/02/2011
Separación del proyecto de tests	2 días	26/01/2011	28/01/2011
Automatización de la construcción del proyecto (1)	2 días	28/01/2011	01/02/2011
Prototipado de vista de catálogo de refactorizaciones	1 días	01/02/2011	02/02/2011
Implementación del modelo del dominio de la clasificación de refactorizaciones	4 días	28/01/2011	03/02/2011
Filtrado de Clasificaciones	2 días	03/02/2011	05/02/2011
Cambio Interno - Sustituir constantes enteras			
RefactoringConstants.Action/Precondition n/Postcondition por Enum	1 días	07/02/2011	08/02/2011
Iteración 2	9 días	09/02/2011	22/02/2011
Automatización de la construcción del proyecto (2)	1 días	09/02/2011	10/02/2011
Configuración de proceso de build diario	1 días	10/02/2011	11/02/2011
XML de las clasificaciones	3 días	11/02/2011	16/02/2011
Actualización del XML de las refactorizaciones	2 días	16/02/2011	18/02/2011
Crear vista de catalogo - Asignar categorias al crear/editar una refact.	2 días	18/02/2011	22/02/2011

Tarea	Duración	Fecha Inicio	Fecha Fin
Iteración 3	9 días	22/02/2011	05/03/2011
Versión de línea de comandos del catálogo de clasificaciones	2 días	22/02/2011	24/02/2011
Primera versión de la vista del catálogo de refactorizaciones en Eclipse	2 días	24/02/2011	26/02/2011
Permitir agregar palabras claves a los metadatos de una refactorización	2 días	28/02/2011	02/03/2011
Añadir al xml de las refactorizaciones el scope al que pertenecen	1 días	02/03/2011	03/03/2011
Vista Catálogo - Agregar botón de refrescar refactorizaciones	2 días	03/03/2011	05/03/2011
Configurar servidor virtual en amazon para servidor de build y métricas	4 días	22/02/2011	26/02/2011
Instalación de plugin desde internet	5 días	25/02/2011	04/03/2011
Iteración 4	9 días	07/03/2011	18/03/2011
Editor de clasificaciones desde Eclipse	3 días	07/03/2011	10/03/2011
Prototipado de la ayuda a la creación de refactorizaciones	1 días	10/03/2011	11/03/2011
Pestañas con información adicional en vista del catálogo	1 días	11/03/2011	12/03/2011
Vista Catalogo - Filtrado desde interfaz	1 días	14/03/2011	15/03/2011
Vista Catálogo - Añadir un Examples Tab al FolderTab	1 días	15/03/2011	16/03/2011
Bug - DynamicRefactoring Wizard - No muestra el javadoc de los elementos	1 días	16/03/2011	17/03/2011
Bug - Error al intentar aplicar la refactorización RenameParameter de un método	1 días	17/03/2011	18/03/2011
Bug - Error al intentar aplicar RenameField	1 días	17/03/2011	18/03/2011
Iteración 5	9 días	21/03/2011	01/04/2011
Debate y prototipado sobre versiones de las refactorizaciones	2 días	21/03/2011	23/03/2011
Implementación de obtención de la metainformación de entradas, predicados y acciones	4 días	23/03/2011	29/03/2011
Implementación de la búsqueda avanzada en la ayuda	2 días	29/03/2011	31/03/2011
Incorporar certificado para la instalación del plugin	1 días	31/03/2011	01/04/2011

Tarea	Duración	Fecha Inicio	Fecha Fin
Bug Wizard - Input Configuration (Step 2 of 7) - flechas orden lista	2 días	25/03/2011	29/03/2011
Cambio Interno - Sustituir InputParameters como HashMap por objeto con atributos	2 días	29/03/2011	31/03/2011
Iteración 6	9 días	04/04/2011	15/04/2011
Implementación de versionado de refactorizaciones	3 días	04/04/2011	07/04/2011
Cambio interno : Sustituir Ambiguous Parameter como Mapa por objeto perteneciente a una precondition, accion o postcondicion	1 días	07/04/2011	08/04/2011
Cambio interno: Repositorio único de refactorizaciones en el proyecto	4 días	08/04/2011	14/04/2011
Cambio interno : Hacer las refactorizaciones accesibles desde un directorio independiente del workspace	1 días	14/04/2011	15/04/2011
Iteración 7	8 días	18/04/2011	28/04/2011
Habilitar un servidor web o ftp con ip fija para instalacion del plugin desde internet	2 días	18/04/2011	20/04/2011
Bug Wizard: (Step 1 of 7) Interfaz se descuadra ante modificaciones de la ventana	2 días	20/04/2011	22/04/2011
Versionado de clasificaciones	2 días	22/04/2011	26/04/2011
Mejoras al editor de clasificaciones	2 días	26/04/2011	28/04/2011
Documentación	28 días	29/04/2011	08/06/2011
Manual de usuario	5 días	29/04/2011	06/05/2011
Documentación técnica	6 días	06/05/2011	14/05/2011
Memoria principal del proyecto	12 días	16/05/2011	01/06/2011
Anexos	5 días	01/06/2011	08/06/2011

Tabla 7: Planificación final del proyecto

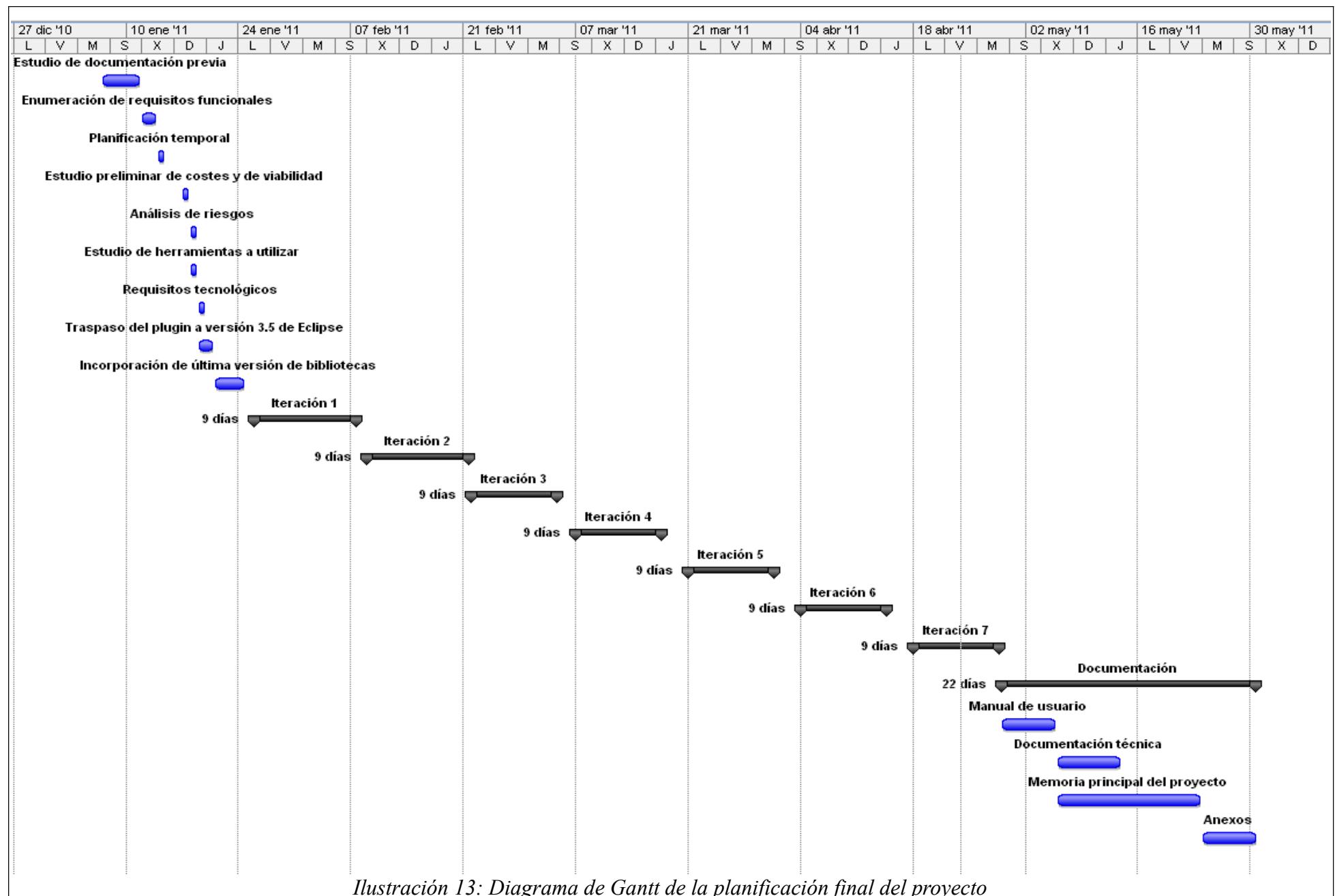


Ilustración 13: Diagrama de Gantt de la planificación final del proyecto

3.3. Desviaciones en la planificación inicial

Como se puede observar si se realiza una comparativa entre las dos planificaciones temporales inicial y final del proyecto se han producido numerosas modificaciones al plan original. El objetivo del plan inicial era servir de guía de la evolución del proyecto con el fin de priorizar aquellas tareas de mayor importancia, dando un cierto sentido al orden en que se deberían de hacer efectivas estas. En ningún momento se tomó el plan de proyecto inicial como una lista de obligaciones inflexibles dado que se era consciente de que el desarrollo del proyecto impondría modificaciones a esa previsión inicial. A continuación se describirán las principales desviaciones que se han producido por ese discurrir del proyecto y las causas que las provocaron.

Las variaciones han consistido no sólo en desviaciones temporales, sino que también ha habido cambios en las características a implementar, tareas que se habían planificado y se han descartado por no considerarse tan importantes o por las dificultades que presentaba su implementación. Por ejemplo, la posibilidad de previsualizar los cambios producidos en un código fuente a consecuencia de la ejecución de una refactorización.

Por otra parte otras tareas que no habían sido consideradas en la planificación inicial han sido agregadas. Ejemplo de esto son ciertas mejoras internas que se han efectuado y que fue imposible planificar inicialmente por desconocimiento del código fuente. Otro tipo de tarea que era muy difícil de planificar inicialmente era la resolución de errores encontrados en el proyecto. Al ser esta la tercera versión del plugin se han encontrado algunos defectos heredados de versiones previas del plugin que han sido necesarios corregir. Algunos de los defectos más importantes corregidos aparecen reflejados en la planificación definitiva.

Otro apartado en el que la variación ha sido mayor es el proceso de construcción del proyecto. Este proceso si aparece reflejado en la primera versión de la planificación pero en ese momento se consideró que sería sencillo de implementar por lo que se planteó que podría quedar terminado para la primera iteración. Sin embargo, la falta de experiencia de los alumnos con la herramienta Maven ha hecho que esa configuración inicial se prolongara a lo largo de varias iteraciones. Además la flexibilidad proporcionada por la herramienta ha hecho que posteriormente se hayan ido agregando nuevas mejoras y funcionalidades adicionales que se consideraba que aportaban valor añadido al proyecto y que se han visto facilitadas por la herramienta. Ejemplos de estas nuevas funcionalidades ha sido la posibilidad de firmar el plugin de forma automática, de generar la cobertura de las pruebas tras cada construcción, la generación del repositorio de instalación del plugin o la

comprobación de métricas sobre el código gracias a Sonar.

En las fases finales del desarrollo se produjo un pequeño giro en la planificación. Debido a la inestabilidad de algunos apartados del plugin se decidió congelar el desarrollo de nuevas características de forma anticipada, previamente a lo que estaba previsto en la planificación inicial. Esto se puede comprobar en el hecho de que las últimas iteraciones están copadas principalmente por modificaciones de diseño y corrección de errores en la planificación definitiva.

Finalmente hay que destacar que todo el desarrollo del plan del proyecto se ha visto facilitado por la herramienta Fogbugz, la cual ya fue destacada en el apartado de *Técnicas y Herramientas de la Memoria*. Esta herramienta ha permitido llevar un control detallado de las tareas y ha facilitado la comunicación entre los miembros del equipo. Además ha sido de gran ayuda a la hora de realizar esta retrospectiva. Gracias a ella y a la labor de documentación de las tareas efectuada por los desarrolladores, se ha encontrado toda la evolución del proyecto reflejada en su página web cuando se ha necesitado para la elaboración de esta sección.

3.4. Descripción de las tareas

3.4.1. Estudio de la documentación previa

Los primeros días de trabajo se realiza un proceso de familiarización con el proyecto. En este periodo se estudia la documentación proporcionada por versiones anteriores del proyecto y se busca información de otras aplicaciones con funcionalidades semejantes. El objetivo es tener una imagen global del trabajo que se tiene que realizar.

3.4.2. Enumeración de requisitos

Se definen los requisitos del proyecto tanto funcionales como no funcionales. Consiste en definitiva en marcar los objetivos del proyecto.

3.4.3. Planificación temporal

Consiste en contextualizar las tareas a desarrollar del proyecto en el tiempo, decidiendo cuánto tiempo se dedicará a cada una de ellas y en qué momento se llevarán a cabo. De este modo se priorizan las importantes y se da sentido al orden de desarrollo de

características en base a cuales son dependientes de otras.

3.4.4. Estudio preliminar de costes

Consiste en plantearse un primer estudio de los costes y de la viabilidad del proyecto basándose en las funcionalidades que se van a obtener, en la competencia y en las necesidades del mercado. Se debe decidir y argumentar el seguir adelante con el proyecto. El estudio de viabilidad tendrá que ser revisado en la documentación al final del proyecto. Esto se debe a que a lo largo del proyecto se han podido utilizar algunas herramientas que no se pensaba adquirir en un principio.

3.4.5. Análisis de riesgos

En esta fase se tienen en cuenta los distintos riesgos conocidos y vigentes en el proyecto. El conjunto de los riesgos se ordena por importancia y se proponer de cada uno de ellos acciones específicas para su mitigación.

3.4.6. Estudio de herramientas a utilizar

Se estudian las necesidades de herramientas software para el desarrollo del proyecto de forma genérica. En base a ese estudio se toman las necesidades y por cada una se sopesan las distintas alternativas comerciales para escoger la que más se adapte a las necesidades del proyecto. En el caso de este proyecto las herramientas más necesarias a estudiar al comienzo del proyecto fueron el sistema de control de versiones a escoger, el de gestión de tareas y bugs y el sistema de construcción del proyecto a partir del código fuente.

3.4.7. Requisitos tecnológicos

En este apartado se plantean todas las técnicas, las herramientas y los entornos de desarrollo que se van a utilizar en la realización del proyecto. Se va a emplear más tiempo que para el resto de los requisitos debido a la necesidad de buscar las ventajas de utilizar determinadas herramientas frente a otras, para lo cual se necesita un estudio de las mismas. Se incluye también en este periodo el tiempo utilizado en la instalación de estos programas.

3.4.8. Traspaso del plugin a versión 3.5 de Eclipse

La última versión del plugin fue desarrollada sobre la versión 3.3 de Eclipse. Con la intención de hacer la nueva versión del plugin compatible con las últimas de Eclipse (3.5 en adelante), fue necesaria realizar una serie de ajustes de compatibilidad en el código fuente.

3.4.9. Incorporación de última versión de bibliotecas

De forma similar a lo que ocurría en la tarea anterior, la evolución del plugin se había congelado desde el proyecto anterior mientras que la de las bibliotecas MOON y JavaMOON había incorporado cambios importantes. De nuevo fue necesario realizar ciertos cambios para hacer posible la integración de las últimas versiones de las bibliotecas. Este es un proceso a destacar dado que las últimas versiones de dichas bibliotecas han incorporado modificaciones de gran importancia que han posibilitado pasar de permitir refactorizaciones exclusivamente sobre proyectos pequeños a permitir refactorizaciones sobre proyectos reales con miles de líneas de código.

También es necesario reseñar que este proceso se realizó en más ocasiones a lo largo del proyecto, aunque sólo se ha marcado como tarea al principio del proyecto por el mayor tiempo que llevó debido a que los cambios en las bibliotecas eran mayores y a que al ser la primera vez que se realizaba el proceso de integración implicó una mayor dificultad.

Además con las nuevas posibilidades abiertas que ofrecía las bibliotecas fue necesario incorporar otros cambios en determinadas partes del plugin, para adaptarlo para permitir hacer uso de esas nuevas opciones que las bibliotecas abrían.

3.4.10. Iteración 1

En esta primera iteración como parte de la configuración del proceso de construcción automático del proyecto se separaron los tests creando un proyecto de Eclipse específico para ellos. Además se consiguió configurar Apache Maven con tycho[Tycho n.d.] para que se descargara sus dependencias de un repositorio en Internet y se compilara correctamente.

En esta fase ya se empezó a desarrollar la vista del catálogo de refactorizaciones que se continuaría desarrollando en iteraciones posteriores. Los pasos en ese sentido dados en esta fase fueron: el prototipado de la interfaz de dicha vista, la implementación de su modelo del dominio y también el modelo de filtrado por varios criterios.

Como último aspecto destacable en esta fase también se realizó una refactorización con el objetivo de eliminar código dependiente de las constantes enteras que representaban a las precondiciones, acciones y postcondiciones por una enumeración. Esto permitió en fases posteriores de la aplicación simplificar el diseño del código.

3.4.11. Iteración 2

En esta segunda iteración se continuó con apartados del proyecto ya tocados en la primera fase. En el apartado de la configuración de la construcción del plugin se consiguió que los tests se ejecutaran como parte del proceso y que se generaran las métricas del proyecto con Sonar [Sonar n.d.]. Además se configuró Hudson [Hudson Continuous Integration n.d.] para que ejecutara de forma diaria el proceso de construcción completo a partir del contenido de la rama principal, master, en el repositorio del proyecto en Github [GitHub n.d.].

Dentro del apartado de la vista del catálogo de refactorizaciones se creó el soporte XML para las clasificaciones y además se modificó el de las refactorizaciones para adaptarse al catálogo. También se modificó la interfaz gráfica de creación de refactorizaciones para permitir agregar clasificaciones a las refactorizaciones creadas.

3.4.12. Iteración 3

Se continua con la configuración del proceso de construcción así como con la vista del catálogo de clasificaciones.

En el primero se consigue configurar un servidor en Internet con un servicio de Amazon llamado EC2 [Amazon Elastic Compute Cloud (Amazon EC2) n.d.] para que se pueda acceder desde Internet al informe de calidad del código del proyecto, a la información sobre los últimos procesos de construcción ejecutados con Hudson. Además se configura el proyecto para que genere un repositorio P2 al construirse. Al hacer disponible este repositorio desde Internet con el repositorio de Amazon ya citado, se alcanza la meta de habilitar al usuario instalar el plugin desde Internet sin necesidad de abandonar el entorno del IDE.

En el segundo se añade la posibilidad de agregar palabras clave a una refactorización para facilitar su búsqueda y clasificación en el catálogo. Esto supone modificaciones en el asistente de creación de refactorizaciones y en el XML de las refactorizaciones. Además se

agregan a los ficheros XML de las refactorizaciones el ámbito al que pertenecen como una categoría. Con todo el modelo del dominio del catálogo ya implementado, se crea una versión de línea de comandos del catálogo para pruebas. A ésta le sigue la primera versión de interfaz gráfica dentro de Eclipse.

3.4.13. Iteración 4

Se implementan nuevas características importantes dentro de la vista del catálogo entre ellas se encuentra el filtrado de refactorizaciones desde la interfaz, nuevas pestañas con información adicional para la refactorización seleccionada y otra pestaña más para mostrar los ejemplos de las refactorizaciones.

Por otro lado se empieza a trabajar en la ayuda a la creación de refactorizaciones. Se crea un prototipo con las modificaciones al asistente necesarias y se corrige un problema en el plugin que hacía que no se mostrara correctamente la documentación de las entradas de las refactorizaciones en dicho asistente.

Con la posibilidad de instalar el plugin desde Internet se considera la necesidad de ralentizar la incorporación de características al proyecto para dedicar ciertos esfuerzos a dotar de estabilidad y liberar de fallos al plugin. Es por eso que ya desde esta fase tan temprana empiezan a incorporarse tareas de cambios internos en el diseño del proyecto y de corrección de errores. Debido a las dimensiones del proyecto y al gran número de características que incorpora este tipo de tareas se encontrarán a lo largo del resto del plan del proyecto.

3.4.14. Iteración 5

Las dos tareas principales de esta fase están relacionadas con el apartado de facilitar al usuario la creación de refactorizaciones. En esta dirección se desarrolla un completo rediseño de la búsqueda para las entradas, precondiciones, acciones y postcondiciones que permite incorporar todas las características de un motor avanzado de búsquedas gracias a la biblioteca Lucene [Apache Lucene n.d.]. También se consigue obtener la metainformación de los elementos anteriormente citados a partir de su documentación en formato JavaDoc.

Se comienza con el estudio de la línea de trabajo para resolver el problema referente al versionado de refactorizaciones. Se realizó un debate sobre las opciones más interesantes para afrontar el problema y una vez se tomó la decisión se creó un prototipo sobre dicha

solución.

Además se incorporó el firmado del plugin al proceso de construcción del proyecto, se solucionaron ciertos errores encontrados en la interfaz del asistente de creación/ediación de refactorizaciones y se incorporó una cambio interno importante en la estructura del código de las refactorizaciones pasando de utilizar un mapa para representar las entradas de las refactorizaciones a un objeto con atributos y métodos. Este cambio facilitó posteriores desarrollos por ejemplo en la línea de trabajo de la mejora del asistente de creación de refactorizaciones.

3.4.15. Iteración 6

En esta fase se llevan a cabo una serie de cambios internos de gran calado.

En primer lugar se modifica la estructura de las refactorizaciones y se pasa de tener los parámetros ambiguos definidos como un objeto difícil de comprender (un array de mapas de listas) por un modelo mucho más sencillo en el que los parámetros ambiguos son entradas y pertenecen a las precondiciones, acciones y postcondiciones.

En segundo lugar y más importante se concluye con una tarea en la que se había comenzado a trabajar en iteraciones anteriores. Esta tarea consiste en sustituir el uso en múltiples puntos del proyecto de modificaciones directas sobre los XML de definición de las refactorizaciones, por un contenedor general encargado de realizar todas las modificaciones relativas a éstas de forma centralizada.

El cambio anterior aunque es costoso nos va a facilitar notablemente implementar el versionado de las refactorizaciones en esta misma fase.

3.4.16. Iteración 7

Por problemas con el servidor web de Amazon se toma un enfoque diferente para la descarga del plugin, creando una cuenta en el servicio gratuito Google code desde la que se puede descargar el plugin. Esta modificación tiene como ventajas que la URL de instalación del plugin se mantiene constante (algo que con el servicio de Amazon no pasaba) y que la instalación es mucho más rápida y el servidor más estable.

Se lleva a cabo la tarea de versionado de clasificaciones que presentaba los mismos problemas iniciales que en el caso de las clasificaciones. Además se incorporan ciertas

mejoras al editor de clasificaciones.

Finalmente es esta fase se soluciona un error que provocaba que los controles se descuadraran en el primer paso del asistente de creación de refactorizaciones.

3.4.17. Documentación

A pesar de que la documentación se había ido realizando de forma simultánea al desarrollo del proyecto, se ha dedicado la fase final del proyecto a ir cerrando los apartados, incorporar ciertas secciones que por sus características no se podían definir hasta que el proyecto estuviera terminado y a revisar lo redactado para corregir despistes.

4. ESTUDIO DE VIABILIDAD

Un proyecto necesita de un estudio de viabilidad para conocer el dinero y tiempo que será necesario invertir y decidir si resulta rentable llevarlo a cabo.

Debemos señalar que nuestro proyecto tiene una finalidad investigadora y además está encaminado a la obtención del título de Ingeniero en Informática, aunque realizaremos el estudio como si tuviera fines comerciales.

Los estudios de viabilidad pueden ser de distintas clases:

Viabilidad técnica: se estudia la funcionalidad de un proyecto y las restricciones que pueden afectarle.

Viabilidad legal: contempla los posibles obstáculos legales.

Viabilidad económica: un análisis coste-beneficio nos indicará si el proyecto es o no rentable. Se va a dividir el estudio en tres puntos. Para empezar se estiman los costes que va a producir la creación del proyecto. Posteriormente se realiza el estudio de mercado que permite ver las posibilidades de la aplicación creada una vez a la venta. Para finalizar se exponen las decisiones tomadas en cuanto al precio del producto y se calcula el punto a partir del cual se obtienen beneficios de su venta.

Viabilidad temporal: sirve para planificar el coste temporal que tendrá un proyecto software.

4.1. Viabilidad técnica

Este es un proyecto de una elevada complejidad técnica. El hecho de que los alumnos no tuvieran ninguna experiencia previa acerca de la programación de plugins para Eclipse ha constituido el factor principal de riesgo. Esto es debido a que el entorno es ciertamente complejo por todas las características específicas que requiere, como por ejemplo el uso de *OSGI* o la dificultad de su biblioteca gráfica específica *SWT*. Además de que la disponibilidad de documentación es muy escasa.

Por otra parte el hecho de que fuera la segunda continuación de un proyecto inicial ha sido otro factor muy a tener en cuenta. Los alumnos han necesitado un comprensión de la aplicación en su totalidad y han tenido que corregir errores o introducir mejoras sobre funcionalidad previa como añadido a la funcionalidad nueva a implementar. Ha sido pues, un proyecto en el que ha sido necesario mantener las características de las versiones anteriores además de implementar nuevas funciones, con toda la carga de trabajo que eso supone.

Otros riesgos de tipo técnico existentes han surgido de la dificultad de implementar nuevas características mediante herramientas con las que los desarrolladores no estaban familiarizados. Ejemplos de esto han sido la utilización de Maven para el proceso de construcción, la incorporación de pruebas de interfaz gráfica, el control de calidad del código del proyecto o la posibilidad de instalar el plugin desde Internet por citar algunos.

4.2. Viabilidad legal

La viabilidad legal se refiere a la necesidad de determinar la inexistencia de posibles obstáculos legales para la distribución, instalación y ejecución normal del proyecto. Después del estudio de las posibles licencias a elegir se ha decidido distribuir el producto bajo un tipo de licencia *GPL*.

Además con finalidad informativa, en la siguiente tabla se muestran las licencias de los productos software usados en el desarrollo de la aplicación.

Software	Descripción	Licencia
Microsoft Windows XP Professional	Sistema operativo	Comercial
Ubuntu 10.04	Sistema operativo	Código Libre
Oracle Java JDK	Kit de desarrollo	Oracle License
Eclipse SDK Version 3.6	Entorno de desarrollo	Eclipse Public License Version (EPL)
OpenOffice 3.3	Suite de ofimática	Oracle License
Microsof Project 2007	Planificación del proyecto	Comercial
Framework de pruebas JUnit	Biblioteca de pruebas unitarias	Código Libre
Apache Maven	Herramienta de automatización del proceso de construcción	Apache Software License (libre)
Sonar	Herramienta de control de calidad del código	GPL
Jing	Capturas de pantalla	Gratuita
Apache Lucene	Biblioteca de búsqueda e indexación	Apache Software License (libre)
Balsamiq Mockup	Prototipos de interfaz gráfica	Comercial
Git	Control de versiones	GPL
Egit	Plugin de Git para Eclipse	Eclipse Public License Version (EPL)
Fogbugz	Herramienta de gestión de tareas	Comercial
SWTBot	Biblioteca para pruebas de interfaz gráfica	Eclipse Public License Version (EPL)
Hudson	Servidor de integración continua	MIT License (libre)
JaCoCo	Informes de cobertura del código de los tests	Eclipse Public License (EPL)
tycho	Plugin de Maven para construir plugins de Eclipse	Eclipse Public License (EPL)
Astah Community	Análisis y diseño UML	Gratuita

Tabla 8: Relación de licencias de las herramientas utilizadas

Como hemos comentado anteriormente se ha decidido distribuir el plugin bajo un tipo de licencia *GPL*, que a continuación va a ser explicada.

La licencia pública general de GNU o más conocida por su nombre en inglés **GNU General Public License** (*GNU GPL*), es una licencia que ha sido creada por la Free Software Foundation.

Es un tipo de licencia sobre la propiedad intelectual en la cual únicamente se exige que aquellos desarrollos hechos con material que use la licencia *GPL*, sean a su vez liberados bajo una licencia *GPL*. Está orientada principalmente a proteger la libre distribución, modificación y uso de software. Su propósito u objetivo es declarar que el software cubierto por esta licencia es software libre y protegerlo de intentos de apropiación que restrinjan esas libertades a los usuarios. Bajo esta filosofía, la licencia *GPL* garantiza a los usuarios de un programa software los derechos de la definición de software libre, asegurando que las libertades se conservan y preservan, incluso cuando el trabajo es modificado o ampliado.

Cabe destacar que es necesaria la compartición del código desarrollado, puesto que permite ayudar a otros desarrolladores compartiendo la resolución de problemas ya resueltos con anterioridad, lo cual permite intercambiar los avances en esta ciencia y generar una evolución más rápida de la misma.

La licencia *GNU GPL* correspondiente a la aplicación se adjunta en el CD dentro de la carpeta `Ejecutable` donde también se encuentra el plugin instalable, y también dentro del directorio con el proyecto de desarrollo del plugin en Eclipse.

4.3. Viabilidad económica

Es necesaria la realización de un estudio de viabilidad económica previo al desarrollo del proyecto, para poder estimar los recursos que van a ser utilizados y la cantidad de beneficios que éste generará.

Este estudio consiste en una evaluación de los costes de desarrollo frente a los beneficios del producto final desarrollado. La parte más importante del estudio de la viabilidad se corresponde con el análisis coste-beneficio.

4.3.1. Análisis de costes

Se debe realizar un análisis completo de todos los costes derivados del desarrollo del proyecto, estos son: costes de *personal*, de recursos *hardware*, de licencias *software* y de otros conceptos que se detallarán convenientemente.

Costes de personal

Para la realización del presente proyecto se necesitan dos desarrolladores noveles que trabajarán durante seis meses con una jornada de siete horas diarias. El salario de los trabajadores, basado en el salario medio de un programador júnior, es 10€/hora. Se debe de tener en cuenta que un mes tiene 20 días laborables.

$$10\text{€}/\text{hora} \times 7\text{horas/día} \times 20\text{días/mes} \times 6\text{meses} \times 2 \text{ sueldos} = \mathbf{16.800 \text{ €}}$$

Para realizar el proyecto se ha contado con la ayuda del tutor, las horas dedicadas por el tutor también deben ser incluidas como costes de personal. Se tiene en cuenta que se han mantenido quince reuniones con el tutor y se calcula que el promedio de duración de cada una ha sido de una hora y media. El sueldo del tutor por tratarse de un trabajador cualificado es muy superior y se ha estimado en 40 €/hora.

$$40\text{€}/\text{hora} \times 1,5\text{horas/reunión} \times 15 \text{ reuniones} = \mathbf{900 \text{ €}}$$

Costes de seguridad social

A los costes de personal hay que sumar los de seguridad social y otros impuestos añadidos que paga el empresario por cada empleado.

Los porcentajes a pagar de cada uno de estos impuestos son establecidos cada año en base a los presupuestos del estado. Para obtener los porcentajes relativos al año 2011 se ha recurrido a la página web de la seguridad social (<http://www.seg-social.es/>) [Seguridad Social n.d.]. Los valores aplicables al proyecto obtenidos son:

- Contingencias comunes: 23,6% del sueldo del trabajador.
- Impuesto de desempleo: 5,50% del sueldo del trabajador.
- Fondo de garantía social: 0,2% del sueldo del trabajador.
- Formación profesional: 0,6 % del sueldo del trabajador.

Estos impuestos deben aplicarse a la totalidad de los costes de personal, es decir, a la suma de los sueldos del tutor y de los desarrolladores.

$$(16.800+900)\text{€} \times 0,299 = \mathbf{5.292,30 \text{ €}}$$

Amortización hardware

En este apartado se incluyen los costes por amortización del hardware utilizado en el desarrollo del proyecto. Se necesitan dos ordenadores personales para el desarrollo.

El precio de adquisición de los equipos está estimado en 600€ cada uno.

Estos ordenadores se suponen que serán empleados en posteriores desarrollos. La inversión realizada en hardware deberá ser amortizada en un plazo de 3 años (36 meses), basándose en la vida útil de las máquinas antes de quedar obsoletas. Aunque el impuesto de sociedades propone como período de amortización entre 4 y 6 años, se han tomado 3 años porque se considera una estimación más real de la vida útil del hardware.

El gasto de amortización del hardware es el siguiente.

$$(6\text{meses}/36\text{meses}) \times 600\text{€} \times 2 \text{ equipos} = \mathbf{200 \text{ €}}$$

Amortización software

En este apartado se incluyen los costes ocasionados por las licencias de software que se han tenido que adquirir.

Los sistemas operativos utilizados en el desarrollo del proyecto han sido Microsoft Windows XP Professional con Service Pack 2, cuyo precio es de 250€ y la distribución Linux Ubuntu en su versión 10,04 (Lucid Lynx). La licencia de este último es de tipo *GNU GPL* [The GNU General Public License v3.0 - GNU Project - Free Software Foundation (FSF) n.d.] por lo que su uso es gratuito. Para la planificación del proyecto se ha utilizado la herramienta Microsoft Project 2007 cuyo coste es de 500€.

Para la gestión de tareas durante el proyecto se ha utilizado la web FogBugz. El coste de esta herramienta es de 25€ por usuario y por mes. Dado que la duración del proyecto ha sido de seis meses el coste de esta herramienta hubiera sido de 300€ . Este coste se va a tener en cuenta a la hora de la amortización del software pero hay que aclarar que los desarrolladores del proyecto no han necesitado incurrir en él debido a que existe una licencia gratuita para proyectos no comerciales, con un máximo de dos usuarios.

Durante el proyecto se ha utilizado la herramienta Balsamiq Mockup para el desarrollo de prototipos de interfaz gráfica. El coste de dicha herramienta es de 79€ por

usuario, con lo que el coste total para los desarrolladores se eleva a 158€. Tampoco ha sido necesario sufragar el coste de esta herramienta debido a que dispone de una versión simplificada, la cual es gratuita y se puede utilizar para la creación de pequeños prototipos.

Se ha empleado como herramienta ofimática OpenOffice. Las fases de análisis y diseño del proyecto se han realizado mediante la herramienta Astah versión *Community*, de uso libre. El entorno de desarrollo utilizado es Eclipse el cual es gratuito, al igual que las librerías de Java utilizadas y otras.

La utilización del servidor de control de versiones ha sido proporcionada por Github, que ofrece este servicio de forma gratuita para proyectos con licencia de código abierto. Lo mismo ocurre para el servidor que permite la instalación del plugin desde Internet con Google Code.

A continuación se adjuntan los costes de las herramientas no gratuitas utilizadas durante desarrollo del proyecto. Para una descripción de las funcionalidades y características de ellas se remite al apartado de *Técnicas y Herramientas de la Memoria*.

Software	Licencia
Microsoft Windows XP Professional	250 €
Microsof Project 2007	500 €
Fogbugz	300 €
Balsamiq Mockup	158 €
Total Software	1.208 €

Tabla 9: Coste de las herramientas utilizadas

El periodo de amortización del software se establece en tres años (36 meses), a partir de los cuales se considera que las aplicaciones pierden su valor. Según este criterio el gasto de amortización es el siguiente:

$$(6\text{meses}/36\text{meses}) \times 1.208\text{€} = \mathbf{201,33\text{ €}}$$

Otros gastos

Se trata de otros gastos que son necesarios, como son el coste del material de oficina empleado cuyo coste aproximado sería de unos 200 €. Se incluye en este apartado el coste porcentual de los suministros (calefacción, energía eléctrica, teléfono, conexión a Internet).

También se debe tener en cuenta el coste del alquiler de las instalaciones de trabajo, puesto que es un recurso necesario para la realización del proyecto. El coste del alquiler se ha calculado en 300€/mes, dado que el período de desarrollo ha sido de seis meses el coste derivado del alquiler de las instalaciones asciende a:

$$300\text{€}/\text{mes} \times 6\text{meses} = 1.800 \text{ €}$$

La suma de los conceptos anteriores, es decir, el total de otros gastos es:

$$200\text{€} + 1.800\text{€} = \mathbf{2.000 \text{ €}}$$

El coste total del proyecto se especifica a continuación.

Concepto	Cantidad
Coste personal	17.700,00 €
Coste Seguridad Social	5.292,30 €
Coste hardware	200,00 €
Coste software	201,33 €
Otros gastos	2.000,00 €
Coste Total	25.393,63 €

Tabla 10: Costes del proyecto

4.3.2. Análisis coste-beneficio

Este apartado esta dedicado al análisis de coste-beneficio, en él se va a realizar un estudio de la competencia así como el estudio de mercado que permite ver las posibilidades de la aplicación creada una vez a la venta. Para finalizar se van a exponer los beneficios económicos y no económicos, en el primero se incluirán las decisiones tomadas en cuanto al precio del producto y se calculará el punto a partir del cual se obtienen beneficios de su venta.

Estudio de la competencia

De entre las alternativas ya estudiadas en la *Memoria* se van a considerar las dos que ofrecen una funcionalidad más similar: Refactory y RefactoringNG. El resto de alternativas no han sido consideradas ya que o no permiten definir refactorizaciones propias o es muy compleja su definición lo que descarta la utilización de esta funcionalidad para la mayoría de los usuarios, por lo tanto no compiten en sentido directo con el plugin de refactorización.

Refactory es una herramienta muy flexible, cuyo punto fuerte es que permite definir refactorizaciones que son totalmente independientes del lenguaje con el que se trabaja. Esto la hace una herramienta de gran valor especialmente para aquellos que utilizan el proceso de desarrollo basado en modelos y definen sus propios lenguajes específicos para su dominio (*DSL*). Sin embargo, por su planteamiento tan general y tan independiente del lenguaje este plugin no va a permitir definir refactorizaciones avanzadas para Java. Además tampoco permite definir refactorizaciones sobre fragmentos de código como por ejemplo *Extract Method*. En definitiva, a pesar de ser un plugin muy interesante Refactory posee desventajas notables respecto al plugin de refactorización cuando se quiere utilizar para el lenguaje de programación Java.

RefactoringNG por su parte sí permite definir refactorizaciones que utilizan todas las propiedades de los *árboles AST* de Java, al contrario de lo que sucedía con Refactory. La desventaja de RefactoringNG es que carece de la semántica necesaria para poder definir precondiciones y poscondiciones o para tener en cuenta las relaciones entre las clases de un modelo. Es por ello que RefactoringNG tampoco supone una competencia directa para el plugin de refactorización.

Análisis del mercado

El perfil del usuario que podría estar interesado en nuestro plugin sería el siguiente:

- Utiliza Eclipse como entorno de programación.
- Aprecia el valor que aportan las refactorizaciones al desarrollo de software.
- Java es su lenguaje de programación.
- Para el uso de ciertas características del plugin de refactorización el usuario debe poseer unos conocimientos de Java y de Eclipse avanzados.
- Presta atención al buen diseño y a la mejora constante de su código.

Beneficios no económicos

Gracias al plugin los desarrolladores que utilizan Eclipse podrán beneficiarse de la siguiente funcionalidad:

- Una vista que permite el estudio de un catálogo amplio de refactorizaciones con abundante información descriptiva y recursos adicionales como imágenes y ejemplos de código.
- La posibilidad de organizar la forma en que se visualizan las refactorizaciones, modificando la clasificación seleccionada y filtrando por una serie de criterios.
- La opción de poder definir nuevas clasificaciones propias y de asignar nuevas categorías a las refactorizaciones.
- La posibilidad de obtener refactorizaciones nuevas por cada actualización del plugin, sin necesidad de eliminar las refactorizaciones que el usuario había definido personalmente.
- Ayuda a la hora de crear las refactorizaciones, con mejoras en el asistente tales como búsqueda avanzada de elementos por nombre y por descripción o relación de las refactorizaciones en las que los elementos participan.
- Posibilidad de instalar el plugin desde Internet y obtener actualizaciones del mismo de forma automática y sin necesidad de abandonar Eclipse.

- Todo esto como añadido a las características de las que el plugin ya disponía como son: la importación y exportación de refactorizaciones, de planes de refactorizaciones o el historial de refactorizaciones aplicadas.

En lo referente al equipo de desarrollo, es fácil ver el enorme beneficio que va a adquirir durante la ejecución del proyecto. Adquirirán el conocimiento necesario para el trabajo en grupo, desarrollarán su liderazgo, potenciarán la toma de decisiones y obtendrán mucha experiencia a nivel técnico; desde programación en Java pasando por el manejo de entornos *OSGI*, la programación de plugins para *Eclipse*, los procesos de construcción basados en *Maven* y el trabajo con ficheros *XML*, entre otras cosas. Con todo ello, se puede deducir que desde un punto de vista no económico el proyecto reporta grandes beneficios.

Beneficios económicos

Puesto que la herramienta desarrollada no está pensada como aplicación comercial, resulta difícil estimar el precio para una licencia. Lo que se va a hacer es utilizar como punto de partida el precio de *IntelliJIDEA*, una solución comercial que proporciona alguna de las características del plugin de refactorización. Dicha herramienta no ha sido considerada como alternativa en la sección de *Comparativa entre alternativas de la Memoria* porque es un entorno integrado de desarrollo y no se centra exclusivamente en las refactorizaciones, además de que no permite definir refactorizaciones propias. Los precios de los distintos tipos de licencias comerciales de *IntelliJIDEA* se reflejan en la siguiente tabla:

Tipo de licencia	Precio
Commercial License	420 €
Personal License	168 €
Academic License	84 €

Tabla 11: Precios de las licencias comerciales de IntelliJIDEA

Se consideró que un paquete que pudiese incluir un entorno de desarrollo como lo es *Eclipse* junto con el plugin de refactorización contaría con unas características agregadas con las que *IntelliJIDEA* no cuenta, por ejemplo la posibilidad de definir refactorizaciones propias o de clasificar las refactorizaciones y visualizarlas en un catálogo, por lo tanto sería interesante para los desarrolladores. Teniendo en cuenta esta consideración, pero también el hecho de que lo que se está ofertando a los clientes no es un entorno de desarrollo integrado sino un extensión del mismo, el precio mínimo que se estableció para la licencia comercial fue de 100 euros.

Presentamos a continuación una tabla comparativa entre el número de licencias que deberán venderse para amortizar el coste estimado partiendo de los 100 euros que se han establecido como precio mínimo. Con ventas mayores al número de licencias indicado en la tabla se obtendrá beneficio.

Precio(€)	Nº licencias
100	≈ 254
120	≈ 212
140	≈ 182
180	≈ 141

Tabla 12: Relación entre el precio y el número de licencias de la aplicación

Definitivamente, teniendo en cuenta las características distintivas existentes respecto de IntelliJIDEA y el número de licencias necesarias para empezar a rentabilizar la inversión se decidió utilizar como precio del producto los 120€.

Con dicho precio sería necesario vender unas 212 licencias para recuperar la inversión y empezar a obtener beneficios:

$$25.393,63 \text{ € de coste total} / 120 \text{ € por licencia} = \underline{\underline{212 \text{ licencias}}}$$

Se consideró que era factible vender ese número de licencias teniendo en cuenta que el precio de la aplicación no es excesivamente alto y que la funcionalidad del plugin es atractiva y puede interesar a una amplia comunidad de desarrolladores que programan en Java utilizando Eclipse. Este objetivo se podría ver facilitado mediante una campaña de promoción del producto en la que se ofreciera a los usuarios una versión con características reducidas instalables sencillamente a través de Internet. Una vez que los usuarios hubieran podido probar dicha versión tendrían la posibilidad de acceder a la versión completa mediante el pago de la licencia. Además otra estrategia a seguir para rentabilizarlo sería la prestación de servicios de mantenimiento y mejora, la creación de módulos o paquetes específicos bajo demanda, etc.

Análisis basado en el V.A.N (Valor actual neto)

Consiste en actualizar a valor presente los flujos de caja futuros que se prevé va a generar el proyecto, descontados a un cierto tipo de interés (la tasa de descuento), y compararlos con el importe inicial de la inversión. Si el VAN es mayor que cero se considera

que el proyecto es rentable y si es menor se considera que no lo es. Desde el punto de vista de un inversor a la hora de escoger entre dos proyectos, elegirá aquel que tenga el mayor VAN.

La fórmula para calcular el VAN es la siguiente:

$$\text{VAN} = \sum (V_t / (1 + k)^t) - I_0$$

donde;

V_t representa los flujos de caja en cada periodo **t**.

I₀ es el valor del desembolso inicial de la inversión.

n es el número de períodos considerado.

k es la tasa de descuento.

Como ejemplo de tasas de descuento (o de corte) que se suelen utilizar indicamos las siguientes:

- Tasa de descuento ajustada al riesgo = Interés que se puede obtener del dinero en inversiones sin riesgo (deuda pública) + prima de riesgo.
- Coste medio ponderado del capital empleado en el proyecto.
- Coste de la deuda, si el proyecto se financia en su totalidad mediante préstamo o capital ajeno.
- Coste medio ponderado del capital empleado por la empresa.
- Coste de oportunidad del dinero, entendiendo como tal el mejor uso alternativo, incluyendo todas sus posibles utilizaciones.

Otra opción es definir una tasa de descuento con el rendimiento que se espera de la inversión para considerarla rentable. Para definir esta tasa se suele tener en cuenta que se obtenga un beneficio superior al que ofrecería el invertir el desembolso inicial a plazo fijo o en bonos del tesoro. En nuestro caso se han tenido en cuenta las estadísticas publicadas en [Banco de España - Tipos de interés n.d.] y en base a ello se han tomado tres tasas de descuento distintas para evaluar la rentabilidad del proyecto: 4,5%, 5,5% y 7%.

La tabla siguiente muestra los valores que se han utilizado para el cálculo del VAN. Cada fila corresponde con el período correspondiente a un año y a la derecha se muestran el número de licencias que se prevé vender a lo largo de ese año y los beneficios que se derivarían de esas ventas. En el primer periodo se refleja el desembolso inicial del proyecto.

Período	Nº licencias	Flujos de caja
0	-	-25.393,63
1	50	6.000
2	125	15.000
3	50	6.000
4	25	3.000

Tabla 13: Valores utilizados para el cálculo del VAN

En el primer año el producto no sería muy conocido por lo que se conseguirían ventas moderadas. En el segundo año con un producto más consolidado en el mercado y más popular las ventas se incrementarían notablemente para luego ir descendiendo en años posteriores porque la evolución de otras herramientas iría convirtiendo el plugin en un producto obsoleto, a menos que este siguiera evolucionando.

El resultado de aplicar la fórmula del VAN sobre las tres tasas de descuento establecidas anteriormente es la siguiente:

$$k = 4,5\% \Rightarrow VAN = \sum (V_t / (i + k)^t) - I_0 = 1.777,43\text{€}$$

$$k = 5\% \Rightarrow VAN = \sum (V_t / (i + k)^t) - I_0 = 1.502,12\text{€}$$

$$k = 7\% \Rightarrow VAN = \sum (V_t / (i + k)^t) - I_0 = 469,07\text{€}$$

Como se puede observar el VAN indica que el proyecto sería rentable para cualquiera de las tres tasas de descuento aplicadas. Incluso cuando se aplica una tasa de descuento exigente del 7% el VAN continua devolviendo un valor positivo, lo que representa un indicador muy favorable.

La rentabilidad del proyecto se debería principalmente a que ofrece una funcionalidad atractiva que puede interesar a una comunidad amplia de desarrolladores. Sin embargo, es necesario puntualizar que el VAN es un estimador impreciso en el caso de proyectos tecnológicos debido a la rápida obsolescencia de los mismos. Si se quisiera hacer una estimación más precisa habría que restar a los beneficios de cada periodo una cantidad

asociada al coste necesario para realizar el mantenimiento de la herramienta.

4.4. Viabilidad temporal

El estudio de la viabilidad temporal sirve para planificar el coste temporal que tendrá un proyecto software. Para ello se tienen en cuenta diversos factores que intervienen en el desarrollo del mismo.

Utilizando los casos de uso, elementos que ya se tienen establecidos por medio de los requisitos exigibles para la elaboración del proyecto, y a través de una serie de cálculos se llegará a una estimación del tiempo necesario para desarrollar el proyecto y comprobar si es viable o no.

4.4.1. Estimación de Casos de Uso

En este apartado vamos a utilizar técnicas de estimación del trabajo basadas en los casos de uso.

Peso de los actores

Se asigna un factor clasificándolos si son actores simples (1), promedios (2) o complejos (3). [Pressman 2006]

En nuestro caso tenemos un único actor:

Usuario: interactúa con una interfaz gráfica. Factor 3.

El peso total de los actores es la suma de los factores de cada actor:

$$\textbf{Peso de los actores} = 3 \text{ (usuario)} = \mathbf{3}$$

Peso de los casos de uso

Sólo se consideran los casos de uso que interactúan con actores, su clasificación depende del número de transacciones del caso de uso, pudiendo ser:

- Simples: representan 3 transacciones o menos (factor 5)
- Promedio: representan de 4 a 7 transacciones (factor 10)
- Complejo: representan más de 7 transacciones (factor 15)

Veamos una tabla con los factores de los casos de uso definidos en el análisis:

Caso de uso	Nº transacciones	Factor
Visualizar refactorizaciones según clasificación	3	5
Refrescar visualización de refactorizaciones	2	5
Añadir filtro de refactorizaciones	6	10
Seleccionar opción aplicar filtro	3	5
Eliminar filtro de refactorizaciones	3	5
Eliminar todos los filtros	2	5
Seleccionar opción ver refactorizaciones filtradas	2	5
Visualizar detalle de refactorización	3	5
Añadir clasificación	3	5
Editar clasificación	4	10
Eliminar clasificación	3	5
Añadir categoría a una clasificación	3	5
Renombrar categoria de una clasificación	4	10
Eliminar categoria de una clasificación	5	10
Mostrar resumen de elemento seleccionado	3	5
Realizar búsqueda de elemento	3	5

Tabla 14: Peso de los casos de uso

El peso total de los casos de uso es la suma de los factores de cada caso de uso:

$$\text{Peso de los casos de uso} = \sum F_i = 100$$

UUCP: Puntos de Casos de Uso sin Ajustar.

Se calcula sumando los pesos de los actores y de los casos de uso.

$$UUCP = \text{Peso actores} + \text{Peso de casos de uso}$$

$$\text{UUCP} = 3 + 100 = 103$$

Peso de los factores técnicos: se calcula con el TFC Factor Técnico de Complejidad

$$\text{TFC} = 0,6 + (0,01 \times \sum (F_i \times \text{Peso}_i))$$

;donde:

Peso es un número de 1 al 5 que se asigna según su importancia a cada uno de los factores de la tabla siguiente:

Factor	F_i	Peso	$F_i \times \text{Peso}$
Sistema Distribuido	2	1	2
Tiempos de respuesta críticos	1	3	3
En Línea	1	1	1
Procesos internos complejos	1	5	5
Código reutilizable	1	4	4
Fácil de instalar	0,5	5	2,5
Fácil de utilizar	0,5	5	2,5
Portable	2	3	6
Fácil de modificar	1	5	5
Concurrencia	1	1	1
Incluye características de seguridad	1	1	1
Acceso a software creado por otras compañías	1	4	4
Incluye facilidades de aprendizaje para el usuario	1	4	4

Tabla 15: Peso de los factores técnicos

$$\sum (F_i \times \text{Peso}_i) = 41$$

$$\text{TFC} = 0,6 + (0,01 \times 41) = 1,01$$

EF: Factor de Entorno. Indica el conocimiento que tiene el programador del tipo de aplicaciones que desarrolla. Para calcularlo se le asignan unos pesos a una serie de factores, desde 0 (nada de conocimiento previo) hasta 5 (experto).

$$\text{EF} = 1,4 + (-0,03 \times \sum (F_i \times \text{Peso}))$$

Factor	F_i	Peso	$F_i \times Peso$
Familiarizado con ciclo de vida	1,5	3	4,5
Experiencia con este tipo de aplicaciones	0,5	2	1
Experiencia en orientación a objetos	1	4	4
Capacidad de liderazgo del analista	0,5	3	1,5
Motivación	1	3	3
Requisitos estables	2	4	8
Trabajadores a tiempo parcial	-1	3	-3
Lenguaje de programación difícil	-1	3	-3

Tabla 16: Peso de los factores de entorno

$$\sum (F_i \times Peso_i) = 16$$

$$EF = 1,4 + (-0,03 \times 16) = 0,92$$

UCP: puntos de casos de uso.

$$UCP = UUCP \times TFC \times EF$$

$$UCP = 103 \times 1,01 \times 0,92 = 95,7076$$

Con todos los datos calculados ya podemos obtener las horas hombre necesarias.

Tenemos que fijarnos en los factores de puntuación del EF que son inferiores al nivel promedio.

$$\text{Horas persona} = 20 \times UCP = 20 \times 95,7076 = 1.914,152$$

Si en el proyecto trabajan 2 personas, durante 7 horas diarias y 20 días al mes:

$$\begin{aligned} \text{Tiempo} &= 1914,152 \text{ Horas Persona}/2 Personas \times 1 \text{ Día}/7 \text{ Horas} \times 1 \text{ Mes}/20 \text{ Días} = \\ &6,8 \text{ meses} \end{aligned}$$

Nuestro proyecto ha sido desarrollado en un periodo total de 6 meses pero durante estos meses han existido etapas en las que se ha trabajado de una forma más intensa superado los 20dias/mes de trabajo que han sido marcados como referencia, debido a necesidades del proyecto y también por la propia motivación del equipo de trabajo, de ahí se justifica que esta estimación obtenida para el tiempo necesario para el desarrollo del proyecto sea algo superior, en concreto algo más de 6 meses y medio.

Universidad de Burgos
ESCUELA POLITÉCNICA SUPERIOR
INGENIERÍA INFORMÁTICA



Anexo 2. Especificación de Requisitos

Clasificación de Refactorizaciones
Dynamic Refactoring Plugin 3.0

Alumnos:
Míryam Gómez San Martín
Íñigo Mediavilla Saiz

Tutor:
Raúl Marticorena Sánchez

Burgos, Julio de 2011

**ANEXO II
ESPECIFICACIÓN DE REQUISITOS**

ÍNDICE DE CONTENIDO

Anexo II	
<i>Especificación de Requisitos.....</i>	151
<i>Lista de cambios.....</i>	157
1. INTRODUCCIÓN.....	159
2. OBJETIVOS DEL PROYECTO.....	160
2.1. Lista de Cambios.....	163
3. LISTA DE USUARIOS PARTICIPANTES.....	164
4. CATÁLOGO DE REQUISITOS DEL SISTEMA.....	164
4.1. Actores.....	164
4.2. Requisitos funcionales.....	165
4.2.1. RF relativos al Análisis del Catálogo de Refactorizaciones.....	165
4.2.2. RF relativos a la Gestión del Clasificaciones.....	167
4.2.3. RF relativos al Análisis de Elementos	169
4.2.4. RF relativos a la Gestión del Plugin.....	169
4.3. Requisitos no funcionales.....	170
4.4. Requisitos de información.....	173
4.4.1. Metamodelo mediante lenguaje minimal.....	174
4.4.2. Definición de refactorizaciones y clasificaciones.....	174
4.4.3. Repositorio de usuario único.....	174
5. ESPECIFICACIÓN DE REQUISITOS.....	175
5.1. Diagramas de casos de uso.....	175
5.2. Plantillas de casos de uso.....	180
6. INTERFAZ DE USUARIO.....	202
6.1. Vista del catálogo de refactorizaciones.....	202
6.2. Primera página del asistente de refactorizaciones.....	204
6.3. Editor de clasificaciones.....	205
6.4. Página del asistente de entradas, predicados y acciones	207

ÍNDICE DE ILUSTRACIONES

Ilustración 14: Diagrama Caso de Uso del Sistema.....	176
Ilustración 15: Diagrama Caso de Uso Analizar Catálogo de Refactorizaciones.....	178
Ilustración 16: Diagrama Caso de Uso Gestionar Clasificaciones.....	179
Ilustración 17: Diagrama Caso de Uso Analizar elementos de Refactorizaciones.....	179
Ilustración 18: Diagrama Caso de Uso Gestionar Plugin.....	180
Ilustración 19: Primer prototipo de la vista del catálogo de refactorizaciones.....	202
Ilustración 20: Segundo prototipo de la vista del catálogo de refactorizaciones.....	203
Ilustración 21: Prototipo definitivo de la vista del catálogo de refactorizaciones.....	204
Ilustración 22: Prototipo con las modificaciones de la primera página del asistente.....	205
Ilustración 23: Primer prototipo del editor de clasificaciones.....	206
Ilustración 24: Segundo prototipo del editor de clasificaciones.....	206
Ilustración 25: Prototipo definitivo del editor de clasificaciones.....	207
Ilustración 26: Prototipo de página del asistente basado en entradas extensibles.....	208
Ilustración 27: Prototipo de página del asistente basado en una tabla.....	209
Ilustración 28: Prototipo de página del asistente definitivo.....	210

ÍNDICE DE TABLAS

Tabla 17: Actor usuario.....	165
Tabla 18: RNF-1: Portabilidad.....	172
Tabla 19: RNF-2: Facilidad de Uso.....	173
Tabla 20: Plantilla caso de uso visualizar refactorizaciones según clasificación.....	181
Tabla 21: Plantilla caso de uso refrescar visualización de refactorizaciones.....	182
Tabla 22: Plantilla caso de uso añadir filtro de refactorizaciones.....	183
Tabla 23: Plantilla caso de uso seleccionar opción aplicar filtro.....	184
Tabla 24: Plantilla caso de uso eliminar filtro de refactorizaciones.....	185
Tabla 25: Plantilla caso de uso eliminar todos los filtros de refactorizaciones.....	186
Tabla 26: Plantilla caso de uso seleccionar opción ver refactorizaciones filtradas.....	187
Tabla 27: Plantilla caso de uso visualizar detalle refactorización.....	188
Tabla 28: Plantilla caso de uso añadir clasificación.....	189
Tabla 29: Plantilla caso de uso editar clasificación.....	190
Tabla 30: Plantilla caso de uso eliminar clasificación.....	191
Tabla 31: Plantilla caso de uso añadir categorías a una clasificación.....	192
Tabla 32: Plantilla caso de uso renombrar categorías de una clasificación.....	193
Tabla 33: Plantilla caso de uso eliminar categorías de una clasificación.....	194
Tabla 34: Plantilla caso de uso mostrar resumen elemento seleccionado.....	195
Tabla 35: Plantilla caso de uso realizar búsqueda de elementos.....	196
Tabla 36: Plantilla caso de uso instalar plugin.....	198
Tabla 37: Plantilla caso de uso actualizar plugin.....	200
Tabla 38: Plantilla caso de uso desinstalar plugin.....	201

LISTA DE CAMBIOS

Número	Fecha	Descripción	Autor/es
0	23/04/11	Añadido apartado dedicado a la introducción y definidos los objetivos del proyecto así como la lista de cambios de los mismos.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
1	07/05/11	Incluida lista de usuarios participantes y apartado dedicado a los requisitos funcionales, no funcionales y de información.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
2	11/05/11	Añadidos casos de uso y apartado de interfaz gráfica.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
3	14/05/11	Incluidas plantillas de casos de uso.	Míryam Gómez San Martín Íñigo Mediavilla Saiz

1. INTRODUCCIÓN

En este documento se describen los requisitos que la aplicación debe de cumplir para tener un comportamiento satisfactorio. Se definen aspectos generales del funcionamiento sin ofrecer detalles de cómo se ha conseguido llevar a cabo.

La relación de requisitos se ha obtenido durante las primeras reuniones con nuestro tutor, Raúl Marticorena Sánchez, en las que se establecieron los objetivos del proyecto. El principal es el de dotar de información añadida a las refactorizaciones con el fin de poder realizar una clasificación de las mismas así como permitir realizar búsquedas sobre estas, además de la mejora funcional del plugin en varios aspectos que posteriormente van a ser detallados.

Como ciclo de vida para el desarrollo software de nuestra aplicación hemos elegido una metodología ágil, tomando aspectos de *SCRUM* [Schwaber 2004], para la gestión y el control del proyecto, con prácticas de *eXtreme Programming* [Beck 2000], para las técnicas de desarrollo, todo ello adaptándolo a las medidas y necesidades de nuestro proyecto.

Para el modelado del presente análisis de requisitos así como para el *Anexo 3 - Especificación del Diseño* se han utilizado diagramas *UML* [Rumbaugh 2005], que ayudan a presentar de forma eficiente los aspectos mas importantes del desarrollo.

A continuación presentaremos los objetivos del proyecto y el catálogo de requisitos, que incluye requisitos funcionales, no funcionales y de información. En la parte final de este anexo encontraremos las especificaciones de los requisitos, con diagramas de casos de uso y sus correspondientes plantillas.

Por último, se muestra la evolución que han ido sufriendo los prototipos de la interfaz del plugin realizados a lo largo del proyecto.

2. OBJETIVOS DEL PROYECTO

Los objetivos del proyecto se han determinado durante las primeras reuniones con el tutor. Algunos han sufrido algunos cambios como se detalla posteriormente.

El objetivo principal del proyecto es hacer evolucionar el plugin mediante la adición de nuevas funcionalidades y la mejora de las ya existentes. Además, también se deberán realizar mejoras sobre la interfaz gráfica de usuario que permitan facilitar la interacción de este con la aplicación, consiguiendo una mejora en la usabilidad de la misma.

Este objetivo principal se puede desglosar en los siguientes:

OBJ 1: Permitir realizar refactorizaciones sobre proyectos reales

Versiones previas del plugin permitían refactorizaciones sobre proyectos software muy simples con limitaciones estrictas en cuanto a tamaño, y a otros aspectos, se refiere. Estas limitaciones hacían que el plugin no pudiera ser utilizado para proyectos reales dado que la mayoría no cumplían con los requisitos. Por esa razón, se marcó como objetivo eliminar esos requisitos con el fin de permitir a desarrolladores utilizar el plugin para sus proyectos independientemente de su tamaño.

A lo largo de esta nueva versión del proyecto se han ido incorporando, según se iban haciendo disponibles, nuevas versiones de las bibliotecas MOON y JavaMOON que superaban esas limitaciones. Además de ser necesario modificar el plugin para adaptarlo y que así pudiese hacer uso de las nuevas versiones de las bibliotecas, también ha sido necesario realizar ciertos cambios para que se hiciera posible el trabajo en este nuevo contexto de proyectos de mayor envergadura.

OBJ 2: Análisis de un catálogo de refactorizaciones

Debido al elevado número de refactorizaciones de las que dispone el plugin, era difícil para el usuario comprenderlas en su totalidad, saber cuales era adecuado utilizar en cada momento y cuales estaban relaciones entre sí.

Dado que no se disponía de ningún mecanismo que diese solución a este problema, se planteó como objetivo prioritario desarrollar una vista que permitiera mostrar al usuario

todas las refactorizaciones disponibles clasificadas según categorías. Además esa vista debía permitir mostrar toda la información disponible sobre cada una de las refactorizaciones, de forma organizada. También se debía permitir al usuario escoger la clasificación a utilizar con la finalidad de agrupar las refactorizaciones, así como aplicar filtros para mostrar exclusivamente las refactorizaciones que cumplieran con ciertos criterios.

Para hacer posible todo esto, sería necesario implementar un completo sistema de clasificación con un editor para crear clasificaciones y una manera de asignar categorías a las refactorizaciones.

Además de asociar a las refactorizaciones categorías a las que pertenecen, también se consideró necesario incluir información adicional en forma de palabras clave. Con todo ello se facilitaría considerablemente la búsqueda y el filtrado de refactorizaciones.

OBJ 3: Simplificación en la creación de refactorizaciones

Uno de los apartados más difíciles de asimilar para los usuarios de la aplicación era el de la creación de refactorizaciones. Conceptos como las entradas, los predicados y las acciones son difíciles de comprender en un primer momento. Además el usuario a la hora de crear una refactorización sólo contaba con los nombres de cada uno de los elementos y su documentación javadoc para entender su funcionamiento. Este hecho unido al gran número de elementos dificultaba mucho al usuario sacar el máximo provecho a las distintas posibilidades que esta herramienta aportaba.

Por todo lo anterior se determinó implementar una serie de mejoras en el asistente de creación de refactorizaciones. En primer lugar se decidió añadir por cada uno de los tipos de elementos (entradas, acciones y predicados) un panel con un resumen y la descripción del objeto seleccionado. En ese resumen se incorporaría información adicional sobre el elemento, con el fin de facilitar al usuario el proceso de creación de refactorizaciones. Finalmente se debería mejorar la búsqueda sobre elementos, pasando de una búsqueda en la que únicamente se utilizaba su nombre para realizar el filtrado a otra en la que se incluyera el campo descripción. Los elementos resultantes de la búsqueda deberían de aparecer ordenados por su relevancia.

OBJ 4: Resolver problema de versionado de refactorizaciones

Ante actualizaciones del plugin se presentaba el problema de que si se querían incorporar nuevas refactorizaciones podían presentarse conflictos con las modificaciones que el usuario hubiera realizado por su cuenta. Lo mismo podía ocurrir para las clasificaciones. Este era un problema importante dado que el conflicto iba a presentarse por cada nueva versión de la herramienta liberada.

La solución a este problema se planteó en base a la definición de dos tipos diferentes de refactorizaciones y clasificaciones: del usuario y del plugin. Las del plugin serían las que el plugin suministra en cada versión y no podrían ser editadas por el usuario. Las de usuario pertenecerían como su propio nombre indica al usuario y podrían ser editadas a su voluntad. Además también existiría la posibilidad de crear nuevas refactorizaciones a partir de cero o como copias de otras, independientemente del tipo que fuesen.

OBJ 5: Disponibilidad de un repositorio único de refactorizaciones

Disponibilidad de un repositorio único de clasificaciones

La evolución que ha ido experimentando el plugin, su crecimiento y la incorporación de numerosas funcionalidades, ha dado lugar a que se hiciese necesaria la gestión de refactorizaciones en diferentes puntos del plugin, provocando que la implementación de la lectura y la modificación de refactorizaciones se repitiese en un excesivo número de clases del mismo. Esto además de poder provocar inconsistencias en la información disponible de las refactorizaciones es menos eficiente y dificulta el mantenimiento de la aplicación, repercutiendo negativamente en la calidad del producto software final.

Es por ello que, a raíz de producirse esta situación, se planteó un nuevo objetivo: la existencia de un repositorio único de refactorizaciones para hacer disponible a todo el plugin la información relativa a las refactorizaciones. Este debería ser el encargado único de su lectura y modificación.

La misma solución se propuso para el caso de las clasificaciones, es decir, un repositorio único para la gestión de estas.

OBJ 6: Instalación automática del plugin

En versiones previas, la instalación del plugin consistía en copiar la carpeta del

proyecto generado sobre la instalación de Eclipse del usuario. Este modelo presentaba las desventajas de que la instalación no era lo suficientemente sencilla y de que se obligaba al usuario a tener que reemplazar carpetas y copiar ficheros para preservar modificaciones por cada nueva versión del plugin que se quisiera instalar. Eso sin contar con que para realizar dicha instalación correctamente se necesitaba disponer de las versiones específicas de Eclipse con las dependencias necesarias.

En este caso, el objetivo era que todo el proceso de instalación se facilitara y se ofreciese la posibilidad de que la actualización del plugin se pudiese realizar de forma automática. El nuevo modelo debería contar con todas las ventajas que el sistema de instalación de plugins de Eclipse ofrece: instalación del plugin, descarga automática de las dependencias, actualización automática, posibilidad de restaurar a versiones anteriores o de desinstalación integrada. Además, como mejora adicional se sugirió la opción de permitir al usuario instalar el plugin a través de Internet.

2.1. *Lista de Cambios*

A continuación se especifican los cambios que han sufrido los objetivos del proyecto que inicialmente se establecieron.

- El objetivo 5 fue añadido durante el desarrollo del proyecto. Fue debido a que se vio clara la necesidad de introducir un repositorio único de refactorizaciones para la gestión de las mismas. Del mismo modo se hizo lo propio para el correspondiente a las clasificaciones.
- El objetivo 6 no era un objetivo inicial. Sin embargo, dado que Eclipse ofrece la posibilidad de implementar la funcionalidad necesaria para hacer posible la instalación del plugin de forma automática decidimos que era conveniente aprovechar esta situación para dotar así al plugin de una valor añadido que la gran mayoría de los plugin comerciales ya ofrecen. Además, se consideró una herramienta de gran utilidad para el usuario ya que le permite realizar el proceso de forma totalmente transparente para él.

3. LISTA DE USUARIOS PARTICIPANTES

Debido a que este no es un proyecto desarrollado ante la demanda de un cliente sino que, más bien, se trata de un proyecto enmarcado dentro de un trabajo de investigación más amplio, en el proceso de obtención de requisitos ha participado principalmente el tutor de dicho proyecto, Raúl Marticorena Sánchez, que ha sido el encargado de dirigir el mismo así como de colaborar activamente en la solución de algunos problemas que han surgido durante su desarrollo.

4. CATÁLOGO DE REQUISITOS DEL SISTEMA

En esta sección se definen formalmente los requisitos que finalmente han sido aceptados, dividiéndose en apartados uno por cada tipo distinto de requisitos. Estos son: requisitos funcionales, requisitos no funcionales y requisitos de información.

Además también se identificarán los actores que van a interactuar con el sistema.

4.1. Actores

Los actores son personas, sistemas o cualquier ente que interactúa con el sistema [Rumbaugh 2005]. Un actor se identifica por uno o varios roles, es decir, una misma persona puede corresponderse con varios actores y viceversa, un actor puede estar interpretado por distintas personas.

En nuestro sistema identificamos un único actor, el usuario.

El usuario puede ser cualquiera que utilice la aplicación, un profesor, un alumno o una persona diferente que necesite realizar las tareas que el plugin ofrece. Por tanto, el usuario, entre otras cosas, será el encargado de crear refactorizaciones, configurar las ya existentes y de aplicarlas en su código fuente cuando así lo crea oportuno.

ACT-1	Usuario
Autores	Miryam Gómez San Martín Iñigo Mediavilla Saiz
Fuentes	Raúl Marticorena Sánchez
Descripción	El usuario solicita la ejecución de las tareas o funcionalidades disponibles en la aplicación.
Comentarios	Se ha favorecido que cualquier usuario con conocimientos básicos de refactorizaciones pueda utilizar nuestra aplicación.

Tabla 17: Actor usuario

4.2. Requisitos funcionales

Definen las acciones que lleva a cabo el sistema para cumplir los objetivos planteados con la información que tiene almacenada [Kroll 2003].

En esta sección se va a definir las funcionalidades que se quiere conseguir incorporar en el sistema final. Los requisitos del sistema tienen mucho que ver con lograr alcanzar los objetivos finales del sistema, es por ello que van a estar estrechamente relacionados con los objetivos del proyecto expuestos con anterioridad.

4.2.1. RF relativos al Análisis del Catálogo de Refactorizaciones

En este apartado se detallan los distintos requisitos funcionales que hacen referencia al análisis del catálogo de refactorizaciones.

RF 1: Visualizar refactorizaciones según clasificación

Debido al gran número de refactorizaciones de las que dispone el plugin y para permitir facilitar la comprensión de estas al usuario, como ya hemos comentado, se deberá desarrollar una vista que le permita visualizar el catálogo de refactorizaciones disponibles.

En esta vista se le ofrecerá la posibilidad al usuario de visualizar las refactorizaciones clasificadas según las categorías de la clasificación seleccionada, para ello bastará con que

éste elija una clasificación de entre las que se muestren como disponibles.

RF 2: Refrescar visualización de refactorizaciones

En la vista destinada a la visualización del catálogo de refactorizaciones se deberá dar la posibilidad de refrescar la misma. Esta será una funcionalidad importante ya que la modificación de refactorizaciones puede realizarse por distintas vías, por ejemplo por un cambio en su propia definición pero también podría ser por un cambio en una categoría a la que esta pertenece. Es por ello que es importante que el usuario tenga presente que cuando realice algún cambio que afecte a las refactorizaciones deberá utilizar esta funcionalidad para que la información que la vista muestre se encuentre actualizada.

RF 3: Añadir filtro de refactorizaciones

Se deberá incorporar en la vista la funcionalidad de poder aplicar filtros al catálogo de refactorizaciones con el objetivo de mostrar exclusivamente aquellas refactorizaciones que cumplan con ciertos criterios que el usuario haya querido establecer. Los filtros creados se mostrarán para que en todo momento el usuario tenga conocimiento de ellos.

Estos filtros se podrán realizar por categorías, texto o palabras clave. La sintaxis a la que deberán responder será la siguiente:

- Categoría category:'clasificación'@'categoría'
Por ejemplo: category:scope@method

- Texto text:'text'
Por ejemplo: text:add

- Palabra clave key: 'palabra clave'
Por ejemplo: key:annotation

RF 4: Seleccionar opción aplicar filtro

Como acabamos de comentar, los filtros que el usuario haya querido establecer para aplicar al catálogo de refactorizaciones se mostrarán para que en todo momento tenga conocimiento de los mismos. Los filtros al crearse serán aplicados directamente sobre el catálogo pero se le deberá de ofrecer la posibilidad al usuario de mantener filtros creados pero que no sean aplicados si no se considera oportuno, es decir, que tenga la libertad de aplicar o no filtros creados.

RF 5: Eliminar filtro de refactorizaciones

De la misma forma que el usuario puede crear diversos filtros para aplicar sobre el catálogo de refactorizaciones, también dispondrá de la posibilidad de una vez que estos han sido creados eliminar aquellos que así lo crea conveniente.

RF 6: Eliminar todos los filtros de refactorizaciones

Para facilitar el borrado de los filtros, el usuario podrá eliminar de una sola vez todos los filtros que haya creado con anterioridad.

RF 7: Seleccionar opción ver refactorizaciones filtradas

Debido a que el usuario puede crear diversos filtros para aplicar sobre el catálogo de refactorizaciones se considera oportuno incluir la opción que permita al usuario seleccionar si desea visualizar o no las refactorizaciones que han sido filtradas conforme a los filtros que se encuentren aplicados en la vista de refactorizaciones, en la que se muestran clasificadas según las categorías de la clasificación que haya sido seleccionada por este.

RF 8: Visualizar detalle refactorización

Además el usuario tendrá la posibilidad de poder seleccionar una refactorización del catálogo de refactorizaciones para la que visualizar toda su información disponible, esta será mostrada de forma organizada.

4.2.2. RF relativos a la Gestión del Clasificaciones

En este apartado se detallan los requisitos funcionales que hacen referencia a la gestión de clasificaciones.

RF 9: Añadir clasificación

Entre la información que se almacena asociada a una refactorización se encuentra la relativa a las clasificaciones de la misma. Para cada uno de los tipos de clasificaciones existentes se podrá clasificarla atendiendo a las categorías disponibles para cada una de ellas. El plugin suministra consigo algunas clasificaciones pero también se le dará al propio usuario la posibilidad de crear las suyas propias como él considere oportuno, es decir, podrá añadir nuevas clasificaciones.

Para añadir una nueva clasificación el usuario deberá facilitar la siguiente información requerida:

- Nombre de la clasificación.
- Descripción representativa de la clasificación.
- Indicar si es uni o multicategoría, es decir, si las refactorizaciones van a poder tener asociada una única o, por el contrario, varias categorías para esta clasificación.
- Añadir las categorías que se consideren oportunas.

RF 10: Editar clasificación

Una vez que una clasificación es creada por parte del usuario esta puede ser editada, es decir, cualquier clasificación propia del usuario podrá ser seleccionada para proceder a su modificación.

RF 11: Eliminar clasificación

De la misma forma, se le ofrecerá la posibilidad al usuario de poder eliminar, por las razones que fuere, las clasificaciones que previamente haya creado el mismo.

RF 12: Añadir categorías a una clasificación

Como acabamos de ver, para poder clasificar las refactorizaciones es necesario que las clasificaciones contengan diferentes categorías a las que asociar estas. Es por ello, que debe de ser accesible al usuario la adición de nuevas categorías a una clasificación que haya sido definida previamente por este.

RF 13: Renombrar categorías de una clasificación

Una vez que una clasificación propia del usuario dispone de categorías estas podrán ser renombradas en caso de que se quisiese la modificación del nombre de la misma.

RF 14: Eliminar categorías de una clasificación

De igual modo, se deberá ofrecer la posibilidad al usuario de eliminar categorías que previamente haya creado él mismo y por lo tanto estén asociadas a clasificaciones propias.

4.2.3. RF relativos al Análisis de Elementos

Este apartado recoge los requisitos funcionales que hacen referencia al análisis de los distintos elementos disponibles (entradas/acciones/predicados).

RF 15: Mostrar resumen elemento seleccionado

Cuando el usuario se encuentre en el asistente dedicado a la creación o modificación de refactorizaciones, en concreto, en las páginas destinadas a mecanismos, es decir, a las entradas, acciones o predicados y seleccione un elemento de la lista en la que se muestran los disponibles, se deberá mostrar una zona dedicada al resumen del elemento en cuestión. En esta se mostrará su descripción así como la relación de refactorizaciones que ya lo están utilizando en su definición.

RF 16: Realizar búsqueda de elementos

Cuando el usuario se encuentre en el asistente dedicado a la creación o modificación de refactorizaciones, en concreto, en las páginas destinadas a mecanismos, es decir, a las entradas, acciones o predicados podrá realizar búsquedas de elementos sobre los que se encuentran disponibles. Internamente la búsqueda se basará en el *parseo* de la descripción del elemento que se encuentra en el correspondiente fichero de documentación javadoc. A partir de esta información y una vez eliminadas las palabras carentes de significado se pasará a realizar un proceso de indexación de las mismas, con el objetivo de poder listar las concordancias encontradas de acuerdo a un orden de relevancia con respecto a los términos buscados por el usuario.

4.2.4. RF relativos a la Gestión del Plugin

Este apartado recoge los requisitos funcionales referentes a la gestión del plugin.

RF 17: Instalar plugin

El usuario deberá de disponer de la posibilidad de realizar la instalación del plugin de refactorizaciones mediante el sistema de instalación de plugins que Eclipse ofrece. Para ello, deberá seguir los pasos que se van indicando en el asistente de instalación de tal forma que este proceso sea totalmente transparente para el usuario.

RF 18: Actualizar plugin

Una vez que ya ha sido instalado el plugin de refactorizaciones, a lo largo del tiempo,

puede darse el caso de que exista una nueva versión disponible para este. El usuario deberá poder chequear la existencia de la nueva versión disponible y en caso de que así fuese poder realizar su actualización de forma automática.

RF 19: Desinstalar plugin

Si así lo considera oportuno, el usuario podrá realizar la desinstalación del plugin de refactorizaciones mediante la opción disponible para tal efecto.

4.3. Requisitos no funcionales

En este apartado se detallan los requisitos no funcionales, los cuales responden a las necesidades técnicas o de calidad del sistema software que el propio proyecto debe cumplir, independientemente de la funcionalidad del mismo. Estos requisitos se han basado en los definidos en [Abran 2010]. Generalmente no es hasta la fase de diseño e implementación cuando se tienen en cuenta.

Funcionalidad

- *Adecuación*: capacidad del producto software para proporcionar un conjunto apropiado de funciones para tareas y objetivos de usuario especificados.
- *Exactitud*: capacidad del producto software para proporcionar los resultados o efectos correctos o acordados, con el grado necesario de precisión.
- *Interoperabilidad*: capacidad del producto software para interactuar con uno o más sistemas especificados.
- *Seguridad*: capacidad del producto software para proteger información y datos de manera que las personas o sistemas no autorizados no puedan leerlos o modificarlos, al tiempo que no se deniega el acceso a personas o sistemas autorizados.

Fiabilidad

- *Madurez*: capacidad del producto software para evitar fallar como resultado de fallos en el software.
- *Tolerancia a Fallos*: capacidad que tiene el software para mantener un nivel especificado de prestaciones en caso de fallos del software o de infringir sus

interfaces especificados.

- *Facilidad de Recuperación:* capacidad del producto software para reestablecer un nivel de prestaciones especificado y de recuperar los datos directamente afectados en caso de fallo.

Facilidad de uso

- *Facilidad de Comprensión:* capacidad del producto software que permite al usuario entender si el software es adecuado y cómo puede ser usado para unas tareas o condiciones de uso particulares.
- *Facilidad de Aprendizaje:* capacidad del producto software que permite al usuario aprender sobre su aplicación.
- *Atracción:* capacidad del producto software para ser atractivo al usuario.

Eficiencia

- *Tiempo de Comportamiento:* capacidad que tiene el producto software para proporcionar tiempos de respuesta, de proceso y de potencia apropiados, bajo condiciones determinadas.
- *Utilización de Recursos:* capacidad del producto software para usar los tipos de recursos y las cantidades adecuadas cuando el software lleva a cabo su función bajo condiciones determinadas.

Facilidad de Mantenimiento

- *Facilidad de Análisis:* capacidad que tiene el producto software para serle diagnosticadas deficiencias o causa de los fallos en el software, o para identificar las partes que han de ser modificadas.
- *Facilidad de Cambio:* capacidad del producto software que permite que una determinada modificación sea implementada.
- *Estabilidad:* capacidad del producto software para evitar efectos inesperados debidos a modificaciones del software.
- *Facilidad de Pruebas:* capacidad del producto software que permite que el software modificado sea validado.

Portabilidad

- *Adaptabilidad*: capacidad del producto software para ser adaptado a diferentes entornos, sin aplicar acciones o mecanismos distintos de los proporcionados para este propósito por el propio software considerado.
- *Facilidad de Instalación*: capacidad del producto software para ser instalado en un entorno especificado.
- *Coexistencia*: capacidad del producto software para coexistir con otro software independiente, en un entorno común, compartiendo recursos comunes.
- *Facilidad de Sustitución*: capacidad del producto software para ser usado en lugar de otro producto software, para el mismo propósito, y en el mismo entorno.

A partir de los requisitos no funcionales que se acaban de exponer, a continuación se detallan algunos más concretos.

RNF-1	Portabilidad
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz
Fuentes	Raúl Marticorena Sánchez
Descripción	El plugin deberá estar diseñado para funcionar correctamente en el entorno de desarrollo Eclipse en máquinas con sistema operativo Windows, Linux y Mac OS. Además se le ha querido dar un valor añadido ofreciendo la posibilidad de que se pueda realizar la instalación de forma automática y por red, favoreciendo así la facilidad en su instalación. También deberá tener la capacidad para coexistir con otro software independiente, en un entorno común como es Eclipse, y compartiendo recursos comunes.
Requisitos asociados	RF 17: Instalar plugin RF 18: Actualizar plugin RF 19: Desinstalar plugin
Frecuencia esperada	Alta
Importancia	Alta
Urgencia	Baja
Comentarios	

Tabla 18: RNF-1: Portabilidad

RNF-2	Facilidad de Uso
Autores	Miryam Gómez San Martín Iñigo Mediavilla Saiz
Fuentes	Raúl Marticorena Sánchez
Descripción	<p>La interfaz debe ser sencilla e intuitiva para facilitar al usuario la comprensión de la misma y la comunicación con la aplicación.</p> <p>Especialmente en la creación/modificación de refactorizaciones, para que el usuario aunque no tenga conocimientos avanzados sobre los elementos de los que se puede componer esta (entradas, acciones y predicados) pueda realizar consultas de manera intuitiva sobre los mismos y obtener la información que considere oportuna para su comprensión.</p> <p>De la misma forma para el análisis del catálogo de refactorizaciones y para la clasificación de estas.</p>
Requisitos asociados	RF 15: Mostrar resumen elemento seleccionado RF 16: Realizar búsqueda de elementos RF 1: Visualizar refactorizaciones según clasificación RF 3: Añadir filtro de refactorizaciones RF 8: Visualizar detalle refactorización RF 9: Añadir clasificación RF 12: Añadir categoría a una clasificación
Frecuencia esperada	Alta
Importancia	Vital
Urgencia	Media
Comentarios	

Tabla 19: RNF-2: Facilidad de Uso

4.4. Requisitos de información

En esta parte se explican los datos relevantes para el usuario, que la aplicación almacena y gestiona.

Aunque éste no es un típico proyecto de gestión, sí tiene un componente importante de manejo de información. La herramienta debe poder manejar información contenida en las clases del sistema software, ser capaz de operar con ella y trasladarla nuevamente a los ficheros fuente correspondientes de donde se obtuvo.

4.4.1. Metamodelo mediante lenguaje minimal

Durante su funcionamiento, la propia herramienta utiliza un formato interno diferente de almacenamiento. Este se trata de un metamodelo descrito mediante el lenguaje minimal MOON. Dicho metamodelo contiene toda la información imprescindible para la aplicación de cambios (producidos por las refactorizaciones) sobre las clases del sistema software tratado. Se puede obtener una descripción más detallada de la definición del metalenguaje MOON en [Crespo 2000] y [Crespo, López, & Marticorena n.d.] .

4.4.2. Definición de refactorizaciones y clasificaciones

Por otro lado la aplicación debe manejar la información contenida en ficheros XML, a partir de los estos obtiene las definiciones de las distintas refactorizaciones, de un plan de refactorizaciones o del conjunto de refactorizaciones disponibles para cada ámbito. Además, debe ser capaz de manejar esa información, operar con ella y obtener como resultado una refactorización ejecutable, un plan ejecutable o la información necesaria para actualizar la vista de refactorizaciones disponibles y el catálogo de refactorizaciones.

Asimismo, deberá ser capaz de construir un fichero XML con la definición de la propia refactorización, en base a los datos que ha suministrado el usuario a través de la interfaz gráfica. También deberá ser capaz de construir el fichero XML con la información relativa al conjunto de entradas de cada una de las refactorizaciones que conforman el plan y de generar y actualizar el XML con las refactorizaciones disponibles para cada ámbito sobre el que se puede ejecutar una refactorización.

Además la aplicación también debe manejar la información correspondiente a la definición de las clasificaciones disponibles, dicha información de igual forma se encuentra contenida en un fichero XML que deberá ser leído para hacer disponible al usuario dicha información en la herramienta. Es por ello que cuando el usuario desee crear nuevas clasificaciones o categorías para una clasificación o bien modificar información relativa a las mismas la aplicación deberá ser capaz de reconstruir el fichero XML.

4.4.3. Repositorio de usuario único

Con esta nueva versión del plugin de refactorizaciones se ha incorporado la existencia de un repositorio de usuario único que será común para todos los espacios de trabajo, el cual contendrá la información relativa a las refactorizaciones y clasificaciones propias del

usuario.

Hasta ahora las modificaciones que el usuario realizaba sobre las refactorizaciones se venían guardando en un directorio dependiente del *workspace* en el que el usuario estaba trabajando. Esto significaba que si el usuario decidía cambiar de espacio de trabajo ya no estaban accesibles los cambios que había realizado con anterioridad en el otro espacio de trabajo. Por tanto, con este repositorio de usuario único se ha conseguido que el usuario preserve las modificaciones que realice independientemente del espacio de trabajo en el que se encuentre, ya que este repositorio es común a todos.

5. ESPECIFICACIÓN DE REQUISITOS

En esta sección se van a desglosar los diferentes requisitos del sistema en casos de uso que posteriormente van a ser detallados mediante el uso de las correspondientes plantillas de casos de uso.

5.1. Diagramas de casos de uso

Como hemos comentado con anterioridad vamos a utilizar los casos de uso [Kroll 2003] como técnica de extracción y especificación de los requisitos funcionales del sistema, describiéndolos a través de plantillas.

Los casos de uso describen lo que el sistema debe de hacer cuando los actores interactúan con él.

En las siguientes ilustraciones se muestran los casos de uso de nuestra aplicación. Cabe mencionar que únicamente se detallan aquellas partes que han sufrido modificaciones significantes respecto a la versión anterior del proyecto y también aquellas funcionalidades de nueva incorporación en el mismo. En caso de querer obtener información de aquellas partes que se mantienen sin sufrir cambio alguno se deberá consultar la documentación de proyectos anteriores [Fuente & Herrero n.d.] y [Fuente de la Fuente n.d.].

Diagrama de Casos de Uso del Sistema

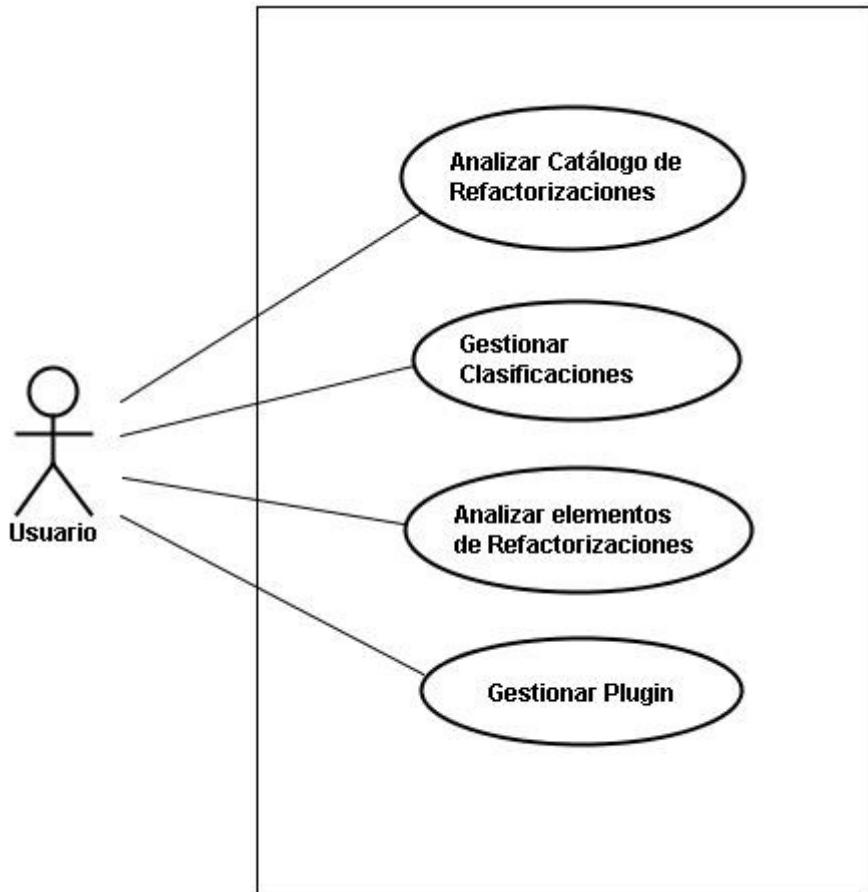


Ilustración 14: Diagrama Caso de Uso del Sistema

Caso de Uso del Sistema

Descripción: Gestiona todas las posibles acciones que un usuario de la aplicación puede llevar a cabo en el sistema.

Caso de Uso Analizar Catálogo de Refactorizaciones

Descripción: Realiza operaciones relativas al análisis del catálogo de refactorizaciones.

Se divide en:

- Visualizar refactorizaciones según clasificación.
- Seleccionar opción aplicar filtro.
- Seleccionar opción ver refactorizaciones filtradas.
- Añadir filtro de refactorizaciones.

- Eliminar filtro de refactorizaciones.
- Eliminar todos los filtros de refactorizaciones.
- Refrescar visualizacion de refactorizaciones.
- Visualizar detalle de refactorizaciones.

Caso de Uso Gestionar Clasificaciones

Descripción: Realiza operaciones relativas a la gestión de clasificaciones.

Se divide en:

- Añadir clasificación.
- Editar clasificación.
- Añadir categorías a una clasificación.
- Renombrar categorías de una clasificación.
- Eliminar categorías de una clasificación.
- Eliminar clasificación.

Caso de Uso Analizar elementos de Refactorizaciones

Descripción: Realiza operaciones relativas al análisis de los elementos de los que se componen las refactorizaciones; entradas, predicados y acciones.

Se divide en:

- Mostrar resumen elemento seleccionado.
- Realizar búsqueda de elementos.

Caso de Uso Gestionar Plugin

Descripción: Permite realizar la gestión del propio plugin, operaciones relativas a la instalación del mismo de forma automática.

Se divide en:

- Instalar plugin.
- Actualizar plugin.
- Desinstalar plugin.

Diagrama de Casos de Uso Analizar Catálogo de Refactorizaciones

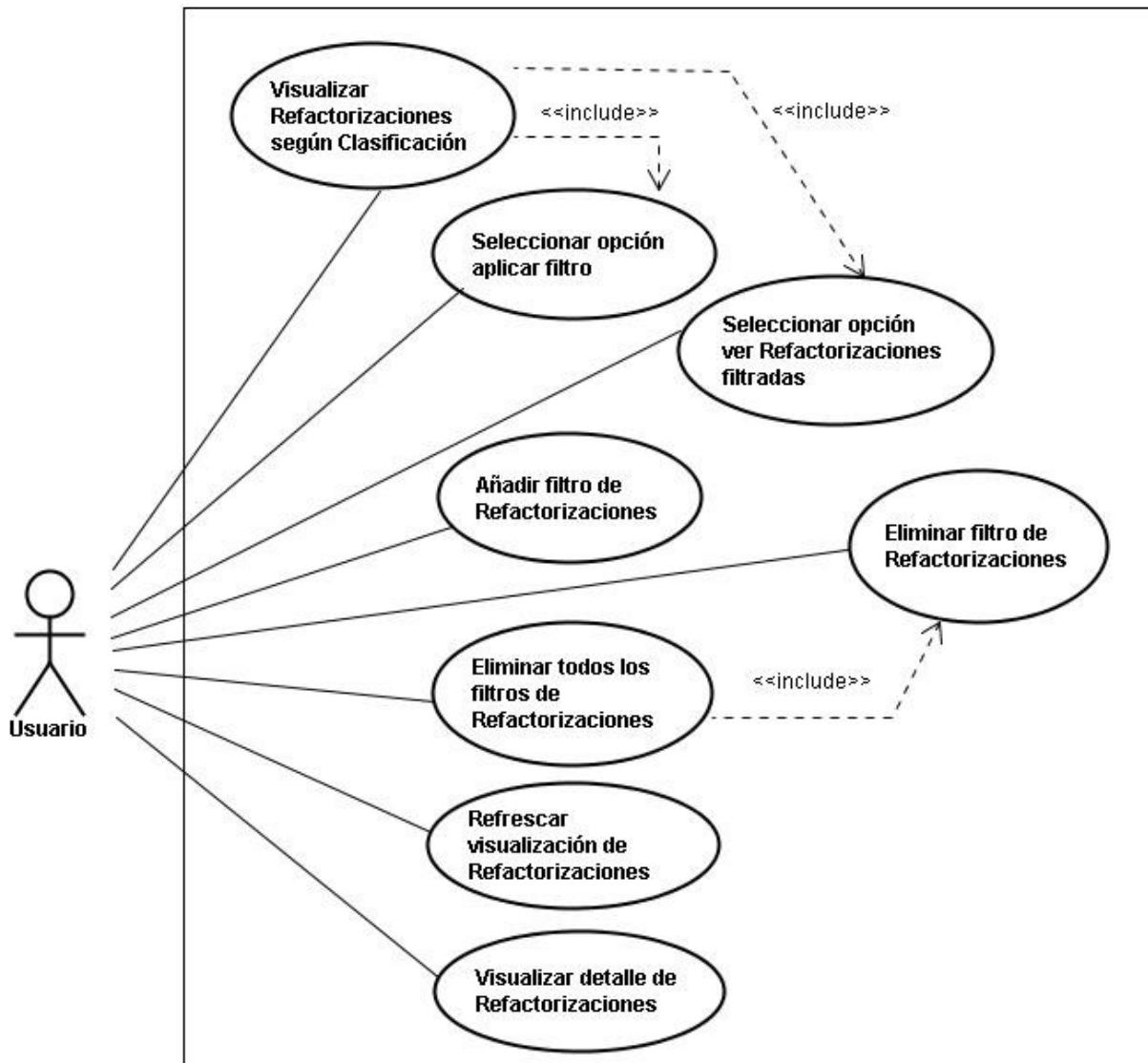


Ilustración 15: Diagrama Caso de Uso Analizar Catálogo de Refactorizaciones

Diagrama de Casos de Uso Gestionar Clasificaciones

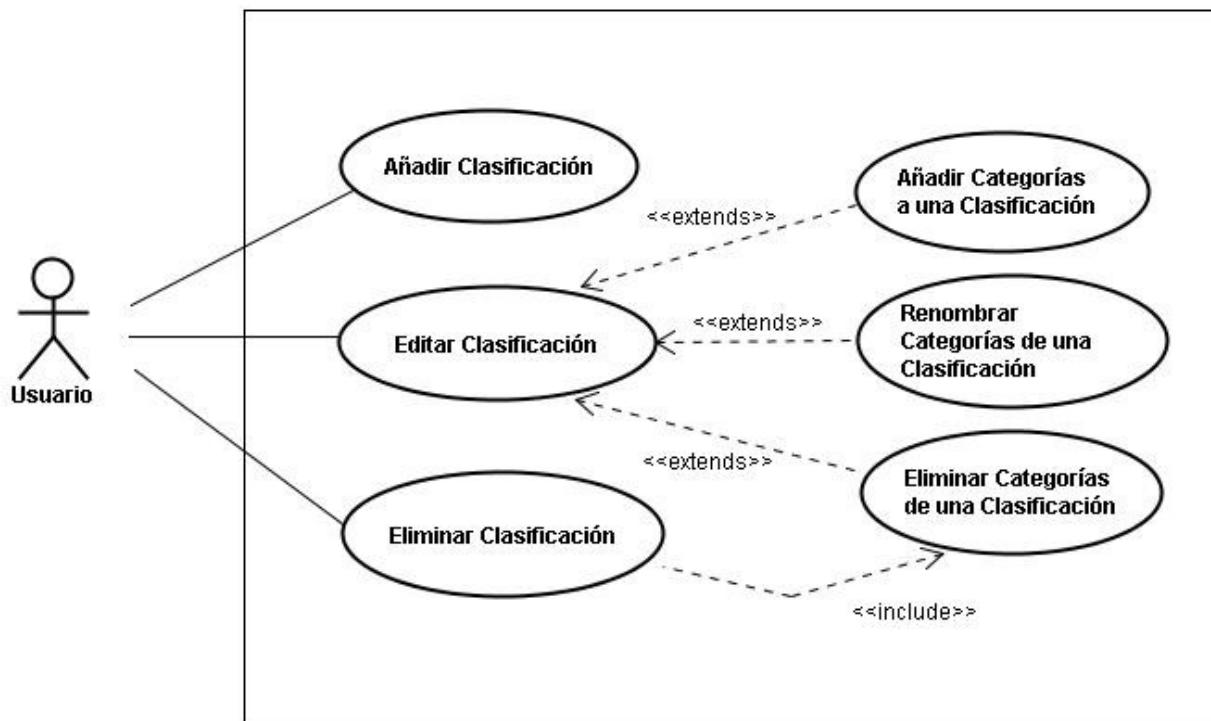


Ilustración 16: Diagrama Caso de Uso Gestionar Clasificaciones

Diagrama de Casos de Uso Analizar elementos de Refactorizaciones

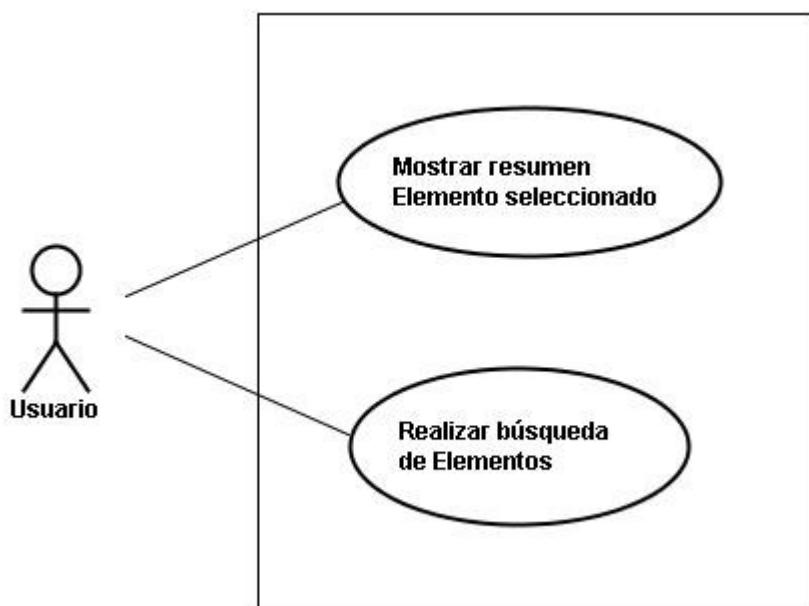


Ilustración 17: Diagrama Caso de Uso Analizar elementos de Refactorizaciones

Diagrama de Casos de Uso Gestionar Plugin

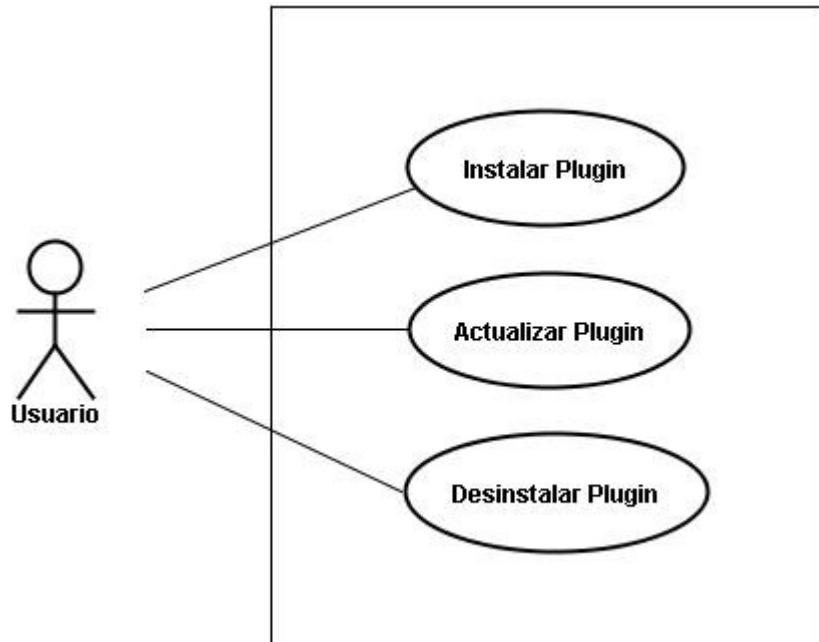


Ilustración 18: Diagrama Caso de Uso Gestionar Plugin

5.2. Plantillas de casos de uso

En este apartado se detallará la información relativa a los casos de uso anteriormente expuestos. Para su descripción se hará uso de una plantilla que recogerá los aspectos más relevantes de cada uno de ellos.

RF-1	Visualizar refactorizaciones según clasificación									
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz									
Fuentes	Raúl Marticorena Sánchez									
Descripción	El usuario visualiza el catálogo de refactorizaciones clasificado según las categorías a las que pertenecen las refactorizaciones, atendiendo a la clasificación que ha seleccionado el usuario.									
Precondición	El usuario se encuentra en la vista del catálogo de refactorizaciones con la intención visualizar las refactorizaciones clasificadas según las categorías a las que pertenecen, atendiendo a la clasificación elegida por el usuario.									
Secuencia normal	<table border="1"> <thead> <tr> <th>Paso</th> <th>Acción</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>El usuario despliega el combo que contiene las clasificaciones para ver las que se encuentran disponibles.</td> </tr> <tr> <td>2</td> <td>El usuario selecciona una clasificación de entre las disponibles, indicando así que desea clasificar a las refactorizaciones mediante esta.</td> </tr> <tr> <td>3</td> <td>El sistema muestra las refactorizaciones clasificadas según las categorías a las que pertenecen, atendiendo a la clasificación elegida.</td> </tr> </tbody> </table>	Paso	Acción	1	El usuario despliega el combo que contiene las clasificaciones para ver las que se encuentran disponibles.	2	El usuario selecciona una clasificación de entre las disponibles, indicando así que desea clasificar a las refactorizaciones mediante esta.	3	El sistema muestra las refactorizaciones clasificadas según las categorías a las que pertenecen, atendiendo a la clasificación elegida.	
Paso	Acción									
1	El usuario despliega el combo que contiene las clasificaciones para ver las que se encuentran disponibles.									
2	El usuario selecciona una clasificación de entre las disponibles, indicando así que desea clasificar a las refactorizaciones mediante esta.									
3	El sistema muestra las refactorizaciones clasificadas según las categorías a las que pertenecen, atendiendo a la clasificación elegida.									
Postcondición	El usuario dispone del catálogo de refactorizaciones clasificado y puede realizar otras tareas sobre el mismo.									
Excepciones	No existe ninguna.									
Frecuencia esperada	Alta / Media / Baja									
Importancia	Alta / Media / Baja									
Urgencia	Alta / Media / Baja									
Comentarios	Este caso de uso tiene relación de dependencia con los casos de uso: <i>Selecciona opción aplicar filtro</i> y <i>Selecciona opción ver refactorizaciones filtradas</i> .									

Tabla 20: Plantilla caso de uso visualizar refactorizaciones según clasificación

RF-2	Refrescar visualización de refactorizaciones	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario refresca la visualización del catálogo de refactorizaciones porque ha realizado algún cambio que afecta a estas.	
Precondición	El usuario se encuentra en la vista del catálogo de refactorizaciones con la intención de refrescar la visualización del catálogo de refactorizaciones para que la información mostrada quede actualizada, ya que previamente ha realizado alguna modificación que las afecta.	
Secuencia normal	Paso	Acción
	1	El usuario pulsa el botón de refresco de la vista que se encuentra en la barra de herramientas de la propia vista.
	2	El sistema refresca la vista actualizando todos los valores y mostrándola con la configuración con la que se encontraba antes de realizar la operación de refresco.
Postcondición	El usuario dispone del catálogo de refactorizaciones refrescado y puede realizar otras tareas sobre el mismo.	
Excepciones	No existe ninguna.	
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	No se considera necesarios.	

Tabla 21: Plantilla caso de uso refrescar visualización de refactorizaciones

RF-3	Añadir filtro de refactorizaciones															
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz															
Fuentes	Raúl Marticorena Sánchez															
Descripción	El usuario añade un filtro de refactorizaciones a la visualización del catálogo de refactorizaciones porque quiere aplicar algún tipo de criterio sobre estas.															
Precondición	El usuario se encuentra en la vista del catálogo de refactorizaciones con la intención de añadir un filtro de refactorizaciones a la visualización del catálogo de refactorizaciones.															
Secuencia normal	<table border="1"> <thead> <tr> <th>Paso</th> <th>Acción</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>El usuario escribe el filtro que desea aplicar.</td> </tr> <tr> <td>2</td> <td>El usuario pulsar el botón para crear filtro.</td> </tr> <tr> <td>3a</td> <td>El sistema valida que la sintaxis del filtro sea correcta.</td> </tr> <tr> <td>3b</td> <td>El sistema valida que no exista ya un filtro idéntico.</td> </tr> <tr> <td>3c</td> <td>El sistema valida que si se trata de un filtro de categorías exista la categoría y/para la clasificación indicada.</td> </tr> <tr> <td>4</td> <td>El sistema aplica el filtro sobre el catálogo de refactorizaciones y en caso de estar seleccionada la opción de mostrar las refactorizaciones filtradas aparecerá un grupo para visualizar y clasificar estas.</td> </tr> </tbody> </table>		Paso	Acción	1	El usuario escribe el filtro que desea aplicar.	2	El usuario pulsar el botón para crear filtro.	3a	El sistema valida que la sintaxis del filtro sea correcta.	3b	El sistema valida que no exista ya un filtro idéntico.	3c	El sistema valida que si se trata de un filtro de categorías exista la categoría y/para la clasificación indicada.	4	El sistema aplica el filtro sobre el catálogo de refactorizaciones y en caso de estar seleccionada la opción de mostrar las refactorizaciones filtradas aparecerá un grupo para visualizar y clasificar estas.
Paso	Acción															
1	El usuario escribe el filtro que desea aplicar.															
2	El usuario pulsar el botón para crear filtro.															
3a	El sistema valida que la sintaxis del filtro sea correcta.															
3b	El sistema valida que no exista ya un filtro idéntico.															
3c	El sistema valida que si se trata de un filtro de categorías exista la categoría y/para la clasificación indicada.															
4	El sistema aplica el filtro sobre el catálogo de refactorizaciones y en caso de estar seleccionada la opción de mostrar las refactorizaciones filtradas aparecerá un grupo para visualizar y clasificar estas.															
Postcondición	El usuario dispone del catálogo de refactorizaciones filtrado y además el filtro que acaba de ser creado en la zona destinada a tal efecto.															
Excepciones	<table border="1"> <thead> <tr> <th>Paso</th> <th>Acción</th> </tr> </thead> <tbody> <tr> <td>3a</td> <td>El sistema no añade el filtro e informa de que la sintaxis no es correcta.</td> </tr> <tr> <td>3b</td> <td>El sistema no añade el filtro e informa de que ya existe un filtro idéntico.</td> </tr> <tr> <td>3c</td> <td>El sistema pregunta al usuario si a pesar de que no existe la categoría y/para la clasificación indicada la quiere incluir. En ese caso, el usuario podrá aceptar o rechazar.</td> </tr> </tbody> </table>		Paso	Acción	3a	El sistema no añade el filtro e informa de que la sintaxis no es correcta.	3b	El sistema no añade el filtro e informa de que ya existe un filtro idéntico.	3c	El sistema pregunta al usuario si a pesar de que no existe la categoría y/para la clasificación indicada la quiere incluir. En ese caso, el usuario podrá aceptar o rechazar.						
Paso	Acción															
3a	El sistema no añade el filtro e informa de que la sintaxis no es correcta.															
3b	El sistema no añade el filtro e informa de que ya existe un filtro idéntico.															
3c	El sistema pregunta al usuario si a pesar de que no existe la categoría y/para la clasificación indicada la quiere incluir. En ese caso, el usuario podrá aceptar o rechazar.															
Frecuencia esperada	Alta / Media / Baja															
Importancia	Alta / Media / Baja															
Urgencia	Alta / Media / Baja															
Comentarios	No se considera necesarios.															

Tabla 22: Plantilla caso de uso añadir filtro de refactorizaciones

RF-4	Seleccionar opción aplicar filtro	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario des/selecciona la opción de aplicar un filtro sobre el catálogo de refactorizaciones.	
Precondición	El usuario se encuentra en la vista del catálogo de refactorizaciones con la intención de des/seleccionar un filtro de refactorizaciones que previamente ya se ha creado.	
Secuencia normal	Paso	Acción
	1	El usuario se situá en el filtro que desea des/habilitar.
	2	El usuario des/selecciona la opción de aplicar un filtro.
	3	El sistema aplica la opción tomada por el usuario referente al filtro actualizando la visualización del catálogo de refactorizaciones.
Postcondición	El usuario dispone del catálogo de refactorizaciones actualizado conforme a la selección de aplicar el filtro que acaba de realizar.	
Excepciones	No existe ninguna.	
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	No se considera necesarios.	

Tabla 23: Plantilla caso de uso seleccionar opción aplicar filtro

RF-5	Eliminar filtro de refactorizaciones	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario elimina, de forma permanente, un filtro de refactorizaciones que estaba disponible para ser aplicado sobre el catálogo de refactorizaciones.	
Precondición	El usuario se encuentra en la vista del catálogo de refactorizaciones con la intención de eliminar un filtro de refactorizaciones que previamente ya se ha creado.	
Secuencia normal	Paso	Acción
	1	El usuario se sitúa en el filtro que desea eliminar.
	2	El usuario pulsar el botón que elimina de forma permanente el filtro.
	3	El sistema elimina permanentemente el filtro y conforme a ello actualiza la visualización del catálogo de refactorizaciones.
Postcondición	El usuario dispone del catálogo de refactorizaciones actualizado conforme al borrado del filtro que acaba de realizar y además el filtro eliminado ya no aparece en la zona destinada a tal efecto.	
Excepciones	No existe ninguna.	
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	No se considera necesarios.	

Tabla 24: Plantilla caso de uso eliminar filtro de refactorizaciones

RF-6	Eliminar todos los filtros de refactorizaciones	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario elimina, de forma permanente, todos los filtros de refactorizaciones que estaban disponible para ser aplicados sobre el catálogo de refactorizaciones.	
Precondición	El usuario se encuentra en la vista del catálogo de refactorizaciones con la intención de eliminar todos los filtros de refactorizaciones que previamente ya se han creado.	
Secuencia normal	Paso	Acción
	1	El usuario pulsar el botón que elimina de forma permanente todos los filtros.
	2	El sistema elimina permanentemente todos los filtros y conforme a ello actualiza la visualización del catálogo de refactorizaciones. En caso de estar seleccionada la opción de mostrar las refactorizaciones filtradas y exista un grupo para visualizar y clasificar estas será eliminado.
Postcondición	El usuario dispone del catálogo de refactorizaciones actualizado conforme al borrado de todos los filtros creados y además ya no aparece ningún filtro en la zona destinada a tal efecto.	
Excepciones	No existe ninguna.	
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	Este caso de uso tiene relación de dependencia con el caso de uso <i>Eliminar filtro de refactorizaciones</i> .	

Tabla 25: Plantilla caso de uso eliminar todos los filtros de refactorizaciones

RF-7	Seleccionar opción ver refactorizaciones filtradas	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario des/selecciona la opción de ver refactorizaciones filtradas en el catálogo de refactorizaciones.	
Precondición	El usuario se encuentra en la vista del catálogo de refactorizaciones con la intención de des/seleccionar la opción de ver refactorizaciones filtradas en el catálogo de refactorizaciones.	
Secuencia normal	Paso	Acción
	1	El usuario des/selecciona la opción de ver refactorizaciones filtradas.
	2	El sistema aplica la opción tomada por el usuario actualizando la visualización del catálogo de refactorizaciones.
Postcondición	El usuario dispone del catálogo de refactorizaciones actualizado conforme a la selección de ver refactorizaciones filtradas que acaba de realizar.	
Excepciones	No existe ninguna.	
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	No se considera necesarios.	

Tabla 26: Plantilla caso de uso seleccionar opción ver refactorizaciones filtradas

RF-8	Visualizar detalle refactorización	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario visualiza el detalle de la refactorización que acaba de seleccionar en el catálogo de refactorizaciones.	
Precondición	El usuario se encuentra en la vista del catálogo de refactorizaciones con la intención de visualizar el detalle de las refactorizaciones del catálogo.	
Secuencia normal	Paso	Acción
	1	El usuario se sitúa en la refactorización de la cual quiere obtener su detalle.
	2	El usuario selecciona la refactorización.
	3	El sistema muestra el detalle de la refactorización que se encuentra seleccionada en forma de pestañas, en la que cada una de ellas muestra la información relevante a un aspecto de la misma.
Postcondición	El usuario dispone del detalle de la refactorización seleccionada con la finalidad de que pueda navegar por la información que se le muestra y con ello obtener más conocimiento de la misma.	
Excepciones	No existe ninguna.	
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	No se considera necesarios.	

Tabla 27: Plantilla caso de uso visualizar detalle refactorización

RF-9	Añadir clasificación	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario añade una nueva clasificación al plugin, quedando disponible para su uso.	
Precondición	El usuario se encuentra en el editor de clasificaciones con la intención de crear una nueva clasificación.	
Secuencia normal	Paso	Acción
	1	El usuario pulsa el botón <i>Add...</i> en el editor.
	2	El usuario introduce el nombre de la nueva clasificación.
	3	El usuario pulsa el botón <i>OK</i> .
Postcondición	La nueva clasificación ha sido creada y esta disponible para su utilización.	
Excepciones	Paso	Acción
	2	Si el nombre de la nueva clasificación ya existe el sistema no deja crear la clasificación y se informa de ello.
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	No se considera necesarios.	

Tabla 28: Plantilla caso de uso añadir clasificación

RF-10	Editar clasificación	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario modifica una clasificación propia del usuario.	
Precondición	El usuario se encuentra en el editor de clasificaciones con la intención de modificar una clasificación propia del usuario ya existente.	
Secuencia normal	Paso	Acción
	1	El usuario selecciona la clasificación que desea modificar.
	2	El usuario renombra la clasificación seleccionando el botón <i>Rename...</i> para ello introduce el nuevo nombre y pulsa <i>OK</i> .
	3	El usuario modifica la descripción mediante la selección del botón <i>Modify...</i> para ello introduce la nueva descripción y pulsa <i>OK</i> .
	4	El usuario modifica la propiedad que indica si la clasificación es uni o multicategoría.
Postcondición	La clasificación ha sido modificada y ya se encuentra disponible con los nuevos valores para su utilización.	
Excepciones	Paso	Acción
	1	Si la clasificación seleccionada no es propia del usuario y por tanto editable el sistema no permite su edición.
	2	Si el nuevo nombre ya existe, el sistema no deja modificar la clasificación y se informa de ello.
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	Existen dos tipos de clasificaciones las que suministra el plugin y las de creación propia del usuario, las primeras no son editables mientras que las segundas si lo son. Por lo tanto, la edición de clasificaciones solo aplica a las propias del usuario.	

Tabla 29: Plantilla caso de uso editar clasificación

RF-11	Eliminar clasificación	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario elimina una clasificación propia del usuario, por lo que ya no está disponible en el plugin para su utilización.	
Precondición	El usuario se encuentra en el editor de clasificaciones con la intención de eliminar una clasificación propia del usuario ya existente.	
Secuencia normal	Paso	Acción
	1	El usuario selecciona la clasificación que desea modificar.
	2	El usuario pulsa el botón <i>Delete...</i> para su borrado.
	3	El sistema le indica que confirme la decisión de borrado pulsando <i>Proceed</i> y sino <i>Cancel</i>
	4	El usuario pulsa el botón <i>Proceed</i> , con ello confirma el borrado.
Postcondición	La clasificación ha sido eliminada y no se encuentra disponible para su utilización.	
Excepciones	Paso	Acción
	1	Si la clasificación seleccionada no es propia del usuario y por tanto editable el sistema no permite su borrado.
	4	Si el usuario pulsa <i>Cancel</i> porque finalmente no quiere realizar el borrado de la clasificación.
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	Existen dos tipos de clasificaciones las que suministra el plugin y las de creación propia del usuario, las primeras no son editables mientras que las segundas si lo son. Por lo tanto, el borrado de clasificaciones solo aplica a las propias del usuario.	

Tabla 30: Plantilla caso de uso eliminar clasificación

RF-12	Añadir categorías a una clasificación	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario añade una nueva categoría a una clasificación propia del usuario, quedando disponible para su uso.	
Precondición	El usuario se encuentra en el editor de clasificaciones con la intención de añadir una nueva categoría a una clasificación propia del usuario ya existente.	
Secuencia normal	Paso	Acción
	1	El usuario selecciona la clasificación que desea modificar.
	2	El usuario en la sección de categorías de la clasificación pulsa el botón <i>Add...</i>
	3	El usuario introduce el nombre para la nueva categoría y pulsa <i>OK</i> .
Postcondición	La nueva categoría para la clasificación ha sido creada y esta disponible para su utilización.	
Excepciones	Paso	Acción
	1	Si la clasificación seleccionada no es propia del usuario y por tanto editable el sistema no permite añadir una categoría.
	3	Si el nombre dado para la nueva categoría ya existe, el sistema no deja crear la categoría y se informa de ello.
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	Las categorías son propias de una clasificación es por ello por lo que la adición de las mismas se considera en si una modificación de la propia clasificación. Esto viene a explicar que este caso de uso sea un punto de extensión del caso de uso editar clasificación, y en este únicamente se detalle lo propio.	

Tabla 31: Plantilla caso de uso añadir categorías a una clasificación

RF-13	Renombrar categorías de una clasificación	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario renombra una categoría de una clasificación propia del usuario.	
Precondición	El usuario se encuentra en el editor de clasificaciones con la intención de renombrar una categoría de una clasificación propia del usuario ya existente.	
Secuencia normal	Paso	Acción
	1	El usuario selecciona la clasificación que desea modificar.
	2	El usuario en la sección de categorías de la clasificación selecciona la categoría que desea renombrar.
	3	El usuario pulsa el botón <i> Rename...</i>
	4	El usuario introduce el nuevo nombre para la categorías y pulsa <i>OK</i> .
Postcondición	La categoría de la clasificación ha sido modificada y ya se encuentra disponible con el nuevo nombre para su utilización.	
Excepciones	Paso	Acción
	1	Si la clasificación seleccionada no es propia del usuario y por tanto editable el sistema no permite renombrar una categoría.
	4	Si el nombre de la nueva categoría para la clasificación ya existe, el sistema no realizar el renombrado y se informa de ello.
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	Las categorías son propias de una clasificación es por ello por lo que el renombrado de las mismas se considera en si una modificación de la propia clasificación. Esto viene a explicar que este caso de uso sea un punto de extensión del caso de uso editar clasificación, y en este únicamente se detalle lo propio.	

Tabla 32: Plantilla caso de uso renombrar categorías de una clasificación

RF-14	Eliminar categorías de una clasificación	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario elimina una categoría de una clasificación propia del usuario.	
Precondición	El usuario se encuentra en el editor de clasificaciones con la intención de eliminar una categoría de una clasificación propia del usuario ya existente.	
Secuencia normal	Paso	Acción
	1	El usuario selecciona la clasificación que desea modificar.
	2	El usuario en la sección de categorías de la clasificación selecciona la categoría que desea eliminar.
	3	El usuario pulsa el botón <i>Delete...</i>
	4	El sistema le indica que confirme la decisión de borrado pulsando <i>Proceed</i> y sino <i>Cancel</i>
	5	El usuario pulsa el botón <i>Proceed</i> , con ello confirma el borrado.
Postcondición	La categoría de la clasificación ha sido eliminada y por lo tanto ya no se encuentra disponible para su utilización.	
Excepciones	Paso	Acción
	1	Si la clasificación seleccionada no es propia del usuario y por tanto editable el sistema no permite renombrar una categoría.
	5	Si el usuario pulsa <i>Cancel</i> porque finalmente no quiere realizar el borrado de la categoría.
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	Las categorías son propias de una clasificación es por ello por lo que el borrado de las mismas se considera en si una modificación de la propia clasificación. Esto viene a explicar que este caso de uso sea un punto de extensión del caso de uso editar clasificación, y en este únicamente se detalle lo propio.	

Tabla 33: Plantilla caso de uso eliminar categorías de una clasificación

RF-15	Mostrar resumen elemento seleccionado	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario se encuentra creando o editando una refactorización y visualiza el resumen asociado a un elemento seleccionado con el objetivo de obtener mayor información del mismo y así poder añadirlo o eliminarlo de la definición de la refactorización.	
Precondición	El usuario se encuentra dentro del asistente de creación o edición de una refactorización en alguna de las páginas destinadas a la gestión de elementos en la definición de la refactorización.	
Secuencia normal	Paso	Acción
	1	El usuario selecciona un elemento de la lista de disponibles para visualizar su resumen.
	2	El sistema muestra el resumen asociado al elemento seleccionado, mostrando su descripción que ha sido obtenida del javadoc y la relación de refactorizaciones que tienen al elemento incluido en su definición.
	3	El usuario se puede situar encima de las refactorizaciones si quiere obtener su información básica.
Postcondición	El usuario ha obtenido la información que solicitaba y en base a ella puede decidir si desea añadir o no el elemento a la definición de la refactorización.	
Excepciones	Paso	Acción
	2	Si no se dispone del javadoc del elemento el sistema no puede mostrar el resumen asociado.
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	Cuando se hace referencia a un elemento se trata de cualquier elemento del que pueda estar compuesta una refactorización, es decir, una entrada, una acción o un predicado.	

Tabla 34: Plantilla caso de uso mostrar resumen elemento seleccionado

RF-16	Realizar búsqueda de elementos	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario se encuentra creando o editando una refactorización y realiza una búsqueda sobre los elementos disponibles con el objetivo de obtener aquellos elementos que cumplen con los criterios establecidos.	
Precondición	El usuario se encuentra dentro del asistente de creación o edición de una refactorización en alguna de las páginas destinadas a la gestión de elementos en la definición de la refactorización.	
Secuencia normal	Paso	Acción
	1	El usuario introduce el criterio de búsqueda.
	2	El usuario pulsa el botón que inicia la búsqueda.
	3	El sistema muestra, ordenados según relevancia, los elementos que cumplen con el criterio de búsqueda.
Postcondición	El usuario dispone la relación de elementos que cumplen con el criterio de búsqueda ordenados según relevancia.	
Excepciones	No existe ninguna.	
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	Si el usuario después de realizar una búsqueda desea eliminarla bastará con que simule una búsqueda vacía.	

Tabla 35: Plantilla caso de uso realizar búsqueda de elementos

RF-17	Instalar plugin	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario realiza el proceso de instalación del plugin de forma automática, quedando disponible para su uso en el entorno de desarrollo.	
Precondición	El usuario una vez arrancado el entorno de desarrollo Eclipse pretende realizar la instalación del plugin. Además se ha asegurado que tiene acceso al repositorio donde este se encuentra disponible este.	
Secuencia normal	Paso	Acción
	1	El usuario selecciona la opción <i>Install New Software...</i> del menú <i>Help</i> de la barra de herramientas de Eclipse.
	2	El usuario pulsa <i>Add</i> e indica el repositorio software donde se encuentra disponible el plugin para realizar su descarga, así como un nombre identificativo que lo represente.
	3	El sistema comprueba la validez del repositorio y si este es correcto muestra la versión que hay disponible del plugin.
	4	El usuario selecciona el plugin a descargar y pulsa <i>Next</i> .
	5	El sistema muestra la licencia del plugin para que el usuario si esta conforme a ella la acepte.
	6	El usuario acepta la licencia del plugin y pulsa <i>Finish</i> para que de comienzo la instalación.
	7	El sistema informa del estado de progreso de la instalación e indica que el software se encuentra firmado, preguntando si se desea confiar en el certificado y por tanto seguir con el proceso de instalación.
	8	El usuario acepta el certificado y continua el proceso de instalación del plugin.
	9	El sistema termina con el proceso de instalación preguntando si se desea reiniciar el entorno de desarrollo para poder comenzar a utilizar las funcionalidades incorporadas por el plugin.
	10	El usuario pulsa la opción <i>Restart Now</i> para reiniciar el entorno de desarrollo, dando como concluido así el proceso de instalación.
Postcondición	El entorno de desarrollo Eclipse cuenta con las funcionalidades incorporadas por el plugin de refactorizaciones disponibles para su uso.	
Excepciones	Paso	Acción
	3	En caso de no ser válido el repositorio, el sistema le rechaza y nos informa de ello.
	6	Si el usuario no esta conforme a la licencia del plugin pulsará <i>Cancel</i> y no comenzará el proceso de instalación.

RF-17	Instalar plugin	
	8	Si el usuario no confía en el certificado del plugin rechazándolo se abortará la instalación del mismo.
	10	Si el usuario rechaza la posibilidad de reiniciar el entorno de desarrollo puede provocar que el sistema quede inestable dando lugar a errores.
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	No se considera necesarios.	

Tabla 36: Plantilla caso de uso instalar plugin

RF-18	Actualizar plugin	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz	
Fuentes	Raúl Marticorena Sánchez	
Descripción	El usuario realiza el proceso de actualización del plugin de forma automática, quedando disponible las nuevas funcionalidades que este incorpora para su uso en el entorno de desarrollo.	
Precondición	El usuario una vez arrancado el entorno de desarrollo Eclipse, el cual ya dispone de una versión del plugin instalada, pretende realizar la actualización del mismo. Además se ha asegurado que tiene acceso al repositorio donde este se encuentra disponible este.	
Secuencia normal	Paso	Acción
	1	El usuario selecciona la opción <i>Check for Updates...</i> del menú <i>Help</i> de la barra de herramientas de Eclipse.
	2	El sistema comprueba la validez del repositorio y si este es correcto y hay disponible una nueva versión del plugin la muestra.
	3	El usuario selecciona el plugin a descargar y pulsa <i>Next</i> .
	4	El sistema muestra la licencia del plugin para que el usuario si esta conforme a ella la acepte.
	5	El usuario acepta la licencia del plugin y pulsa <i>Finish</i> para que de comienzo la actualización.
	6	El sistema informa del estado de progreso de la actualización e indica que el software se encuentra firmado, preguntando si se desea confiar en el certificado y por tanto seguir con el proceso de actualización.
	7	El usuario acepta el certificado y continua el proceso de actualización del plugin.
	8	El sistema termina con el proceso de actualización preguntando si se desea reiniciar el entorno de desarrollo para poder comenzar a utilizar las nuevas funcionalidades incorporadas por el plugin.
	9	El usuario pulsa la opción <i>Restart Now</i> para reiniciar el entorno de desarrollo, dando como concluido así el proceso de actualización.
Postcondición	El entorno de desarrollo Eclipse cuenta con las nuevas funcionalidades, que han sido incorporadas en la nueva versión del plugin de refactorizaciones, disponibles para su uso.	
Excepciones	Paso	Acción
	2	En caso de no ser válido el repositorio, el sistema le rechaza y nos informa de ello. En caso de no existir actualización hará lo propio.
	5	Si el usuario no esta conforme a la licencia del plugin pulsará <i>Cancel</i> y no comenzará el proceso de instalación.
	7	Si el usuario no confía en el certificado del plugin rechazándolo se abortará la instalación del mismo.

RF-18	Actualizar plugin	
	9	Si el usuario rechaza la posibilidad de reiniciar el entorno de desarrollo puede provocar que el sistema quede inestable dando lugar a errores.
Frecuencia esperada	Alta / Media / Baja	
Importancia	Alta / Media / Baja	
Urgencia	Alta / Media / Baja	
Comentarios	No se considera necesarios.	

Tabla 37: Plantilla caso de uso actualizar plugin

RF-19	Desinstalar plugin																	
Autores	Míryam Gómez San Martín Iñigo Mediavilla Saiz																	
Fuentes	Raúl Marticorena Sánchez																	
Descripción	El usuario realiza el proceso de desinstalación del plugin de forma automática, eliminando todas las funcionalidades que hubiese incorporado el plugin al entorno de desarrollo.																	
Precondición	El usuario una vez arrancado el entorno de desarrollo Eclipse, el cual ya dispone de una versión del plugin instalada, pretende realizar la desinstalación del mismo.																	
Secuencia normal	<table border="1"> <thead> <tr> <th>Paso</th> <th>Acción</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>El usuario selecciona la opción <i>About Eclipse SDK</i> del menú <i>Help</i> de la barra de herramientas de Eclipse.</td> </tr> <tr> <td>2</td> <td>El usuario pulsa el botón <i>Installation Details</i> para obtener la relación de plugins que se encuentran instalados.</td> </tr> <tr> <td>3</td> <td>El usuario selecciona el plugin de refactorizaciones entre la lista de plugins instalados que se encuentran en la pestaña <i>Installed Software</i> y pulsa el botón <i>Uninstall</i>.</td> </tr> <tr> <td>4</td> <td>El sistema solicita la confirmación del software a desinstalar.</td> </tr> <tr> <td>5</td> <td>El usuario confirma que desea desinstalar el plugin, pulsa el botón <i>Finish</i>.</td> </tr> <tr> <td>6</td> <td>El sistema termina con el proceso de desinstalación preguntando si se desea reiniciar el entorno de desarrollo para que los cambios producidos por la desinstalación hagan efecto.</td> </tr> <tr> <td>7</td> <td>El usuario pulsa la opción <i>Restart Now</i> para reiniciar el entorno de desarrollo, dando como concluido así el proceso de desinstalación.</td> </tr> </tbody> </table>		Paso	Acción	1	El usuario selecciona la opción <i>About Eclipse SDK</i> del menú <i>Help</i> de la barra de herramientas de Eclipse.	2	El usuario pulsa el botón <i>Installation Details</i> para obtener la relación de plugins que se encuentran instalados.	3	El usuario selecciona el plugin de refactorizaciones entre la lista de plugins instalados que se encuentran en la pestaña <i>Installed Software</i> y pulsa el botón <i>Uninstall</i> .	4	El sistema solicita la confirmación del software a desinstalar.	5	El usuario confirma que desea desinstalar el plugin, pulsa el botón <i>Finish</i> .	6	El sistema termina con el proceso de desinstalación preguntando si se desea reiniciar el entorno de desarrollo para que los cambios producidos por la desinstalación hagan efecto.	7	El usuario pulsa la opción <i>Restart Now</i> para reiniciar el entorno de desarrollo, dando como concluido así el proceso de desinstalación.
Paso	Acción																	
1	El usuario selecciona la opción <i>About Eclipse SDK</i> del menú <i>Help</i> de la barra de herramientas de Eclipse.																	
2	El usuario pulsa el botón <i>Installation Details</i> para obtener la relación de plugins que se encuentran instalados.																	
3	El usuario selecciona el plugin de refactorizaciones entre la lista de plugins instalados que se encuentran en la pestaña <i>Installed Software</i> y pulsa el botón <i>Uninstall</i> .																	
4	El sistema solicita la confirmación del software a desinstalar.																	
5	El usuario confirma que desea desinstalar el plugin, pulsa el botón <i>Finish</i> .																	
6	El sistema termina con el proceso de desinstalación preguntando si se desea reiniciar el entorno de desarrollo para que los cambios producidos por la desinstalación hagan efecto.																	
7	El usuario pulsa la opción <i>Restart Now</i> para reiniciar el entorno de desarrollo, dando como concluido así el proceso de desinstalación.																	
Postcondición	El entorno de desarrollo Eclipse ya no cuenta con las funcionalidades que habían sido incorporadas por el plugin de refactorizaciones.																	
Excepciones	<table border="1"> <thead> <tr> <th>Paso</th> <th>Acción</th> </tr> </thead> <tbody> <tr> <td>7</td> <td>Si el usuario rechaza la posibilidad de reiniciar el entorno de desarrollo puede provocar que el sistema quede inestable dando lugar a errores.</td> </tr> </tbody> </table>		Paso	Acción	7	Si el usuario rechaza la posibilidad de reiniciar el entorno de desarrollo puede provocar que el sistema quede inestable dando lugar a errores.												
Paso	Acción																	
7	Si el usuario rechaza la posibilidad de reiniciar el entorno de desarrollo puede provocar que el sistema quede inestable dando lugar a errores.																	
Frecuencia esperada	Alta / Media / Baja																	
Importancia	Alta / Media / Baja																	
Urgencia	Alta / Media / Baja																	
Comentarios	No se considera necesarios.																	

Tabla 38: Plantilla caso de uso desinstalar plugin

6. INTERFAZ DE USUARIO

En este apartado se pretende mostrar el prototipado que se ha ido realizando de las diferentes interfaces que se han creado nuevas y de las que se han visto modificadas en la nueva versión del plugin y que compondrán la interfaz gráfica de usuario final. Se realizaron varios prototipos que se han ido depurando hasta que se ha obtenido la versión final que se ha creído más conveniente para cada uno de ellos. A lo largo del desarrollo estas versiones definitivas de los prototipos sufrirán una serie de modificaciones, bien impuestas por las limitaciones de las propias librerías gráficas que han sido utilizadas o bien con el fin de aprovechar las posibilidades que estas ofrecían.

6.1. Vista del catálogo de refactorizaciones

Para cumplir con el segundo requisito funcional del proyecto se pensó inicialmente en una vista que permitiera mostrar en su parte derecha todas las refactorizaciones disponibles agrupadas por categorías. Puesto que podían existir distintos tipos de clasificaciones, debía existir un *combo* que permitiera seleccionar la clasificación por la que las refactorizaciones se iban a agrupar. Para el filtrado se incorporaría un cuadro de texto con una ayuda contextual sobre la sintaxis de las búsquedas. En la parte derecha se mostraría toda la información de la refactorización.

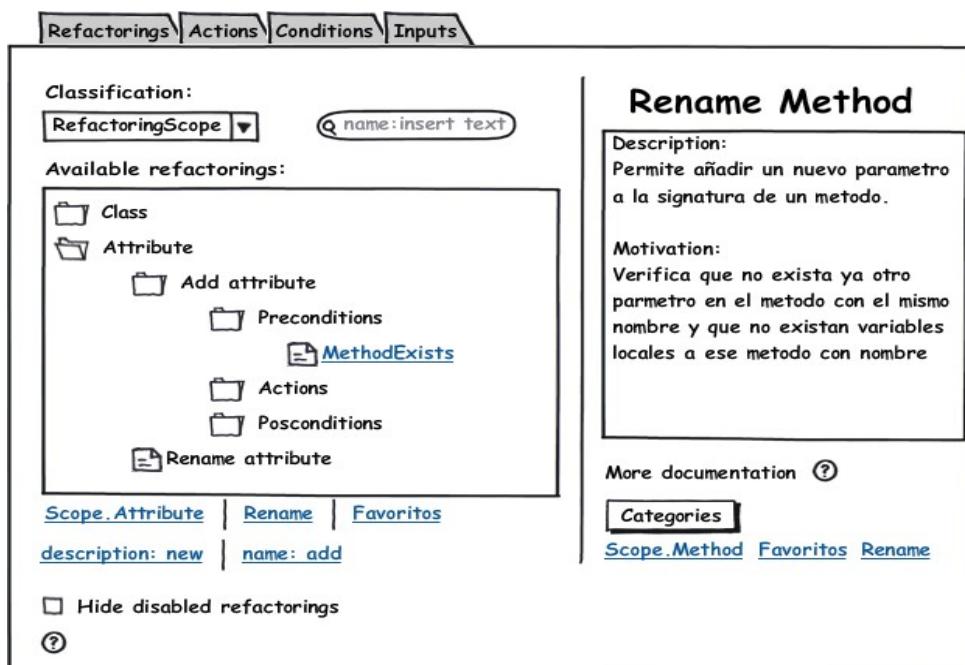


Ilustración 19: Primer prototípico de la vista del catálogo de refactorizaciones

Tras una reunión con el tutor en la que se le expuso el prototipo creado se realizaron algunos ajustes fruto de ciertas ideas que se aclararon y otras nuevas que se añadieron. Se clarificó que las clasificaciones iban a tener categorías pero estas no tendrían subcategorías y que no habría categorías de un sólo nivel. En lugar de este tipo de categorías de un sólo nivel existiría el concepto de palabras clave que permitirían identificar las refactorizaciones de forma similar a como se hace con las publicaciones.

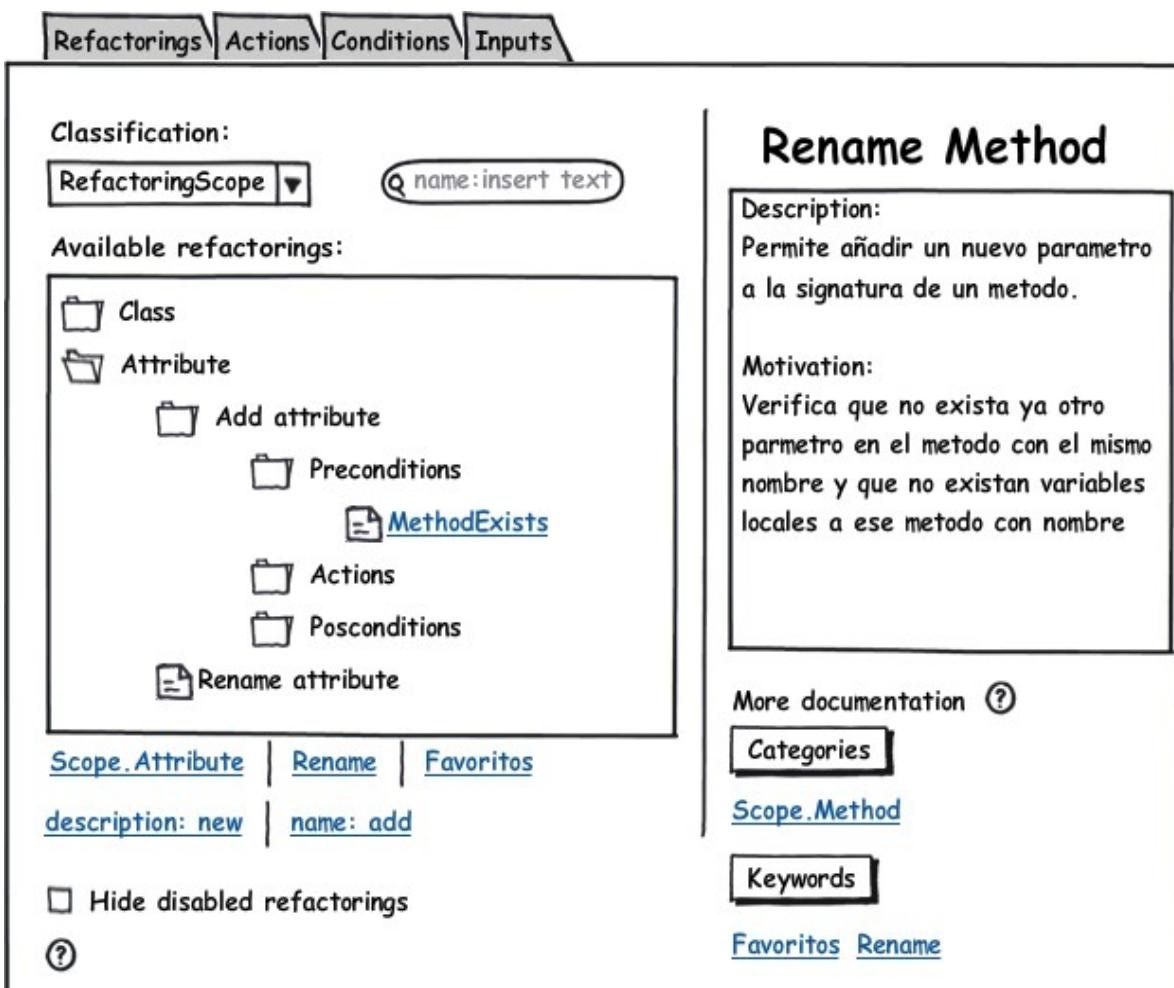


Ilustración 20: Segundo prototipo de la vista del catálogo de refactorizaciones

En la versión definitiva incorporada en el plugin ya se aclaró que sólo se crearía una vista de este tipo para las refactorizaciones y que las entradas, las acciones y los predicados se tratarían de una forma distinta. Además se agregaron pestañas en la parte derecha para incorporar más información de la refactorización como sus entradas, sus mecanismos y sus imágenes y ejemplos si los tenía. Se simplificó el panel de los filtros que aparecía abajo del todo en la parte izquierda para ajustarlo a las posibilidades de *SWT* y se agregó el botón de limpiar todos los filtros.

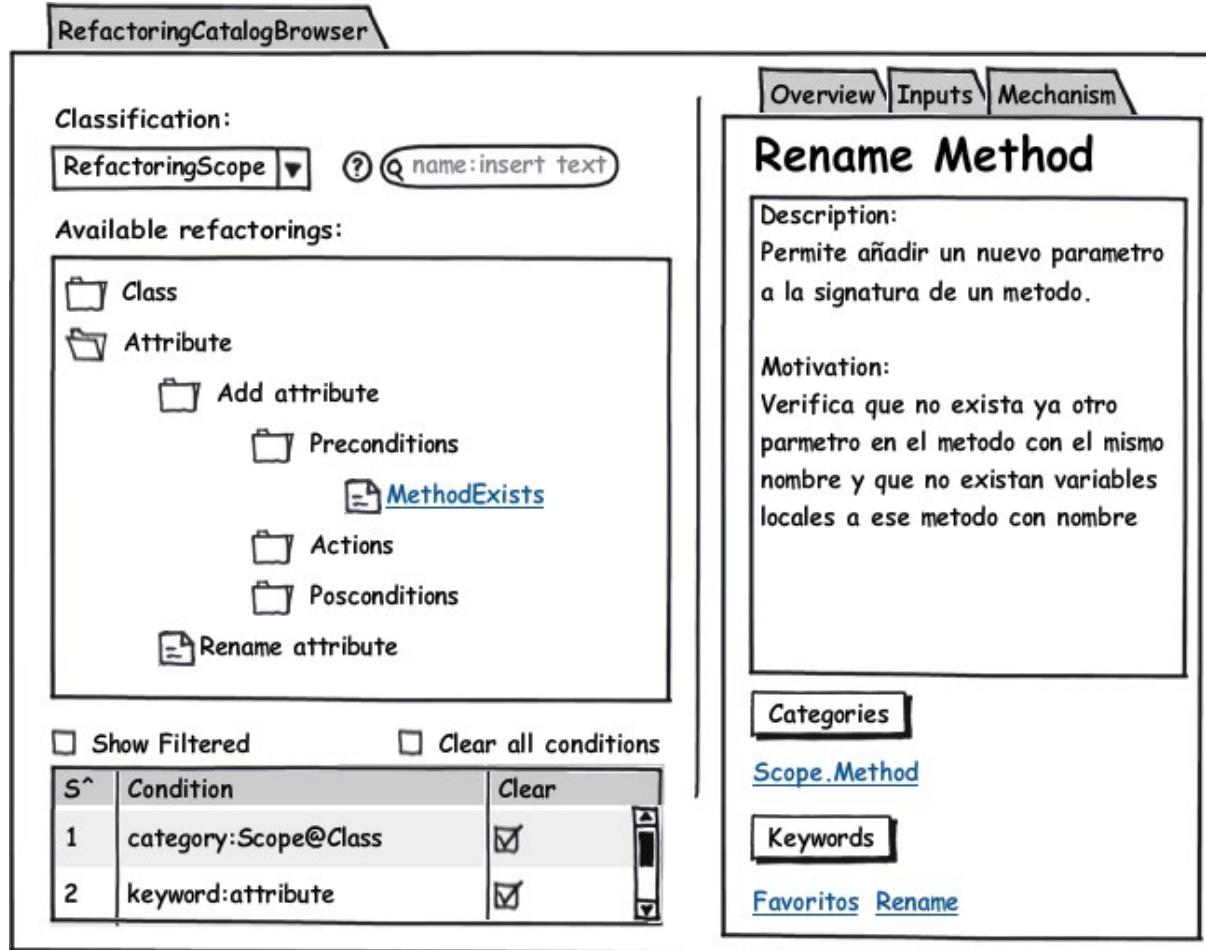


Ilustración 21: Prototipo definitivo de la vista del catálogo de refactorizaciones

6.2. Primera página del asistente de refactorizaciones

Si se quería organizar las refactorizaciones asignándolas una categoría por cada una de las clasificaciones existentes y una serie de palabras clave descriptivas había que ofrecer al usuario el medio de establecer las categorías y las palabras clave de una refactorización. El lugar más adecuado para esta asignación se consideró que sería la primera página del asistente de creación de refactorizaciones. A continuación se muestra el primer y definitivo prototipo con las modificaciones necesarias a dicha página:

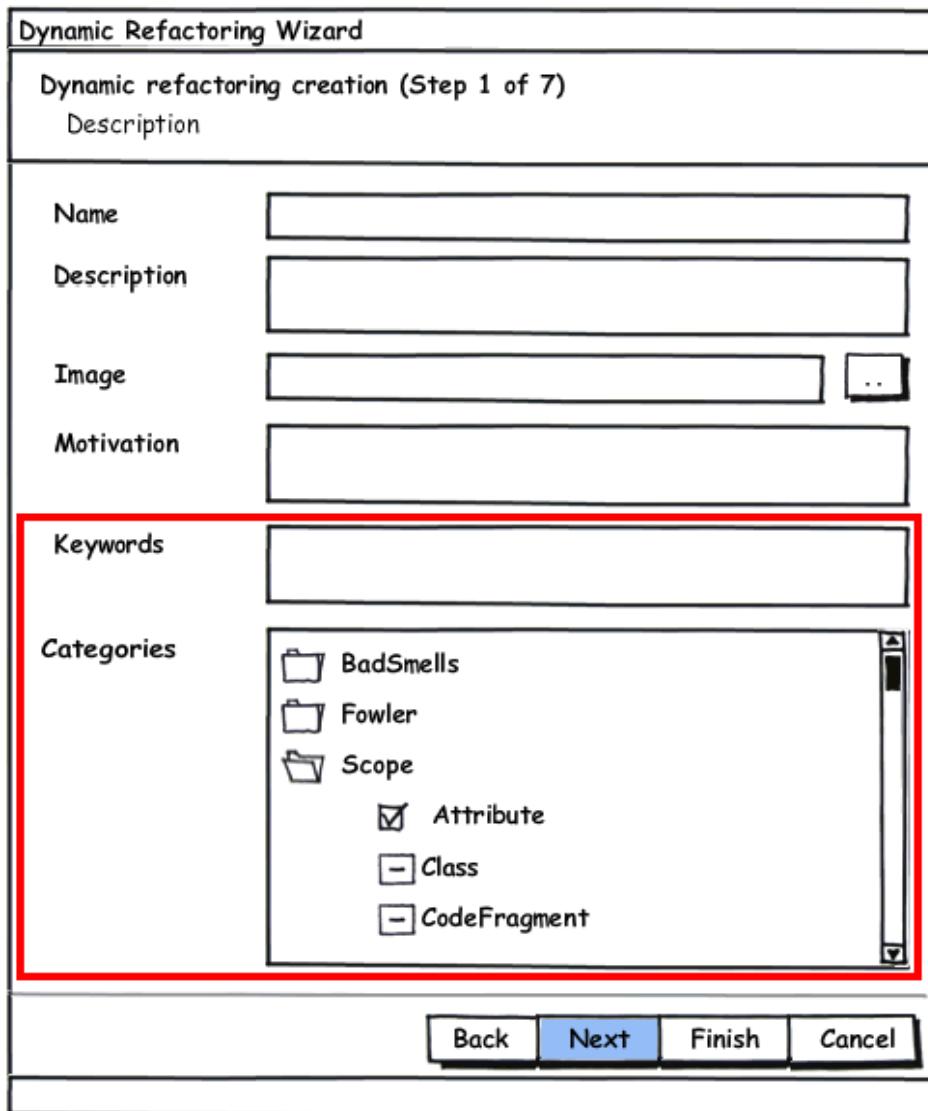


Ilustración 22: Prototipo con las modificaciones de la primera página del asistente

6.3. Editor de clasificaciones

Para permitir al usuario crear sus propias clasificaciones y que pudiese modificar tanto sus atributos como las categorías que las componen se contempló en un principio la posibilidad de que hiciese estas tareas modificando el fichero XML de clasificaciones de forma manual. Sin embargo, finalmente se vio que esta no era una idea muy acertada y que sería más apropiado ofrecer un editor de clasificaciones.

El primer prototipo fue el de un modelo muy simple basado en dos secciones una lista de clasificaciones disponibles a la izquierda y otra para los atributos y categorías de la clasificación seleccionada a la derecha.

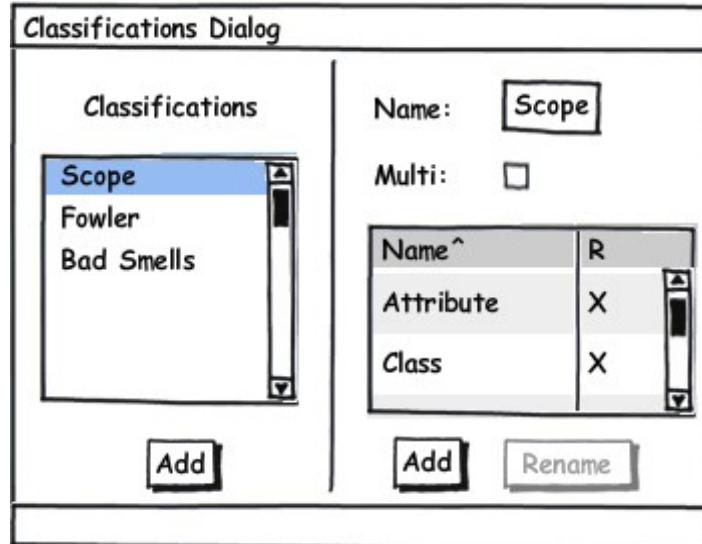


Ilustración 23: Primer prototipo del editor de clasificaciones

En la segunda versión del prototipo ya se consideró que el editor podía hacer uso de las posibilidades que ofrecía de la librería de Eclipse, Eclipse Forms. Teniendo esto en cuenta se decidió agrupar los apartados por secciones. A la derecha se situaría la sección de clasificaciones con la lista de las mismas y los botones para modificarlas. A la izquierda dos secciones, en la de arriba los atributos de la clasificación seleccionada y en la de debajo la lista de categorías de la clasificación con sus botones de modificación.

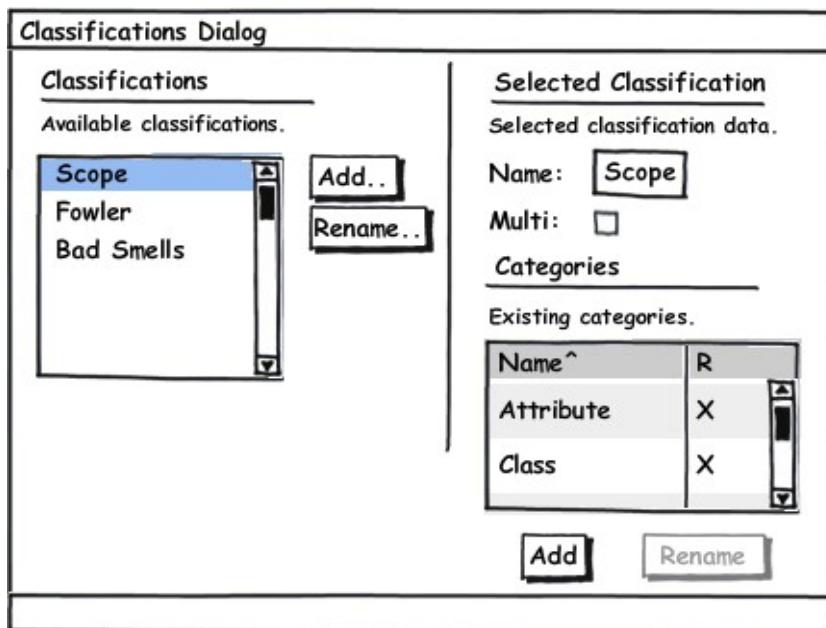


Ilustración 24: Segundo prototipo del editor de clasificaciones

Finalmente en el último prototipo se modificaron algunos pequeños detalles que eran necesarios pero que se habían ignorado en los prototipos previos como los botones de borrado o el campo de descripción de una clasificación. Además lo que en un principio se habían considerado un diálogo de Eclipse se sustituyó por un editor, más acorde con la función de la interfaz.

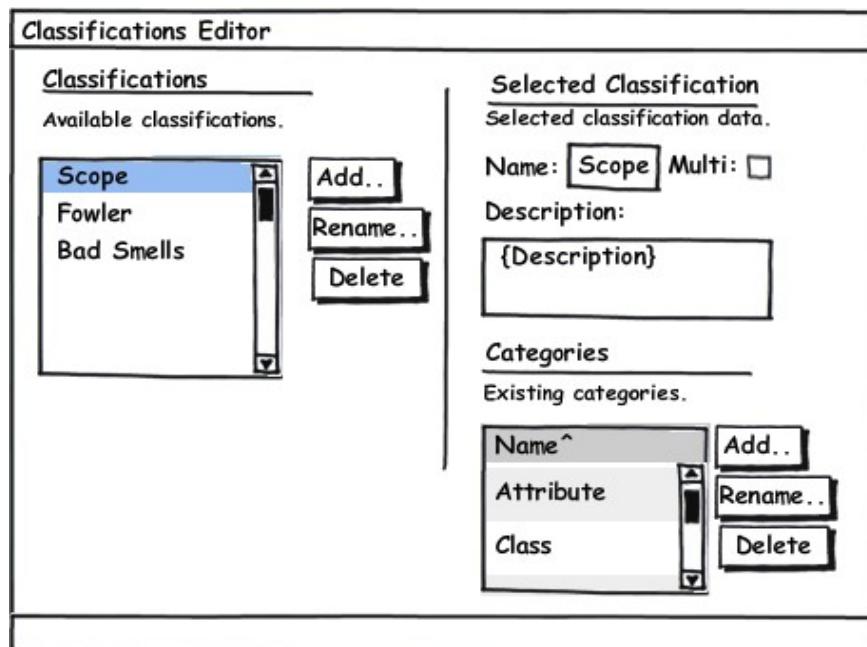


Ilustración 25: Prototipo definitivo del editor de clasificaciones

6.4. Página del asistente de entradas, predicados y acciones

Con la intención de cumplir los objetivos del tercer requisito funcional relativo a la simplificación del proceso de creación de refactorizaciones era necesario incluir una serie de modificaciones en su asistente para los pasos de asignación de entradas, precondiciones, acciones y postcondiciones.

Para esta parte de la interfaz se crearon dos prototipos diferentes cada uno con sus ventajas y desventajas con la intención de presentárselos al tutor y decidir con él cual era el más conveniente para el usuario.

El primero era un prototipo que cambiaba en fuerte medida el diseño ya existente de las ventanas. Eliminaba la ventana del javadoc de la entrada seleccionada por considerarlo información redundante. Bajaba las entradas seleccionadas por el usuario y los parámetros de las entradas al fondo de la ventana y arriba dejaba un campo de texto con un botón de

búsqueda. Debajo del botón las entradas existentes que se ajustaran a la búsqueda que se hubiese realizado, las cuales aparecerían ordenadas según prioridad y dentro de un *widget* que se abriera al ser pulsado para mostrar la descripción y otra información importante de la entrada.

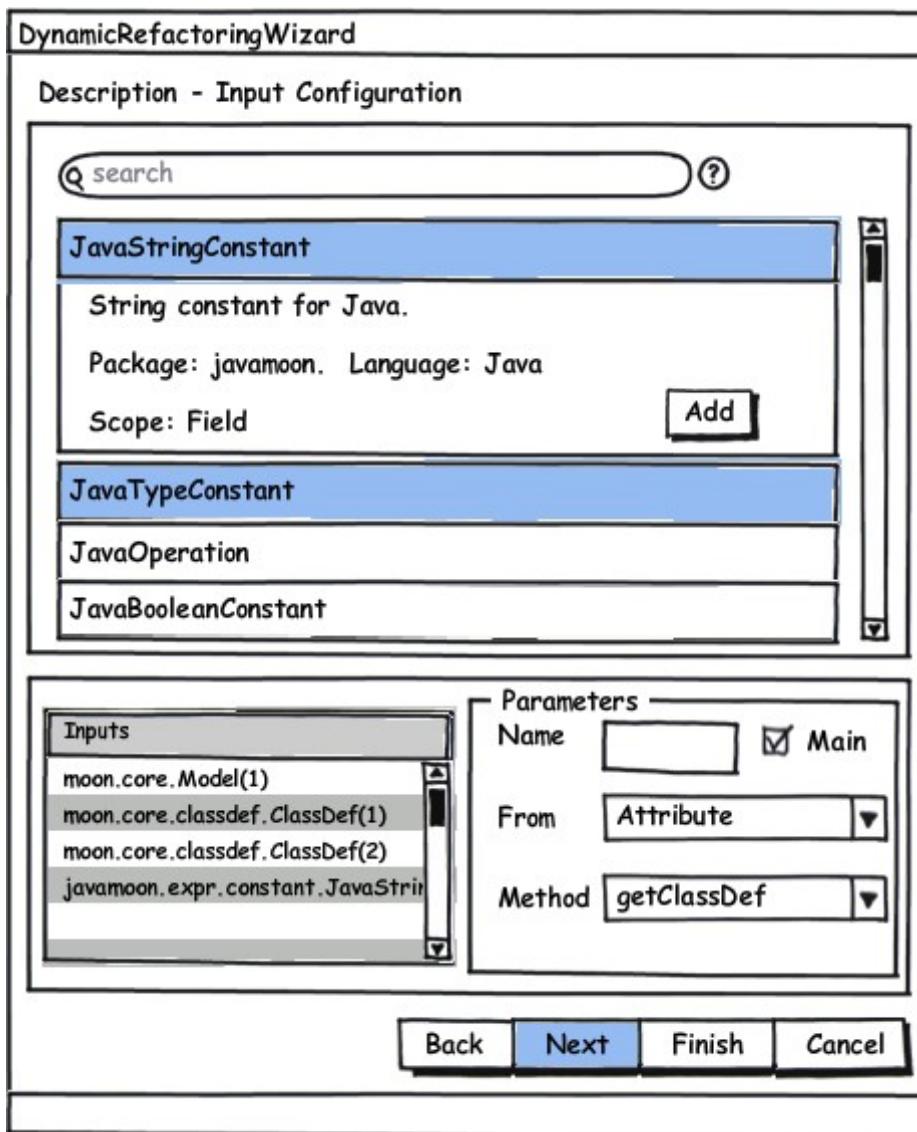


Ilustración 26: Prototipo de página del asistente basado en entradas extensibles

El segundo prototipo incorporaba muchas de las modificaciones del prototipo anterior como la distribución de los elementos, pero optaba por un modelo más sencillo que el de las entradas que se despliegan al seleccionarlas. En su lugar optaba por una tabla y un panel que mostraba la descripción del elemento seleccionado dentro de un pequeño marco.

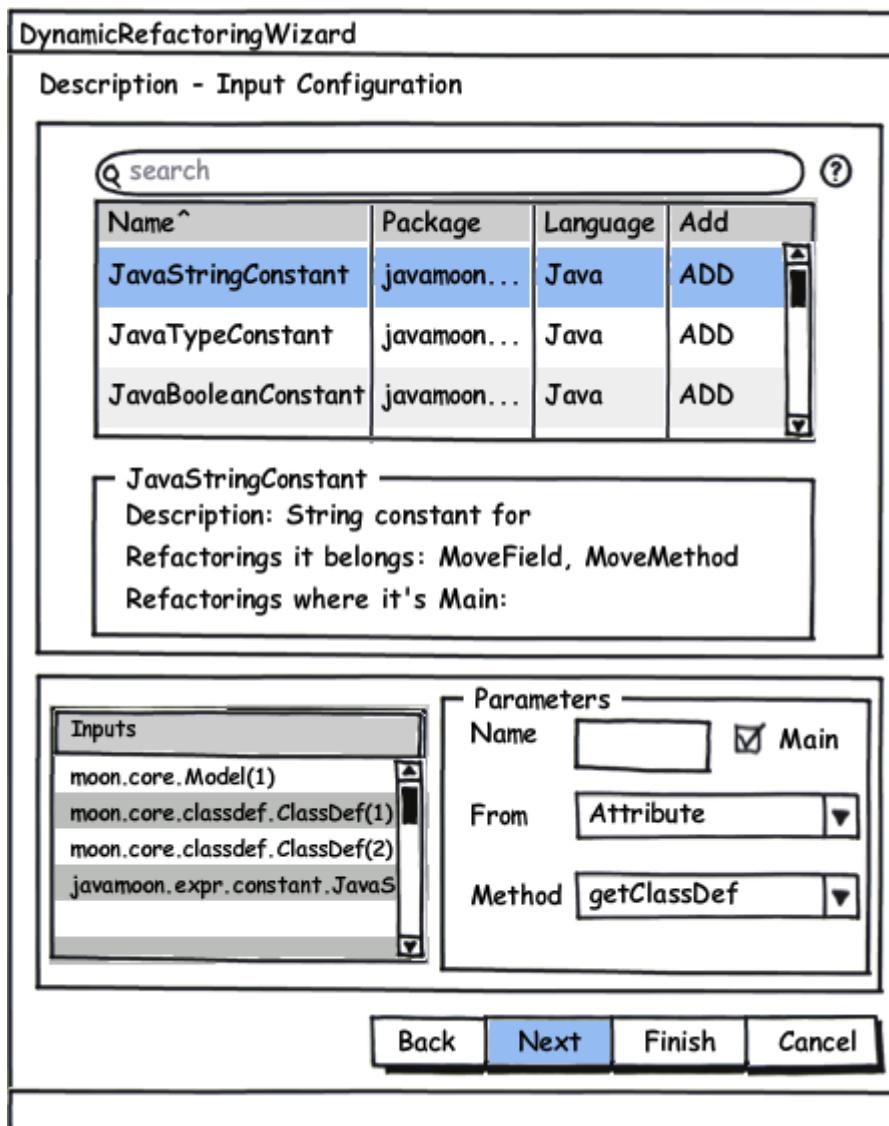


Ilustración 27: Prototipo de página del asistente basado en una tabla

En la opción definitiva se optó por mantener la distribución de los elementos bajo la lógica de que así se mantendrían los elementos más importantes en la zona de arriba y el flujo de información de izquierda a derecha (a la izquierda se ve el elemento y se pasa a la derecha al seleccionarlo como elemento de la refactorización). En la zona de abajo se deja espacio para los elementos de información adicional como lo es el cuadro de resumen del elemento seleccionado y debajo de él, el javadoc que se puede ocultar si no se desea visualizar.

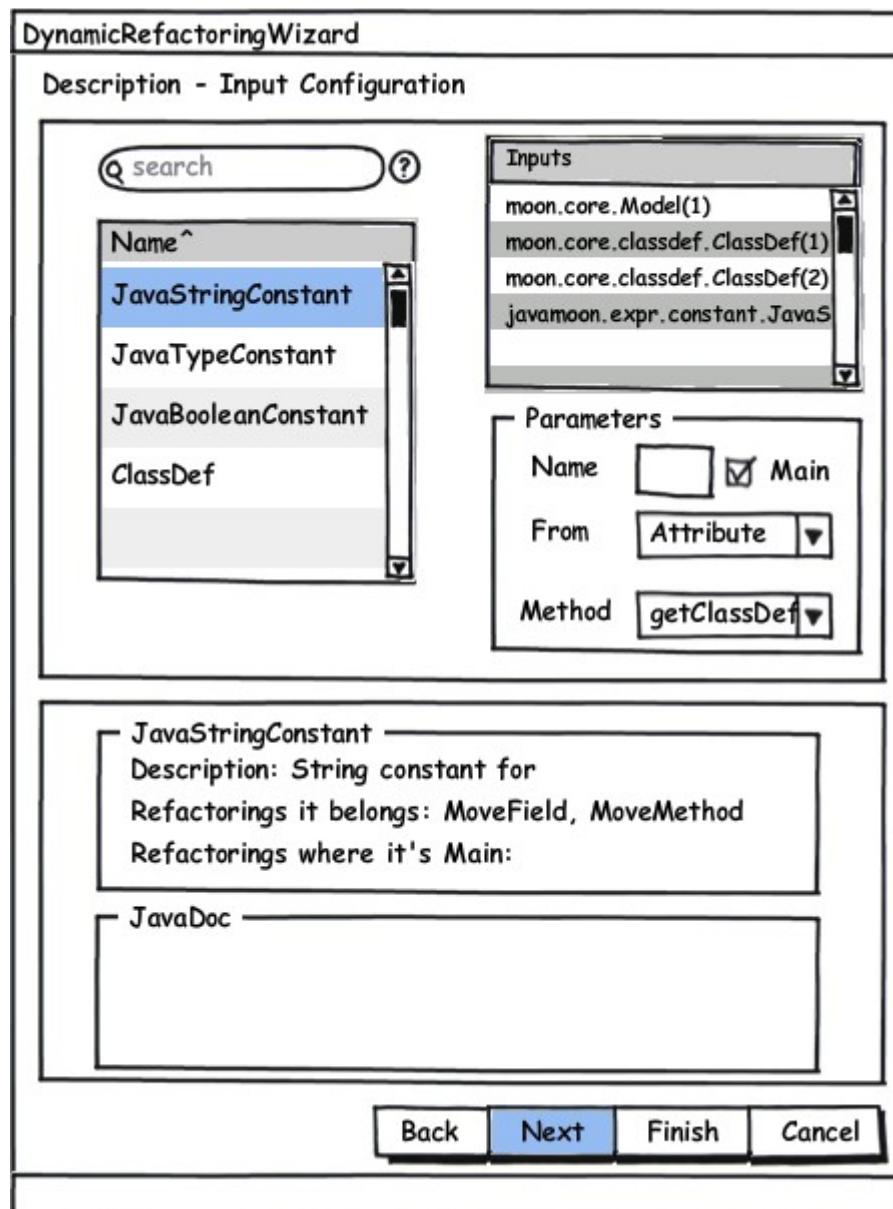


Ilustración 28: Prototipo de página del asistente definitivo

Universidad de Burgos
ESCUELA POLITÉCNICA SUPERIOR
INGENIERÍA INFORMÁTICA



Anexo 3. Especificación del Diseño

Clasificación de Refactorizaciones
Dynamic Refactoring Plugin 3.0

Alumnos:
Míryam Gómez San Martín
Íñigo Mediavilla Saiz

Tutor:
Raúl Marticorena Sánchez

Burgos, Julio de 2011

**ANEXO III
ESPECIFICACIÓN DEL DISEÑO**

ÍNDICE DE CONTENIDO

Anexo III	
<i>Especificación del Diseño.....</i>	213
<i>Lista de cambios.....</i>	221
1. INTRODUCCIÓN.....	223
2. DISEÑO DE DATOS.....	224
<i>2.1. DTD de refactorizaciones.....</i>	<i>224</i>
<i>2.2. DTD de clasificaciones.....</i>	<i>227</i>
3. DISEÑO ARQUITECTÓNICO.....	230
<i>3.1. Arquitectura lógica.....</i>	<i>231</i>
<i>3.2. Diagramas de clases.....</i>	<i>232</i>
<i>3.2.1. dynamicrefactoring.domain.....</i>	<i>232</i>
<i>3.2.2. dynamicrefactoring.domain.metadata.....</i>	<i>234</i>
<i>3.2.3. dynamicrefactoring.domain.metadata.classifications.xml.imp.....</i>	<i>237</i>
<i>3.2.4. dynamicrefactoring.domain.metadata.condition.....</i>	<i>238</i>
<i>3.2.5. dynamicrefactoring.interfaz.view.....</i>	<i>239</i>
<i>3.2.6. dynamicrefactoring.interfaz.editor.classifieditor.....</i>	<i>240</i>
<i>3.2.7. dynamicrefactoring.interfaz.wizard.classificationscombo.....</i>	<i>241</i>
<i>3.2.8. dynamicrefactoring.interfaz.wizard.search.internal.....</i>	<i>242</i>
<i>3.2.9. dynamicrefactoring.interfaz.wizard.search.javadoc.....</i>	<i>243</i>
4. DISEÑO DE LA INTERFAZ.....	245
<i>4.1. Asistente para creación y edición de refactorizaciones.....</i>	<i>245</i>
<i>4.2. Interfaz para la exportación de refactorizaciones.....</i>	<i>255</i>
<i>4.3. Vista Available Refactorings.....</i>	<i>257</i>
<i>4.4. Vista Refactoring Catalog Browser.....</i>	<i>259</i>
<i>4.5. Editor de Clasificaciones.....</i>	<i>260</i>
5. DISEÑO PROCEDIMENTAL.....	262
<i>5.1. Diagramas de secuencia.....</i>	<i>262</i>
<i>5.1.1. RF 1: Visualizar refactorizaciones según clasificación.....</i>	<i>263</i>
<i>5.1.2. RF 2: Refrescar visualización de refactorizaciones</i>	<i>264</i>
<i>5.1.3. RF 3: Añadir filtro de refactorizaciones</i>	<i>265</i>
<i>5.1.4. RF 4: Seleccionar opción aplicar filtro</i>	<i>266</i>
<i>5.1.5. RF 5: Eliminar filtro de refactorizaciones</i>	<i>267</i>
<i>5.1.6. RF 6: Eliminar todos los filtros de refactorizaciones</i>	<i>268</i>
<i>5.1.7. RF 7: Seleccionar opción ver refactorizaciones filtradas</i>	<i>269</i>
<i>5.1.8. RF 8: Visualizar detalle refactorización</i>	<i>270</i>
<i>5.1.9. RF 9: Añadir clasificación</i>	<i>271</i>
<i>5.1.10. RF 10: Editar clasificación</i>	<i>272</i>
<i>5.1.11. RF 11: Eliminar clasificación</i>	<i>274</i>
<i>5.1.12. RF 12: Añadir categoría a una clasificación</i>	<i>275</i>

5.1.13. RF 13: Renombrar categoría de una clasificación	276
5.1.14. RF 14: Eliminar categoría de una clasificación	277
5.1.15. RF 15: Mostrar resumen elemento seleccionado	278
5.1.16. RF 16: Realizar búsqueda de elementos	279
6. REPRESENTACIÓN CTTE DE DYNAMIC REFACTORING.....	280
6.1. Modificaciones en la primera página del wizard.....	280
6.2. Visualizar catálogo de Refactorizaciones.....	281
6.3. Editar clasificaciones.....	282
6.4. Añadir una nueva clasificación.....	283
6.5. Eliminar una clasificación.....	283
7. PATRONES DE DISEÑO.....	284
7.1. Patrón Objeto Constructor - Builder.....	284
7.2. Patrón Fachada.....	286
7.3. Patrón Singleton.....	288
7.4. Patrón Comando	289
8. REFERENCIA CRUZADA CON LOS REQUISITOS.....	292
9. DISEÑO DE PRUEBAS.....	294
9.1. RF 1: Visualizar refactorizaciones según clasificación	294
9.2. RF 2: Refrescar visualización de refactorizaciones	295
9.3. RF 3: Añadir filtro de refactorizaciones	296
9.4. RF 4: Seleccionar opción aplicar filtro	297
9.5. RF 5: Eliminar filtro de refactorizaciones	297
9.6. RF 6: Eliminar todos los filtros de refactorizaciones	298
9.7. RF 7: Seleccionar opción ver refactorizaciones filtradas	298
9.8. RF 8: Visualizar detalle refactorización	299
9.9. RF 9: Añadir clasificación	299
9.10. RF 10: Editar clasificación	300
9.11. RF 11: Eliminar clasificación	301
9.12. RF 12: Añadir categoría a una clasificación	301
9.13. RF 13: Renombrar categoría de una clasificación	302
9.14. RF 14: Eliminar categoría de una clasificación.....	302
9.15. RF 15: Mostrar resumen elemento seleccionado	303
9.16. RF 16: Realizar búsqueda de elementos	303
10. SELECCIÓN DE MÉTRICAS.....	304
11. ENTORNO TECNOLÓGICO DE LA APLICACIÓN.....	308
12. PLAN DE DESARROLLO E IMPLANTACIÓN.....	308
12.1. Diagrama de componentes.....	309
12.2. Diagrama de despliegue.....	313

ÍNDICE DE ILUSTRACIONES

Ilustración 29: DTD de clasificaciones.....	228
Ilustración 30: Diagrama global de paquetes.....	231
Ilustración 31: D. de clases de dynamicrefactoring.domain.....	232
Ilustración 32: Detalle clase Enumeración Scope.....	234
Ilustración 33: D. de clases de dynamicrefactoring.domain.metadata.....	236
Ilustración 34: D. de clases de domain.metadata.classifications.xml.imp.....	237
Ilustración 35: D. de clases de dynamicrefactoring.domain.metadata.condition.....	238
Ilustración 36: D. de clases relevantes de dynamicrefactoring.interfaz.view.....	239
Ilustración 37: D. de clases de dynamicrefactoring.interfaz.editor.classifieditor.....	240
Ilustración 38: D. de clases dynamicrefactoring.interfaz.wizard.classificationscombo.....	241
Ilustración 39: D. de clases de dynamicrefactoring.interfaz.wizard.search.internal.....	242
Ilustración 40: D. de clases de dynamicrefactoring.interfaz.wizard.search.javadoc.....	243
Ilustración 41: Asistente creación/edición refactorizaciones.....	246
Ilustración 42: Selección refactorización antes de modificaciones.....	247
Ilustración 43: Selección refactorización después de modificaciones.....	247
Ilustración 44: Primera página del asistente antes de modificaciones.....	248
Ilustración 45: Primera página del asistente después de modificaciones.....	249
Ilustración 46: Segunda página del asistente antes de modificaciones.....	250
Ilustración 47: Segunda página del asistente después de modificaciones.....	251
Ilustración 48: Páginas 3,4 y 5 antes de modificaciones.....	252
Ilustración 49: Páginas 3, 4 y 5 después de modificaciones.....	253
Ilustración 50: Séptima página del asistente antes de modificaciones.....	254
Ilustración 51: Séptima página del asistente después de modificaciones.....	255
Ilustración 52: Interfaz exportación refactorizaciones antes de modificaciones.....	256
Ilustración 53: Interfaz exportación refactorizaciones después de modificaciones.....	257
Ilustración 54: Vista Available Refactorings antes de modificaciones.....	258
Ilustración 55: Vista Available Refactorings después de modificaciones.....	259
Ilustración 56: Vista Refactoring Catalog Browser.....	260
Ilustración 57: Editor de clasificaciones.....	261
Ilustración 58: D. de secuencia de RF 1: Visualizar ref. según clasificación.....	263
Ilustración 59: D. de secuencia de RF 2: Refrescar visualización de refactorizaciones	264
Ilustración 60: D. de secuencia de RF 3: Añadir filtro de refactorizaciones.....	265
Ilustración 61: D. de secuencia de RF 4: Seleccionar opción aplicar filtro.....	266
Ilustración 62: D. de secuencia de RF 5: Eliminar filtro de refactorizaciones.....	267
Ilustración 63: D. de secuencia de RF 6: Eliminar todos los filtros de refactorizaciones	268
Ilustración 64: D. de secuencia de RF 7: Seleccionar opción ver ref. filtradas.....	269
Ilustración 65: D. de secuencia de RF 8: Visualizar detalle refactorización.....	270
Ilustración 66: D. de secuencia de RF 9: Añadir clasificación.....	271
Ilustración 67: D. de secuencia de RF 10: Detalle - Renombrar una clasificación.....	272
Ilustración 68: D. de secuencia de RF 10: Detalle - Modificar desc. de clasificación..	273
Ilustración 69: D. de secuencia de RF 11: Eliminar clasificación.....	274
Ilustración 70: D. de secuencia de RF 12: Añadir categoría a una clasificación.....	275
Ilustración 71: D. de secuencia de RF 13: Renombrar categoría.....	276
Ilustración 72: D. de secuencia de RF 14: Eliminar categoría de una clasificación.....	277
Ilustración 73: D. de secuencia de RF 15: Mostrar resumen elemento seleccionado...	278
Ilustración 74: D. de secuencia de RF 16: Realizar búsqueda de elementos.....	279
Ilustración 75: Ctte de la primer página del asistente.....	280
Ilustración 76: CTTE de visualizar catálogo de refactorizaciones.....	281

Ilustración 77: CTTE de edición de clasificaciones.....	282
Ilustración 78: CTTE de renombrar una categoría.....	282
Ilustración 79: CTTE de eliminar categoría.....	282
Ilustración 80: CTTE de agregar una nueva clasificación.....	283
Ilustración 81: CTTE de eliminar una clasificación.....	283
Ilustración 82: D. de clases del PD Builder aplicado a InputParameter.....	286
Ilustración 83: D. de clases del PD Facade aplicado a SearchingFacade.....	287
Ilustración 84: D. de clases del PD Singleton aplicado a EclipseBasedJavadocReader.....	289
Ilustración 85: D. de clases del PD Command aplicado a RefreshViewAction.....	291
Ilustración 86: Diagrama de componentes.....	312
Ilustración 87: Diagrama de despliegue.....	314

ÍNDICE DE TABLAS

Tabla 39: Requisitos funcionales.....	292
Tabla 40: Referencia cruzada de requisitos con módulos de diseño.....	293
Tabla 41: Pruebas de RF1: Visualizar refactorizaciones según clasificación.....	294
Tabla 42: Pruebas de RF 2: Refrescar visualización de refactorizaciones.....	295
Tabla 43: Pruebas de RF 3: Añadir filtro de refactorizaciones.....	296
Tabla 44: Pruebas de RF 4: Seleccionar opción aplicar filtro.....	297
Tabla 45: Pruebas de RF 5: Eliminar filtro de refactorizaciones.....	297
Tabla 46: Pruebas de RF 6: Eliminar todos los filtros de refactorizaciones.....	298
Tabla 47: Pruebas de RF 7: Seleccionar opción ver refactorizaciones filtradas.....	298
Tabla 48: Pruebas de RF 8: Visualizar detalle refactorización.....	299
Tabla 49: Pruebas de RF 9: Añadir clasificación.....	299
Tabla 50: Pruebas de RF 10: Editar clasificación.....	300
Tabla 51: Pruebas de RF 11: Eliminar clasificación.....	301
Tabla 52: Pruebas de RF 12: Añadir categoría a una clasificación.....	301
Tabla 53: Pruebas de RF 13: Renombrar categoría de una clasificación.....	302
Tabla 54: Pruebas de RF 14: Eliminar categoría de una clasificación.....	302
Tabla 55: Pruebas de RF 15: Mostrar resumen elemento seleccionado.....	303
Tabla 56: Pruebas de RF 16: Realizar búsqueda de elementos.....	303

LISTA DE CAMBIOS

Número	Fecha	Descripción	Autor/es
0	10/05/11	Agregados primera parte del diseño arquitectónico.	Miryam Gómez e Íñigo Mediavilla
1	15/05/11	Terminada parte del diseño arquitectónico.	Miryam Gómez e Íñigo Mediavilla
2	20/05/11	Incluido diseño procedimental, diagramas de secuencia.	Miryam Gómez e Íñigo Mediavilla
3	24/05/11	Incluido apartados dedicados a la selección de métricas, entorno tecnológico de la aplicación y plan de desarrollo e implantación	Miryam Gómez e Íñigo Mediavilla
4	25/05/11	Diseño de pruebas del plugin y patrones de diseño.	Miryam Gómez e Íñigo Mediavilla
5	26/05/11	Incluido apartado diseño de la interfaz y referencia cruzada con requisitos.	Miryam Gómez e Íñigo Mediavilla
6	29/05/11	Diseño de pruebas del plugin y diseño de datos.	Miryam Gómez e Íñigo Mediavilla
7	07/06/11	Última revisión del anexo, se incluyen pequeños detalles.	Miryam Gómez e Íñigo Mediavilla

1. INTRODUCCIÓN

En este anexo se plantea la solución software del problema. En él se expondrán los detalles del diseño de la aplicación para conseguir los factores de calidad externos e internos que marcarán la calidad del producto software final, basándose en las directivas indicadas en la fase de análisis se toman decisiones en cuanto a la arquitectura, los datos y la interfaz.

El diseño marca las directrices al programador para continuar con el proyecto en su fase de implementación.

Según el paradigma objetual, describimos la arquitectura software utilizando diferentes vistas que representan la misma información pero desde puntos de vista distintos. Estas representaciones se realizarán de la siguiente forma:

Diseño arquitectónico

- Diagramas de clases

Diseño procedimental

- Diagramas de secuencia
- Diagramas de colaboración

Plan de desarrollo e implantación

- Diagrama de despliegue
- Diagrama de componentes

2. DISEÑO DE DATOS

El apartado de diseño de datos es habitual y muy importante en aplicaciones orientadas a la gestión de datos, basadas normalmente en el uso de algún tipo de base de datos y sistema gestor de bases de datos. Sin embargo, en este proyecto los datos no son almacenados en bases de datos sino que se utilizan ficheros XML para su persistencia.

En este proyecto, se ha visto modificada la estructura del fichero XML que almacena las refactorizaciones dinámicas para la adición de nuevos campos. Además, se ha creado una nueva estructura de ficheros XML para guardar determinada información relativa a las clasificaciones que serán utilizadas por la aplicación.

La estructura utilizada en cada uno de los archivos XML se declara el fichero *DTD* (*Document Type Definition*) correspondiente. Dicho fichero, constituye en sí mismo una fuente de documentación acerca de la forma en que se almacenan las refactorizaciones y clasificaciones de la aplicación. Además, se utiliza a la hora de cargar los ficheros XML para su lectura con el fin de comprobar que los archivos cargados cumplen con la especificación impuesta.

2.1. DTD de refactorizaciones

Los ficheros XML que cumplen con la estructura de datos marcada en este DTD almacenan la información relativa a una determinada refactorización.

Estos ficheros XML son utilizados en varias funcionalidades que ofrece la aplicación, como puede ser cuando se quiere mostrar la información asociada a una refactorización, al ejecutar la propia refactorización sobre un código fuente, así como en el proceso de exportación/importación de refactorización o de plan de refactorizaciones.

A continuación se muestra al completo la estructura del DTD de refactorizaciones. En él, marcado en rojo, se muestran los nuevos elementos que el presente proyecto ha incorporado. Posteriormente, se detallará cada uno de estos elementos con el objetivo de que quede clara la finalidad de los mismos.

Estructura DTD de refactorizaciones:

```

<!ELEMENT refactoring ( information, inputs, mechanism, examples? ) >
<!ATTLIST refactoring name NMTOKENS #REQUIRED >
<!ATTLIST refactoring version CDATA #IMPLIED>

<!ELEMENT information ( description, image?, motivation, keywords?, categorization ) >
<!ELEMENT description ( #PCDATA ) >
<!ELEMENT image EMPTY >
<!ATTLIST image src CDATA #REQUIRED >
<!ELEMENT motivation ( #PCDATA ) >

<!ELEMENT keywords ( keyword+ ) >
<!ELEMENT keyword ( #PCDATA ) >
<!ELEMENT categorization ( classification+ ) >
<!ELEMENT classification ( category+ ) >
<!ATTLIST classification name CDATA #REQUIRED >
<!ELEMENT category ( #PCDATA ) >

<!ELEMENT inputs ( input+ ) >
<!ELEMENT input EMPTY >
<!ATTLIST input type NMTOKEN #REQUIRED >
<!ATTLIST input name ID #IMPLIED >
<!ATTLIST input from IDREF #IMPLIED >
<!ATTLIST input method NMTOKEN #IMPLIED >
<!ATTLIST input root (false|true) #IMPLIED >
<!ELEMENT mechanism ( preconditions, actions, postconditions ) >
<!ELEMENT preconditions ( precondition* ) >
<!ELEMENT precondition ( param* ) >
<!ATTLIST precondition name NMTOKEN #REQUIRED >
<!ELEMENT actions ( action* ) >
<!ELEMENT action ( param* ) >
<!ATTLIST action name NMTOKEN #REQUIRED >
<!ELEMENT postconditions ( postcondition* ) >
<!ELEMENT postcondition ( param* ) >
<!ATTLIST postcondition name NMTOKEN #REQUIRED >
<!ELEMENT param EMPTY >
<!ATTLIST param name IDREF #REQUIRED >
<!ELEMENT examples ( example* ) >
<!ELEMENT example EMPTY >
<!ATTLIST example before CDATA #REQUIRED >
<!ATTLIST example after CDATA #REQUIRED >

```

Elementos del DTD de refactorizaciones:Elemento refactoring/information/

```
<!ELEMENT information ( description, image?, motivation, keywords?, categorization ) >
```

Se trata del primer subelemento que conforma el nodo raíz de la definición de una

refactorización (elemento requerido), el cual ha sido modificado para incluir en su definición las palabras clave y la categorización asociadas a la refactorización. Por lo tanto los nuevos subelementos son los siguientes:

keywords: subelemento opcional correspondiente a las palabras clave.

categorization: subelemento requerido correspondiente a la categorización.

Elemento refactoring/information/keywords

```
<!ELEMENT keywords ( keyword+ ) >
```

Cuarto subelemento del nodo information (elemento opcional).

No contiene datos en sí mismo, sino que alberga la secuencia de palabras clave asociadas a la refactorización.

Elemento refactoring/information/keywords/keyword

```
<!ELEMENT keyword ( #PCDATA ) >
```

Único tipo de subelemento que contiene el nodo keyword (elemento necesario y repetible).

Contiene datos de tipo carácter que corresponden con las palabras clave de la refactorización.

Elemento refactoring/information/categorization

```
<!ELEMENT categorization ( classification+ ) >
```

Quinto subelemento del nodo information (elemento requerido).

No contiene datos en sí mismo, sino que alberga la secuencia de clasificaciones asociadas a la refactorización.

Elemento refactoring/information/categorization/classification

```
<!ELEMENT classification ( category+ ) >
<!ATTLIST classification name CDATA #REQUIRED >
```

Único tipo de subelemento que contiene el nodo classification (elemento necesario y repetible).

Alberga la secuencia de categorías a las que pertenece la refactorización para la clasificación que se está tratando. Además tiene un único atributo denominado name que contiene datos de tipo carácter y corresponde con el nombre de la propia clasificación.

Elemento refactoring/information/categorization/classification/category

```
<!ELEMENT category ( #PCDATA ) >
```

Único tipo de subelemento que contiene el nodo classification (elemento necesario y repetible).

Contiene datos de tipo carácter que corresponden con la categoría de la clasificación a la que pertenece la refactorización.

2.2. DTD de clasificaciones

Los ficheros XML que cumplen con la estructura de datos marcada en este DTD almacenan la información relativa a una determinada clasificación.

Estos ficheros XML son utilizados en varias funcionalidades que ofrece la aplicación, como puede ser cuando se quiere mostrar la información asociada a una clasificación para visualizarla o editarla, en la creación/edición de refactorizaciones para poder clasificar a esta según se crea conveniente, así como a la hora de seleccionar la clasificación por la que se desea visualizar el catálogo de refactorizaciones clasificado en la vista disponible para ello.

A continuación se muestra la estructura del DTD de clasificaciones, posteriormente se detalla la finalidad de cada uno de estos elementos de los que se compone una clasificación.

Estructura DTD de clasificaciones:

```
<!-- ===== Defined Types ===== -->
<!-- A "Boolean" is the string representation of a boolean variable. -->

<!ENTITY % Boolean "(true|false)">

<!-- ===== Top Level Elements ===== -->

<!ELEMENT classifications ( classification+ ) >
<!ATTLIST classifications version CDATA #IMPLIED>

<!ELEMENT classification ( categories+ ) >
<!ATTLIST classification name NMTOKENS #REQUIRED >
<!ATTLIST classification description CDATA #REQUIRED >
<!ATTLIST classification multicategory %Boolean; "false" >

<!ELEMENT categories ( category+ ) >

<!ELEMENT category ( #PCDATA ) >
```

Ilustración 29: DTD de clasificaciones

Elementos del DTD de clasificaciones:

Previamente a la definición de los elementos se ha definido una entidad (Boolean) para realizar la representación de una variable booleana en formato cadena. Esta es la siguiente:

```
<!ENTITY % Boolean "(true|false)">
```

Elemento classifications

```
<!ELEMENT classifications ( classification+ ) >
<!ATTLIST classifications version CDATA #IMPLIED>
```

Nodo raíz del fichero de clasificaciones.

Alberga la secuencia de clasificaciones disponibles y además tiene un único atributo denominado *version* que contiene datos de tipo carácter y corresponde con la versión. Este atributo se puede omitir sin que se adopte automáticamente un valor por defecto.

Elemento classifications/classification

```
<!ELEMENT classification ( categories+ ) >
```

```
<!ATTLIST classification name NMOKENS #REQUIRED >
<!ATTLIST classification description CDATA #REQUIRED >
<!ATTLIST classification multicategory %Boolean; "false" >
```

Único subelemento del nodo classifications (necesario y repetible). Alberga la secuencia de categorías disponibles asociadas a la clasificación que se está tratando. Además tiene los siguientes atributos:

name: nombre de la clasificación. Se trata de un atributo de tipo NMOKENS (Name TOKENS) que se diferencian de los CDATA (Character DATA) en que solo aceptan caracteres válidos para nombrar cosas, como son: letras, números, puntos, guiones, subrayados y dos puntos.

description: descripción asociada a la clasificación, contiene datos de tipo carácter.

multicategory: indica si se trata de una clasificación multicategoría o no, para ello hace uso de la entidad Boolean anteriormente definida.

Elemento classifications/classification/categories

```
<!ELEMENT categories ( category+ ) >
```

Único tipo de subelemento que contiene el nodo classification (elemento necesario y repetible).

Alberga la secuencia de todas las categorías que tiene disponibles la clasificación.

Elemento classifications/classification/categories/category

```
<!ELEMENT category ( #PCDATA ) >
```

Único tipo de subelemento que contiene el nodo categories (elemento necesario y repetible).

Contiene datos de tipo carácter que corresponden con una de las categorías que tiene disponible la clasificación que se está tratando.

3. DISEÑO ARQUITECTÓNICO

Cuando se trabaja en orientación a objetos, la descripción de la arquitectura puede hacerse describiendo los paquetes y sus relaciones, para posteriormente describir cada una de las clases de los paquetes con sus atributos y métodos.

El diseño arquitectónico de un sistema software representa las partes o módulos de que se compone así como las relaciones entre ellos. En este apartado se van a describir los paquetes más importantes modificados en esta última versión del plugin: sus funciones, las clases que los componen y las relaciones con otros paquetes.

3.1. Arquitectura lógica

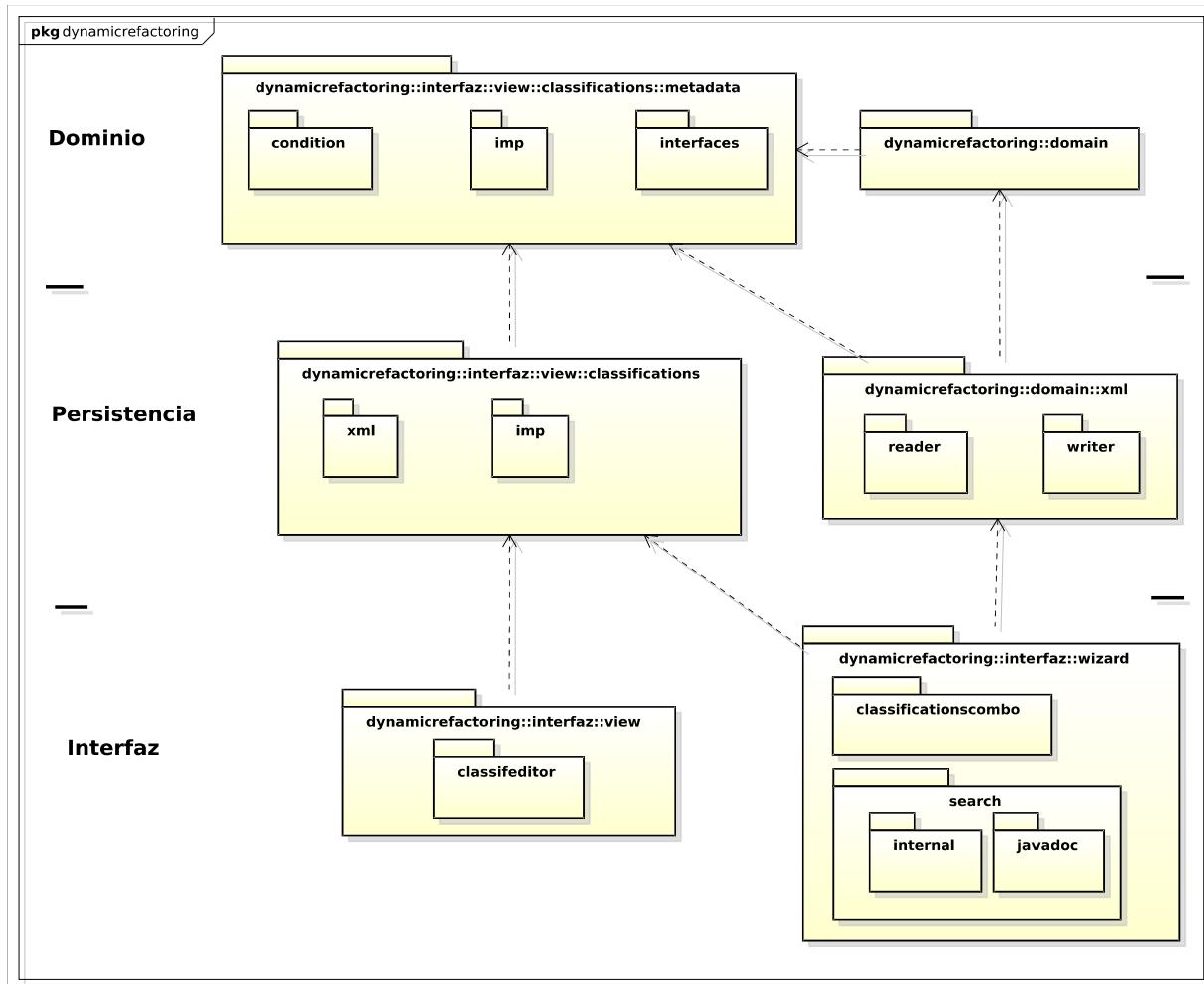


Ilustración 30: Diagrama global de paquetes

Con el diagrama de paquetes obtenemos una visión global de la arquitectura del sistema, además podemos ver las relaciones entre los paquetes que van a ser descritos en las secciones siguientes. Como se puede comprobar las relaciones entre los paquetes están bien definidas según una estructura de capas. La capa inferior es la capa del dominio. Esta capa es utilizada por las otras dos capas existentes, la de persistencia y la de la interfaz. Se ha colocado inmediatamente por debajo de la capa de dominio la capa que corresponde a la de persistencia ya que, a pesar de que las dos hacen uso de la capa de dominio, la capa de la interfaz utiliza a la capa de persistencia para salvar los datos de las modificaciones de los usuarios mientras que la capa de persistencia desconoce cualquier tipo de existencia de la capa de interfaz.

Además de esta jerarquía que se representa de forma vertical en el diagrama según

que paquetes hacen uso de otros, también se puede comprobar la existencia de otra división que se representa de forma horizontal. A la izquierda se encuentran los paquetes con clases relacionadas con los metadatos de las refactorizaciones tanto en el dominio, como en el apartado de persistencia como de la interfaz. A la derecha ocurre lo mismo con las clases del dominio puro de las refactorizaciones. Estas últimas utilizan los paquetes de su izquierda, dado que las refactorizaciones cuentan con metadatos representados en dichos paquetes.

3.2. Diagramas de clases

Mediante los diagramas de clases plasmaremos la estructura estática del sistema.

3.2.1. *dynamicrefactoring.domain*

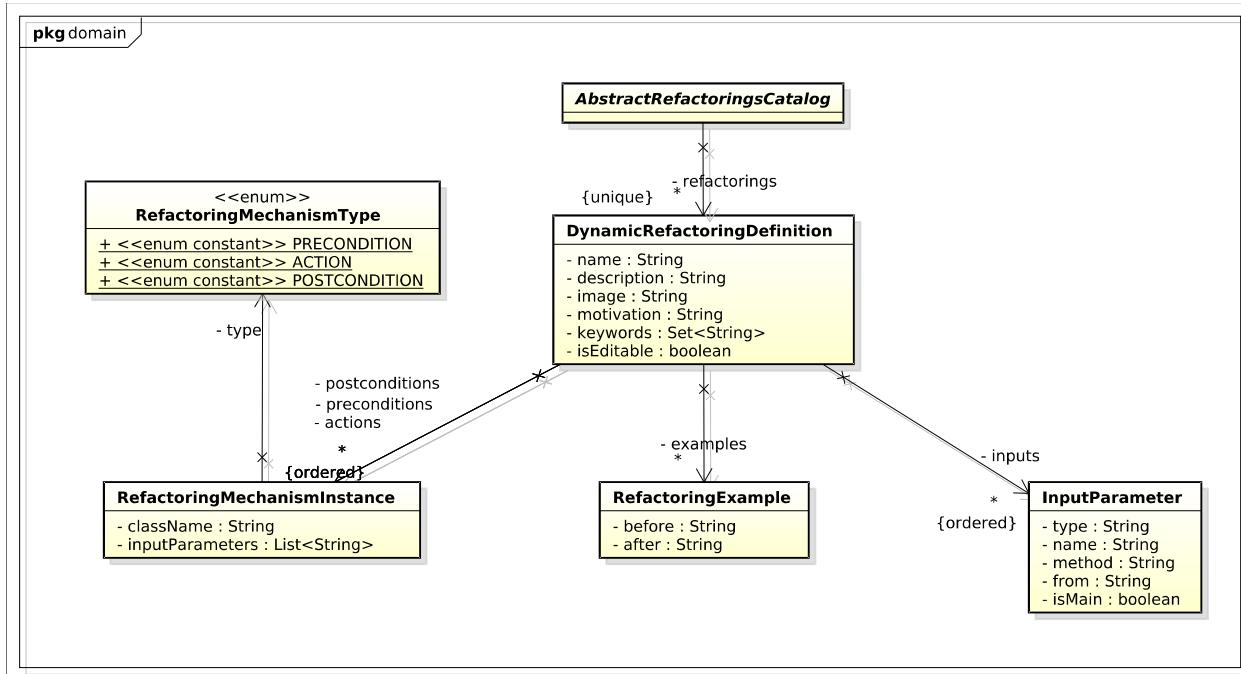


Ilustración 31: D. de clases de *dynamicrefactoring.domain*

En el centro del diagrama se muestra una de las clases más importantes del proyecto *DynamicRefactoringDefinition*. Sobre esta clase se han añadido nuevas funcionalidades y se han realizado una serie de refactorizaciones importantes.

Entre los nuevas características añadidas se podrían destacar las categorías a las que

pertenece una refactorización así como las palabras clave que la describen, atributos que se han añadido para hacer posible clasificar la refactorización en la vista del catálogo de refactorizaciones. Estos cambios han tenido por supuesto un reflejo en el correspondiente fichero XML de definición de las refactorizaciones. Otra mejora destacable consiste en haber añadido un atributo que indica si la refactorización es editable (pertenece al usuario) o no lo es (pertenece al plugin). Este es un cambio que ha sido necesario para hacer frente al problema del versionado de las refactorizaciones que requería que se crearan dos tipos de refactorizaciones las de usuario y las del plugin.

En el aspecto de las refactorizaciones las mejoras han sido notables y se reflejan en la existencia de las otras clases que aparecen en el diagrama y de las que la definición de una refactorización depende. Anteriormente las clases de RefactoringExample, InputParameter y RefactoringMechanismInstance estaban representadas en el plugin por mapas, arrays de cadenas o combinaciones de ambos. Esto hacía el código muy difícil de leer puesto que era difícil comprender a qué campo se refería cada cadena utilizada. Esto era especialmente así, para los parámetros ambiguos de una refactorización que se formaban por un array de mapas de cadenas. Con la sustitución de estas construcciones por clases, el código se ha visto notablemente mejorado.

En cuanto a su función la clase RefactoringExample representa la ruta de un fichero de ejemplo para una refactorización, la clase InputParameter un parámetro de entrada y RefactoringMechanismInstance puede representar cualquiera de los mecanismos de una refactorización, es decir: una precondition, una acción o una postcondición. El tipo de elemento que cada instancia de la clase representa está indicado por un atributo del tipo de la enumeración RefactoringMechanismType, que contiene las tres opciones ya citadas. De este modo se consigue agrupar todas las acciones comunes a los tres tipos de elementos mientras que siempre se puede realizar acciones especializadas para cada instancia ya que se conoce su tipo.

El catálogo AbstractRefactoringsCatalog es una clase abstracta que representa acciones comunes a los catálogos de refactorizaciones pero siempre en memoria, es decir, no se ocupa de la persistencia de los cambios responsabilidad que cede a las subclases que la extiendan.

Otra enumeración a destacar es dynmaicrefactoring.domain.Scope. Esta clase contiene una lista de los posibles ámbitos existentes para una refactorización. Scope sustituye constantes enteras que realizaban esta función en versiones anteriores. Sin embargo, el hecho de agrupar los valores en una enumeración ha proporcionado una

sustancial mejora. Gracias a esto se han podido eliminar un gran número de sentencias switch que complicaban el código, se han eliminado muchos errores provenientes de valores de las constantes incorrectos, se han hecho innecesarios muchas comprobaciones de estas constantes y además se ha ganado en legibilidad del código y en comprobación de tipos automática por parte del compilador.

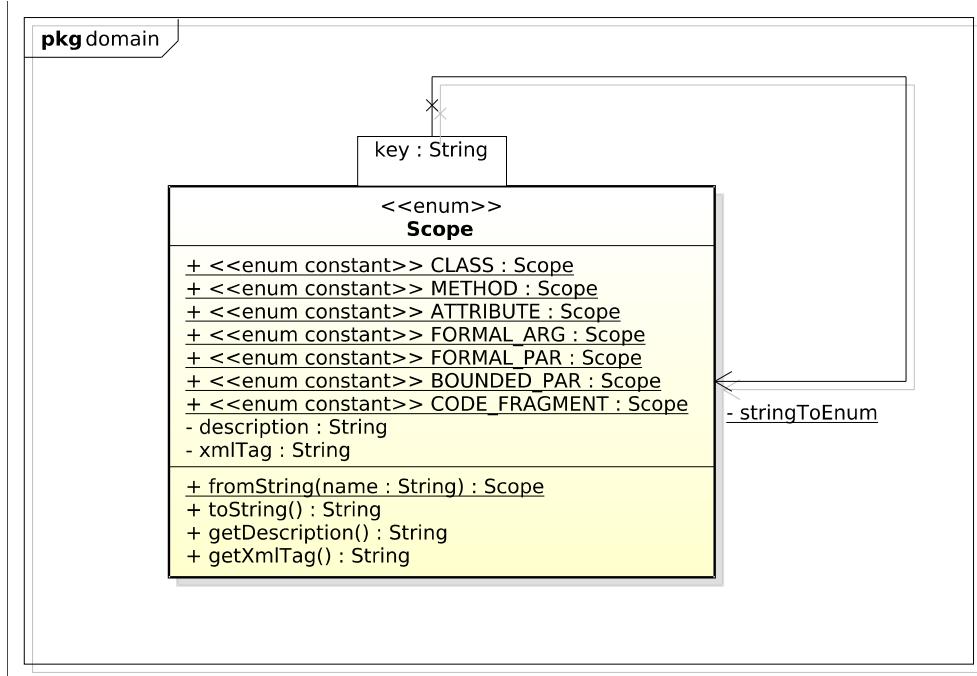


Ilustración 32: Detalle clase Enumeración Scope

3.2.2. dynamicrefactoring.domain.metadata

Este paquete y sus subpaquetes contienen los objetos del modelo de clasificación de las refactorizaciones. De los subpaquetes `condition` y `classifications` se hablará en apartados posteriores por centrarse en cuestiones específicas de este modelo. De momento nos centraremos en el paquete más importante que es el paquete `interfaces`.

Este paquete define como su propio nombre indica, las interfaces correspondientes, únicos elementos públicos del paquete que el resto de módulos deberá conocer. Las interfaces más importantes son `Element`, `Category` y `Classification`. `Element` representa un elemento que se puede agrupar por categorías y por tanto tiene un método `getCategories()` que devuelve las categorías a las que pertenece. `Classification` contiene ciertos atributos descriptivos de la clasificación como su nombre o su descripción y

un conjunto de categorías que definen la clasificación. Las categorías son del tipo `Category`. Este tipo tiene un nombre, y un padre que es el nombre de la clasificación a la que pertenece. Dado que no puede haber dos clasificaciones con el mismo nombre en el plugin, ni dos categorías con el mismo nombre dentro de una clasificación estos atributos identifican de forma única a la categoría.

Sobre las dos interfaces anteriores se definen otras clases que permiten manipular colecciones de elementos clasificables. La interfaz `ClassifiedFilterableCatalog` es un contenedor de elementos clasificados por categorías que incluye los elementos que no pertenecen a ninguna categoría en una categoría especial llamada `None`. Este contenedor tiene la característica especial de que permite aplicar filtros sobre los elementos clasificados de modo que se pueden descartar los elementos que no cumplen con alguno de los filtros aplicados . Los filtros se pueden aplicar de forma acumulativa y también se pueden eliminar con lo que los elementos volverían a pertenecer al grupo de elementos no descartados. Los métodos `getClassificationOfElements()` y su homólogo para los elementos filtrados son los que obtienen la lista de elementos que cumplen o no con los filtros. Estos métodos devuelven objetos de tipo `ClassifiedElements` que facilitan las consultas para conocer qué elementos pertenecen a una categoría.

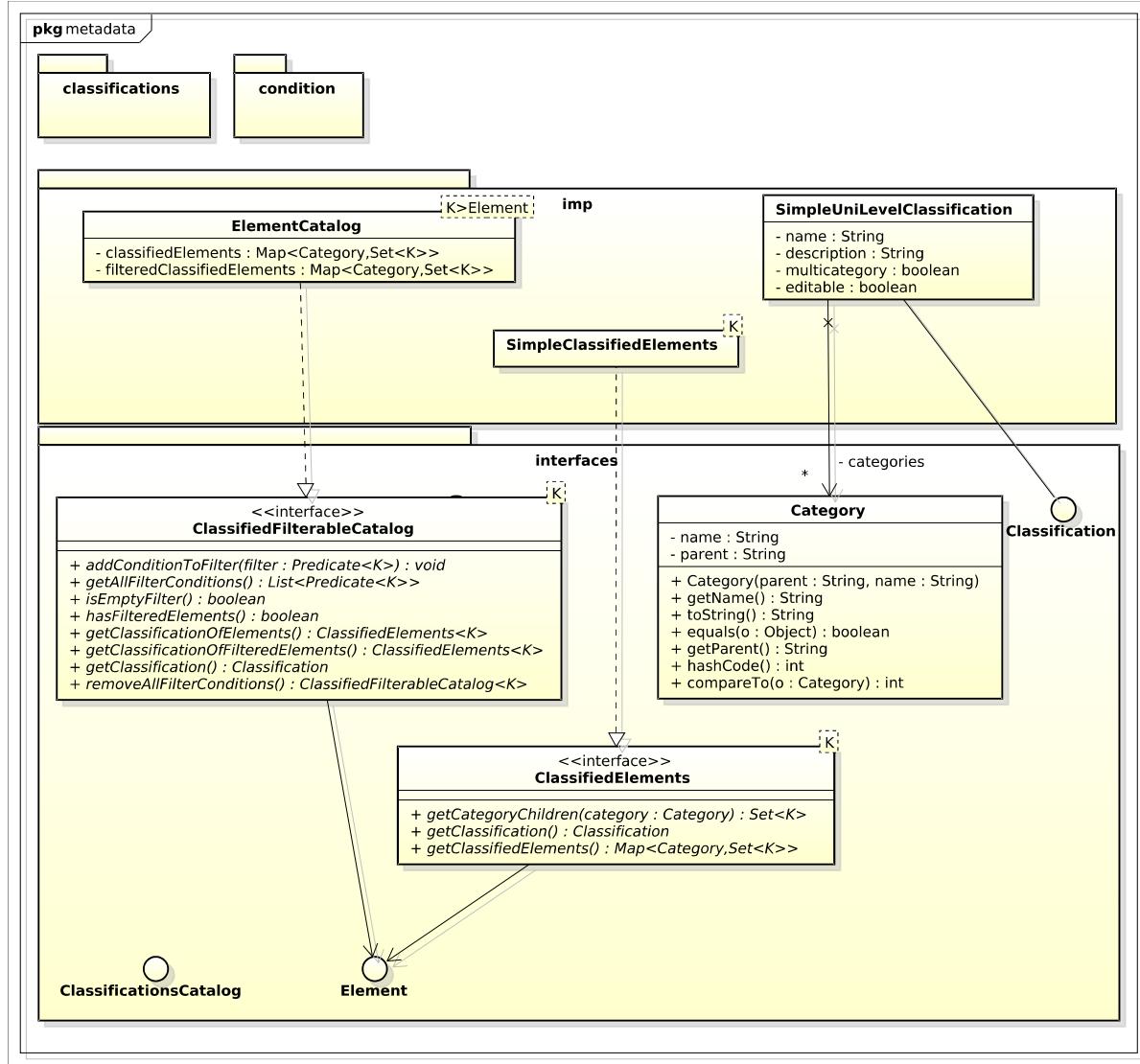


Ilustración 33: D. de clases de `dynamicrefactoring.domain.metadata`

Sobre las interfaces en la figura Ilustración 41 se pueden ver las implementaciones proporcionadas por defecto para estas. No se va a entrar en los detalles de las implementaciones concreta, aunque como detalles a destacar se puede hablar del hecho de que el catálogo de elementos este basado en dos mapas de elementos uno para los elementos filtrados y otro para los no filtrados y el detalle de que la implementación por defecto de `Classification` sólo permite clasificaciones con un nivel de categorías, es decir, sin subcategorías.

3.2.3. *dynamicrefactoring.domain.metadata.classifications.xml.imp*

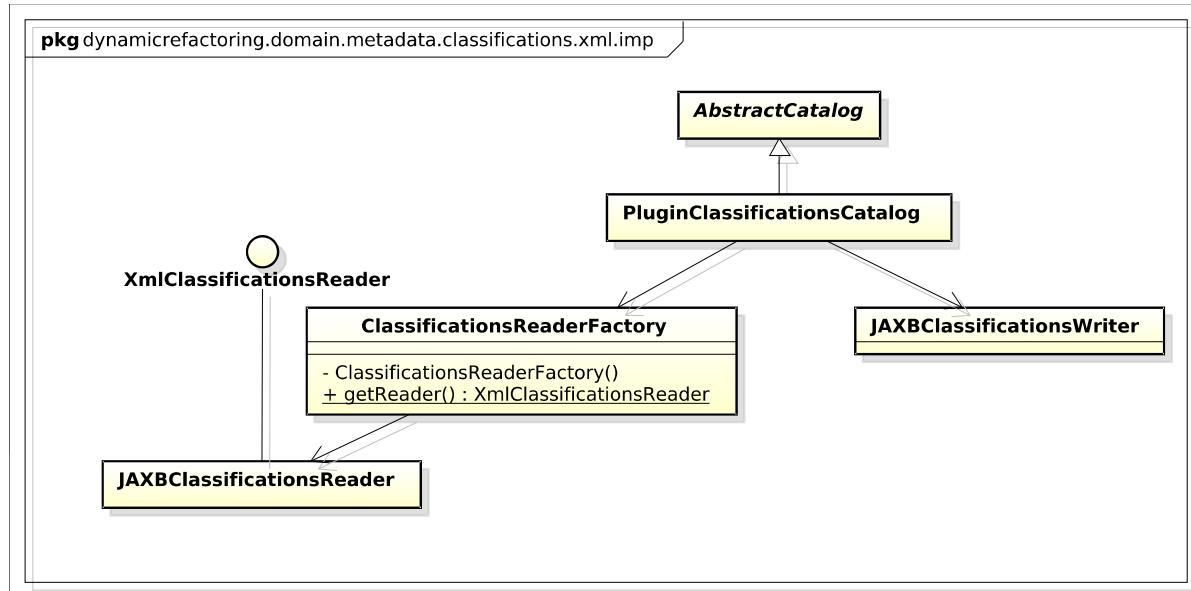


Ilustración 34: D. de clases de domain.metadata.classifications.xml.imp

Este paquete contiene junto con su paquete padre las clases principales encargadas de la lectura y escritura de las clasificaciones en ficheros XML. No se va a entrar a describir en profundidad el funcionamiento del paquete padre dado que sus clases han sido generadas de forma automática con JAXB que es una framework de Java para acceso a XML y por tanto sus detalles son irrelevantes. Basta con decir, que dicho paquete proporciona una serie de clases que encapsulan la lectura de los ficheros XML permitiendo acceder directamente a objetos generados a partir de los ficheros XML leídos.

Centrándonos ya en el módulo del apartado, la única clase que es de interés para el resto de paquetes y esta declarada como pública es `PluginClassificationsCatalog`. Esta clase hace la función de catálogo y repositorio único de clasificaciones. Todas las operaciones de consulta o modificación de las clasificaciones pasan por ella haciendo así una función de fachada sobre este paquete. Así se evita que otras clases fuera de este paquete tengan que conocer detalles sobre el almacenamiento de las clasificaciones. Además hace las funciones de protector, para evitar que ciertas clases modifiquen clasificaciones cuando no deberían y de fuente fiable de información sobre las refactorizaciones.

`PluginClassificationsCatalog` utiliza la clase `JAXBClassificationsWriter` para transformar los objetos de las clasificaciones en ficheros XML. Por otro lado utiliza `ClassificationsReaderFactory` para obtener una instancia de un lector de clasificaciones que en concreto será la clase `JAXBClassificationsReader`. En un principio ante la inseguridad

en el manejo de la biblioteca de XML. *JAXB*, se decidió implementar otro lector basado en *JDOM* biblioteca ya utilizada en otras partes del plugin lo que daba más sentido a la existencia de la fábrica. Sin embargo, una vez se comprobó lo sencillo que era el uso de *JAXB*, se eliminó el lector basado en *JDOM* por sus dificultades de mantenimiento y se decidió mantener la fábrica para así permitir al programador escoger entre otras variantes de fábricas en caso de que aparecieran.

3.2.4. *dynamicrefactoring.domain.metadata.condition*

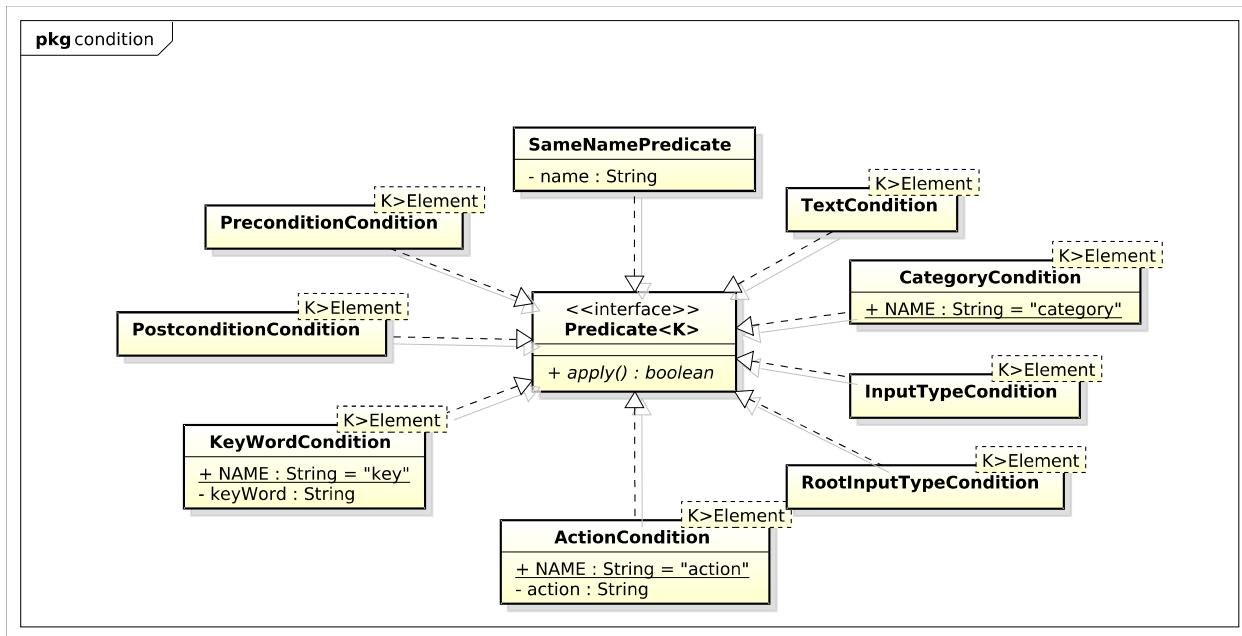


Ilustración 35: D. de clases de `dynamicrefactoring.domain.metadata.condition`

Este paquete, como se puede ver en el diagrama, está formado por un conjunto de condiciones. Todas ellas implementan la interfaz `Predicate<K>` que ha sido incluido en el diagrama para facilitar la visualización pero realmente pertenece a la biblioteca de google para java guava [guava-libraries n.d.]. La función de esta interfaz es la de comprobar condiciones sobre un objeto de tipo `K` y sólo contiene un método `apply()` que recibe un objeto de dicho tipo y devuelve si ese objeto cumple la condición definida por el predicado.

Las condiciones existentes sirven principalmente como filtros para el clasificador de elementos `ClassifiedFilterableCatalog`, el cual como ya se ha descrito permite aplicar condiciones para el filtrado de sus elementos.

Por citar un ejemplo que puede servir para comprender mejor el funcionamiento de estas clases escogeremos la clase `CategoryCondition`. Esta clase recibe en su constructor

un objeto de tipo `Category`. Cuando se llama al método `apply()` de `CategoryCondition`, esta devuelve verdadero si el elemento que se le ha pasado pertenece a la categoría que se pasó al constructor de la condición. Si un objeto de `ClassifiedFilterableCatalog` recibe un filtro de tipo `CategoryCondition` lo que hará será pasar al grupo de elementos descartados aquellos que no pertenezcan a la categoría indicada por la condición.

Lo positivo de este enfoque es que el catálogo es independiente de la forma en que estan definidas las condiciones y podría recibir una condición de tipo `KeywordCondition` o cualquier otra que implementara `Predicate<K>` sin ningún problema.

3.2.5. *dynamicrefactoring.interfaz.view*

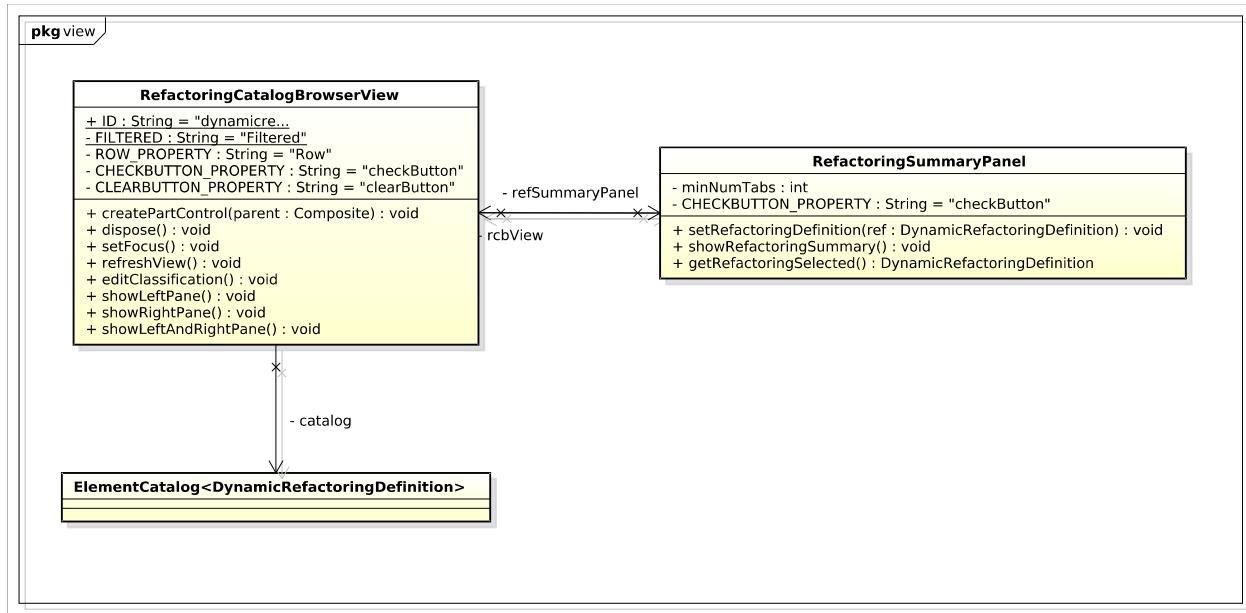


Ilustración 36: D. de clases relevantes de *dynamicrefactoring.interfaz.view*

En el diagrama reflejado en Ilustración 42 aparecen las clases más relevantes de las que se han agregado en esta última versión del plugin para el paquete `dynamicrefactoring.interfaz. view`. La clase `RefactoringCatalogBrowserView` representa la vista para el catálogo con todas las refactorizaciones. Como se puede ver esta clase contiene un campo de tipo `ElementCatalog`, este objeto es el que le permite obtener las refactorizaciones clasificadas por categorías y a su vez agregar filtros para ocultar las refactorizaciones que no cumplen con los criterios establecidos por el usuario.

La otra clase importante añadida al paquete de las vistas es la clase del panel, es decir, `RefactoringSummaryPanel`. Esta clase representa el panel derecho de la vista del

catálogo que muestra las pestañas con toda la información sobre la refactorización: su nombre, descripción, motivación, categorías, palabras claves, mecanismos que la componen y ejemplos o imágenes representativas si la refactorización las tiene.

3.2.6. *dynamicrefactoring.interfaz.editor.classifieditor*

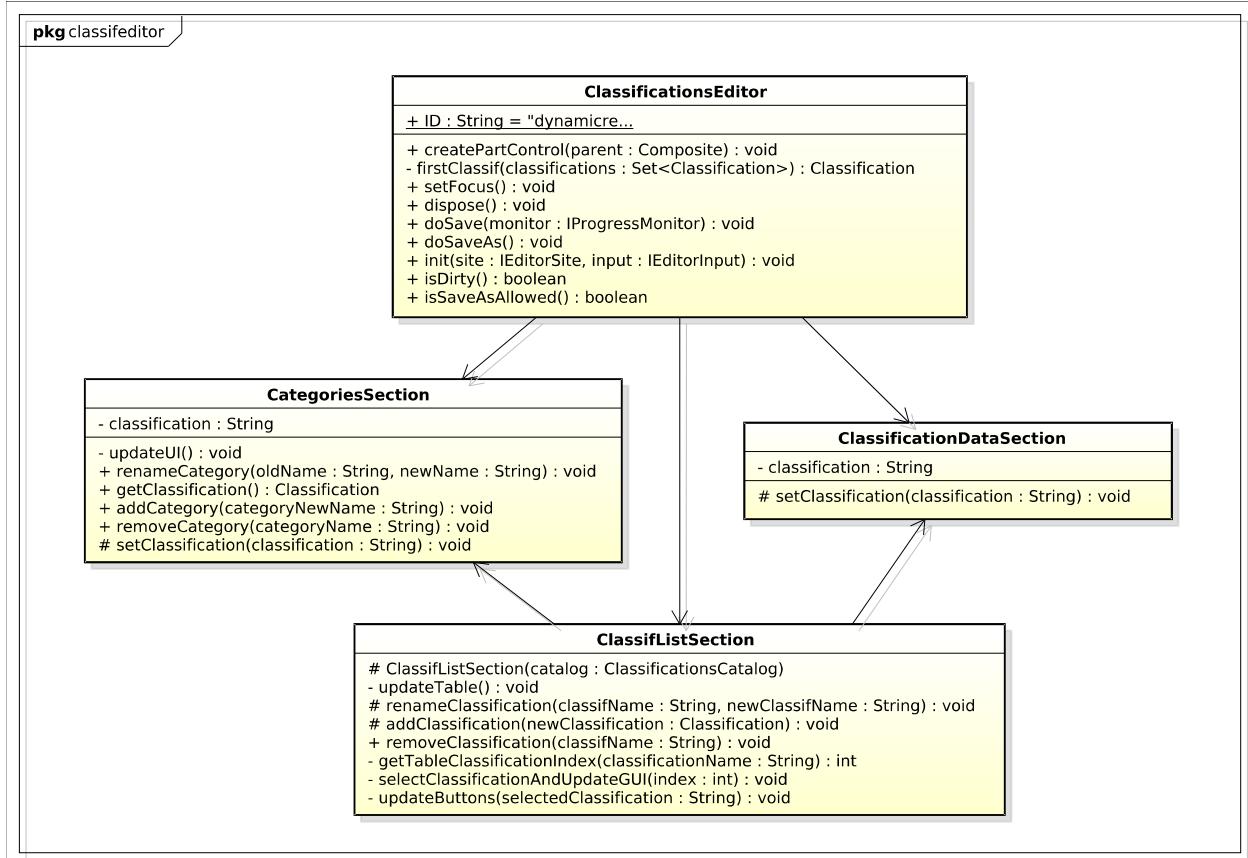


Ilustración 37: D. de clases de dynamicrefactoring.interfaz.editor.classifieditor

Este paquete contiene las clases relacionadas con el editor de clasificaciones. En la parte superior del plugin se encuentra el editor que implementa la clase `EditorPart` como requisito imprescindible para convertirse en un editor de Eclipse. La clase `ClassificationsEditor` implementa los métodos necesarios de `EditorPart` y delega el resto de funciones al resto de clases del paquete. Del resto de clases del paquete cada una se ocupa de una sección distinta del editor.

`ClassifListSection` contiene la sección en la que se listan las clasificaciones existentes y se puede añadir nuevas clasificaciones o seleccionar una de ellas para modificar sus categorías o sus atributos. De la edición de sus atributos se encarga la clase `ClassificationDataSection` y de la de las categorías `CategoriesSection`.

ClassifListSection está relacionada con las otras dos secciones dado que cuando el usuario selecciona una nueva clasificación, ésta se encarga de notificárselo a los otros dos apartados para que actualicen sus datos en la interfaz.

3.2.7. *dynamicrefactoring.interfaz.wizard.classificationscombo*

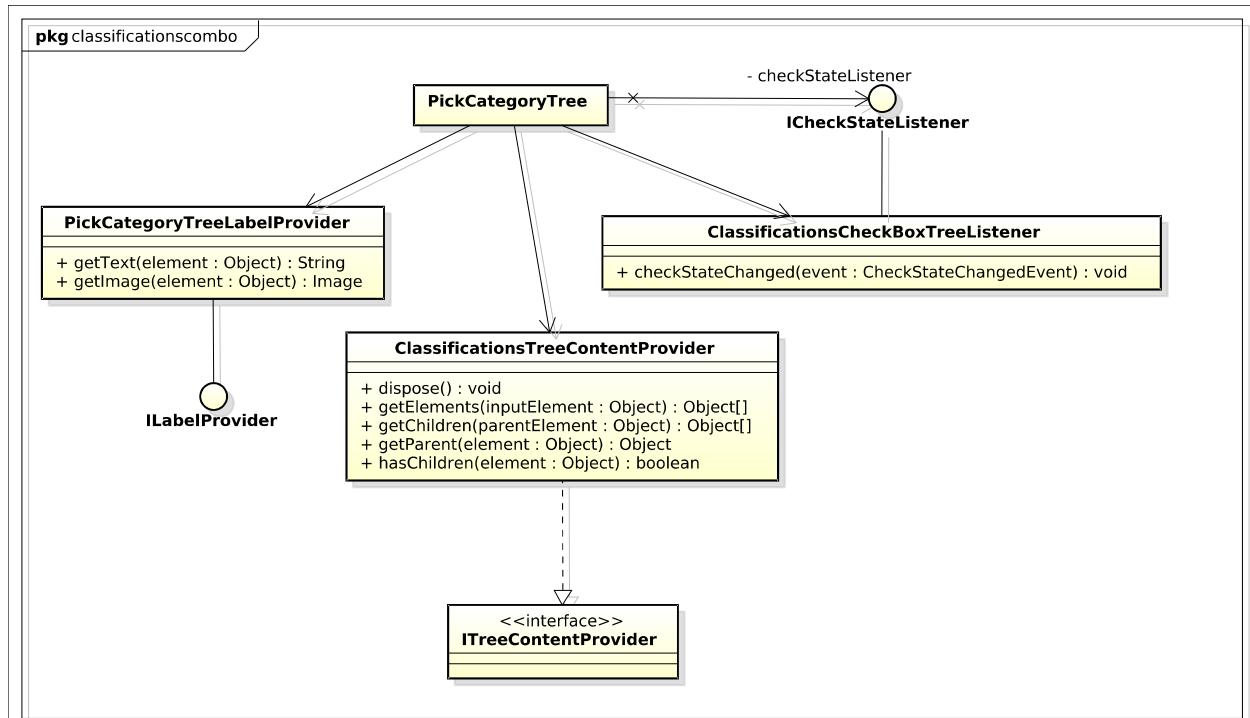


Ilustración 38: D. de clases *dynamicrefactoring.interfaz.wizard.classificationscombo*

Esta paquete contiene las clases que se encargan de mostrar el catálogo que permite seleccionar las categorías a las que una refactorización pertenece en el asistente de creación de refactorizaciones.

PickCategoryTree es la clase principal de este paquete dado que se ocupa de crear el árbol, que es el objeto principal en la elección de categorías, junto con otros objetos que se encargan de gestionar el comportamiento del árbol. Estos objetos son instancias del resto de clases que aparecen en el diagrama.

Una instancia de ClassificationsCheckBoxTreeListener escucha eventos de cambios en los elementos seleccionados en el árbol. Su función es asegurarse de que no se permite asignar combinaciones de categorías no permitidas al usuario. Por ejemplo, ciertas clasificaciones a las que hemos calificado de unicategoría no permiten que un elemento pertenezca a más de una de sus categorías. La función del listener es controlar que esto no

se produzca no permitiendo al usuario marcar más de una categoría para esas clasificaciones.

`ITreeContentProvider` junto con `ILabelProvider` son las interfaces que el API de Eclipse ofrece para permitir establecer una división entre la representación del árbol en memoria y cómo se muestra en la interfaz gráfica. `ITreeContentProvider` y su implementación en el paquete que nos ocupa `ClassificationsTreeContentProvider` tienen como función proporcionar los elementos que conforman el árbol, es decir, esta clase representa la capa del modelo del árbol. `ILabelProvider` y su implementación `PickCategoryTreeLabelProvider` forman la capa de la vista encargadas de devolver la representación de un elemento en el árbol tanto de forma textual como gráfica.

3.2.8. *dynamicrefactoring.interfaz.wizard.search.internal*

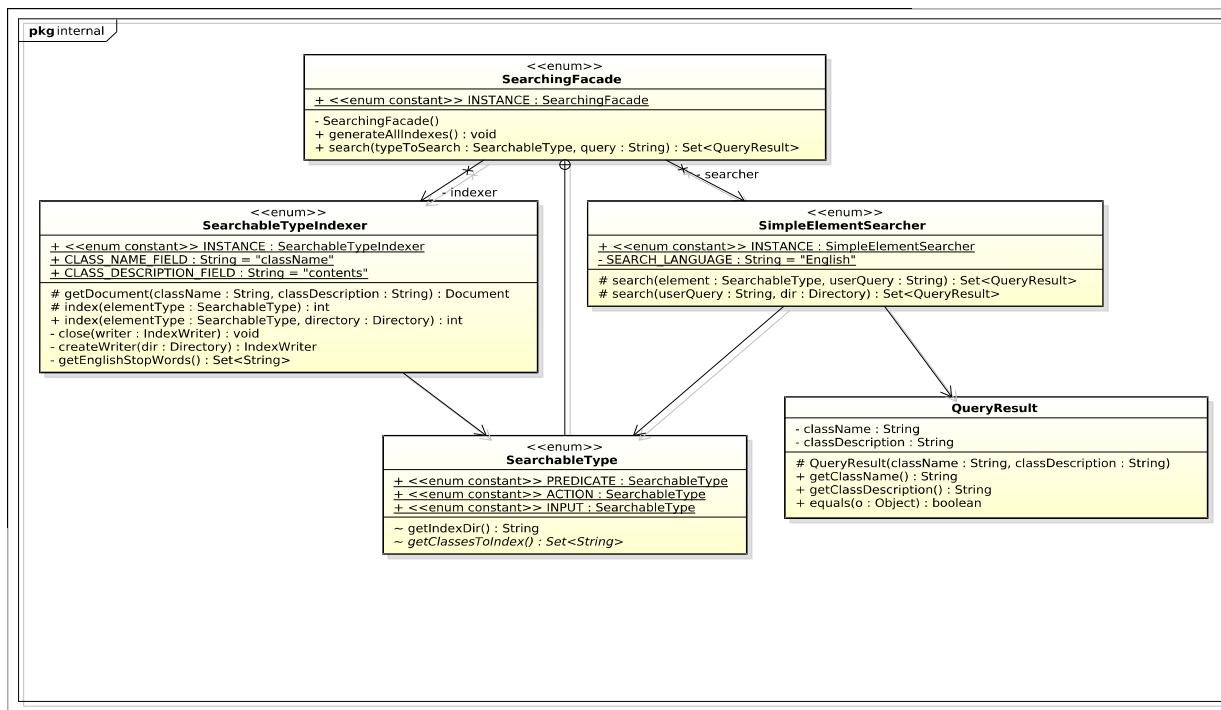


Ilustración 39: D. de clases de `dynamicrefactoring.interfaz.wizard.search.internal`

Este paquete contiene las clases encargadas de implementar la búsqueda para las pantallas de entradas, precondiciones, acciones y postcondiciones del asistente de creación de refactorizaciones.

De nuevo se ha recurrido al patrón fachada para aislar a los clientes de este paquete de las peculiaridades de la implementación de la búsqueda. A los ojos de los clientes la

fachada sólo expone los únicos dos métodos que se considera que pueden ser de interés para estos: `search()` y `generateAllIndexes()`.

El primero genera los índices de todos los elementos (entradas, precondiciones, acciones y postcondiciones) para hacer posible la búsqueda posteriormente. La búsqueda se realiza a través del segundo que recibe dos argumentos: el primero es el tipo de elemento a buscar de entre los ya citados anteriormente y el segundo es la búsqueda a realizar. El resultado de la búsqueda es un conjunto de objetos de tipo `QueryResult`. Los objetos de esta clase hacen accesibles todos los campos que se pueden obtener por cada elemento obtenido como resultado de una búsqueda.

Las clases puramente internas de este paquete son el indizador y el buscador. El indizador (`SearchableTypeIndexer`) genera directorios con índices para los distintos tipos de elementos según se le pida. El buscador (`SimpleElementSearcher`) realiza las búsquedas sobre los directorios de índices anteriormente generados por el indizador.

3.2.9. *dynamicrefactoring.interfaz.wizard.search.javadoc*

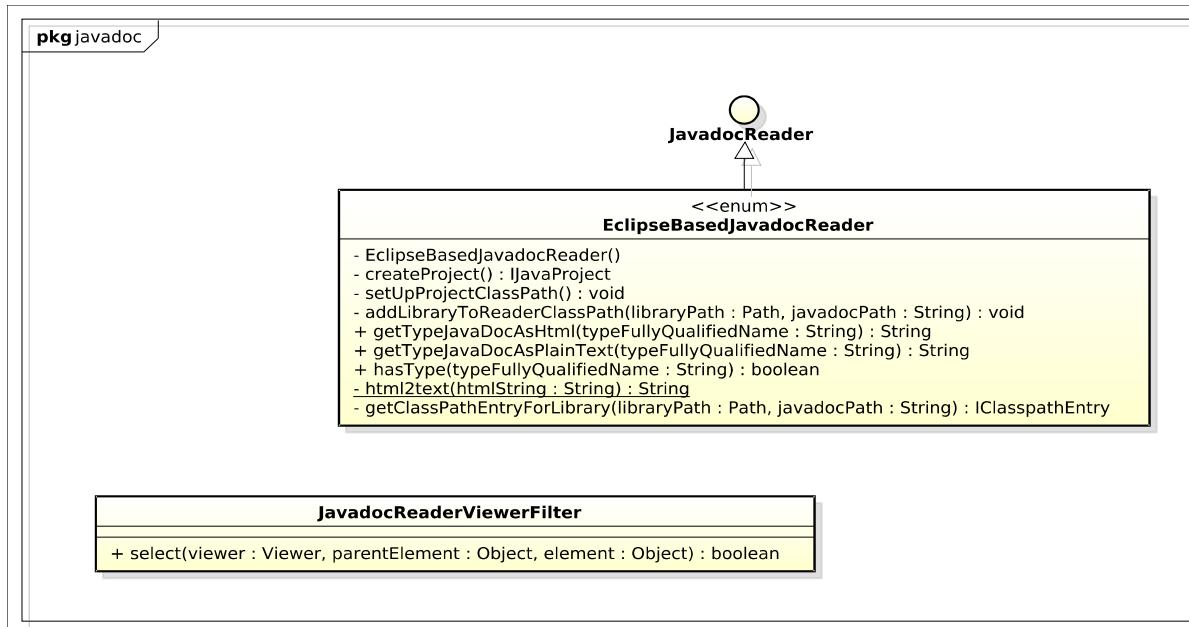


Ilustración 40: D. de clases de *dynamicrefactoring.interfaz.wizard.search.javadoc*

Este paquete contiene las clases que se encargan de leer la documentación de los ficheros *HTML* de la documentación de las clases en formato javadoc. Esta función la realiza la clase `EclipseBasedJavadocReader` que implementa la interfaz `JavadocReader`. El lector de documentación javadoc se basa en utilizar las funcionalidades de los proyectos de Eclipse para obtener la documentación de las clases. Para evitar que los proyectos de

Eclipse que se crean temporalmente para la lectura de la documentación sean percibidos por el usuario se ha utilizado JavadocReaderViewerFilter que filtra dichos proyectos ocultándolos a la vista de los usuarios del plugin.

4. DISEÑO DE LA INTERFAZ

El diseño de la interfaz gráfica de usuario es un aspecto de gran relevancia en un proyecto software. En concreto en nuestro proyecto, el desarrollo de la interfaz ha supuesto una parte considerable del tiempo total invertido, fundamentalmente la implementación.

Gran parte de la interfaz gráfica ya se encontraba implementada en las versiones anteriores de la aplicación [Fuente & Herrero n.d.] y [Fuente de la Fuente n.d.], es por ello que más que explicar el conjunto de elementos que conforman la misma, nos centraremos en este apartado en explicar cómo se han modificado aquellas pantallas ya creadas y también detallaremos las que han sido fruto de nueva creación, con el objetivo de aclarar las diferencias que se presentan respecto a la situación anterior de la aplicación.

Una parte fundamental de una aplicación es su interfaz gráfica. La usabilidad de un producto depende en su mayor parte de la interfaz de usuario, por ello es necesario un buen diseño de la misma.

Las interfaces de usuario se tienen que crear lo suficientemente intuitivas para que el usuario pueda saber cómo completar la tarea en cada paso y permitir un aprendizaje rápido sobre la utilización de la herramienta.

Con el fin de facilitar el manejo del plugin se han realizado una serie de mejoras que permiten un uso más intuitivo de la misma.

4.1. Asistente para creación y edición de refactorizaciones

El proceso de creación y edición de refactorizaciones es, probablemente, la operación más compleja de cara al usuario de entre todas las que permite llevar a cabo la funcionalidad proporcionada por el plugin. Por este motivo, se ha llevado a la práctica mediante un asistente que, paso a paso, guía al usuario a lo largo de la configuración de todos los puntos que deben componer una refactorización.

El diseño de todas las páginas del asistente consta de tres partes comunes; una franja de título, un área de configuración y una barra de botones. Estos componentes se pueden visualizar en la siguiente imagen del asistente.

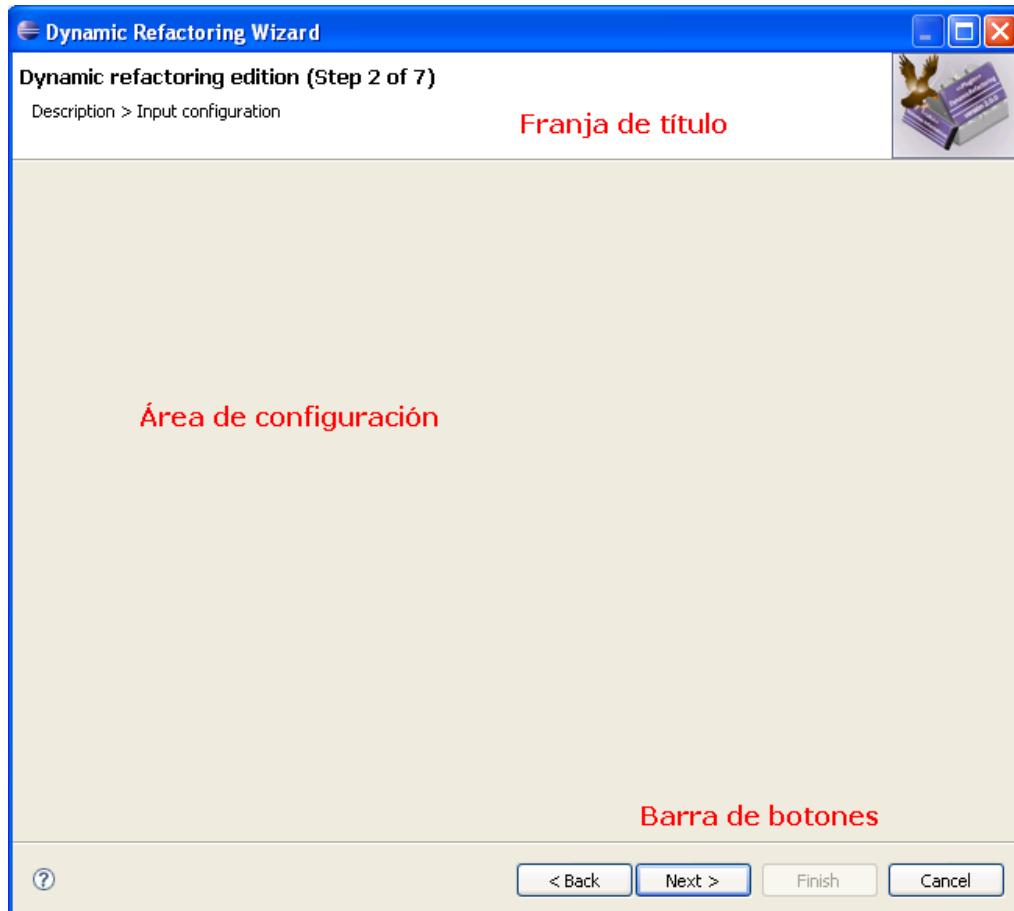


Ilustración 41: Asistente creación/edición refactorizaciones

Para ver una descripción más detallada de cada uno de los componentes que conforman la página de un asistente consultar [Fuente de la Fuente n.d.].

En el caso de que vayamos a realizar la modificación de una refactorizacion previamente a realizar el proceso mediante el asistente aparecerá una ventana para seleccionar la refactorización a editar. Esta ventana ha sufrido varios cambios respecto a la versión anterior, estos son:

- lista de refactorizaciones disponibles: se incluye icono representativo para indicar si la refactorizacion viene suministrada con el plugin o si es propia del usuario. Además aparece un botón que permite ocultar aquellas que no son del usuario.
- acciones a realizar: además de editar se añade la posibilidad de crear una nueva refactorizacion a partir de una ya existente, es decir, crear una copia que poder customizar.

A continuación se muestran de forma visual los campos sufridos:

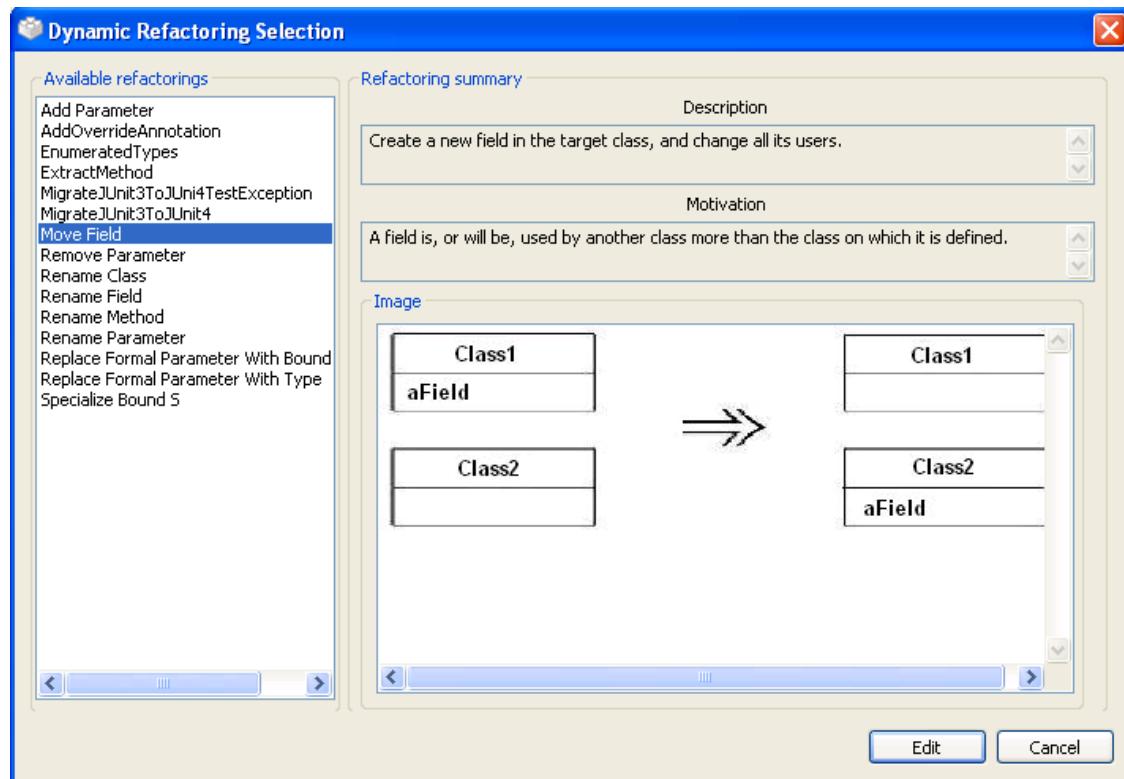


Ilustración 42: Selección refactorización antes de modificaciones

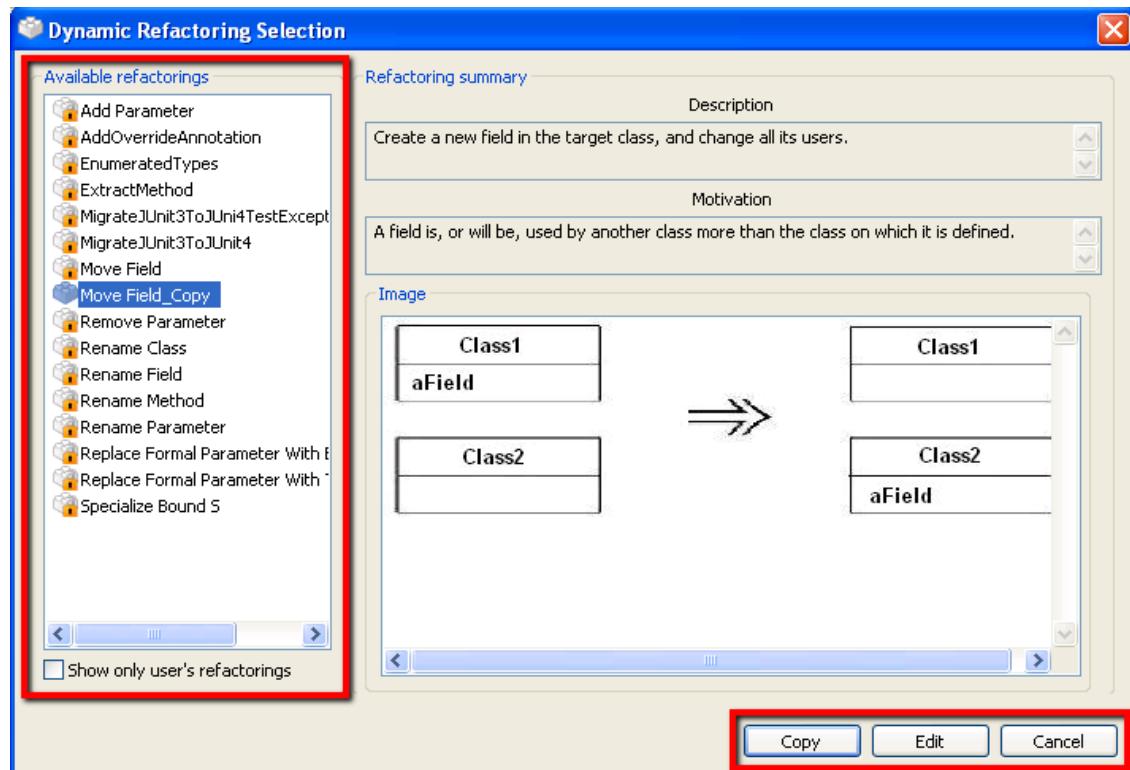


Ilustración 43: Selección refactorización después de modificaciones

Una vez ya nos encontramos en el asistente de creación y edición de refactorizaciones y a lo largo de cada una de las páginas que lo componen se van a apreciar los cambios que a continuación se detallan.

La primera página del wizard de creación y edición de refactorizaciones permite introducir la información descriptiva básica y de alto nivel de la refactorización. En esta página se han incluido, con respecto al anterior proyecto, los apartados dedicados a las palabras clave y a las categorías. En el primer apartado se indicarán las palabras con las cuales se quiere caracterizar a la refactorización, mientras que en el segundo apartado se indicará las categorías a las que va a pertencer la refactorización para cada una de las clasificaciones disponibles.

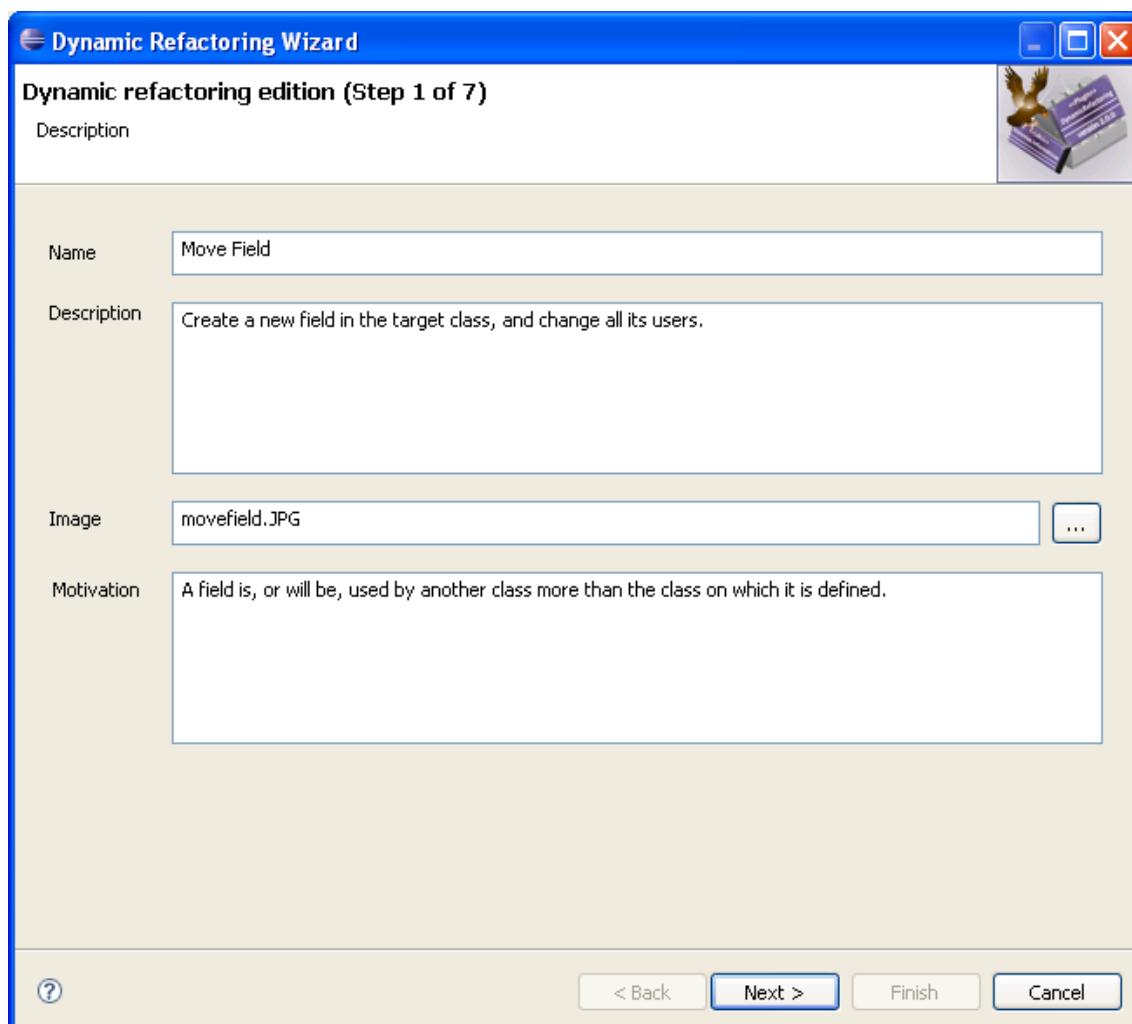


Ilustración 44: Primera página del asistente antes de modificaciones

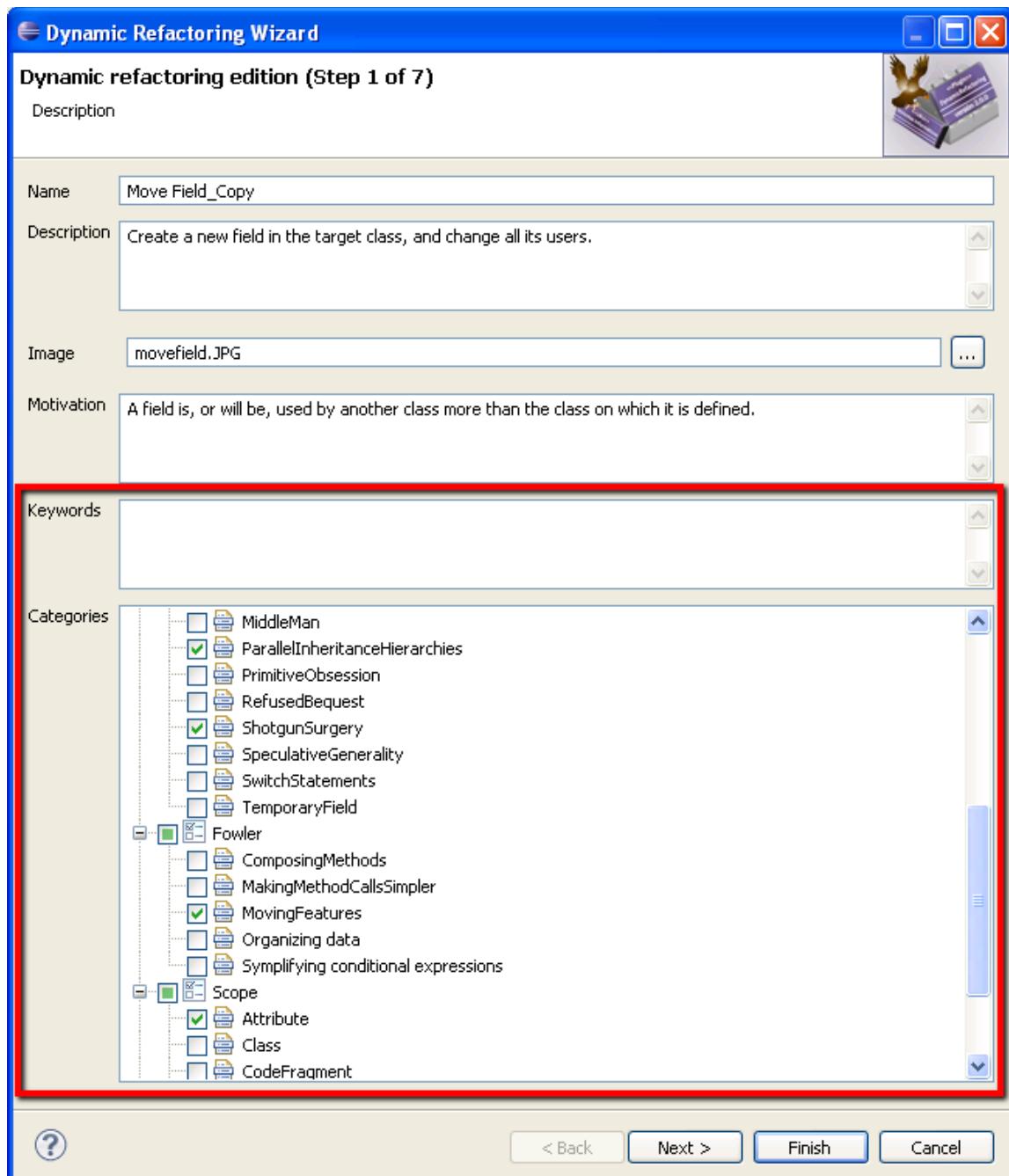


Ilustración 45: Primera página del asistente después de modificaciones

La segunda página del wizard de creación y edición de refactorizaciones esta dedicada a la definición de los tipos de entrada para la refactorización. En esta página además de modificaciones funcionales, como es el caso de cambio total del proceso de búsqueda, se ha incluido una nueva zona en la que se muestra un resumen del tipo de

entrada que el usuario seleccione el lista de disponibles. En el resumen se muestra la descripción del tipo de entrada y la relación de las refactorizaciones que la están utilizando en su definición así como las que la tienen como entrada principal.

Además, también se ha modificado la presentación del navegador `html` donde se muestra la información `javadoc` relativa a la clase mejorando su usabilidad. Para que el usuario puede leer la información más cómodamente este navegador se puede ampliar desplazándolo hacia arriba con el ratón, de esta forma el navegador consigue un tamaño mayor que permite visualizar más detalles al usuario.

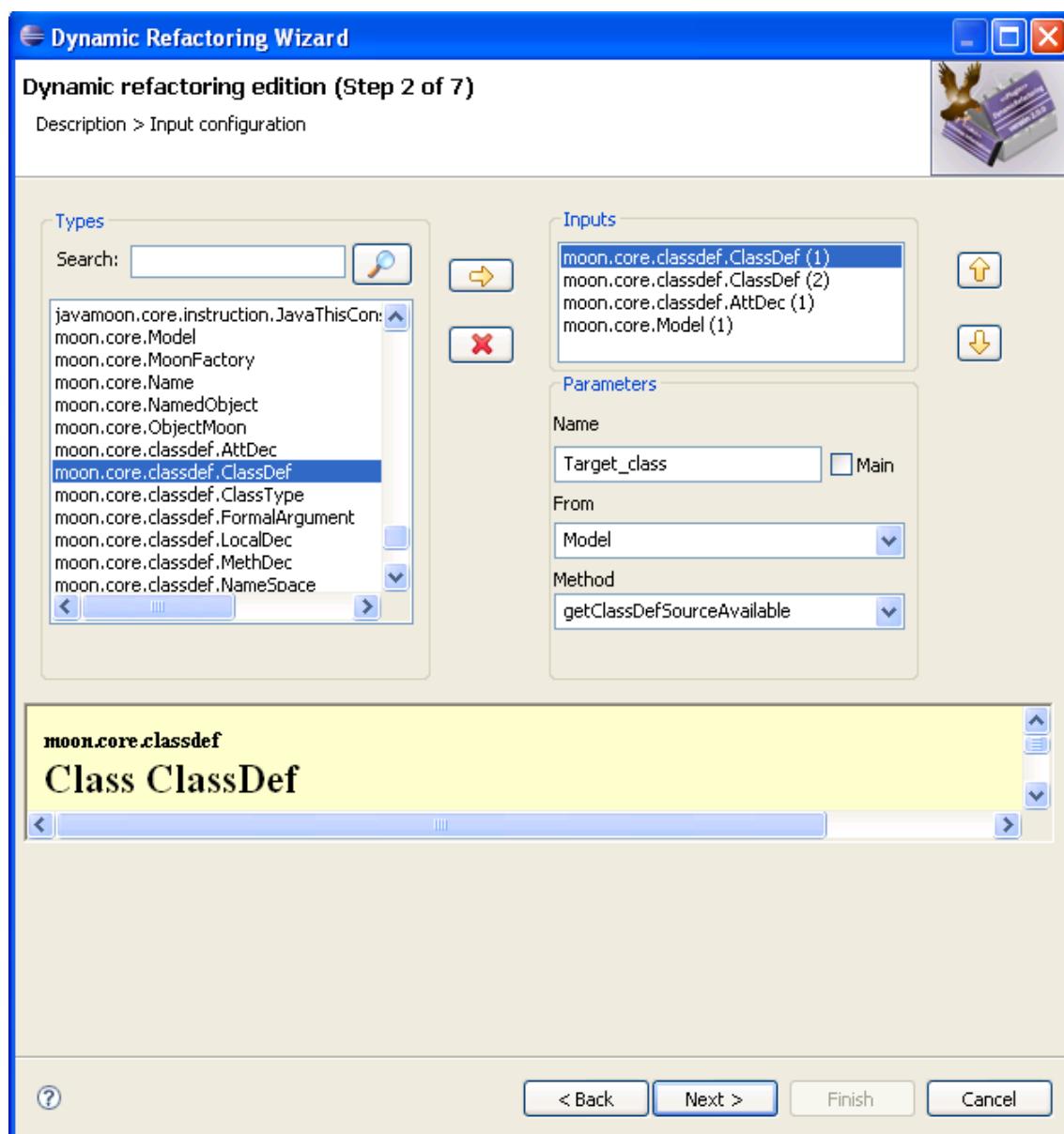


Ilustración 46: Segunda página del asistente antes de modificaciones

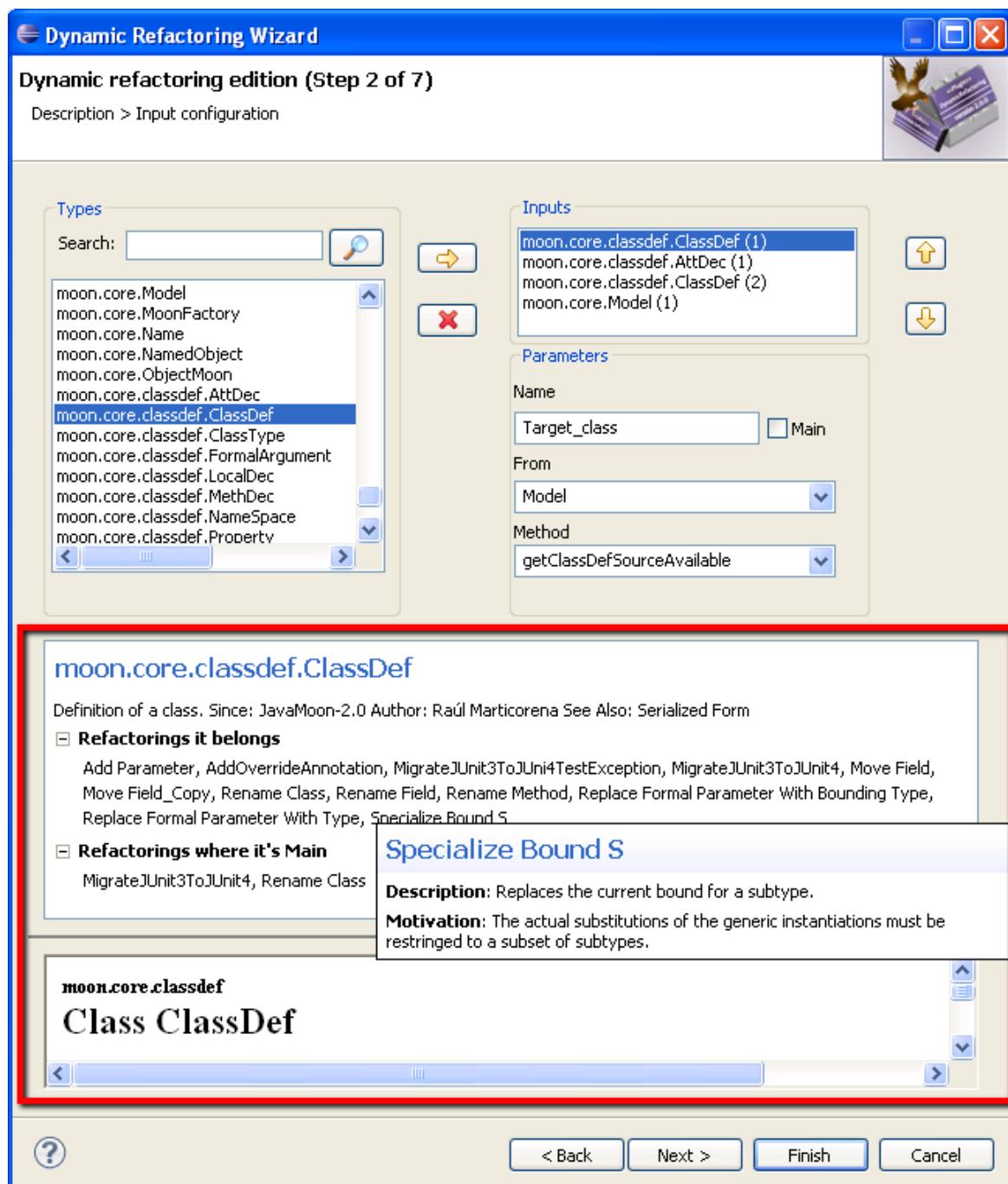


Ilustración 47: Segunda página del asistente después de modificaciones

De la misma forma, en páginas sucesivas correspondientes a la definición de los mecanismos de refactorización, es decir, la página de precondiciones, acciones y postcondiciones se ha incorporado la zona destinada al resumen del tipo de elemento que el usuario ha seleccionado en el lista de disponibles. En el resumen se muestra la descripción del tipo de elemento y la relación de las refactoriaciones que le estan utilizando en su

definición. Además, se ha modificado la presentación del navegador como anteriormente ya se ha comentado para el caso de los tipos de entrada.

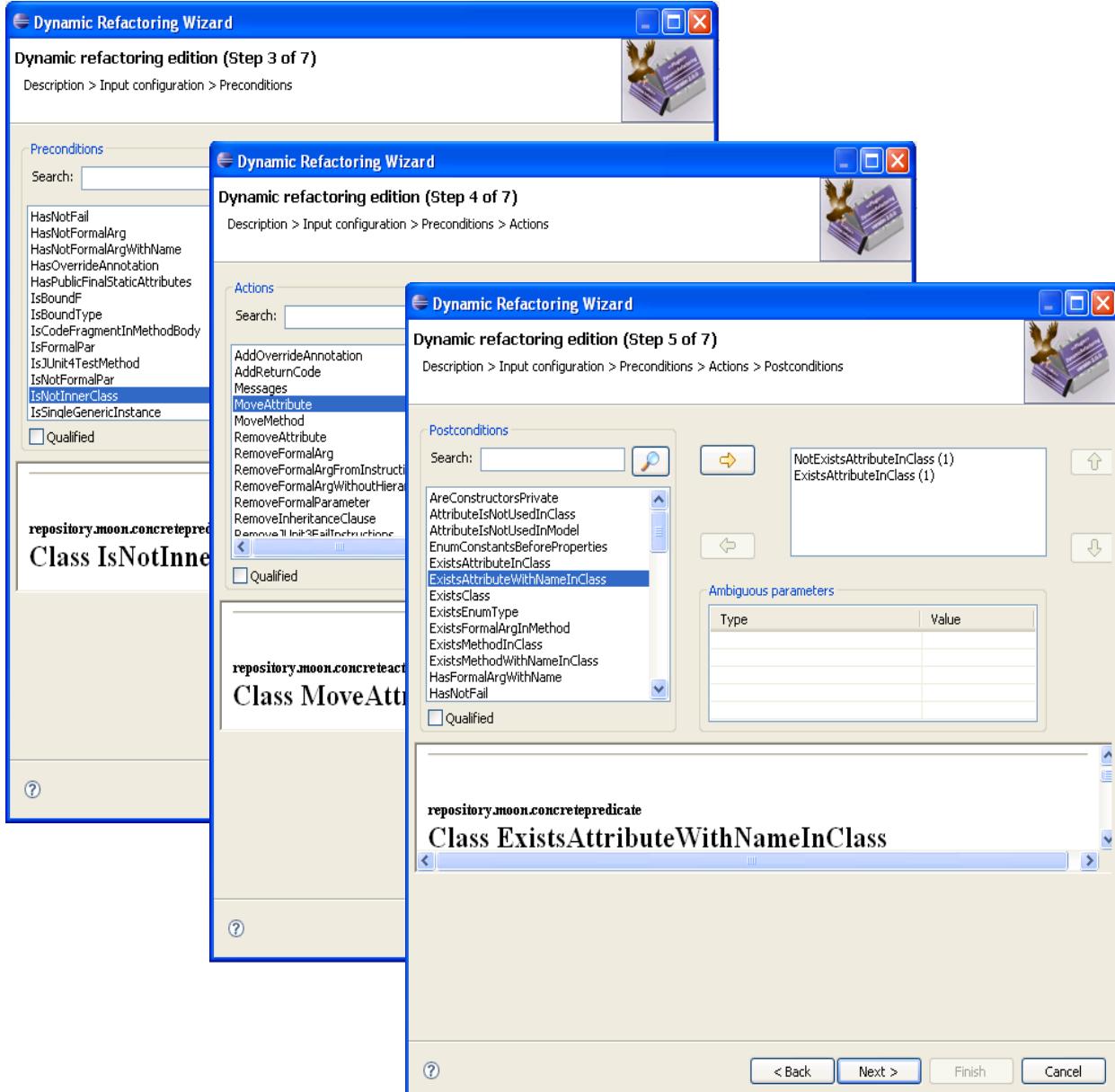


Ilustración 48: Páginas 3,4 y 5 antes de modificaciones

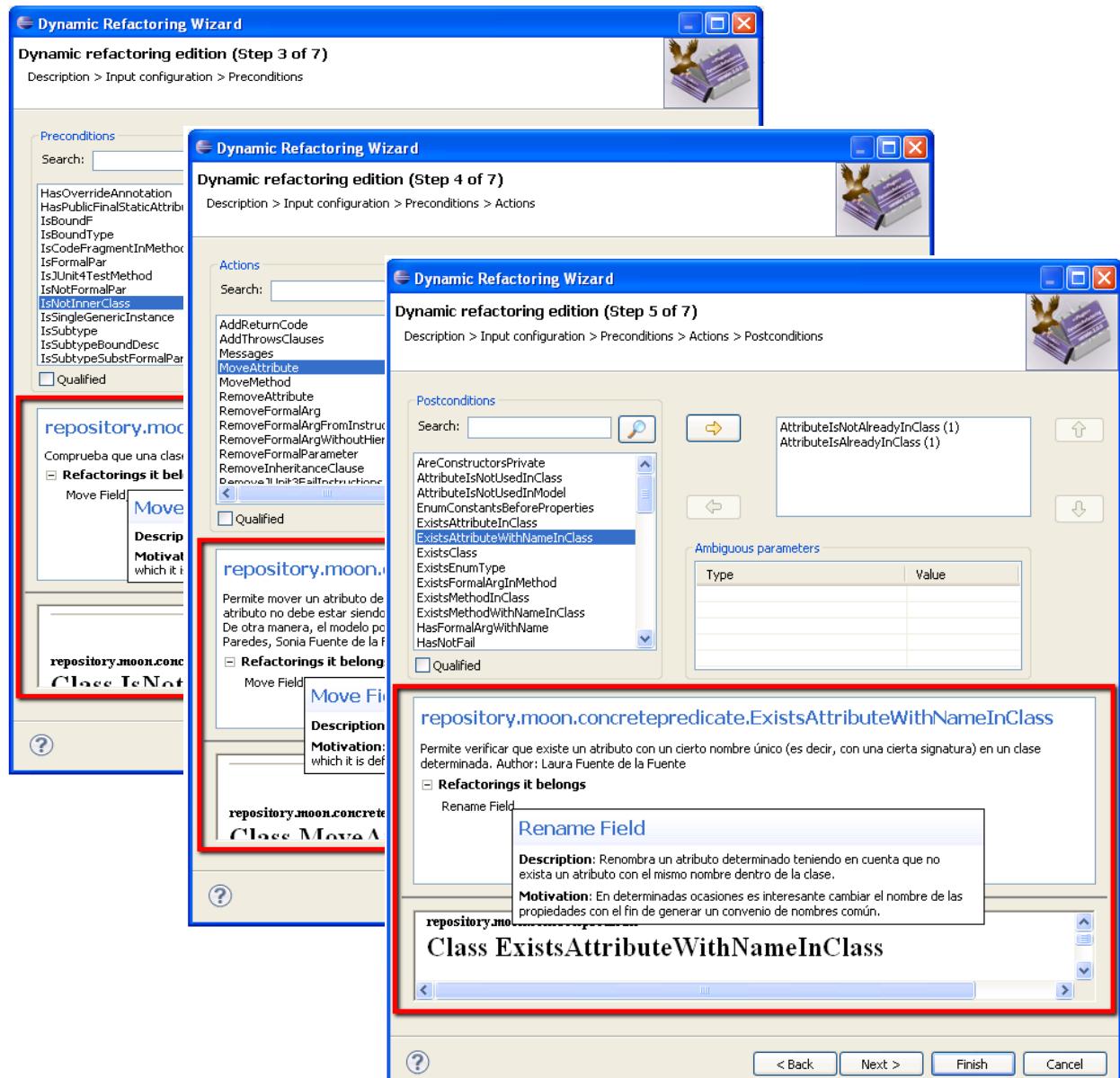


Ilustración 49: Páginas 3, 4 y 5 después de modificaciones

Por último, la última página del wizard de creación y edición de refactorizaciones es la encargada de mostrar al usuario la información que ha sido recogida a lo largo de todo el proceso y que sirve como comprobación previa antes de hacer efectiva la creación o edición de la refactorización que se este tratando.

Esta página también se ha visto modificada para incluir la información relativa a las palabras clave y las categorías a las que pertenece la refactorización, la cual ha sido introducida en la primera página del asistente.

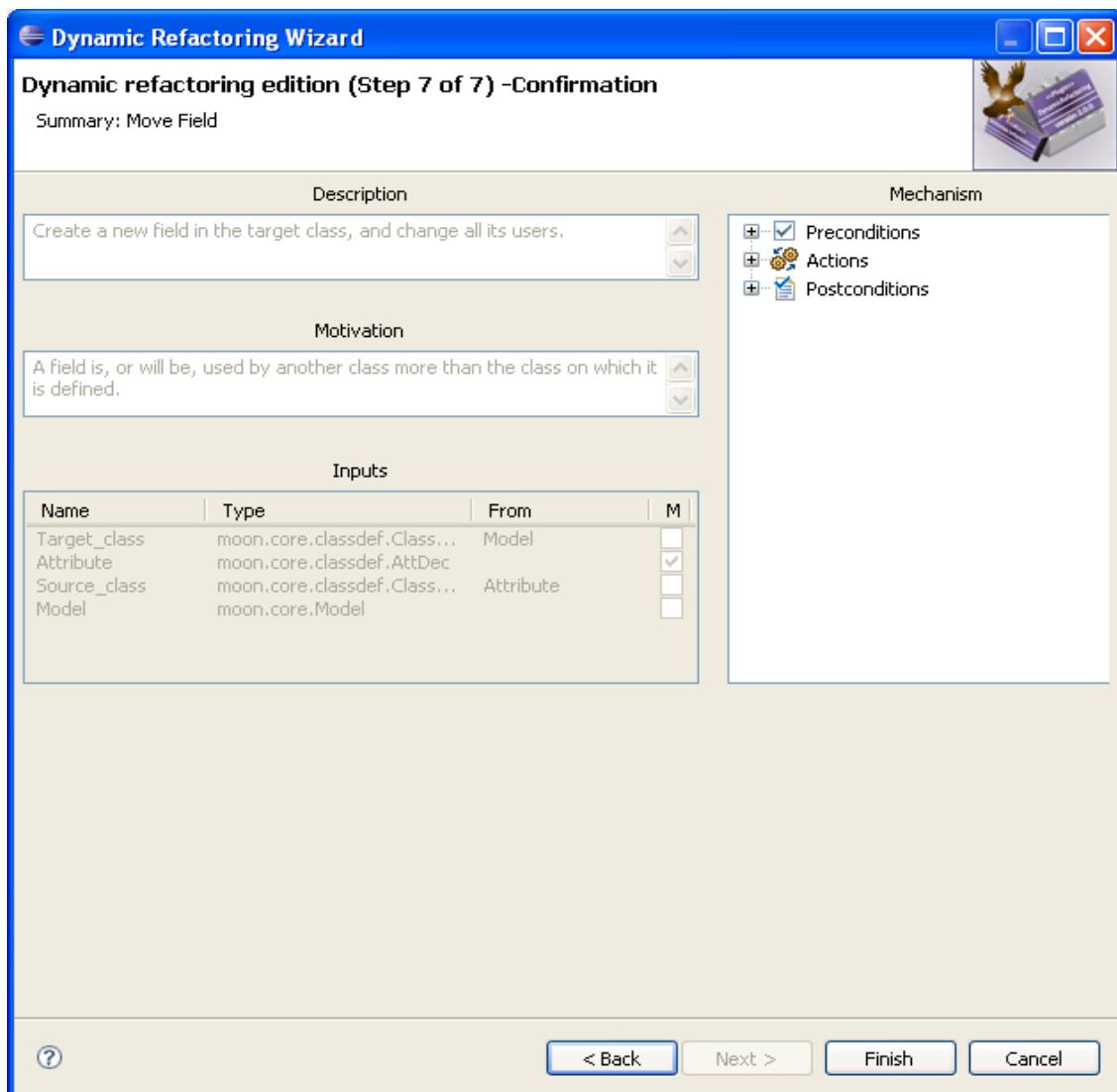


Ilustración 50: Séptima página del asistente antes de modificaciones

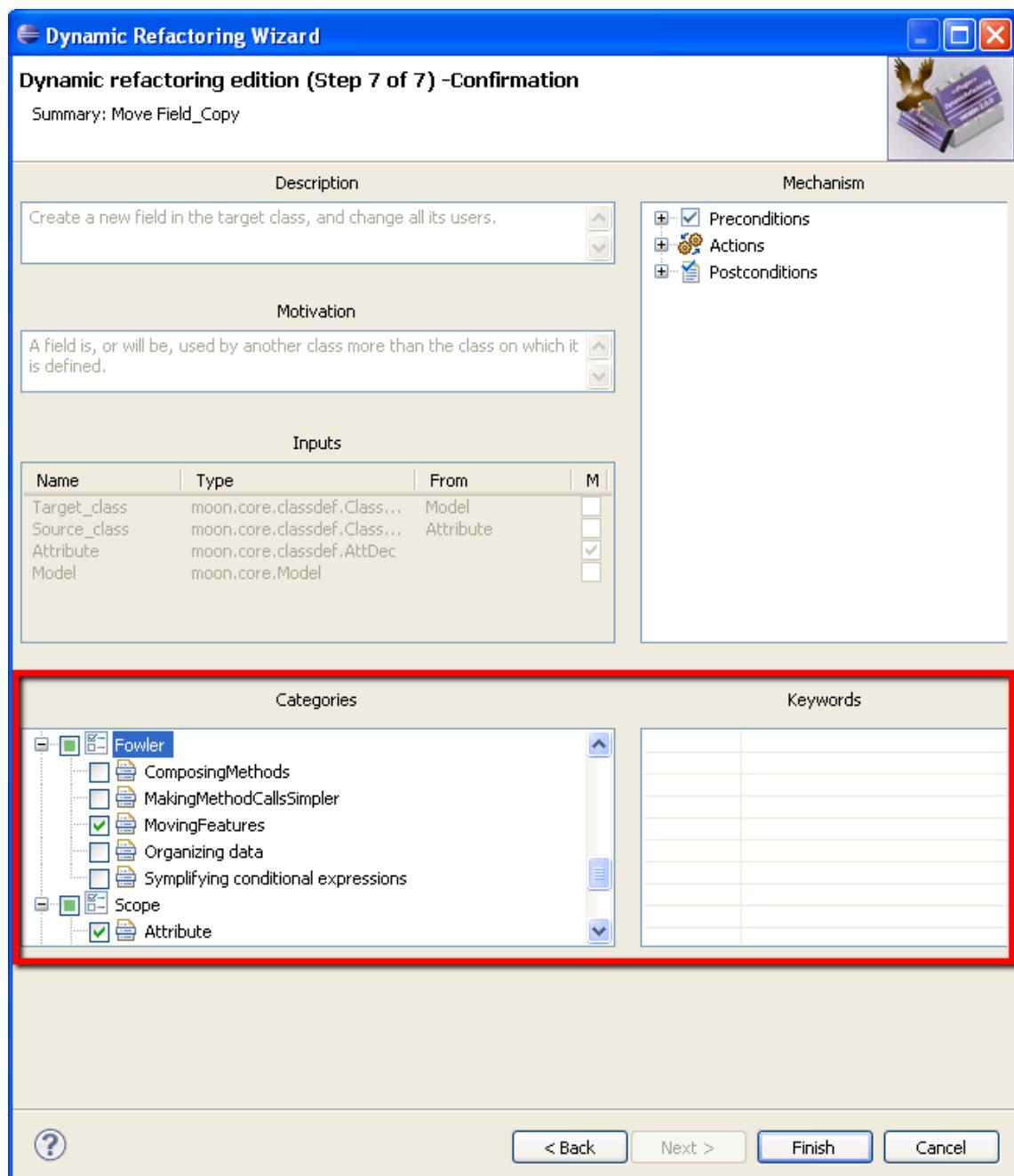


Ilustración 51: Séptima página del asistente después de modificaciones

4.2. Interfaz para la exportación de refactoriaciones

Otra de las funcionalidades con las que cuenta el plugin de refactorizaciones es la exportación e importación de refactorizaciones. En este caso, nos centraremos en la interfaz que permite realizar exportaciones de las refactorizaciones disponibles ya que es en la que

se ha incorporado cambios respecto a la versión del plugin del proyecto anterior.

La interfaz que hace posible la exportación de refactorizaciones muestra la relación de refactorizaciones que se encuentran disponibles. De estas, el usuario seleccionará las que desea exportar e indicará la ruta en la que lo quiere realizar.

Las modificaciones que ha sufrido afectan a la lista de refactorizaciones disponibles, en ella se incluye el icono representativo para indicar si la refactorización viene suministrada con el plugin o si es propia del usuario. Además, aparece un botón que el usuario puede marcar si desea ocultar aquellas que son del plugin, con la finalidad de que únicamente aparezcan las propias del usuario y con ello mejorar su visualización facilitándole el trabajo.

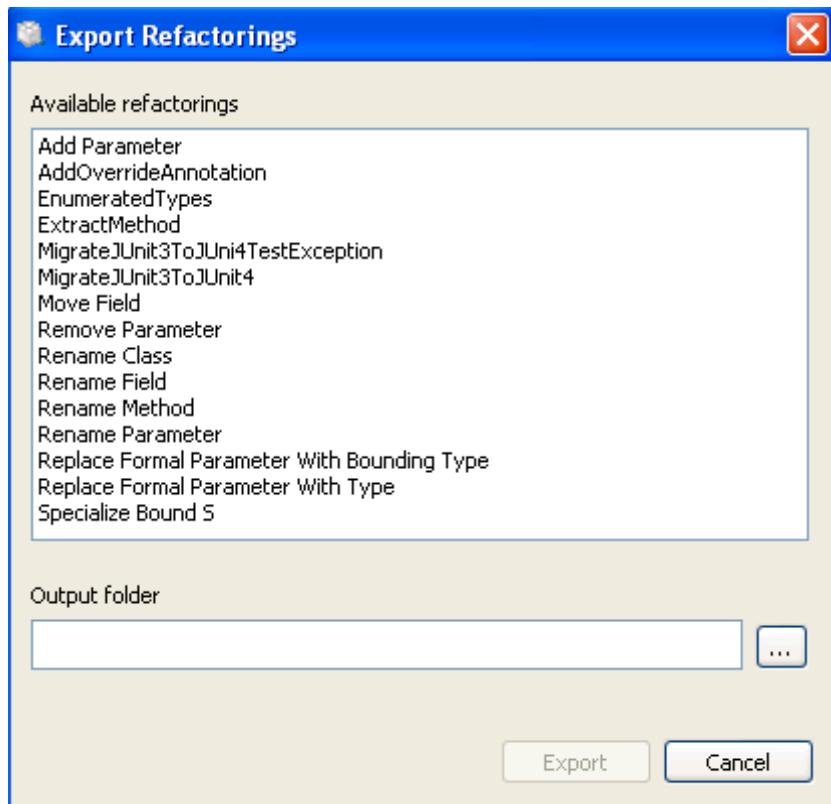


Ilustración 52: Interfaz exportación refactorizaciones antes de modificaciones

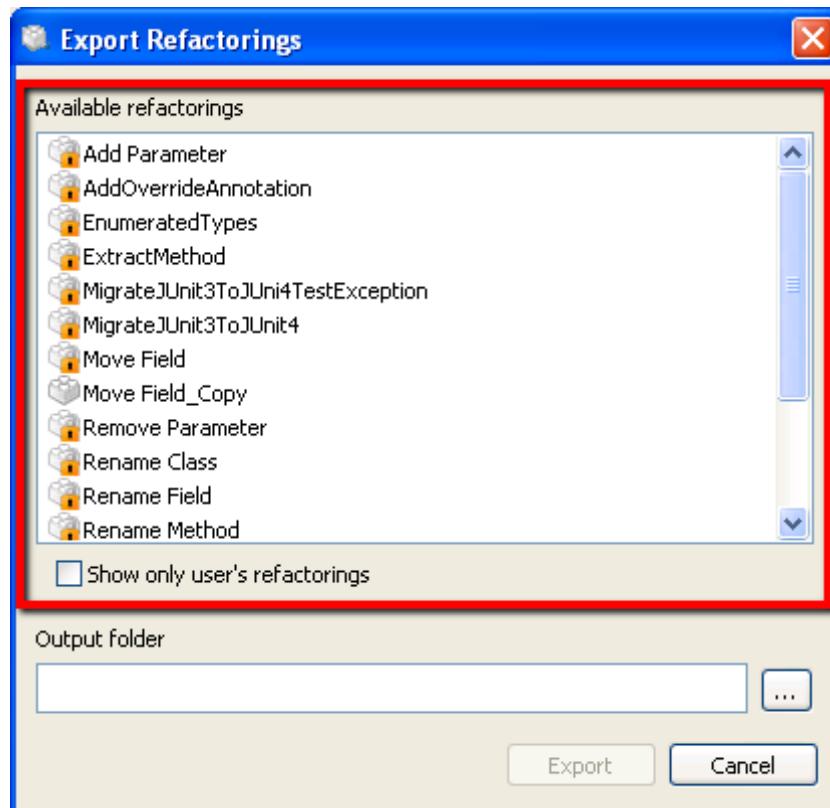


Ilustración 53: Interfaz exportación refactorizaciones después de modificaciones

4.3. Vista Available Refactorings

Una característica que enriquece favorablemente a una interfaz gráfica es la posibilidad de guiar dinámicamente al usuario hacia la tarea final. Esta es la causa por la que en la versión del plugin del proyecto anterior se incluyó una nueva vista que actualiza su contenido cada vez que se selecciona un elemento que sirve como entrada principal de una refactorización. Esta vista se encarga de mostrar las distintas refactorizaciones que pueden ser ejecutadas tomando dicho elemento como entrada principal, esto evita buscar el menú adecuado para poder ejecutar una refactorización, facilitando con ello la interacción con la herramienta.

Nuevamente, como hemos visto en las modificaciones de la interfaz del plugin que se han explicado hasta ahora, también en esta vista se ha incluido la posibilidad de identificar a las refactorizaciones como propias del usuario o bien como suministradas por el plugin, mediante la visualización del ícono correspondiente.

Además, se ha creado en la vista una barra de herramientas en la que incluir las distintas acciones que se pueden llevar a cabo sobre esta. En esa barra de herramientas se han incluido dos acciones; una que permiten ocultar o mostrar las refactorizaciones suministradas con el plugin y otra que hace lo propio con las refactorizaciones creadas por el usuario, todo ello mediante el deshabilitado o habilitado del botón correspondiente. Con esto se consigue mejorar la visualización de las refactorizaciones disponibles facilitándole el trabajo al usuario, sobre todo cuando se da el caso de la existencia de un número elevado de estas.

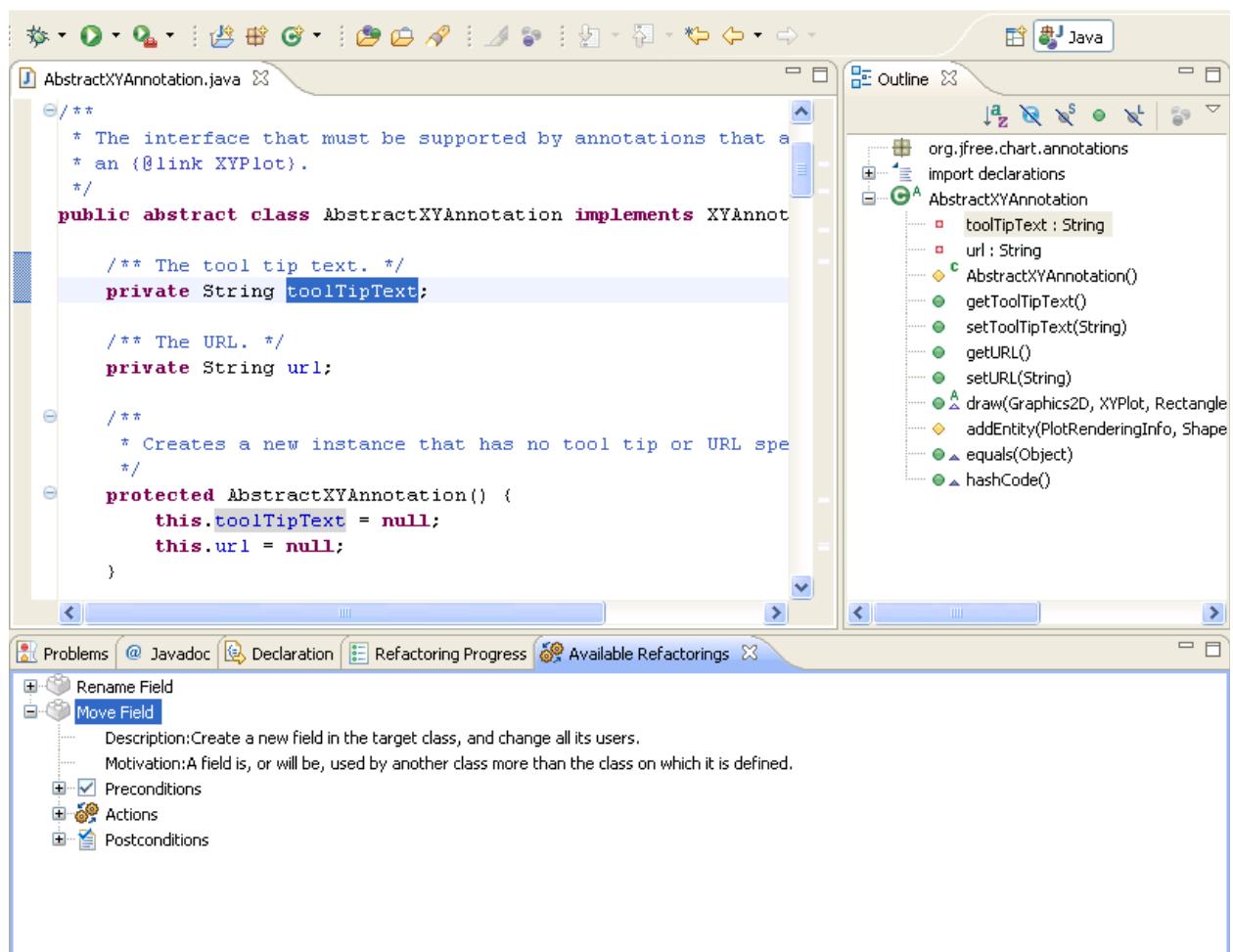


Ilustración 54: Vista Available Refactorings antes de modificaciones

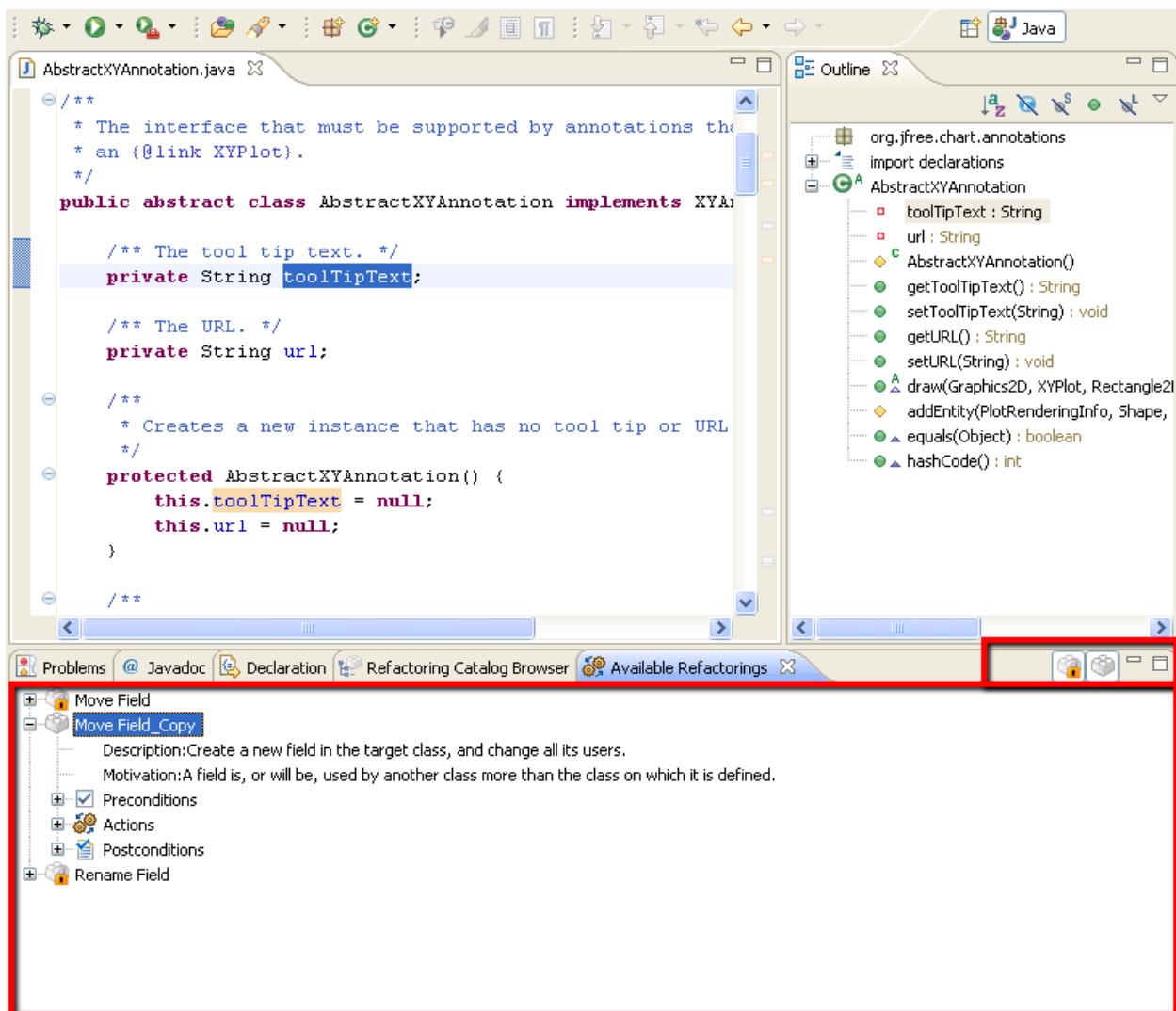


Ilustración 55: Vista Available Refactorings después de modificaciones

4.4. Vista Refactoring Catalog Browser

Debido al gran número de refactorizaciones de las que dispone el plugin, se hacía difícil para el usuario su comprensión, saber cual era adecuado utilizar en cada momento y cuales estaban relaciones entre sí. Por esta razón y porque no se disponía de ningún mecanismo para dar solución a este problema se ha incluido en el plugin una nueva vista denominada *Refactoring Catalog Browser*.

Esta vista recoge toda la información asociada a cada una de las refactorizaciones disponibles presentandola de una forma organizada y dando la posibilidad al usuario de clasificarlas y realizar búsquedas sobre las mismas, favoreciendo con ello que el usuario

adquiera el conocimiento de estas de una forma mucho más amigable.

La ilustración que a continuación se presenta muestra la nueva vista:

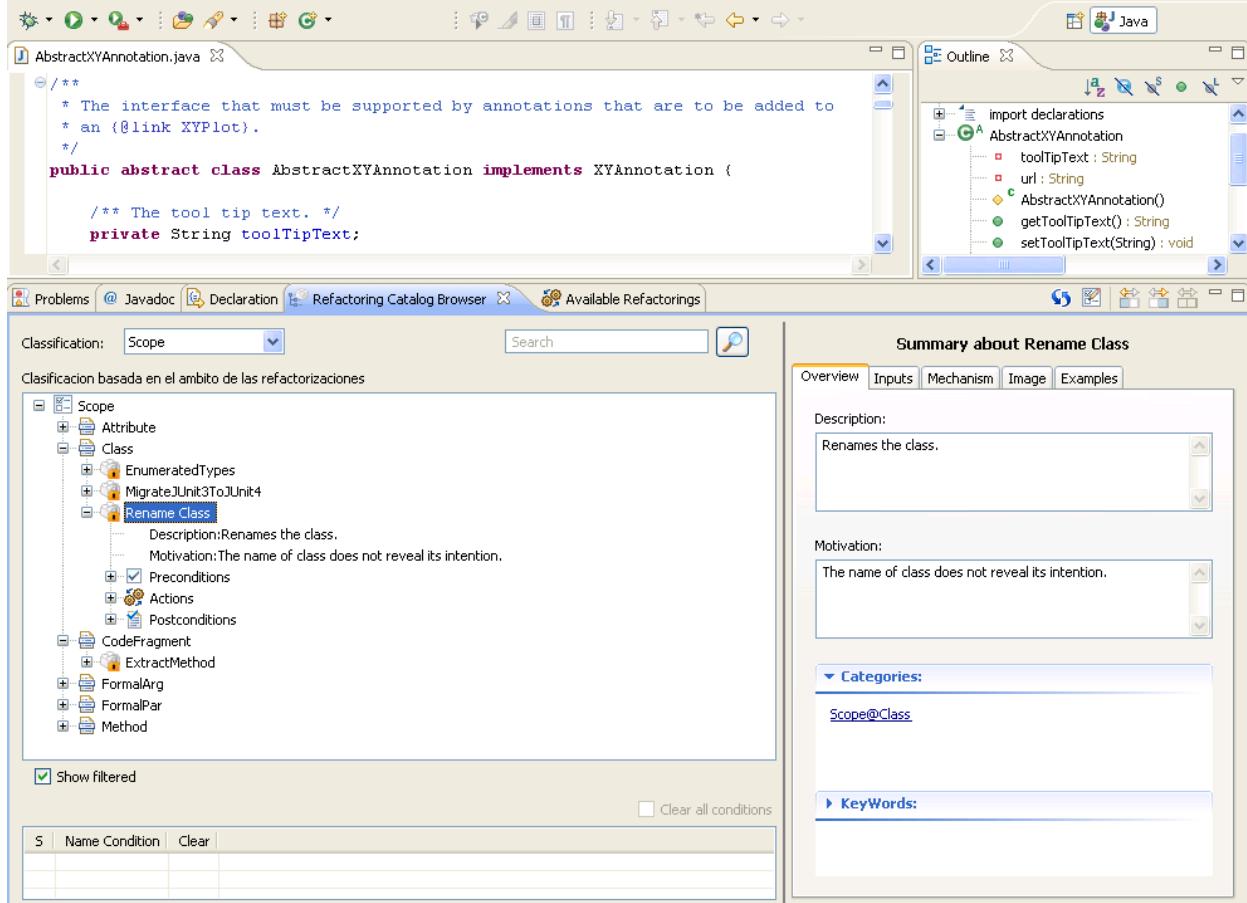


Ilustración 56: Vista Refactoring Catalog Browser

4.5. Editor de Clasificaciones

Como hemos comentado anteriormente en la nueva vista que muestra el catálogo de refactorizaciones se puede realizar la clasificación de las mismas. Para ello, previamente en la definición de las refactorizaciones se ha tenido que definir las categorías a las que pertenece la refactorización para cada una de las clasificaciones disponibles.

Con el objetivo de dotar a la aplicación de funcionalidad añadida y dar flexibilidad al usuario se ha decidido permitir crear clasificaciones propias del usuario. Es por tanto que, para hacer esto posible se ha creado un editor en el que el usuario puede crear sus propias clasificaciones y definir las categorías que tendrá disponible cada una de ellas.

A continuación se muestra el editor de clasificaciones:

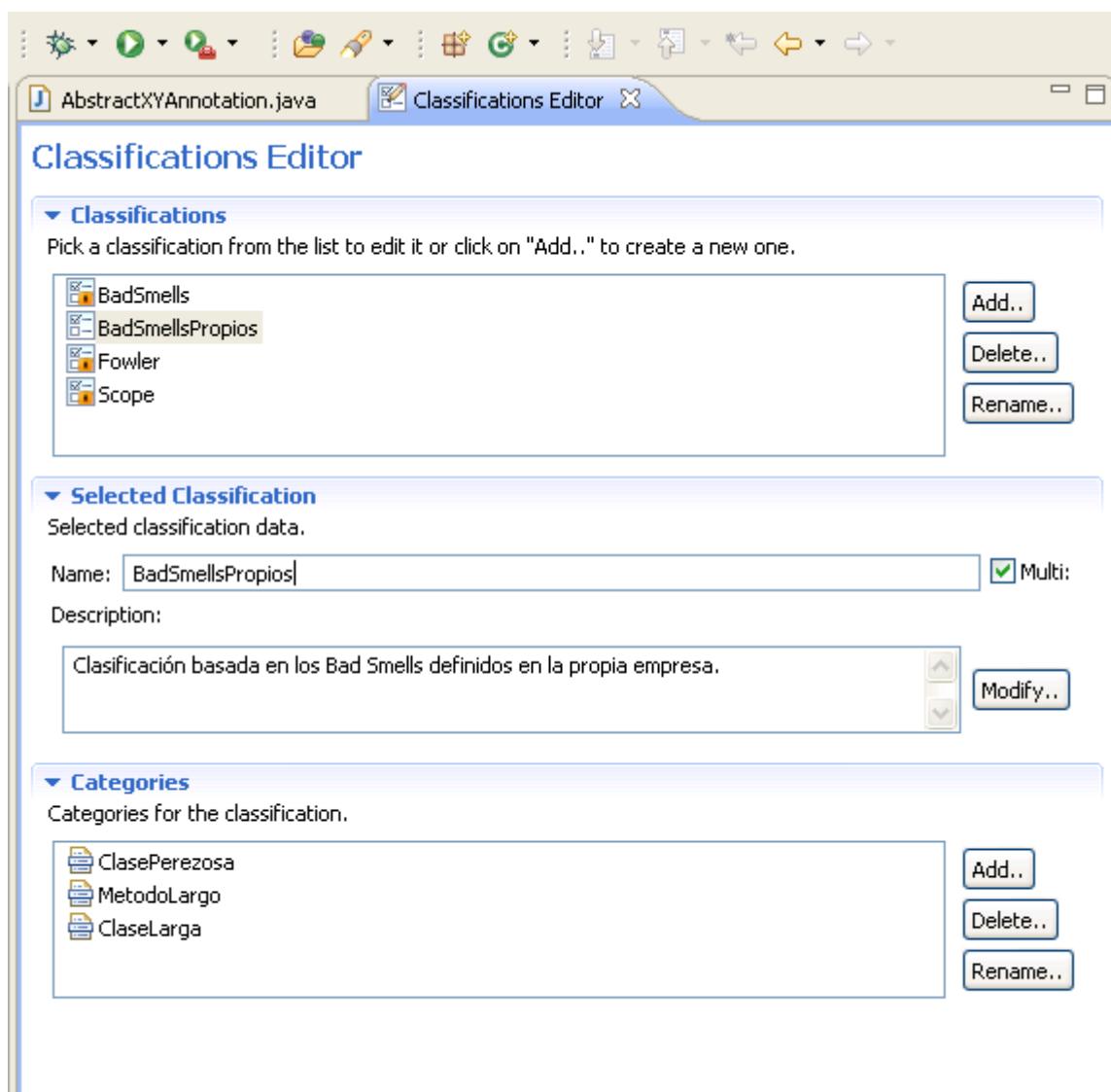


Ilustración 57: Editor de clasificaciones

5. DISEÑO PROCEDIMENTAL

Este apartado está dedicado a presentar el diseño procedimental del sistema mediante los diagramas de secuencia y colaboración, que describen la interacción entre los objetos ordenados en secuencia temporal. Estos diagramas aportan la vista dinámica del apartado de diseño. Los diagramas se basan en los casos de uso definidos en el anexo 2 y únicamente serán incluidos los diagramas de secuencia ya que estos y los diagramas de colaboración pueden ser utilizados indistintamente para expresar las mismas interacciones entre objetos. Es por ello que, hemos recurrido al que se ha considerado más adecuado para reflejar cada interacción determinada.

5.1. Diagramas de secuencia

Para este anexo se va a utilizar diagramas de secuencia por considerar que se ajustan mejor a la presentación que se desea mostrar.

5.1.1. RF 1: Visualizar refactorizaciones según clasificación

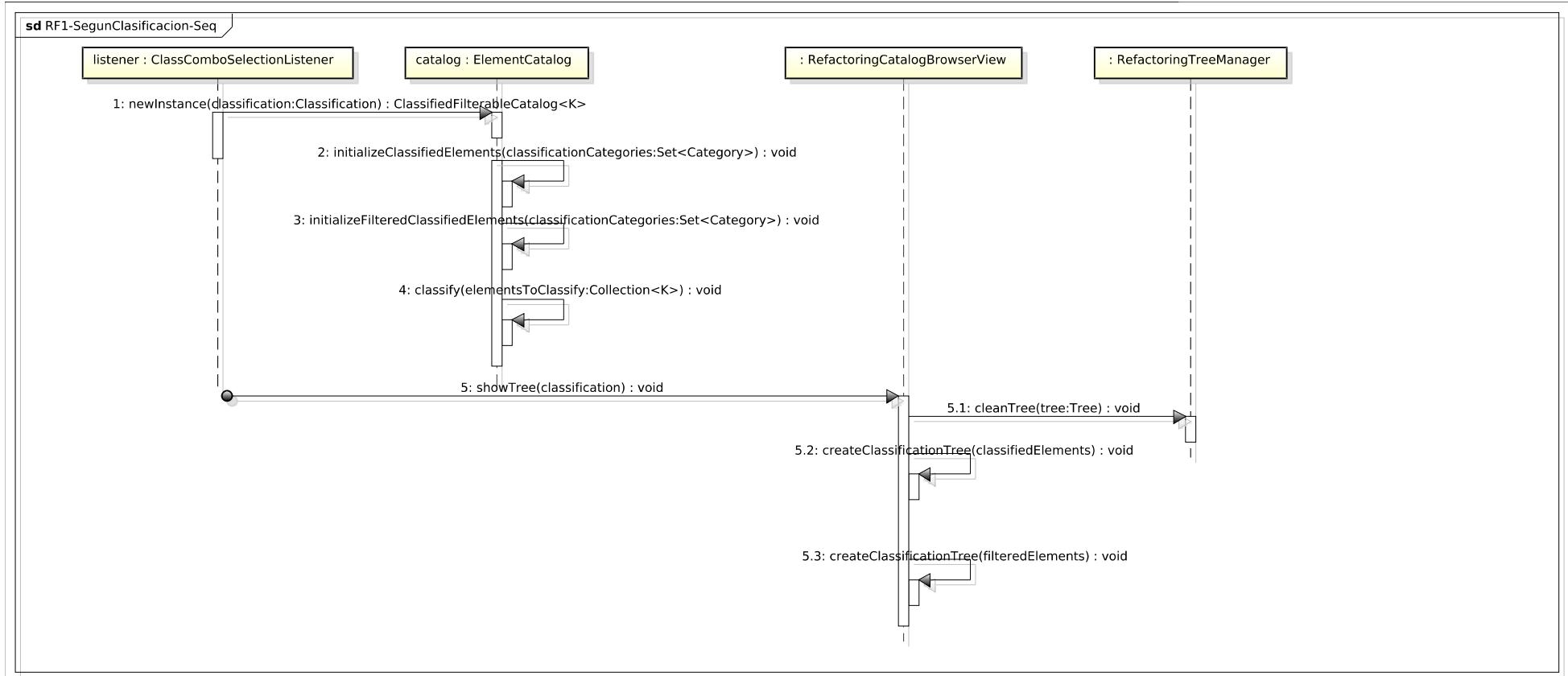


Ilustración 58: D. de secuencia de RF 1: Visualizar ref. según clasificación

En este diagrama se muestra el proceso necesario para visualizar las refactorizaciones categorizadas según una clasificación. Para llevar a cabo esta función, en primer lugar el `ComboListener` recibe un evento de que el usuario o cambiado la clasificación seleccionada. Tras esto el `listener` actualiza el catálogo para que ahora este categorizado según la nueva clasificación. Finalmente se llama al `RefactoringCatalogBrowser` para que este se encargue de actualizar el árbol de la interfaz gráfica.

5.1.2. RF 2: Refrescar visualización de refactorizaciones

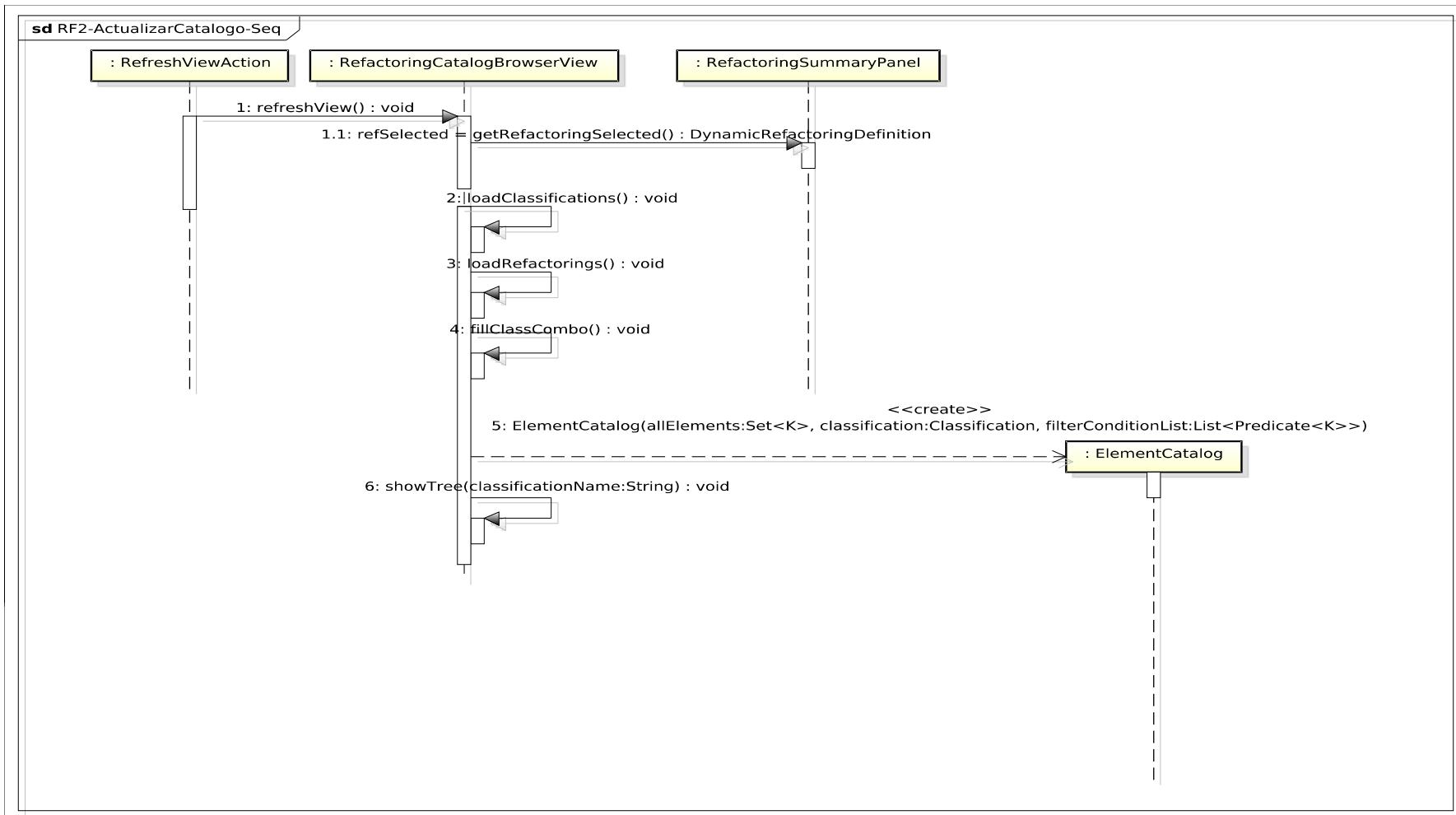


Ilustración 59: D. de secuencia de RF 2: Refrescar visualización de refactorizaciones

En este caso la secuencia es lanzada por la clase RefreshActionView. Esta clase advierte a la vista del catálogo de que alguien ha solicitado la actualización de la vista. Como respuesta ésta carga todas las refactorizaciones y las clasificaciones del catálogo y se dedica a ir refrescando todos los elementos de la interfaz con estos nuevos datos.

5.1.3. RF 3: Añadir filtro de refactorizaciones

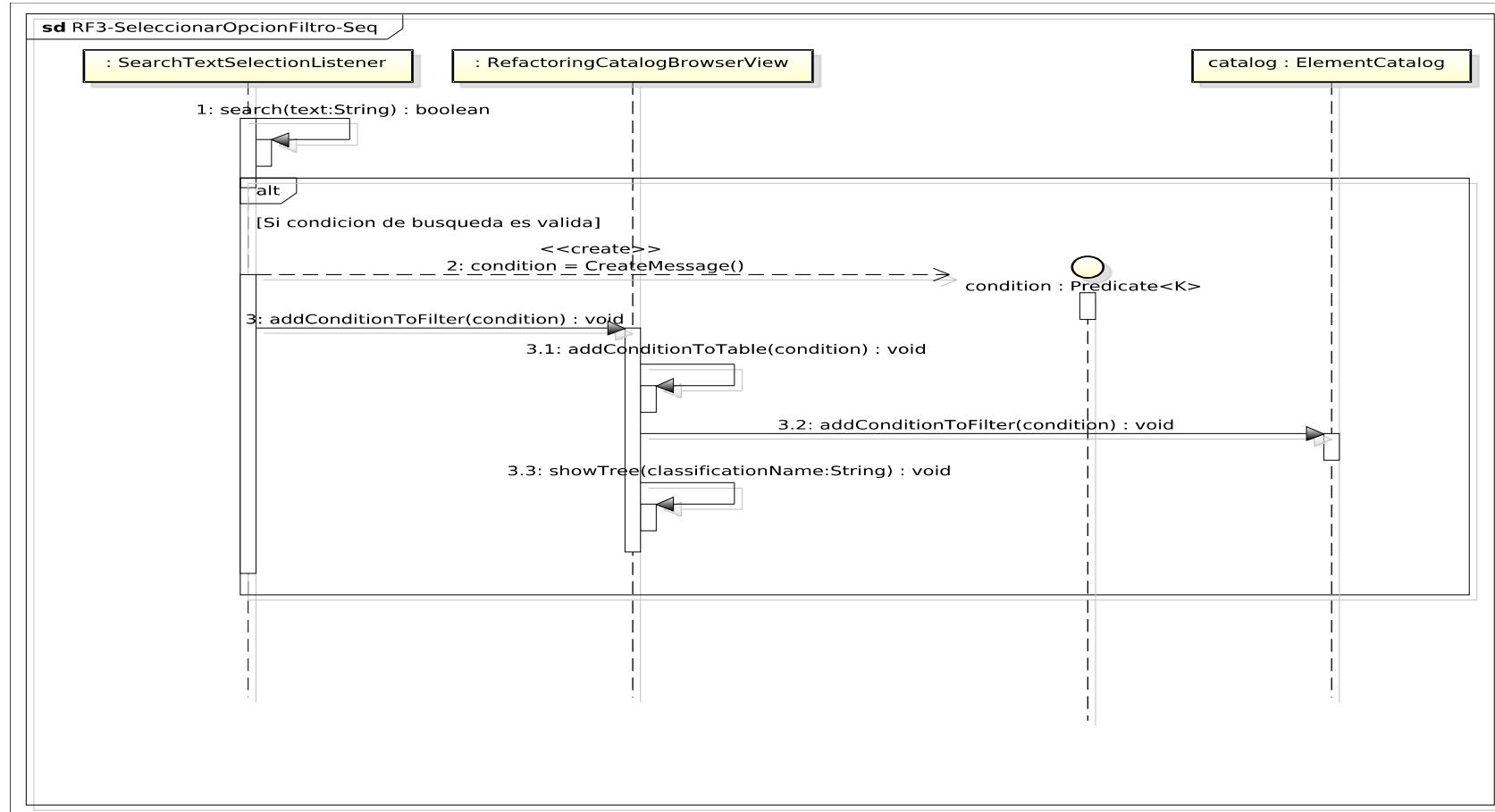


Ilustración 60: D. de secuencia de RF 3: Añadir filtro de refactorizaciones

Cuando el usuario decide añadir un filtro salta un evento que controla `SearchTextSelectionListener`. Este llama a su método `search()` que se encarga de comprobar si se puede crear una condición a partir del texto de filtro introducido por el usuario. Si es así, este método crea un objeto con dicha condición. Esa condición se agrega a la lista de filtros en la interfaz y también al catálogo de refactorizaciones filtrable. También se actualiza el árbol de refactorizaciones con los cambios en el catálogo.

5.1.4. RF 4: Seleccionar opción aplicar filtro

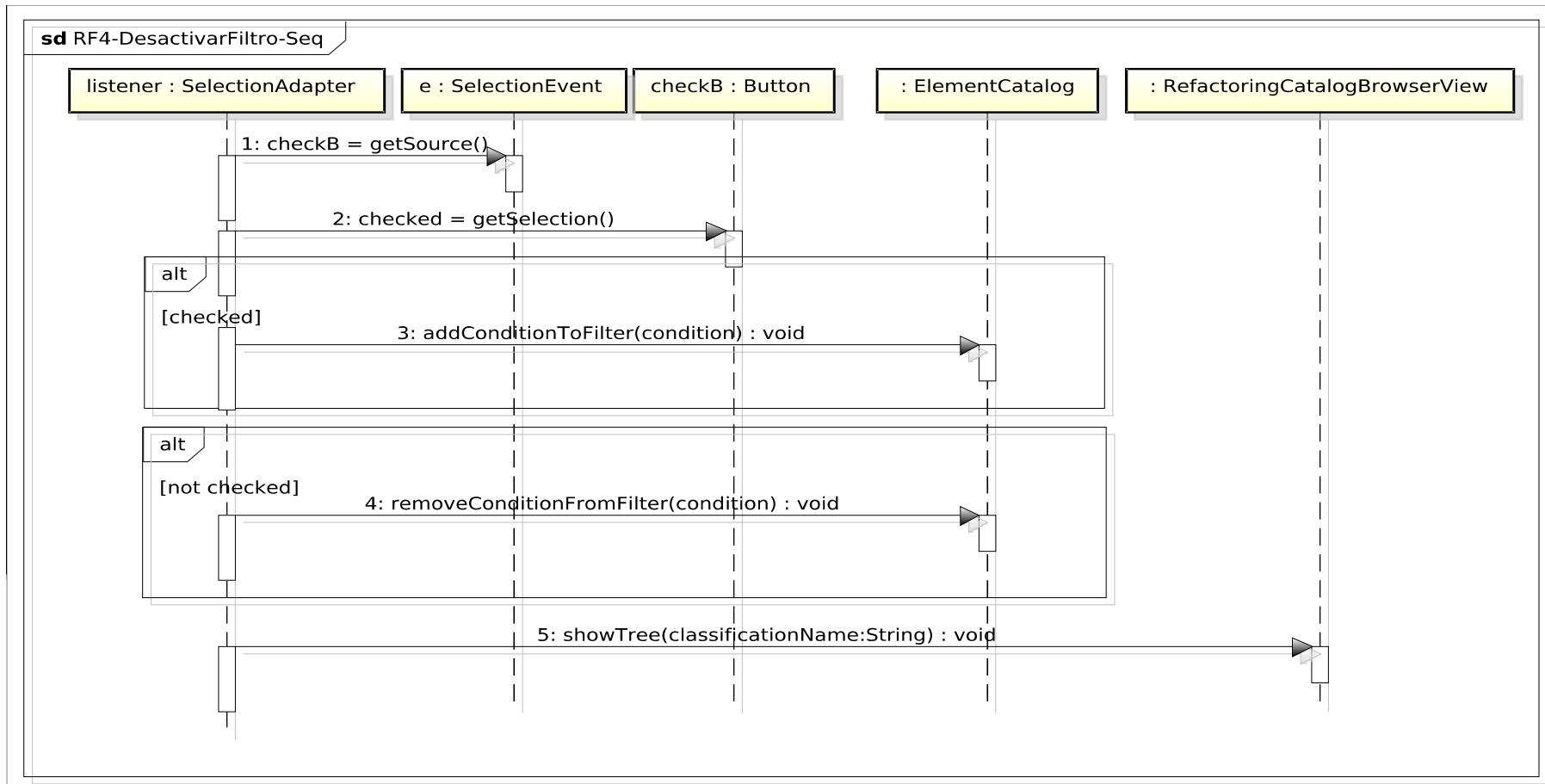


Ilustración 61: D. de secuencia de RF 4: Seleccionar opción aplicar filtro

El usuario hace saltar un evento de ratón que es capturado por un observador de tipo SelectionAdapter. El observador obtiene del evento lanzado el botón (en este caso de tipo checkbox) que lo lanzó y comprueba si dicho checkbox está marcado o no. Si esta marcada la condición correspondiente a ese botón se debe hacer efectiva, si no se debe desactivar. Finalmente se llama a la vista para que actualice el árbol de refactorizaciones.

5.1.5. RF 5: Eliminar filtro de refactorizaciones

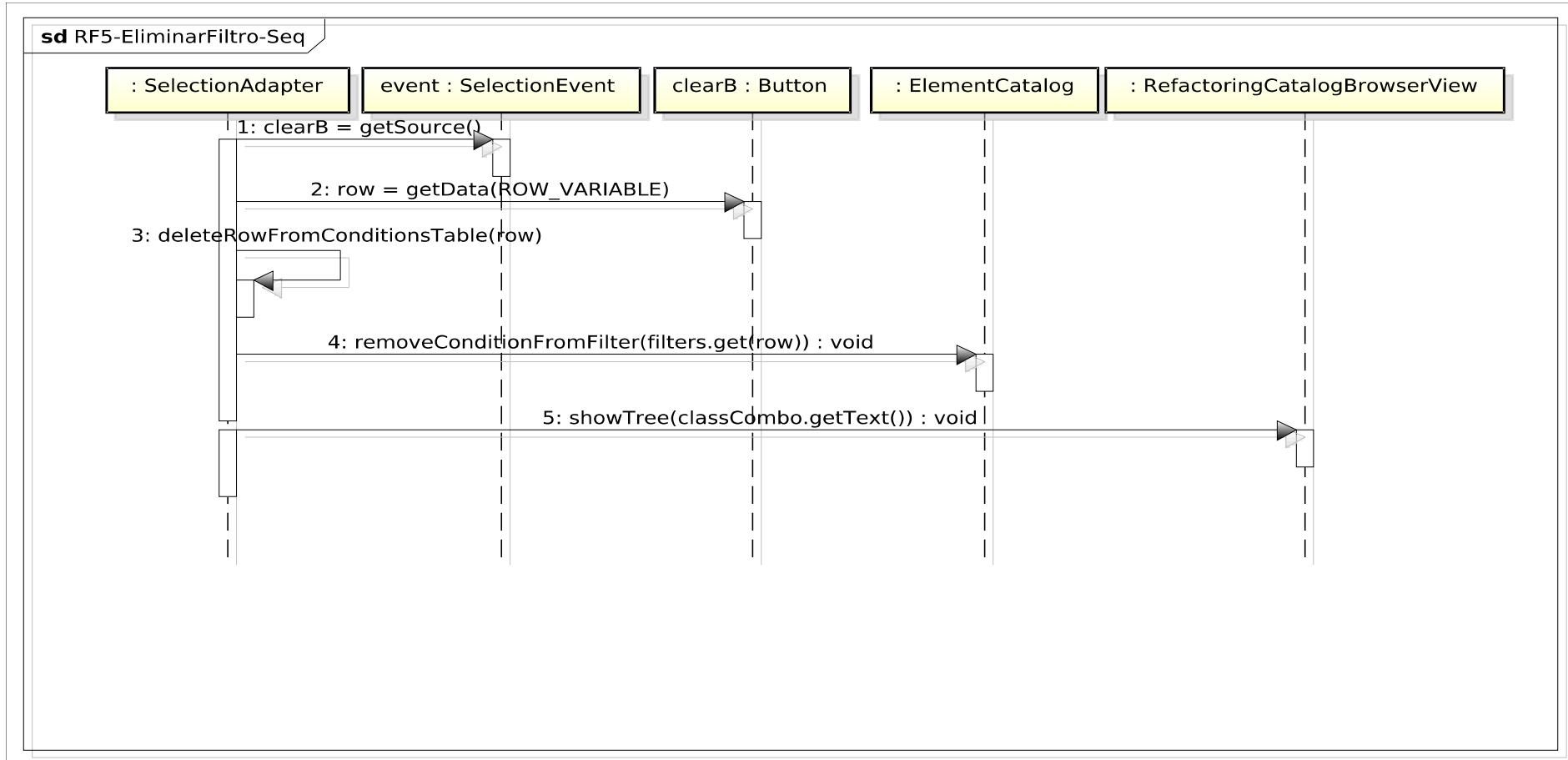


Ilustración 62: D. de secuencia de RF 5: Eliminar filtro de refactorizaciones

El evento lanzado por el usuario provoca una relación similar a la del caso de uso RF4. El observador del evento obtiene el botón origen del evento. A partir de él obtiene el número de fila de la condición que se va a eliminar. Esto le permite en primer lugar eliminar la fila de la tabla de condiciones, para luego eliminar la condición asociada de entre los filtros del catálogo y finalmente actualizar el árbol de refactorizaciones con estos cambios.

5.1.6. RF 6: Eliminar todos los filtros de refactorizaciones

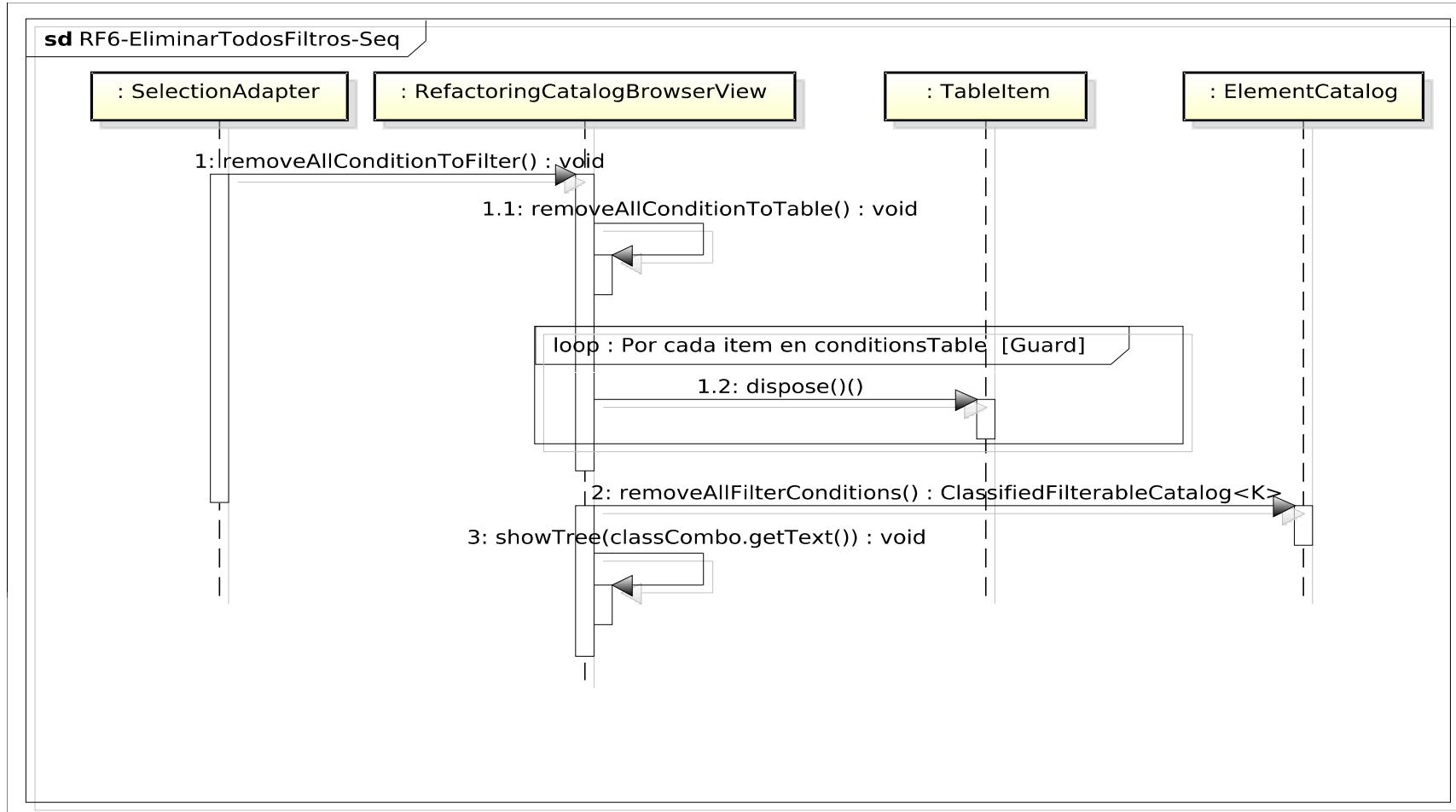


Ilustración 63: D. de secuencia de RF 6: Eliminar todos los filtros de refactorizaciones

Esta vez la respuesta al evento no necesita saber que elemento de la interfaz lo lanzó porque sólo puede ser uno. Por tanto, inmediatamente tras recibir el evento se inicia la cadena de respuesta que consiste en delegar en la vista del catálogo la eliminación de todas las filas de la tabla de condiciones, todas las condiciones del catálogo y tras ello la actualización del árbol.

5.1.7. RF 7: Seleccionar opción ver refactorizaciones filtradas

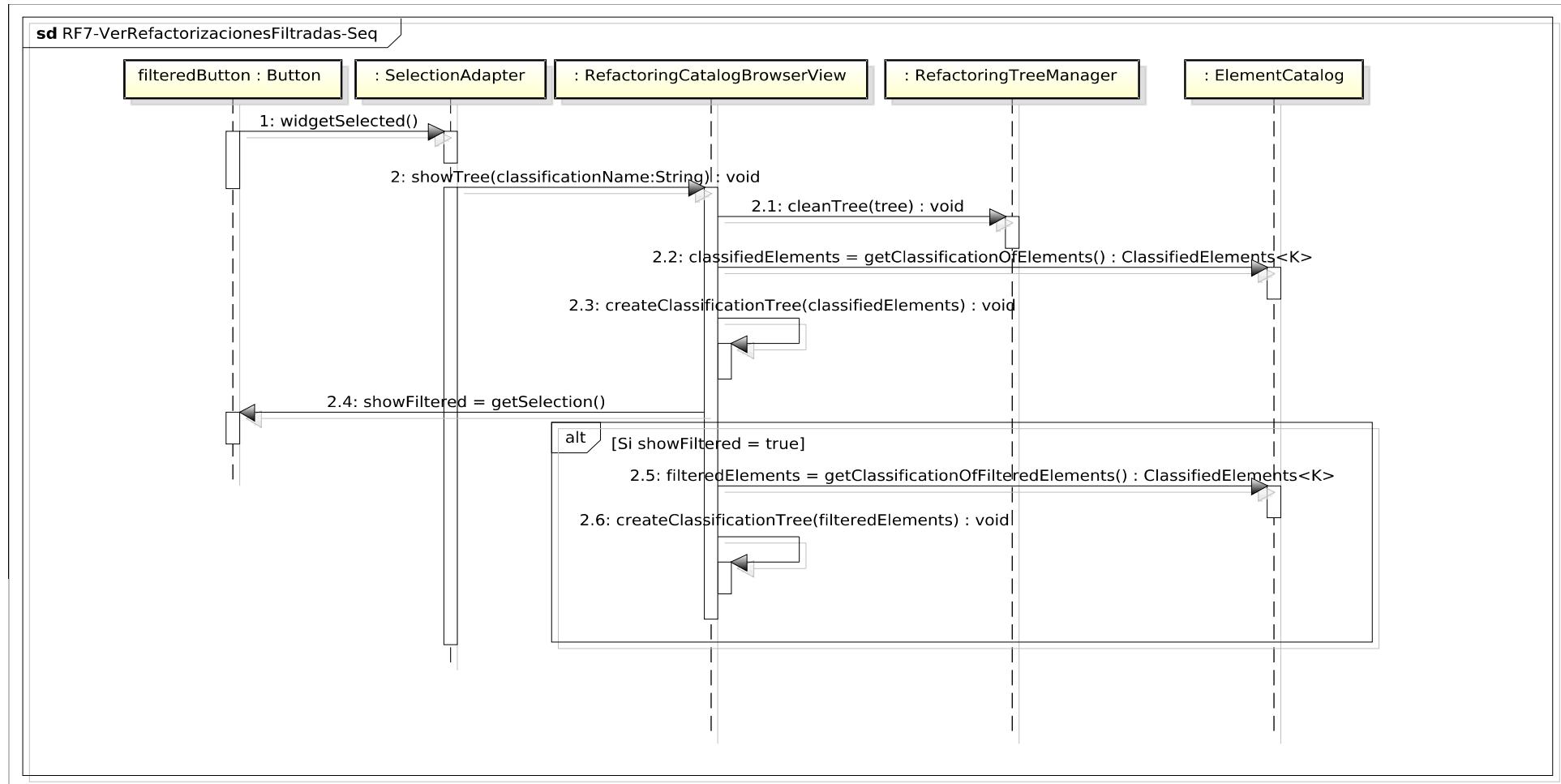


Ilustración 64: D. de secuencia de RF 7: Seleccionar opción ver ref. filtradas

En este caso cuando el usuario pulsa el botón de tipo checkbox, la única función del observador del evento es repintar el árbol. Como se puede ver en el diagrama al repintar el árbol se comprueba el estado del checkbox para decidir si se pintan o no los elementos filtrados.

5.1.8. RF 8: Visualizar detalle refactorización

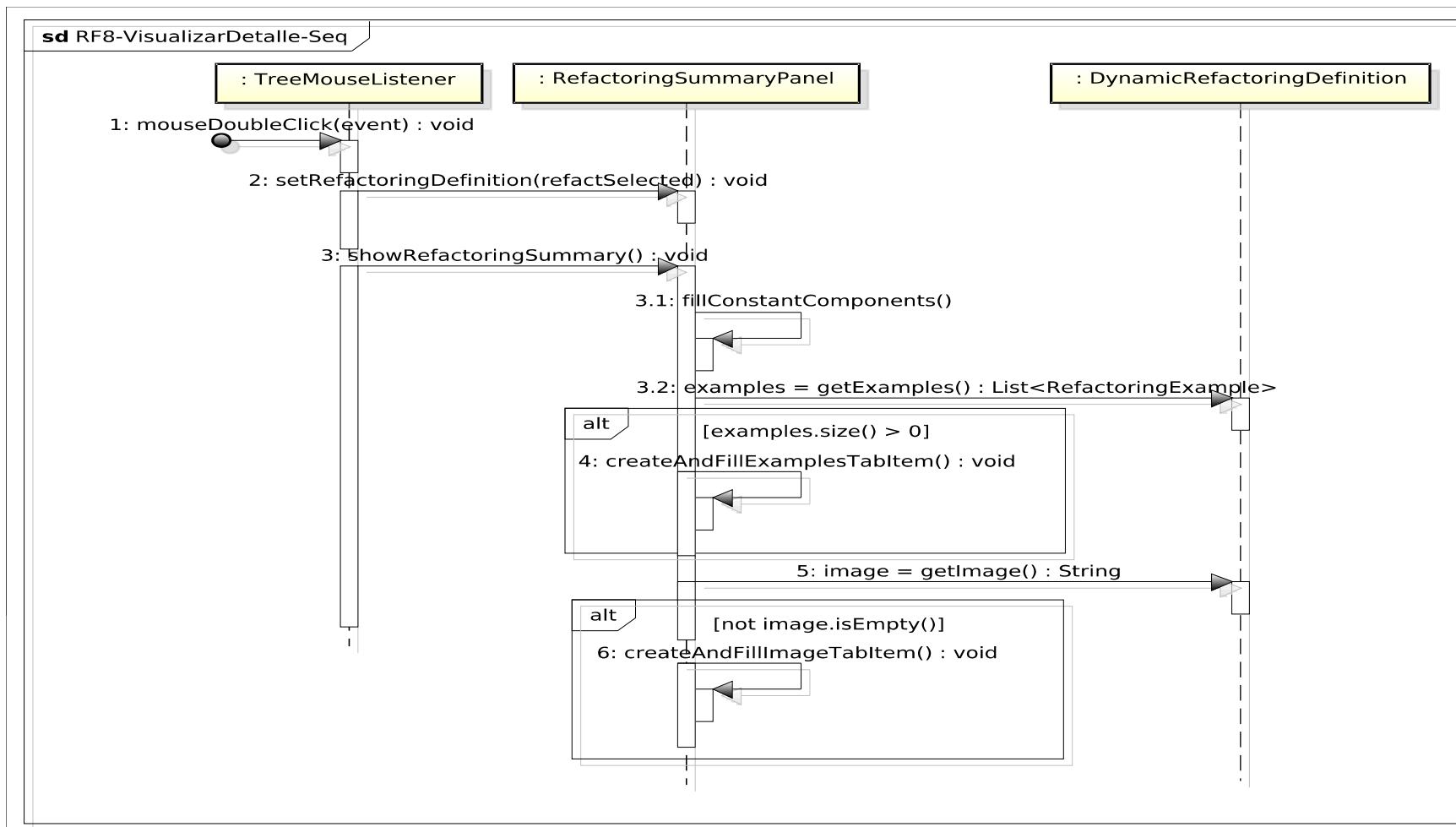


Ilustración 65: D. de secuencia de RF 8: Visualizar detalle refactorización

Para mostrar los detalles de la refactorización en el RefactoringSummaryPanel, primero se asigna la refactorización seleccionada al panel y luego se muestra el panel. Al mostrarse el panel, este llama a su método para mostrar las pestañas comunes y posteriormente a los de mostrar la imagen y los ejemplos si la refactorización cuenta con imágenes o ejemplos.

5.1.9. RF 9: Añadir clasificación

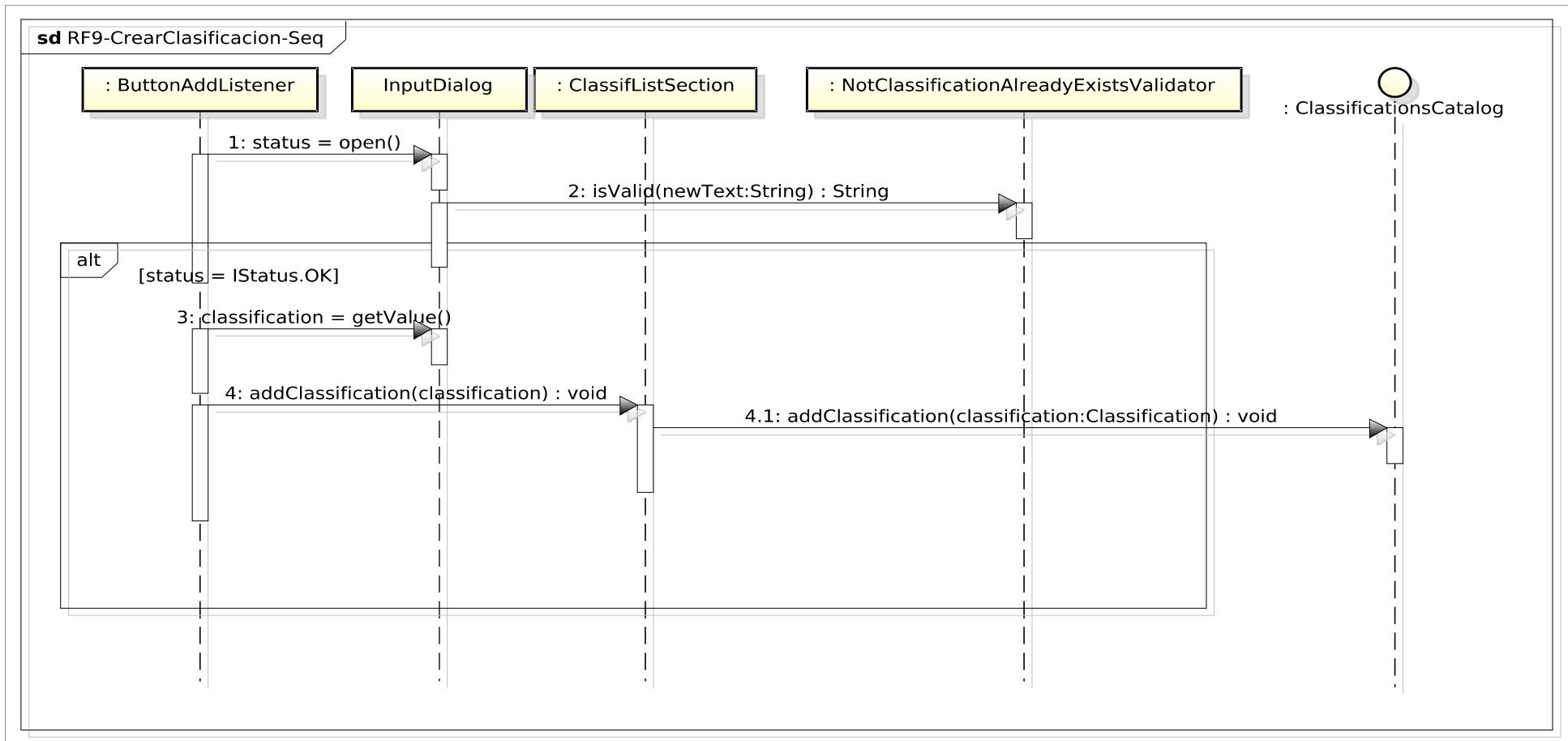


Ilustración 66: D. de secuencia de RF 9: Añadir clasificación

El observador de tipo `ButtonAddListener` crea un diálogo. Ese diálogo solicita el nombre de la clasificación a crear al usuario y comprueba que el usuario está introduciendo un valor válido como nombre llamando al validador. Cuando el diálogo se cierra si el status es OK, significa que el usuario ha introducido un valor válido por lo que el listener llama a la sección para que cree la nueva clasificación y actualice su interfaz.

5.1.10. RF 10: Editar clasificación

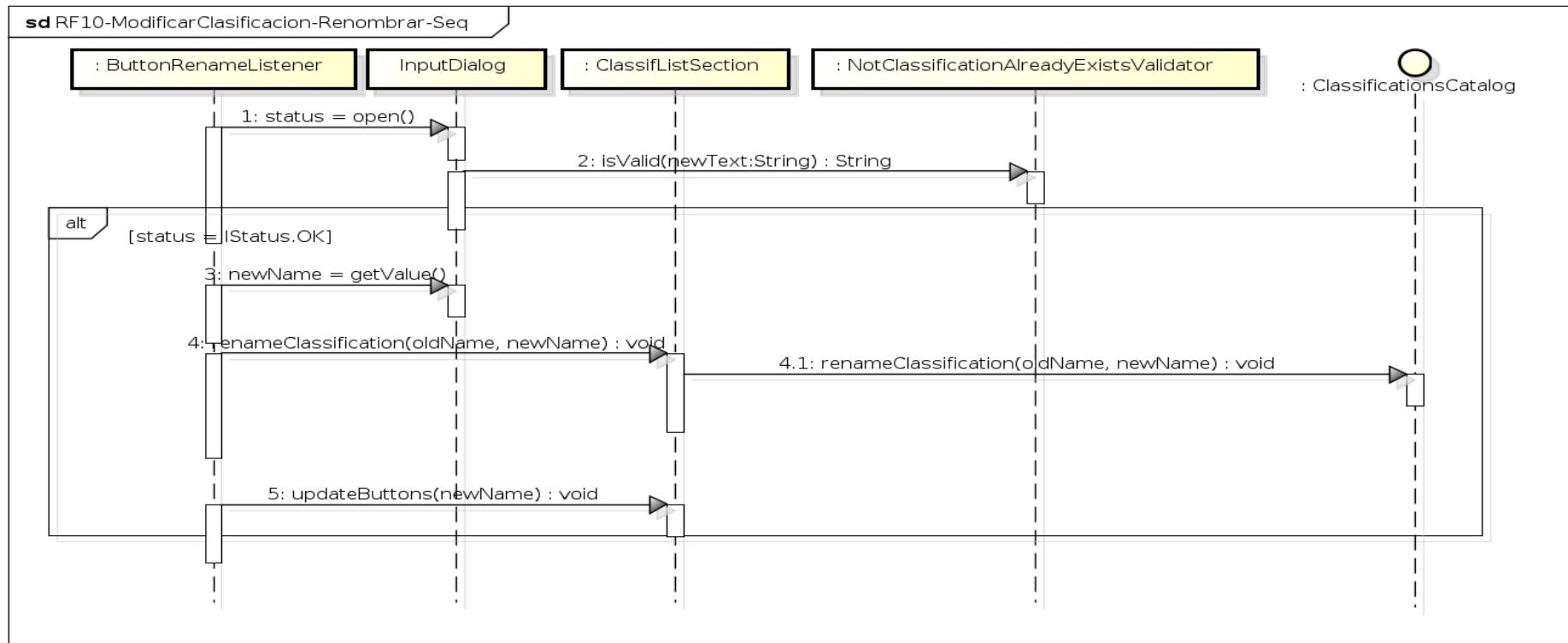


Ilustración 67: D. de secuencia de RF 10: Detalle - Renombrar una clasificación

Este requisito funcional se va a dividir en varios diagramas de secuencia dado que las actividades relacionadas con modificar los atributos de una clasificación comprenden varias acciones distintas dentro de la interfaz gráfica. El primer apartado que podemos ver en la imagen Ilustración 43 refleja lo que ocurre en la aplicación cuando el usuario renombra una clasificación existente. Para ello el usuario pulsa un botón que lanza un evento en `ButtonRenameListener` de `ClassifListSection`. Este lanza un cuadro de diálogo que comprueba la validez del nombre asignado a la clasificación mediante un validador de tipo `NotClassificationAlreadyExistsValidator`. Cuando el diálogo finaliza correctamente el observador del evento llama a la sección la cual llama de forma recursiva al catálogo de clasificaciones para que se encargue de aplicar la modificación.

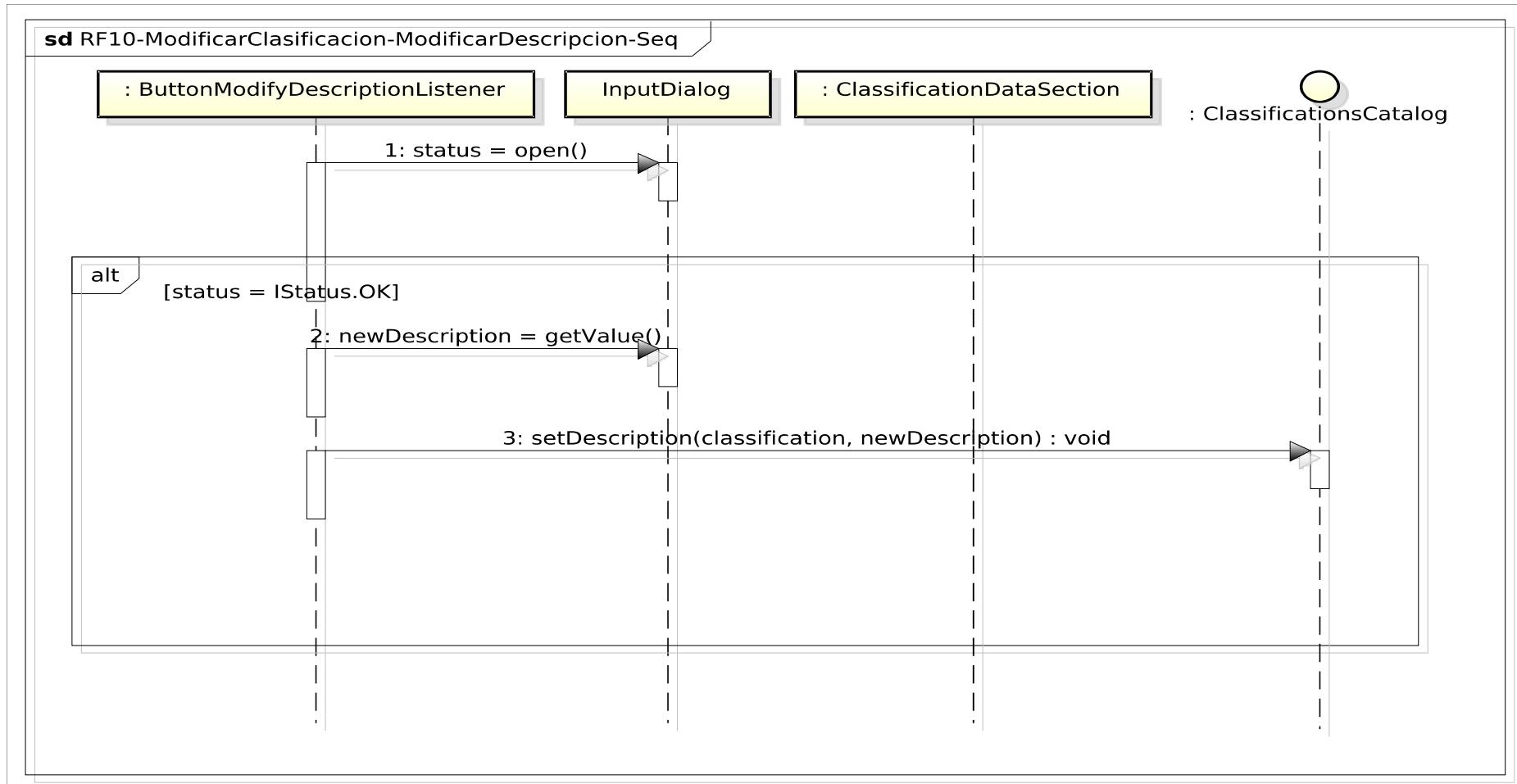


Ilustración 68: D. de secuencia de RF 10: Detalle - Modificar desc. de clasificación

Para la acción de modificar la descripción de la clasificación el observador del evento es un objeto del tipo `ButtonModifyDescriptionListener` de la clase `ClassificationDataSection`. Este observador llamará a la sección para que aplique la modificación en caso de que el usuario pulse aceptar en el diálogo dado que en este caso no se necesita validar el texto introducido por el usuario.

5.1.11. RF 11: Eliminar clasificación

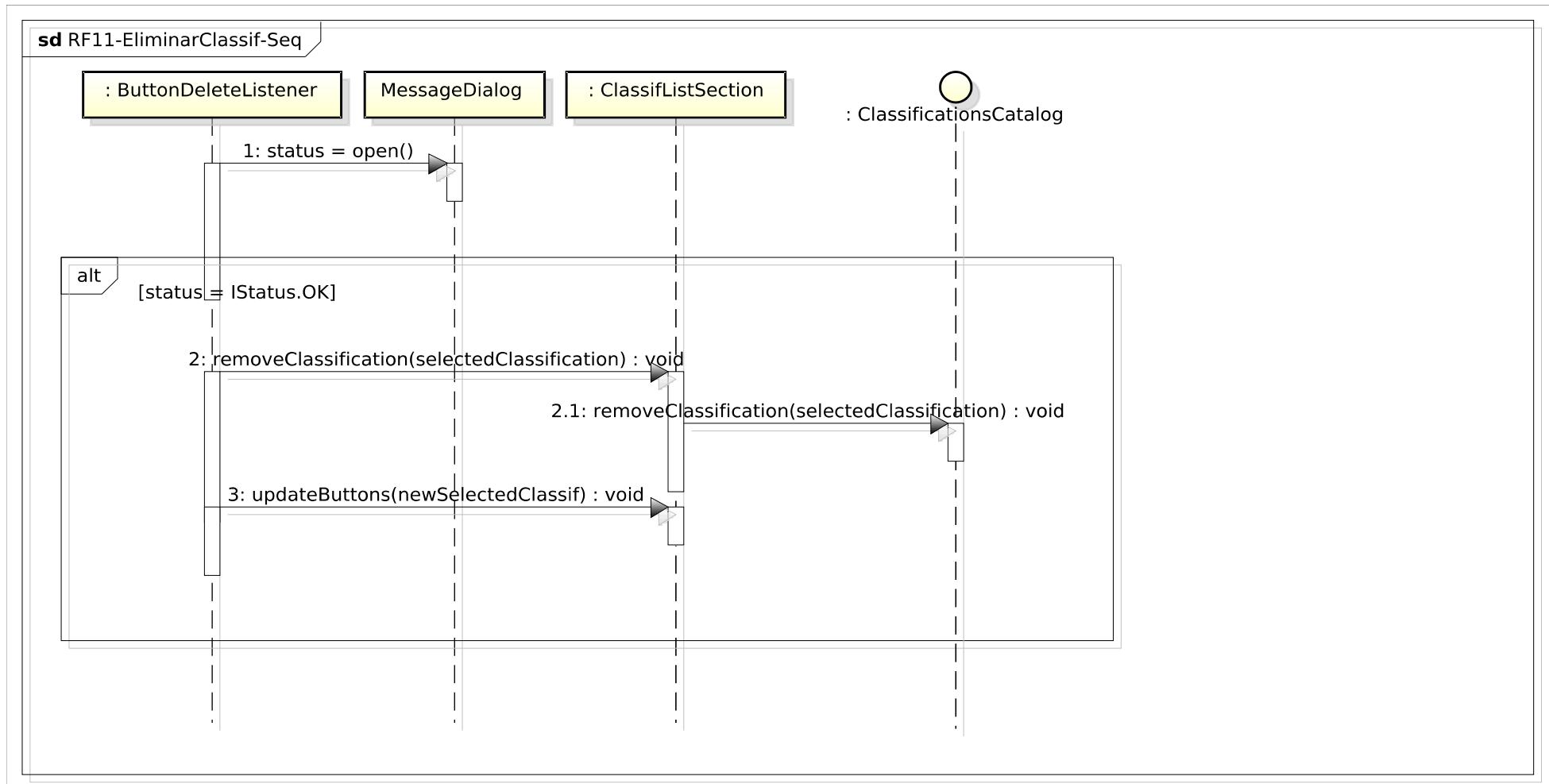


Ilustración 69: D. de secuencia de RF 11: Eliminar clasificación

De nuevo el iniciador es un observador que lanza un cuadro de diálogo. Cuando el cuadro de diálogo termina de forma correcta el observador avisa a la sección de tipo ClassifListSection que solicita al catálogo de clasificaciones que elimine la clasificación y luego actualiza su propia interfaz gráfica.

5.1.12. RF 12: Añadir categoría a una clasificación

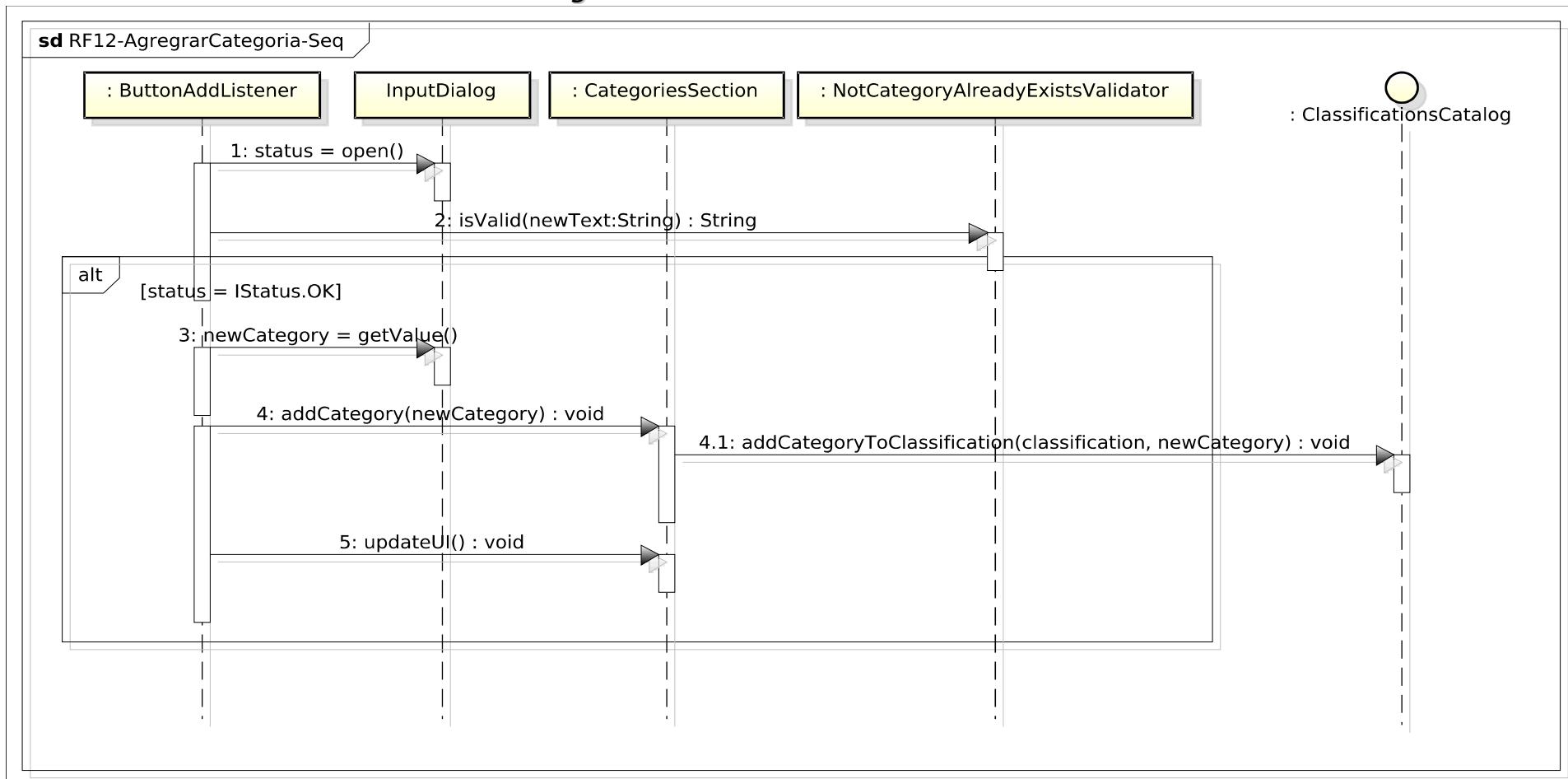


Ilustración 70: D. de secuencia de RF 12: Añadir categoría a una clasificación

Este diagrama es muy similar al del caso de uso RF9 que agregaba una clasificación nueva. Sigue exactamente la misma estructura de abrir un diálogo, comprobar su entrada, aplicar los cambios y actualizar la interfaz. Lo único que cambia es que en este caso el validador que valida la entrada comprueba que no existe una categoría con el nombre introducido por el usuario y las llamadas al catálogo añaden una categoría en lugar de una clasificación.

5.1.13. RF 13: Renombrar categoría de una clasificación

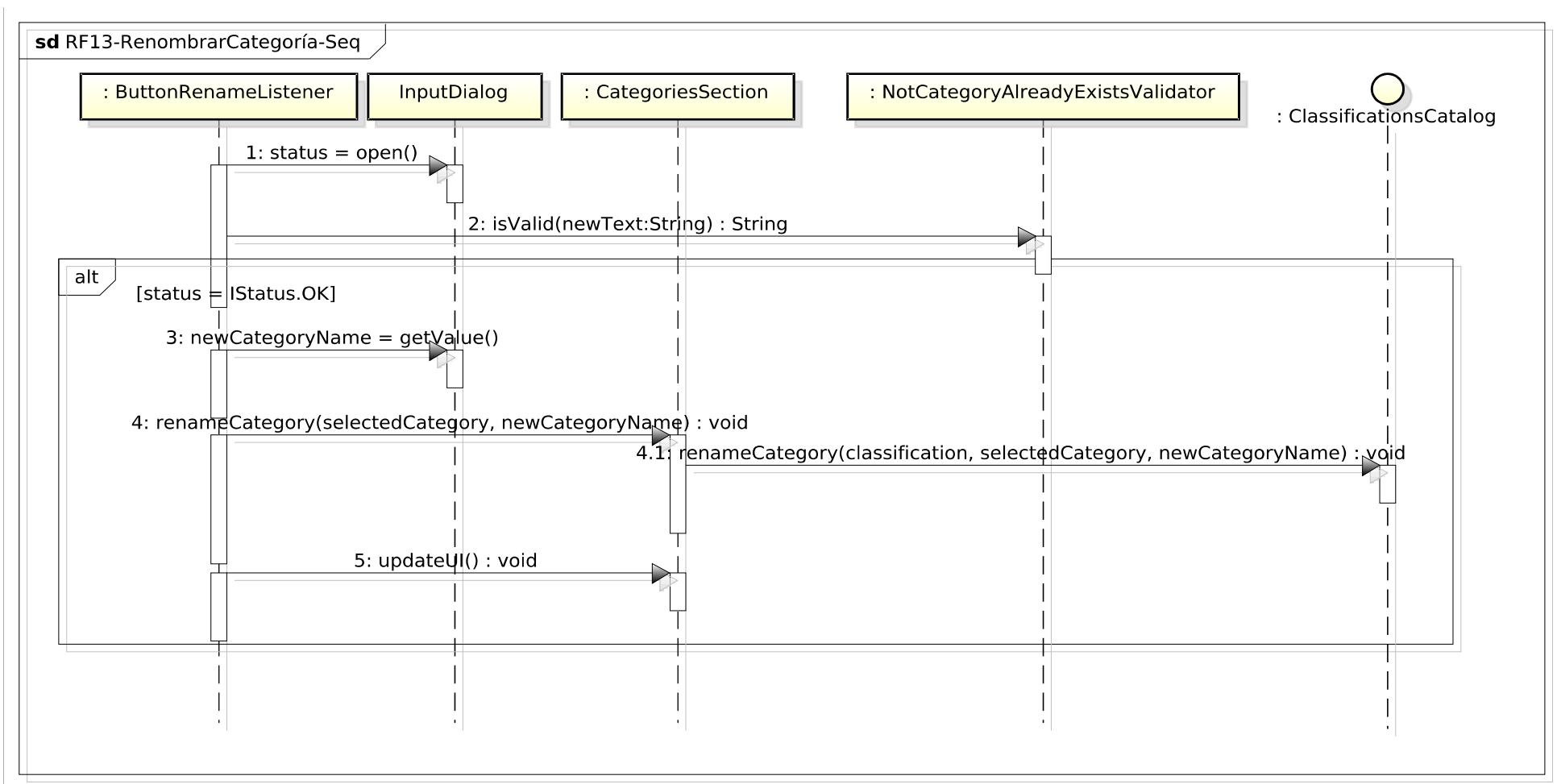


Ilustración 71: D. de secuencia de RF 13: Renombrar categoría

Este diagrama guarda paralelismo con el diagrama de Ilustración 43, de nuevo al igual que con el diagrama anterior lo único que cambia son el validador de la entrada y la sección que en este caso es `CategoriesSection`.

5.1.14. RF 14: Eliminar categoría de una clasificación

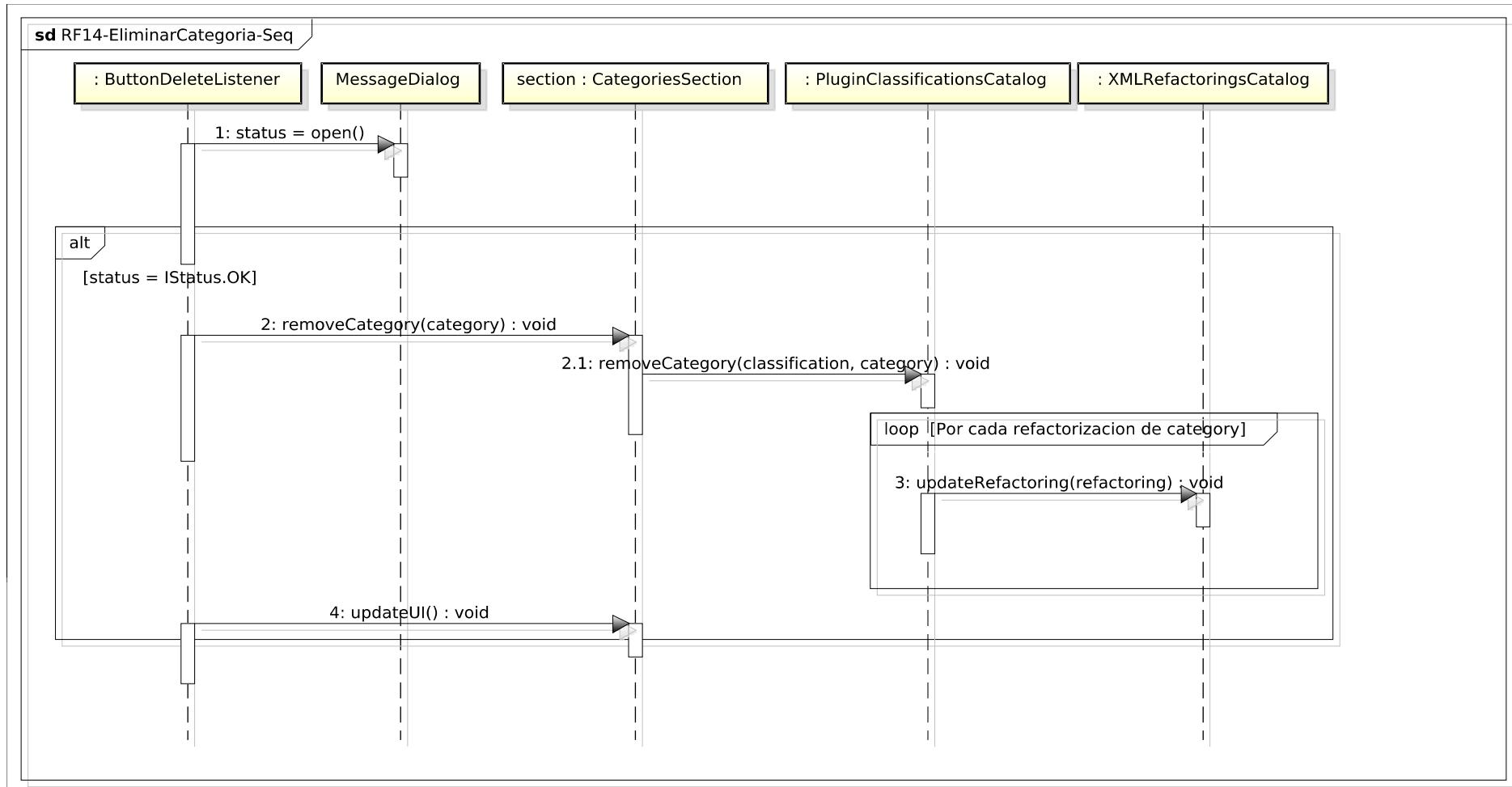


Ilustración 72: D. de secuencia de RF 14: Eliminar categoría de una clasificación

Lo más interesante de este diagrama es la parte final. En ella podemos ver que el catálogo de clasificaciones cuando elimina la categoría también se tiene que encargar de actualizar todas las refactorizaciones que pertenecían a dicha categoría a través del `XMLRefactoringsCatalog`.

5.1.15. RF 15: Mostrar resumen elemento seleccionado

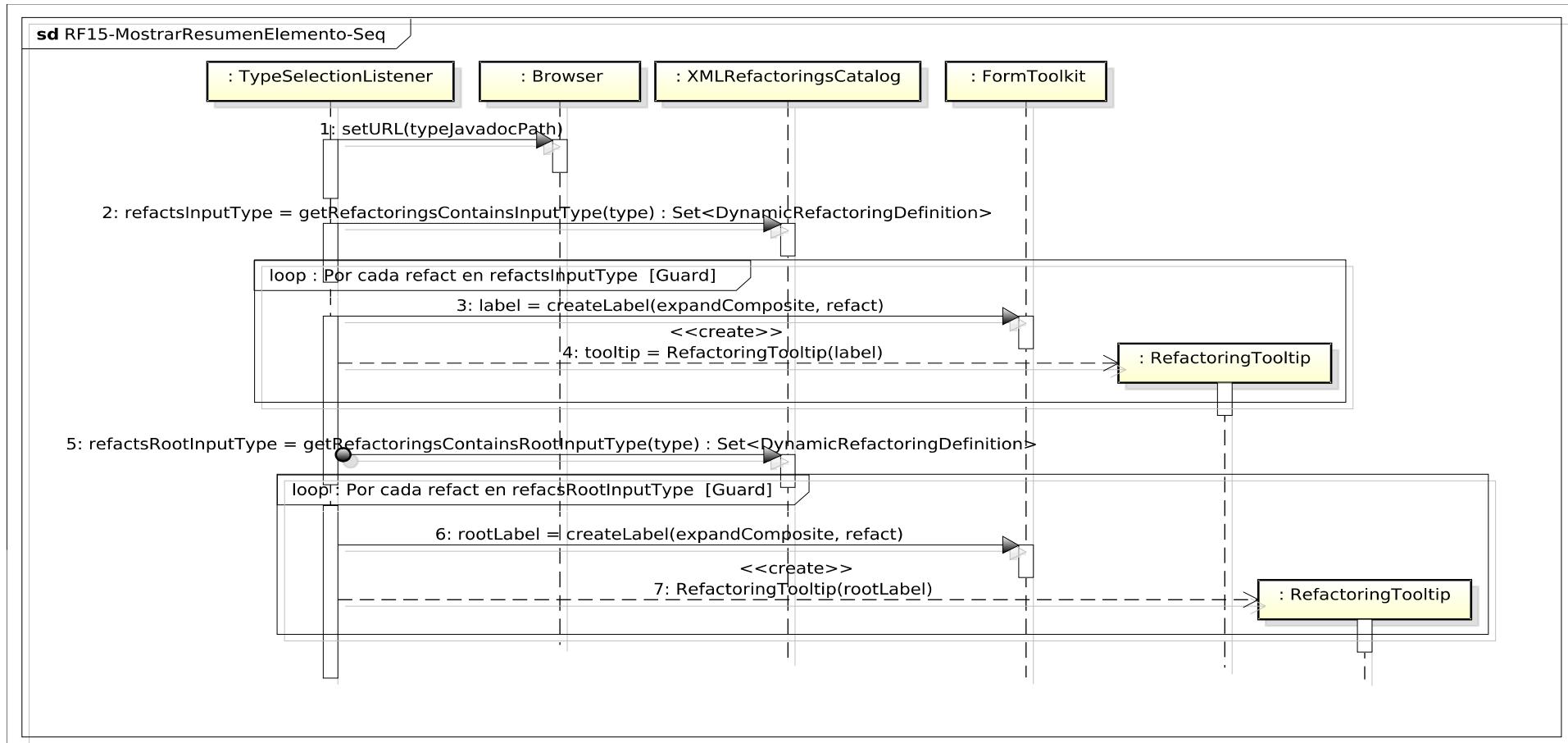


Ilustración 73: D. de secuencia de RF 15: Mostrar resumen elemento seleccionado

Ante el evento de selección de una entrada el `TypeSelectionListener` asignador al navegador el Javadoc correspondiente al tipo seleccionado. Además consulta al catálogo de refactorizaciones cuales son las que contienen una entrada del tipo seleccionado y dibuja un label con un tooltip para cada una de ellas. Hace lo mismo para las refactorizaciones que tienen como tipo raíz el seleccionado. Resaltar que en la imagen se ha tomado el caso de las entradas pero un caso similar se aplicaría para precondiciones, acciones y postcondiciones.

5.1.16. RF 16: Realizar búsqueda de elementos

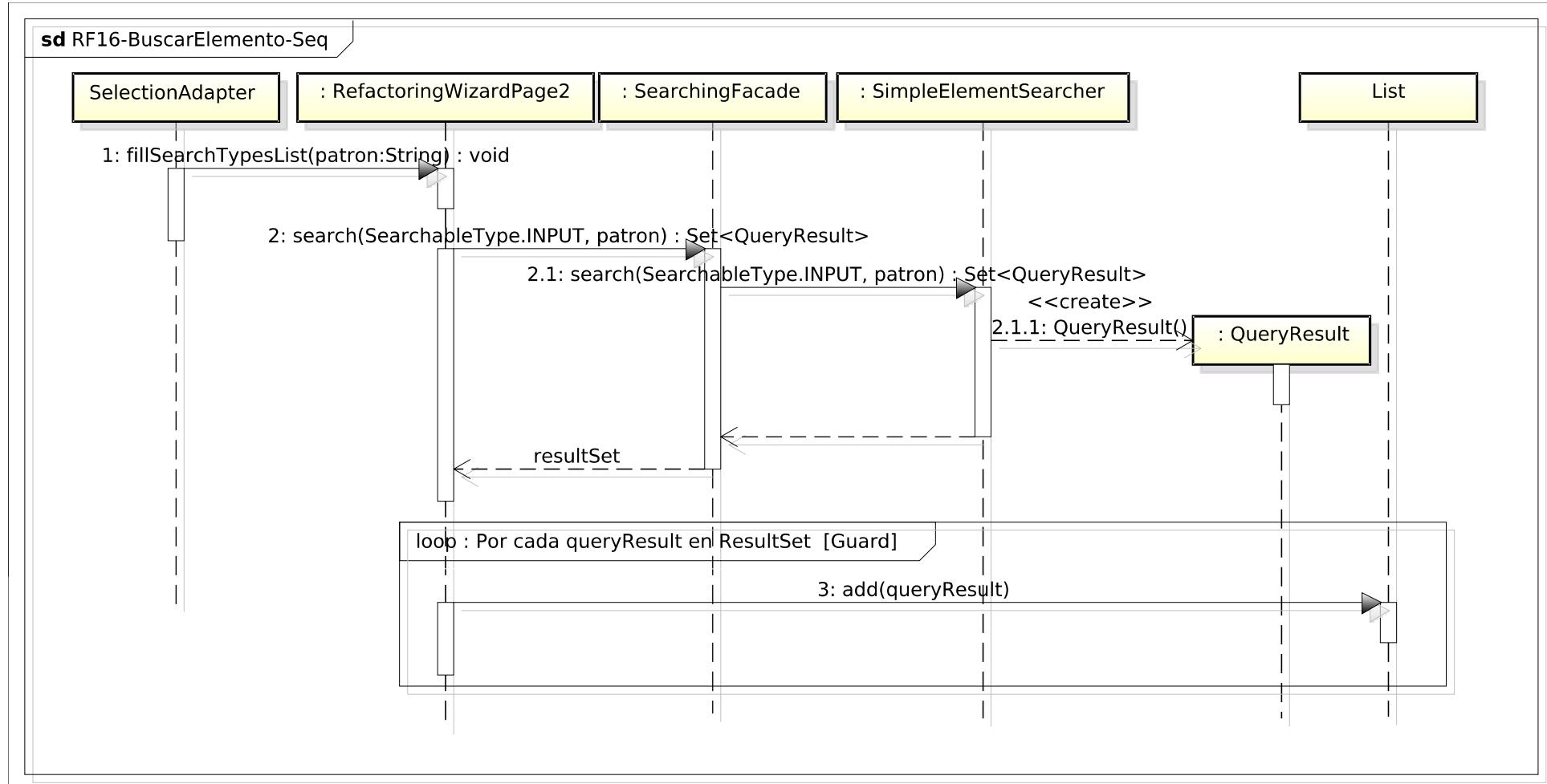


Ilustración 74: D. de secuencia de RF 16: Realizar búsqueda de elementos

Para realizar la búsqueda de elementos la página del asistente llama a la fachada de búsqueda con el patrón a buscar y el tipo sobre el que se realizará la búsqueda (en este caso de tipo entrada). Esta delega al buscador quien se encarga de crear los resultados de la búsqueda y devolvérselos. Finalmente la página crea una entrada en la lista de tipos de la interfaz por cada objeto devuelto en la búsqueda.

6. REPRESENTACIÓN CTTE DE DYNAMIC REFACTORING

A continuación se muestra una representación en forma de árbol de la estructura principal de las principales tareas de la interfaz desarrolladas en esta versión del plugin o de aquellas en las que se han introducido mejoras sustanciales.

6.1. Modificaciones en la primera página del wizard

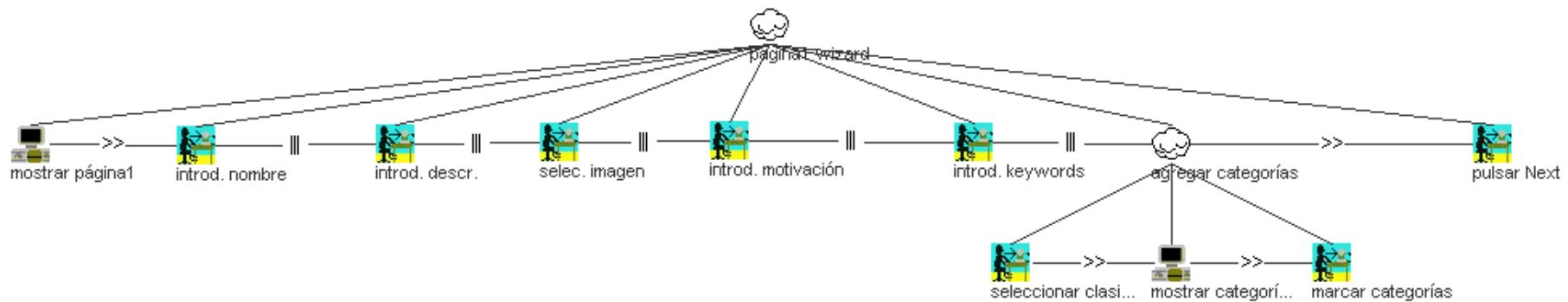


Ilustración 75: Cte de la primer página del asistente

6.2. Visualizar catálogo de Refactorizaciones

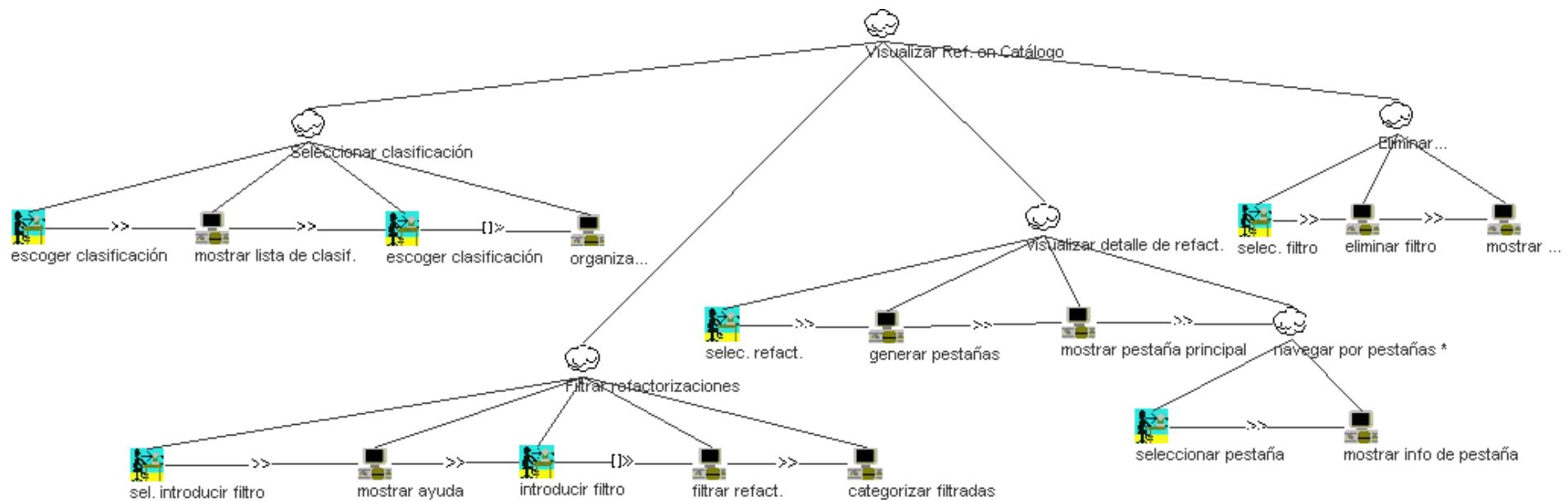


Ilustración 76: CTTE de visualizar catálogo de refactorizaciones

6.3. Editar clasificaciones

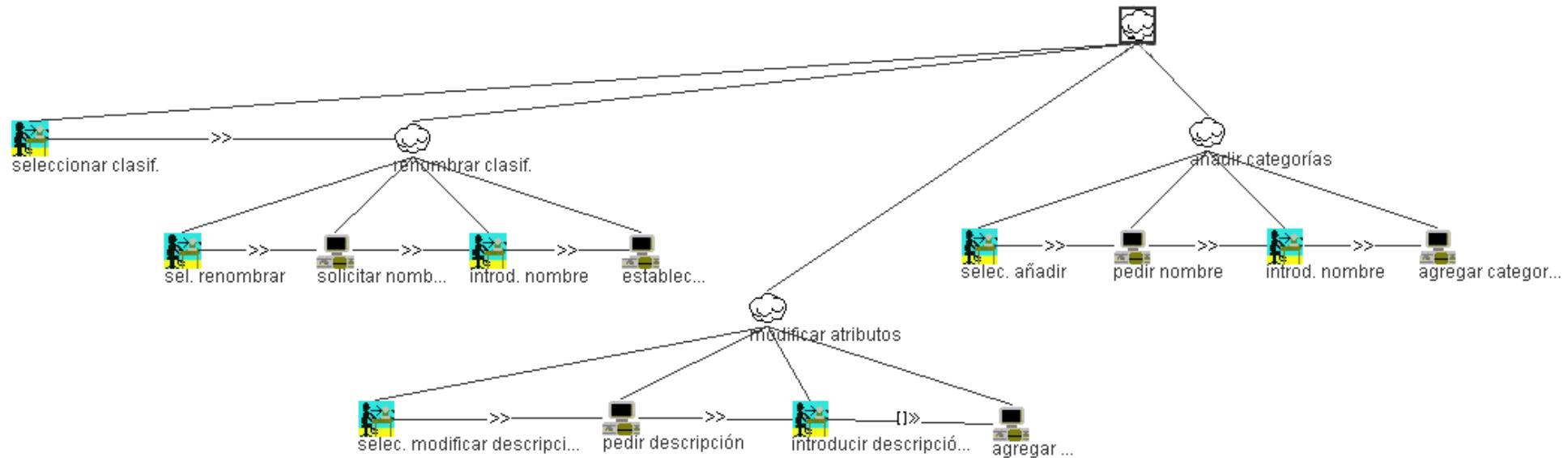


Ilustración 77: CTTE de edición de clasificaciones

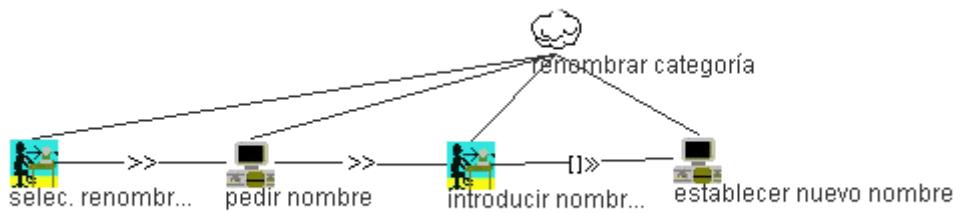


Ilustración 78: CTTE de renombrar una categoría

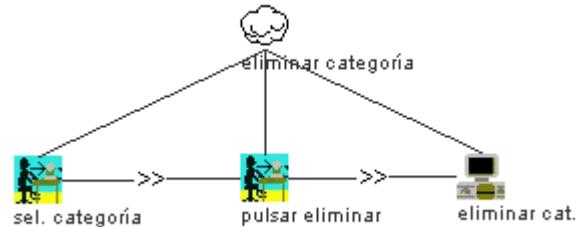


Ilustración 79: CTTE de eliminar categoría

6.4. Añadir una nueva clasificación

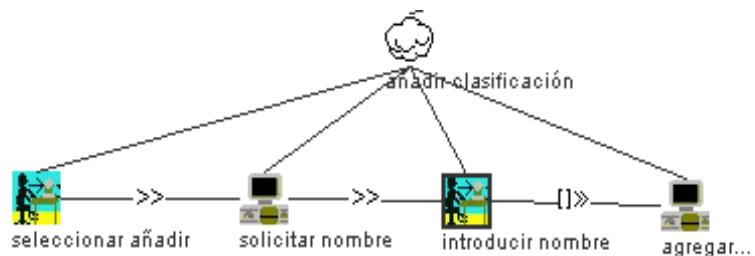


Ilustración 80: CTTE de agregar una nueva clasificación

6.5. Eliminar una clasificación



Ilustración 81: CTTE de eliminar una clasificación

7. PATRONES DE DISEÑO

Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí adaptada para resolver un problema de diseño general en un contexto particular [Gamma 1995].

Con un patrón de diseño podemos solucionar un problema que se nos presente desarrollando software, gracias a que ese problema lo tuvieron desarrolladores antes y se estableció un mecanismo para resolverlo.

En la Memoria principal se describen los tipos de patrones de diseño y se muestra una tabla resumen. Ahora veamos los patrones de diseño utilizados para este proyecto.

7.1. Patrón Objeto Constructor - *Builder*

Los constructores de las clases se encuentran con un problema cuando se necesita utilizar secuencias de parámetros muy largos cuando varios de los parámetros son opcionales. Tradicionalmente los programadores han utilizado el patrón *telescoping constructor* [Dr Dobb's Journal n.d.] . Este patrón funciona pero complica el escribir código cliente para este tipo de clases.

Una alternativa a este patrón son los *JavaBeans* en los que se llama a un constructor sin parámetros para crear la instancia del objeto y se utilizan métodos `Set` para asignar valores a los atributos. Este patrón era el utilizado en ciertas clases del plugin inicialmente. Sin embargo este patrón tiene ciertas desventajas. Un objeto creado de esta manera puede estar en un estado inconsistente cuando se está construyendo y la clase no tiene la opción de obligar a cumplir con los requisitos que aseguran dicha consistencia. Intentar utilizar el objeto en estado inconsistente puede provocar errores que se revelan en secciones del código alejadas de donde se originó el error y por tanto difíciles de depurar. Una desventaja relacionada es el hecho de que este patrón impide hacer una clase inmutable [Bloch 2008].

La solución que incluye las ventajas de los dos anteriores enfoques es el patrón *Builder* [Gamma 1995]. En lugar de crear el objeto deseado de forma directa, el cliente llama a un constructor con todos los parámetros obligatorios y obtiene un objeto constructor. Entonces el cliente llama a los métodos `Set` en el objeto constructor para asignar un valor al parámetro opcional que interese. Finalmente, el cliente llama al método sin parámetros `build()` para generar el objeto que es inmutable. Con este patrón el código

para instanciar objetos de la clase es sencillo de escribir y lo que es más importante de leer. Además simula los parámetros opcionales por nombre de lenguajes como Ada o Python.

Este patrón se ha utilizado para varias clases en el plugin, algunas tan importantes como `DynamicRefactoringDefinition` que contiene la definición de una refactorización con todos sus atributos: su nombre, su descripción, sus parámetros de entrada y sus precondiciones, acciones y postcondiciones entre otros.

Un ejemplo más sencillo de clase en el que se ha utilizado este patrón ha sido la clase `InputParameter`. La clase `InputParameter` ha tenido una evolución muy importante durante el desarrollo. En un principio la clase no existía y su papel lo cumplía un array de cadenas de caracteres lo que hacía la lectura del código de los parámetros de entrada en la aplicación muy difícil. Posteriormente se reemplazo la implementación original por una clase con atributos para incrementar la legibilidad del código lo que hizo posible otras mejoras en la lectura de los parámetros ambiguos. Para construir esta clase que disponía de varios atributosopcionales se decidió optar por el patrón constructor. Este patrón ha permitido facilitar la creación de objetos del tipo. Un ejemplo de código para crear un objeto con este patrón es el siguiente:

```
InputParameter.Builder(type).name("Method").from("Class").main(false).build()
```

Además debido al patrón Builder, en el método se puede comprobar una serie de condiciones en forma de contratos que debe cumplir todo parámetro de entrada como por ejemplo que todo parámetro de entrada principal no puede tener un atributo `from` asociado.

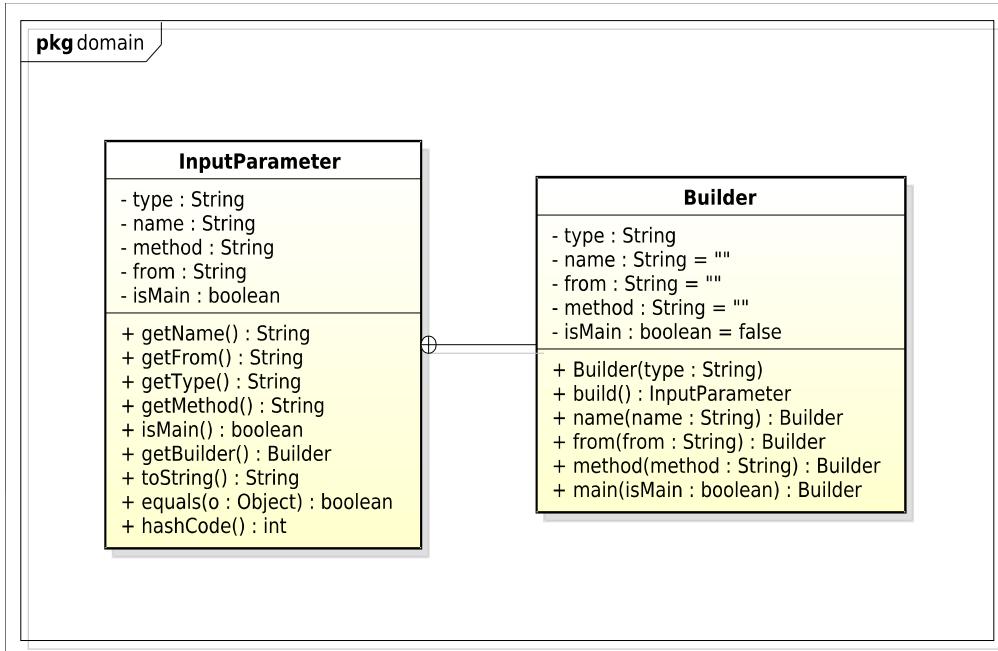


Ilustración 82: D. de clases del PD Builder aplicado a InputParameter

7.2. Patrón Fachada

El patrón fachada se basa en un objeto que proporciona una interfaz simplificada para trabajar con una biblioteca o un conjunto de clases. Las ventajas que el patrón fachada proporciona son:

- Simplifica la comprensión, el uso y la escritura de pruebas para la biblioteca debido a que la fachada proporciona métodos de conveniencia para las tareas comunes.
- Facilita la escritura de código que utilice las funciones de la biblioteca.
- Reduce las dependencias del código cliente de la biblioteca con el funcionamiento interno de la misma, dado que la mayoría del código cliente utiliza exclusivamente la fachada, lo que incrementa la flexibilidad a la hora de desarrollar el sistema.
- Cuando la arquitectura y el código de la biblioteca están mal diseñados, una fachada puede permitir ocultar las interfaces mal diseñadas bajo un único API más adecuado.

Un subsistema del plugin en el que se ha aplicado el patrón fachada ha sido el de búsqueda de elementos dentro del asistente de creación de refactorizaciones. Este sistema es un sistema bastante complejo que utiliza la biblioteca Lucene para las búsquedas y clases del API de Eclipse para la lectura de la descripción de las clases a partir de su documentación en formato javadoc. De forma interna se ocupa de funciones como el parseado de la documentación, el indexado de la información de los elementos, la configuración de parámetros de idioma y palabras de parada o la realización de la búsqueda.

Sin embargo, desde un punto de vista externo los clientes sólo requerían acceso a las funciones de búsqueda y a la posibilidad de decidir en qué momento se iba a necesitar realizar la indexación de los elementos. Es por eso que la clase fachada del subsistema SearchingFacade sólo expone dos métodos para cada una de las funciones anteriores. De este modo se ha aislado al cliente de toda la complejidad del funcionamiento de la búsqueda y éste sólo necesita comprender dos métodos muy sencillos para poder realizar tareas cuya complejidad subyacente es mucho mayor de lo que el cliente pudo percibir. Como ventaja añadida, el patrón fachada proporciona la flexibilidad de que la biblioteca puede variar su implementación sin provocar problemas de compatibilidad en sus clientes.

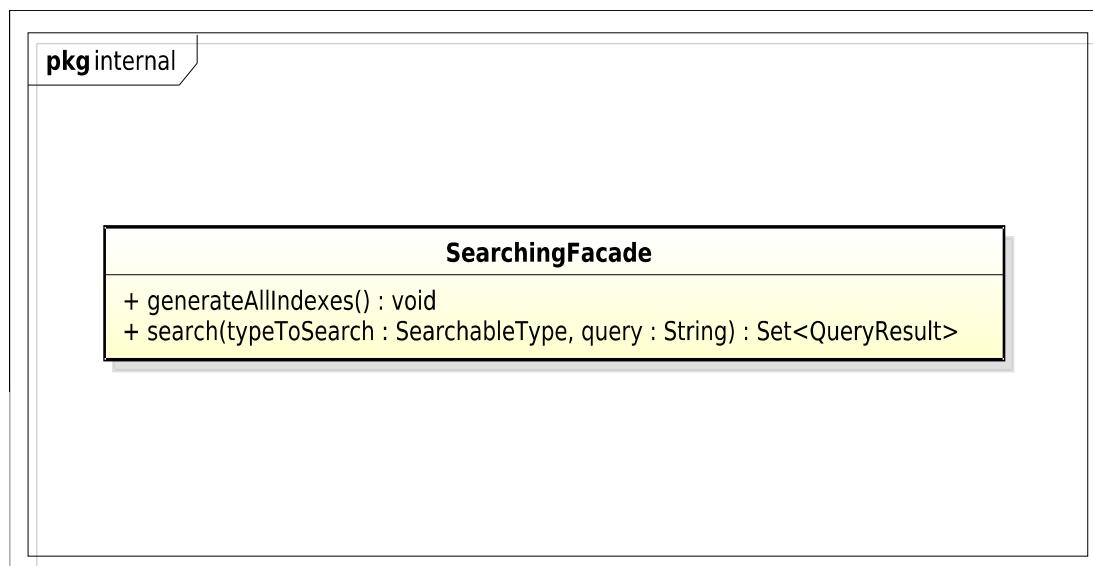


Ilustración 83: D. de clases del PD Facade aplicado a SearchingFacade

7.3. Patrón Singleton

El patrón de diseño singleton (instancia única) está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

El patrón singleton se implementa creando en nuestra clase un método fábrica que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor haciéndolo protegido o privado.

Las situaciones más habituales de aplicación de este patrón son aquellas en las que dicha clase controla el acceso a un recurso físico único (como puede ser el ratón o un archivo abierto en modo exclusivo) o cuando cierto tipo de datos debe estar disponible para todos los demás objetos de la aplicación.

El patrón singleton provee una única instancia global gracias a que:

- La propia clase es responsable de crear la única instancia.
- Permite el acceso global a dicha instancia mediante un método de clase.
- Declara el constructor de clase como privado para que no sea instanciable directamente.

Otra implementación del patrón *Singleton* en Java es sugerida en [Bloch 2008] como alternativa a la implementación basada en un método fábrica. La implementación basada en un método otorga la flexibilidad de que da la posibilidad de cambiar de opinión y sustituir el comportamiento como *Singleton* por una clase de la que se pueden crear múltiples instancias. Sin embargo, para casos en los que este requisito no es necesario es una solución mucho más sencilla de implementar es utilizar una enumeración de un único elemento que se convierte en la única instancia del objeto. Este enfoque tiene la ventaja de que es funcionalmente equivalente y proporciona un seguro adicional ante múltiples instanciaciones tanto en el caso de que el objeto sea serializable, como en el caso de que se intente utilizar reflexión. Por encima de estas ventajas está el hecho de que el código necesario para implementar un *Singleton* de esta manera es increíblemente sencillo.

Un ejemplo de clase en el que se ha utilizado este último enfoque comentado es la clase `EclipseBasedJavadocReader`. Esta clase necesita ser un *Singleton* debido a que crea

un recurso de Eclipse que no debe ser duplicado cada vez que es instanciada. Gracias a haber utilizado el enfoque basado en la enumeración todo el código necesario para hacer la clase de instancia única ha sido el siguiente:

```
public enum EclipseBasedJavadocReader implements JavadocReader {  
    INSTANCE;  
  
    private EclipseBasedJavadocReader() {  
        ...  
    }  
}
```

Como se puede ver se ha evitado tener que escribir el código necesario para la inicialización ante demanda del lector y el código es notablemente más sencillo. De hecho, el constructor privado ha sido necesario para realizar ciertas tareas de inicialización de la clase pero no es estrictamente necesario, lo que pudiera haber hecho el código de la clase más sencillo. El diagrama de la clase se muestra a continuación:

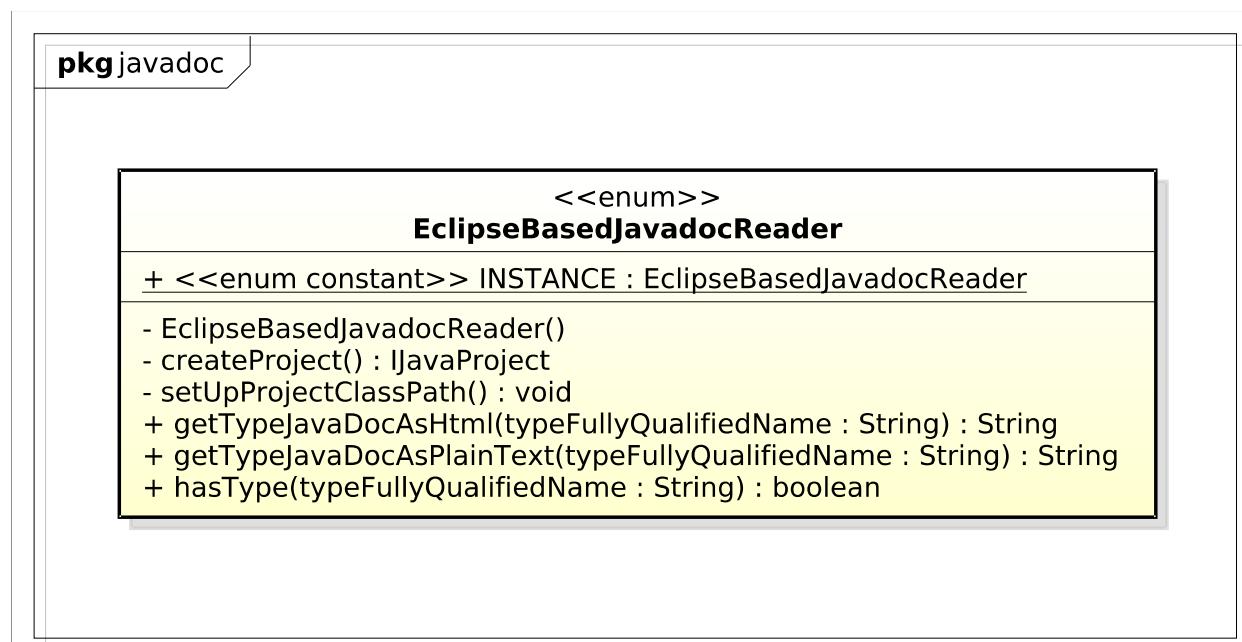


Ilustración 84: D. de clases del PD Singleton aplicado a EclipseBasedJavadocReader

7.4. Patrón Comando

El patrón Comando permite que objetos del toolkit hagan peticiones a objetos de la

aplicación no especificados convirtiendo la propia petición en un objeto. La clave está en una clase abstracta Orden- Comando que declara una interfaz para ejecutar operaciones. Las subclases concretas de Orden- Comando especifican un par receptor/acción. Se guardan el receptor como variable de instancia e implementan ejecutar para que invoque a la petición.

El patrón orden desacopla el objeto que invoca la operación de aquel que posee el conocimiento para realizarla. Gracias a esto por ejemplo un elemento del menú y un botón pueden realizar la misma función compartiendo una instancia de la misma subclase concreta de orden.

El patrón comando permite además:

- Parametrizar objetos mediante una acción a realizar.
- Especificar, encolar y ejecutar peticiones en momentos diferentes
- Soportar comandos reversibles
- Mantener un registro (log) de operaciones realizadas.
- Estructurar un sistema en base a transacciones.

La solución del comando se basa en:

- Comando: declara una interfaz para ejecutar una operación
- ComandoConcreto: define una ligadura entre un receptor y una acción/operación y redefine la operación Ejecutar para que se envíe una petición al receptor.
- Cliente: crea un ComandoConcreto al que le indica cual es su receptor.
- Emisor: pide al comando que lleve a cabo una petición.
- Receptor: sabe como realizar la operación asociada a una petición.

En el caso de los plugins, Eclipse proporciona su propio mecanismo que funciona como una extensión de las ideas del patrón comando: las acciones. Las acciones permiten definir objetos que encapsulan respuestas a peticiones del usuario y que pueden ser accesibles desde distintos puntos de la interfaz. De este modo se separan la representación

en la interfaz de la acción, de la propia acción lo que hace que un usuario pueda ejecutar la misma acción desde un menú contextual o desde cualquier barra de herramientas sin esfuerzo adicional para el programador.

Un ejemplo de comando concreto en el plugin de refactorizaciones es la clase `RefreshViewAction` la cual encapsula la acción dedicada a actualizar la vista del catálogo de refactorizaciones. Esta clase implementa la interfaz `IviewActionDelegate` que cumple el rol de la interfaz comando.

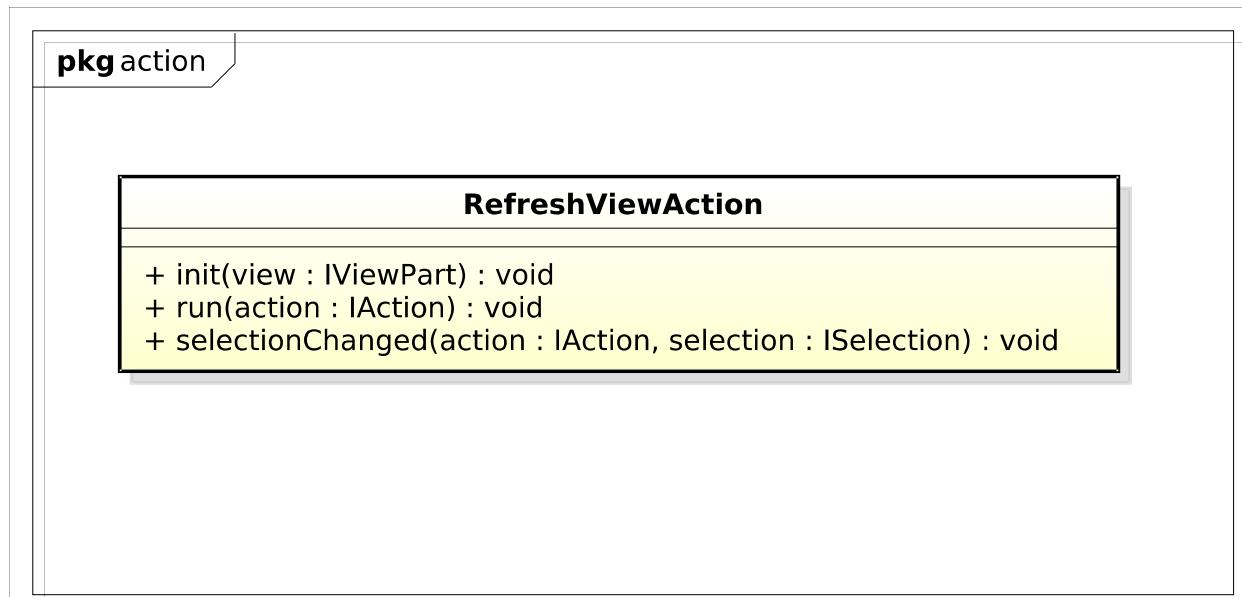


Ilustración 85: D. de clases del PD Command aplicado a RefreshViewAction

8. REFERENCIA CRUZADA CON LOS REQUISITOS

En este apartado relacionamos los requisitos funcionales con los módulos de diseño. Veremos si se han satisfecho todos los requisitos tras la etapa de diseño y qué módulos implementan cada uno de los requisitos.

Los requisitos funcionales definidos en el Anexo 2 se enumeran en esta tabla:

Requisito	Descripción
RF-1	Visualizar refactorizaciones según clasificación
RF-2	Refrescar visualización de refactorizaciones
RF-3	Añadir filtro de refactorizaciones
RF-4	Seleccionar opción aplicar filtro
RF-5	Eliminar filtro de refactorizaciones
RF-6	Eliminar todos los filtros de refactorizaciones
RF-7	Seleccionar opción ver refactorizaciones filtradas
RF-8	Visualizar detalle refactorización
RF-9	Añadir clasificación
RF-10	Editar clasificación
RF-11	Eliminar clasificación
RF-12	Añadir categorías a una clasificación
RF-13	Renombrar categorías de una clasificación
RF-14	Eliminar categorías de una clasificación
RF-15	Mostrar resumen elemento seleccionado
RF-16	Realizar búsqueda de elementos

Tabla 39: Requisitos funcionales

A continuación se muestra una tabla con las referencias cruzadas entre los requisitos funcionales y los módulos de diseño.

	dynamicrefactoring										
		dynamicrefactoring.action			dynamicrefactoring.domain						
			dynamicrefactoring.domain.metadata.classifications.xml.imp			dynamicrefactoring.domain.metadata.condition					
				dynamicrefactoring.domain.metadata.imp			dynamicrefactoring.domain.metadata.interfaces				
					dynamicrefactoring.domain.xml			dynamicrefactoring.interfaz			
						dynamicrefactoring.interfaz.view			dynamicrefactoring.interfaz.editor.classifieditor		
							dynamicrefactoring.interfaz.wizard				
								dynamicrefactoring.interfaz.wizard.wizard.search.internal			
									dynamicrefactoring.interfaz.wizard.wizard.search.javadoc		
										dynamicrefactoring.util	
RF-1											
RF-2											
RF-3											
RF-4											
RF-5											
RF-6											
RF-7											
RF-8											
RF-9											
RF-10											
RF-11											
RF-12											
RF-13											
RF-14											
RF-15											
RF-16											

Tabla 40: Referencia cruzada de requisitos con módulos de diseño

9. DISEÑO DE PRUEBAS

En este apartado se definen las pruebas que se deben desarrollar para comprobar que la aplicación cumple con los requisitos funcionales. Se va a presentar una tabla por cada requisito funcional con todas las pruebas relacionadas con dicho requisito mostrando los aspectos que se deben probar y los resultados esperados de dichas pruebas.

9.1. RF 1: Visualizar refactorizaciones según clasificación

Requisito	Aspectos a probar	Resultado esperado
RF 1: Visualizar refactorizaciones según clasificación	Las lista de clasificaciones se muestran correctamente al usuario.	En el combo de selección de clasificaciones el usuario cuenta con todas las clasificaciones existentes.
	Al seleccionar una clasificación, la vista muestra las refactorizaciones correctamente agrupadas.	Cada refactorización aparece dentro del grupo correspondiente a su categoría en el árbol de la vista.
	Al seleccionar una clasificación multicategoría, la vista muestra las refactorizaciones correctamente agrupadas.	Las refactorizaciones que pertenecen a más de una categoría en la clasificación son mostradas en el grupo de cada categoría.
	La vista muestra correctamente las refactorizaciones que no pertenecen a ninguna categoría dentro de la clasificación seleccionada.	Las refactorizaciones que no pertenecen a ninguna categoría deberán mostrarse dentro de la categoría NONE.
	El árbol representa de forma correcta las refactorizaciones de usuario y del plugin.	Si una refactorización es del plugin deberá representarse con el icono de refactorización y un candado superpuesto para indicar que no es editable.
	Los grupos que representan las categorías de la clasificación están ordenados para facilitar la visualización al usuario.	Las categorías aparecen ordenadas alfabéticamente en el árbol.
	Por cada refactorización se muestra correctamente su información general.	Cada nodo que representa una refactorización en el árbol muestra la descripción, motivación, precondiciones, postcondiciones y acciones que componen la refactorización.

Tabla 41: Pruebas de RF1: Visualizar refactorizaciones según clasificación

9.2. RF 2: Refreshar visualización de refactorizaciones

Requisito	Aspectos a probar	Resultado esperado
RF 2: Refreshar visualización de refactorizaciones	Si el usuario añade una refactorización la vista debe reflejarlo al pulsar el botón de refreshar.	La nueva refactorización se muestra dentro de la categoría que le corresponde de entre las que posee la clasificación actualmente seleccionada.
	Si el usuario modifica una refactorización la vista debe reflejarlo .	La refactorización modificada muestra sus atributos actualizados, sin dejar ningún vestigio de sus atributos anteriormente reflejados.
	Si el usuario elimina una refactorización la vista debe reflejarlo.	La refactorización eliminada por el usuario debe desaparecer del árbol de la vista.
	Si el usuario agrega una clasificación la vista debe reflejarlo.	En el control de selección de la clasificación la nueva clasificación debe aparecer entre las listadas.
	Si el usuario modifica una clasificación la vista debe reflejarlo.	La clasificación debe aparecer en el control de elección de clasificación con el nuevo nombre y deben listarse todas las categorías que la clasificación tiene tras las modificaciones.
	Si el usuario elimina una refactorización la vista debe reflejarlo.	En el control de selección de la clasificación la nueva clasificación debe desaparecer. Si era la clasificación seleccionada deberá seleccionarse otra y actualizar la vista con la nueva selección.

Tabla 42: Pruebas de RF 2: Refreshar visualización de refactorizaciones

9.3. RF 3: Añadir filtro de refactorizaciones

Requisito	Aspectos a probar	Resultado esperado
RF 3: Añadir filtro de refactorizaciones	El filtrado de las refactorizaciones por categoría es correcto.	Al filtrar las refactorizaciones por categoría, todas aquellas refactorizaciones que no pertenezcan a la categoría del filtrado pasaran a la lista de elementos ocultos. * Esto es independiente de la clasificación seleccionada para la vista.
	El filtrado de las refactorizaciones por texto es correcto.	Al filtrar las refactorizaciones por texto, todas aquellas refactorizaciones cuya descripción no contenga las palabras pasadas pasaran a la lista de elementos ocultos.
	El filtrado de las refactorizaciones por palabra clave es correcto.	Al filtrar las refactorizaciones por palabra clave, todas aquellas refactorizaciones que no tengan entre sus palabras clave la pasada pasaran a la lista de elementos ocultos.
	El plugin responde de forma correcta ante peticiones de filtrado con sintaxis incorrecta.	Si la petición de filtrado solicitada por el usuario no es correcta, la aplicación deberá mostrar un mensaje de error y no aplicar ningún tipo de filtro.
	Respuesta ante un filtro por una categoría inexistente.	Si se intenta aplicar un filtrado por una categoría que no existe se notificará al usuario de que dicha categoría no existe y se le hará la opción de cancelar el filtrado o seguir adelante con él.

Tabla 43: Pruebas de RF 3: Añadir filtro de refactorizaciones

9.4. RF 4: Seleccionar opción aplicar filtro

Requisito	Aspectos a probar	Resultado esperado
RF 4: Seleccionar opción aplicar filtro	El filtro se aplica por defecto al ser creado.	Cuando el usuario crea cualquier tipo de filtro tal y como se define en el RF3, inmediatamente el filtro creado será aplicado por defecto.
	Un filtro activado se desactiva correctamente.	Si un filtro que está activado se desactiva, todas las refactorizaciones que estaban ocultas exclusivamente porque no cumplían los requisitos de dicho filtro volverán a mostrarse en sus categorías correspondientes.
	Un filtro desactivado se activa correctamente.	Si nun filtro desactivado se activa, todas las refactorizaciones que no cumplen con dicho filtro volverán a ocultarse.

Tabla 44: Pruebas de RF 4: Seleccionar opción aplicar filtro

9.5. RF 5: Eliminar filtro de refactorizaciones

Requisito	Aspectos a probar	Resultado esperado
RF 5: Eliminar filtro de refactorizaciones	Eliminar un filtro, que es el único responsable de que una refactorización este oculta.	Si un filtro que está activado se desactiva, todas las refactorizaciones que estaban ocultas porque no cumplían los requisitos de dicho filtro volverán a mostrarse en sus categorías correspondientes. El filtro desaparecerá de la tabla de filtros.
	Eliminar un filtro, que es responsable junto con otros de que una refactorización este oculta.	Sólo se volverán a mostrar las refactorizaciones que estaban siendo ocultadas exclusivamente debido dicho filtro. Refactorizaciones que no cumplen con el filtro pero tampoco con otros en vigor seguirán ocultas. El filtro deberá desaparecer de la tabla de filtros.

Tabla 45: Pruebas de RF 5: Eliminar filtro de refactorizaciones

9.6. RF 6: Eliminar todos los filtros de refactorizaciones

Requisito	Aspectos a probar	Resultado esperado
RF 6: Eliminar todos los filtros de refactorizaciones	En un principio cuando no hay ningún filtro aplicado la interfaz no debe de dar la opción al usuario de ejecutar esta acción.	El botón de eliminar todos los filtros debe aparecer desactivado.
	La eliminación de los filtros funciona correctamente.	Todos los filtros se eliminan. Por lo tanto, todos los filtros deben desaparecer de la tabla de filtros y todas las refactorizaciones deben dejar de estar ocultas y volver a sus categorías. Posteriormente el botón de eliminar todos los filtros debe desactivarse dado que no hay filtros a desactivar.

Tabla 46: Pruebas de RF 6: Eliminar todos los filtros de refactorizaciones

9.7. RF 7: Seleccionar opción ver refactorizaciones filtradas

Requisito	Aspectos a probar	Resultado esperado
RF 7: Seleccionar opción ver refactorizaciones filtradas	Activar el checkbox de mostrar refactorizaciones filtradas funciona correctamente.	Todas las refactorizaciones que estaban ocultas, pasan a mostrarse dentro de un grupo denominado "Filtradas". Dentro del grupo de "Filtradas" las refactorizaciones deben aparecer también categorizadas por la clasificación seleccionada y deben atenuarse para contrastarlas con las no filtradas.
	Desactivar el checkbox de mostrar refactorizaciones filtradas funciona correctamente.	Las refactorizaciones que pertenecían al grupo "Filtradas" deben pasar a estar ocultas.

Tabla 47: Pruebas de RF 7: Seleccionar opción ver refactorizaciones filtradas

9.8. RF 8: Visualizar detalle refactorización

Requisito	Aspectos a probar	Resultado esperado
RF 8: Visualizar detalle refactorización	Al seleccionar una refactorización sin ejemplos, ni imagen se muestra correctamente.	Se muestra en un panel toda la información de la refactorización organizada por pestañas. Puesto que la refactorización seleccionada no tiene imagen ni ningún ejemplo las pestañas de imagen y ejemplos no se muestran.
	Al seleccionar una refactorización con algún ejemplo o con una imagen los detalles de esta se muestran correctamente.	Se muestran las pestañas comunes, más las pestañas de imagen y ejemplos.
	Al seleccionar una refactorización con ejemplos o con una imagen cuyo fichero no puede ser encontrado se advierte al usuario.	Se muestran las pestañas comunes y deberá saltar un diálogo de error para indicar que no se ha encontrado un fichero.

Tabla 48: Pruebas de RF 8: Visualizar detalle refactorización

9.9. RF 9: Añadir clasificación

Requisito	Aspectos a probar	Resultado esperado
RF 9: Añadir clasificación	La clasificación se crea correctamente.	La clasificación es creada correctamente y añadida al catálogo de clasificaciones. La clasificación debe ser creada como clasificación de usuario y por tanto se mostrará con el icono de editable.
	No se permite crear una clasificación con el nombre de otra ya existente.	Cuando el usuario está creando una clasificación e intenta asignarle el nombre de una clasificación ya existente se le muestra un mensaje de que la clasificación ya existe y no se le permite proceder a la creación.
	El usuario cancela la acción de crear una clasificación.	No se realiza ninguna modificación sobre el catálogo de clasificaciones.

Tabla 49: Pruebas de RF 9: Añadir clasificación

9.10. RF 10: Editar clasificación

Requisito	Aspectos a probar	Resultado esperado
RF 10: Editar clasificación	La modificación de la descripción se realiza correctamente.	Se asigna la nueva descripción a la clasificación y esto se refleja en la interfaz.
	El renombrado de la clasificación se realiza correctamente.	Se asigna el nuevo nombre a la clasificación y esto se refleja en la interfaz.
	No se permite asignar a una clasificación el nombre de otra ya existente al renombrar.	Cuando el usuario está renombrando una clasificación e intenta asignarle el nombre de una clasificación ya existente se le muestra un mensaje de que la clasificación ya existe y no se le permite proceder al renombrado.
	Una clasificación unicategoría se convierte en clasificación multicategoría correctamente.	Se asigna el tipo unicategoría a la clasificación y esto se refleja en la interfaz.
	Una clasificación multicategoría sin refactorizaciones en varias categorías se convierte en clasificación unicategoría correctamente.	Se asigna el tipo multicategoría a la clasificación y esto se refleja en la interfaz.
	No se permite convertir una clasificación multicategoría con refactorizaciones en varias categorías en clasificación unicategoría.	La opción de convertir la clasificación a tipo unicategoría en este caso debe aparecer desactivada.

Tabla 50: Pruebas de RF 10: Editar clasificación

9.11. RF 11: Eliminar clasificación

Requisito	Aspectos a probar	Resultado esperado
RF 11: Eliminar clasificación	La clasificación se elimina correctamente.	Antes de eliminar la clasificación se advierte al usuario de las consecuencias de la acción. Si el usuario decide proseguir se elimina la clasificación y se elimina la pertenencia a las categorías de la clasificación para todas las refactorizaciones. A continuación se actualiza la interfaz del editor.
	El usuario cancela el proceso de eliminar la clasificación.	No se realiza ninguna modificación sobre el catálogo de clasificaciones.

Tabla 51: Pruebas de RF 11: Eliminar clasificación

9.12. RF 12: Añadir categoría a una clasificación

Requisito	Aspectos a probar	Resultado esperado
RF 12: Añadir categoría a una clasificación	La categoría se crea correctamente.	La categoría es creada correctamente y añadida a la clasificación.
	No se permite crear una categoría con el nombre de otra ya existente en la clasificación seleccionada.	Cuando el usuario está creando una categoría e intenta asignarle el nombre de otra categoría ya existente se le muestra un mensaje de que la categoría ya existe y no se le permite proceder a la creación.
	El usuario cancela la acción de crear una clasificación.	No se realiza ninguna modificación sobre el catálogo de clasificaciones.

Tabla 52: Pruebas de RF 12: Añadir categoría a una clasificación

9.13. RF 13: Renombrar categoría de una clasificación

Requisito	Aspectos a probar	Resultado esperado
RF 13: Renombrar categoría de una clasificación	El renombrado de la categoría se realiza correctamente.	Se asigna el nuevo nombre a la categoría y esto se refleja en la interfaz.
	No se permite asignar a una categoría el nombre de otra ya existente en la clasificación al renombrar.	Cuando el usuario está renombrando una categoría e intenta asignarle el nombre de otra categoría ya existente se le muestra un mensaje de que la categoría ya existe y no se le permite proceder al renombrado.
	El usuario cancela la acción de renombrar la clasificación.	No se realiza ninguna modificación sobre el catálogo de clasificaciones.

Tabla 53: Pruebas de RF 13: Renombrar categoría de una clasificación

9.14. RF 14: Eliminar categoría de una clasificación

Requisito	Aspectos a probar	Resultado esperado
RF 14: Eliminar categoría de una clasificación	La categoría se elimina correctamente.	Antes de eliminar la categoría se advierte al usuario de las consecuencias de la acción. Si el usuario decide proseguir se elimina la categoría y se elimina la pertenencia a las categoría para todas las refactorizaciones. A continuación se actualiza la interfaz del editor.
	El usuario cancela el proceso de eliminar la categoría.	No se realiza ninguna modificación sobre el catálogo de clasificaciones.

Tabla 54: Pruebas de RF 14: Eliminar categoría de una clasificación

9.15. RF 15: Mostrar resumen elemento seleccionado

Requisito	Aspectos a probar	Resultado esperado
RF 15: Mostrar resumen elemento seleccionado	Si existe documentación para el elemento su descripción se muestra correctamente.	En el panel de resumen del elemento se muestra la descripción del elemento.
	Si no existe documentación para el elemento.	Se muestra un mensaje indicando al usuario que no existe documentación para el elemento seleccionado.

Tabla 55: Pruebas de RF 15: Mostrar resumen elemento seleccionado

9.16. RF 16: Realizar búsqueda de elementos

Requisito	Aspectos a probar	Resultado esperado
RF 16: Realizar búsqueda de elementos	Búsqueda de un elemento por su término exacto.	En el panel de elementos se deberá mostrar exclusivamente el único elemento que coincide con el patrón buscado.
	Búsqueda de un elemento coincidente en su raíz.	Se deberá mostrar exclusivamente el único elemento que coincide en su raíz con el patrón buscado, igual que en el caso anterior.
	Búsqueda con múltiples resultados concordantes.	Se muestran todos los resultados concordantes por orden de concordancia.
	Búsqueda sin resultados concordantes.	No se muestra ningún elemento.
	Mostrar todos los elementos buscando "*" o vacío.	Se muestran todos los elementos existentes.

Tabla 56: Pruebas de RF 16: Realizar búsqueda de elementos

10. SELECCIÓN DE MÉTRICAS

Con la finalidad de obtener una adecuada evaluación de la calidad del producto software, conviene no solo conformarse con la consecución de los objetivos definidos en la especificación del proyecto sino que estos se hayan logrado construyendo un producto de calidad cuya arquitectura haya sido diseñada e implementada dentro de unos intervalos establecidos, con anterioridad al desarrollo, para una serie de parámetros elegidos.

Para la evaluación del presente proyecto, se han tomado en consideración las métricas establecidas en la Universidad de Burgos para la evaluación de los proyectos fin de carrera desarrollados en la asignatura Sistemas Informáticos del último curso de Ingeniería Informática. En la página dedicada a esta asignatura¹, durante los últimos años se viene realizando un proceso de análisis de proyectos entregados. Entre otros aspectos, se detallan los resultados obtenidos en la evaluación de estos proyectos para un conjunto de métricas de código establecido.

Para poder evaluar el nivel de idoneidad de los diseños realizados así como de las correspondientes implementaciones, se han determinado intervalos de valores esperados para algunas de las métricas más importantes. Dichos valores se han fijado a partir de las recomendaciones de los propios autores de las métricas y de las herramientas de medida.

Sin pretender hacer de ello un análisis exhaustivo del proyecto pero si con el objetivo de que nos permita tener una referencia general del mismo, en primer lugar se van a tomar unas métricas que nos van a servir como indicadoras del tipo de proyecto que se trata, en cuanto a lenguaje de programación y tamaño del mismo se refiere, para posteriormente centrarnos en una serie de métricas de código asociadas a valores medios y otras asociadas a valores máximos o mínimos. Estas últimas entendidas más bien como orientadas a la mejora de casos concretos dentro del código que a la evaluación del propio producto a nivel general, es por ello que no serán interpretadas pero si se realizará una representación a través de gráficas de kiviat de las mismas.

Por otra parte, cabe destacar que todo producto software puede presentar ciertas desviaciones respecto a los valores esperados en función de aspectos tales como son su naturaleza, la tecnología utilizada, el tipo de interfaz, etc.

A continuación se detallan las métricas que utilizaremos, muchas de ellas se han

¹ <http://pisuerga.inf.ubu.es/lsl/Asignaturas/SI>

tomado del documento [López, Rodríguez, & Marticorena n.d.]. Este documento se encuentra disponible en la carpeta Documentación\Documentación Adicional del CD que se suministra con el proyecto:

Lenguaje de programación

Lenguaje de programación en el que se ha implementado el producto software.

Líneas de código

Total de líneas de código del producto software.

Número de sentencias

Número total de sentencias del producto software.

Porcentaje de sentencias condicionales

Porcentaje de sentencias condicionales del producto software, relación entre el número total de sentencias y el número de sentencias condicionales. Es un claro indicador de la complejidad del producto software.

Número de llamadas a métodos

Número medio de métodos que tienen posibilidad de ejecutarse como respuesta a un mensaje que recibe un objeto de cada clase. Es un indicador de la respuesta que tiene el conjunto de clases que compone el producto software. Un valor elevado puede ser signo de ser un software propenso a errores y más difícil de comprender y de probar.

Porcentaje de líneas de comentarios

Porcentaje de líneas de comentarios del producto software, relación entre el número total de líneas de código y el número de líneas de comentarios. Es un claro indicador de la densidad de comentarios en el producto software.

Intervalo de valores recomendados: [8, 22]

Número de clases e interfaces

Número total de clases e interfaces del producto software.

Número de métodos por clase

Número medio de métodos por clase del producto software. Por un lado, para esta métrica es deseable un número elevado desde el punto de vista de la reutilización del código. Sin embargo, una cifra demasiado elevada es signo de mala distribución de responsabilidades ya que existiría una tendencia a clases demasiado largas y por lo tanto a dificultar la extensibilidad. Es un claro indicador de la responsabilidad de una clase en el producto software.

Intervalo de valores recomendados: [4, 16]

Media de sentencias por método

Número medio de sentencias por método del producto software. El tamaño medio de los métodos puede ser un indicador de diseños pobres en lo que a la orientación a objetos se refiere, ya que un número alto en esta métrica representa una tendencia a desarrollar métodos demasiado largos lo que podría ser signo de un desarrollo orientado más a la función que al objeto y más difícil de comprender ya que probablemente se traduzca en una mayor complejidad del mismo.

Intervalo de valores recomendados: [6, 12]

Media de profundidad de bloques

Número medio de profundidad de bloques del producto software. La profundidad de bloque mide la profundidad del anidamiento condicional en los métodos. Si tiene valores elevados, será señal de que existen complejas estructuras de control de flujo en el programa, lo que da lugar a código difícil de comprender, de reutilizar y mantener, y altamente propenso a errores.

Intervalo de valores recomendados: [1, 2.2]

Media de complejidad

Número medio de complejidad a nivel de método del producto software, también llamado complejidad ciclomática. Es una medida de la complejidad global evaluada para cada método. Valores altos indican métodos con demasiadas responsabilidades y control de flujo complejo. Serán difíciles de entender, reutilizar y mantener.

Intervalo de valores recomendados: [2, 4]

Máxima profundidad de bloques

Número máximo de profundidad de bloques del producto software.

Intervalo de valores recomendados: [3, 7]

Complejidad máxima

Número máximo de complejidad global evaluada para cada método del producto software, también llamado complejidad ciclomática.

Intervalo de valores recomendados: [2, 8]

11. ENTORNO TECNOLÓGICO DE LA APLICACIÓN

En el entorno tecnológico de la aplicación software desarrollada durante el proyecto, Dynamic Refactoring Plugin, se puede distinguir entre requerimientos software y los hardware, que a continuación se detallan.

Los requerimientos software, es decir, las características que debe tener el software instalado en un equipo para poder soportar y ejecutar nuestra aplicación son los siguientes:

- instalación de la herramienta *JRE (Java Runtime Environment)*. No obstante como el objetivo final es el desarrollo de aplicaciones, para ello se requerirá la instalación de la herramienta de desarrollo *JDK (Java Development Kit)*, la cual incluye la de ejecución, en su versión 1.6.0 o superior.
- instalación del entorno de desarrollo o *IDE (Integrated Development Environment)* Eclipse, para el cual ha sido desarrollado el plugin y que es imprescindible para la utilización del mismo. Se recomienda el uso de una versión de Eclipse igual o superior a la versión 3.5, Eclipse Galileo.

En cuanto a los requerimientos hardware, la aplicación en si misma no necesita de requisitos que no pueda cumplir cualquier equipo actual de nivel medio en cuanto a prestaciones y características se refiere. Aunque si bien es verdad, para un funcionamiento óptimo del plugin es recomendable disponer de buena capacidad de procesamiento y memoria RAM necesarios en los procesos internos del mismo, así como por parte del interprete de Java y del propio Eclipse. Por tanto, se puede establecer como requisitos mínimos un procesador de 1 Ghz y 1 GB de memoria RAM para versiones de 32 bits, y 2 GB para versiones de 64 bits.

12. PLAN DE DESARROLLO E IMPLANTACIÓN

Hasta ahora en este anexo nos hemos centrado en describir el diseño lógico del proyecto. A continuación brevemente presentaremos los módulos físicos necesarios para la instalación y ejecución de nuestra aplicación. Para ello, nos basaremos en los diagramas de componentes y de despliegue.

12.1. Diagrama de componentes

En el siguiente diagrama se muestran los ejecutables necesarios además de los correspondientes al plugin de refactorizaciones, **dynamicrefactoring.plugin_3.0.8.jar** y **dynamicrefactoring.feature_3.0.8.jar**.

Dependencias con plugins de Eclipse

- **org.eclipse.core.databinding**: proporciona clases que se pueden utilizar para sincronizar el estado entre pares de objetos observables enlazados. Por ejemplo, los componentes de interfaz de usuario y los objetos del modelo.
- **org.eclipse.core.databinding.beans**: proporciona clases que se pueden utilizar para observar los objetos que se ajusten a la especificación JavaBean.
- **org.eclipse.core.expressions**: proporciona la *API* y las clases de implementación para definir un lenguaje unificado XML de expresión, el cual consiste en un conjunto de etiquetas predefinidas que representan expresiones.
- **org.eclipse.core.filesystem**: especifica la API para interactuar con el sistema de archivos de plugins, pudiendo ser utilizado este para la consulta y la manipulación de los archivos almacenados asociados a los mismos.
- **org.eclipse.core.resources**: proporciona el soporte básico para manejar el *workspace* y sus *recursos*.
- **org.eclipse.core.runtime**: proporciona el núcleo que soporta la ejecución de los *plugins*, el registro de *plugins* y todas las operaciones relacionadas con la ejecución de la plataforma en general.
- **org.eclipse.help**: proporciona el soporte central para el sistema de ayuda.
- **org.eclipse.jdt.core**: contiene las clases del modelo Java, las cuales implementan los comportamientos específicos Java para los recursos y la descomposición de recursos Java en elementos del modelo.
- **org.eclipse.jdt.launching**: módulo para la programación de interfaces que permite la interacción con el soporte de despliegue de Eclipse. Proporciona soporte para describir los entornos de ejecución de Java instalados y

desplegar sus máquinas virtuales de Java.

- **org.eclipse.jdt.ui:** módulo para la programación de interfaces para la interacción con la interfaz de usuario de Eclipse en relación con proyectos Java. Este paquete proporciona clases de soporte para presentar elementos Java en la interfaz de usuario.
- **org.eclipse.jface.databinding:** *proporciona* un que conecta el modelo de dominio y la interfaz de usuario y proporciona una API para su correcta utilización.
- **org.eclipse.jface.text:** proporciona un *framework* para la creación, manipulación, visualización y edición de documentos de texto.
- **org.eclipse.team.core:** proporciona la API para definir y trabajar con proveedores de repositorio.
- **org.eclipse.ui:** contiene el soporte para la programación y ejecución de interfaces gráficas y permite extender la interfaz de usuario de la plataforma Eclipse.
- **org.eclipse.ui.editors:** se trata de un punto de extensión para el *workbench* ya que permite a los *plugins* diseñar nuevos editores con el fin de añadir a estos al mismo.
- **org.eclipse.ui.forms:** proporciona un conjunto de *widgets* basados en formularios para su uso en vistas, editores y asistentes, así como otras clases de utilidad.
- **org.eclipse.ui.workbench.texteditor:** proporciona un *framework* para los editores de texto.

Bibliotecas auxiliares

- **commons-io-2.0.jar:** proporciona una extensión a la funcionalidad de los paquetes de Entrada/Salida que nos ofrece la plataforma J2SE.
- **guava-r07.jar:** proporciona distintos paquetes que contienen clases de utilidad para trabajar con entrada y salida, datos primitivos y para la programación concurrente y lo equivalente a la Google Collections

Library.

- **j2h.jar:** permite convertir ficheros fuente Java a ficheros HTML.
- **javymoon-2.6.1.jar:** proporciona las clases que permiten generar los modelos a partir de clases Java y también las clases que son necesarias para la recuperación de código a partir de un modelo MOON. Constituye, en definitiva, la extensión de MOON para el lenguaje Java.
- **jdom.jar:** permite leer, escribir, crear y manipular ficheros XML cómodamente desde Java.
- **jgraph-jdk1.5.jar:** se centra en las estructuras de datos proporcionando objetos de teoría de grafos y algoritmos.
- **jsoup-1.5.2.jar:** proporciona una API muy conveniente para la extracción y manipulación de documentos HTML.
- **log4j-1.2.15.jar:** proporciona un API para manejar el registro -log- de operaciones en los programas.
- **lucene-core-3.0.3.jar:** ofrece un API para la recuperación de información que permite agregarle funcionalidades de búsqueda e indexación a las aplicaciones.
- **lucene-snowball-3.0.3.jar:** ofrece una buena colección de stemmers del algoritmo Snowball para varios idiomas.
- **moon-2.4.2.jar:** contiene las clases que componen un modelo MOON minimal. Sus elementos mantienen la independencia del lenguaje y son los que permiten representar las clases, los métodos, etc. en un modelo orientado a objetos genérico.
- **refactoringengine-1.1.1.jar:** contiene las clases fundamentales del motor de refactorizaciones.
- **swtdesigner.jar:** proporciona clases de utilidad para el manejo de recursos del sistema operativo asociados a controles SWT, tales como: colores, fuentes, imágenes, etc.

Diagrama de Componentes

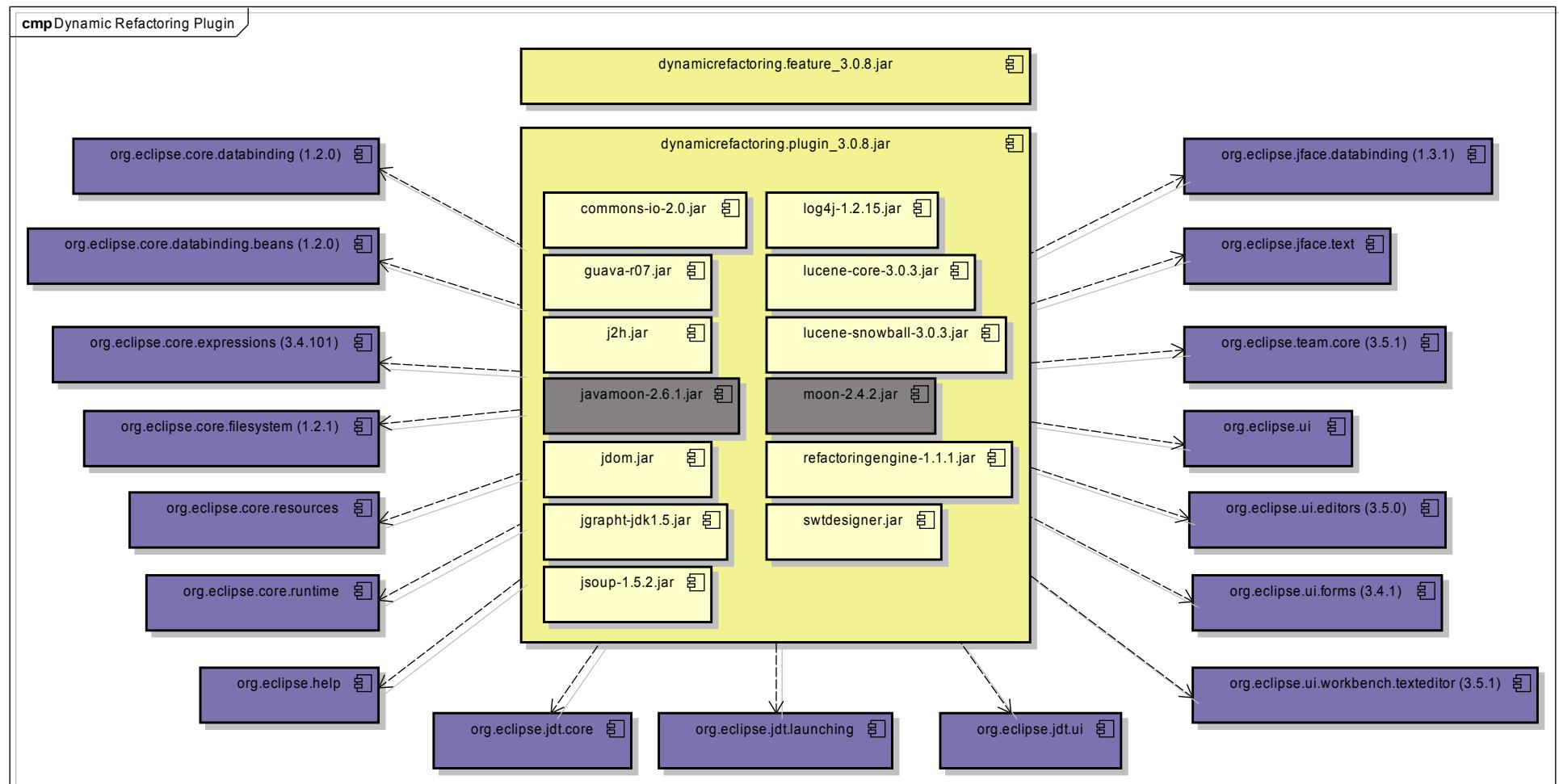


Ilustración 86: Diagrama de componentes

Leyenda:

Dynamic Refactoring Plugin

Bibliotecas auxiliares

Plugins Eclipse

Bibliotecas MOON y JavaMOON

12.2. Diagrama de despliegue

En este diagrama se muestra la distribución de los módulos necesarios para un funcionamiento correcto. En la estación que se ejecute, además de los componentes detallados en el apartado anterior el usuario deberá contar con el entorno de ejecución de Java (*JRE*) en su versión 1.6.0 y Eclipse.

Diagrama de Despliegue

A continuación se presenta el diagrama de despliegue.

La leyenda para el mismo es la siguiente:

Leyenda:	
Dynamic Refactoring Plugin	Bibliotecas auxiliares
Eclipse y plugins Eclipse	Bibliotecas MOON y JavaMOON
Entorno de ejecución	

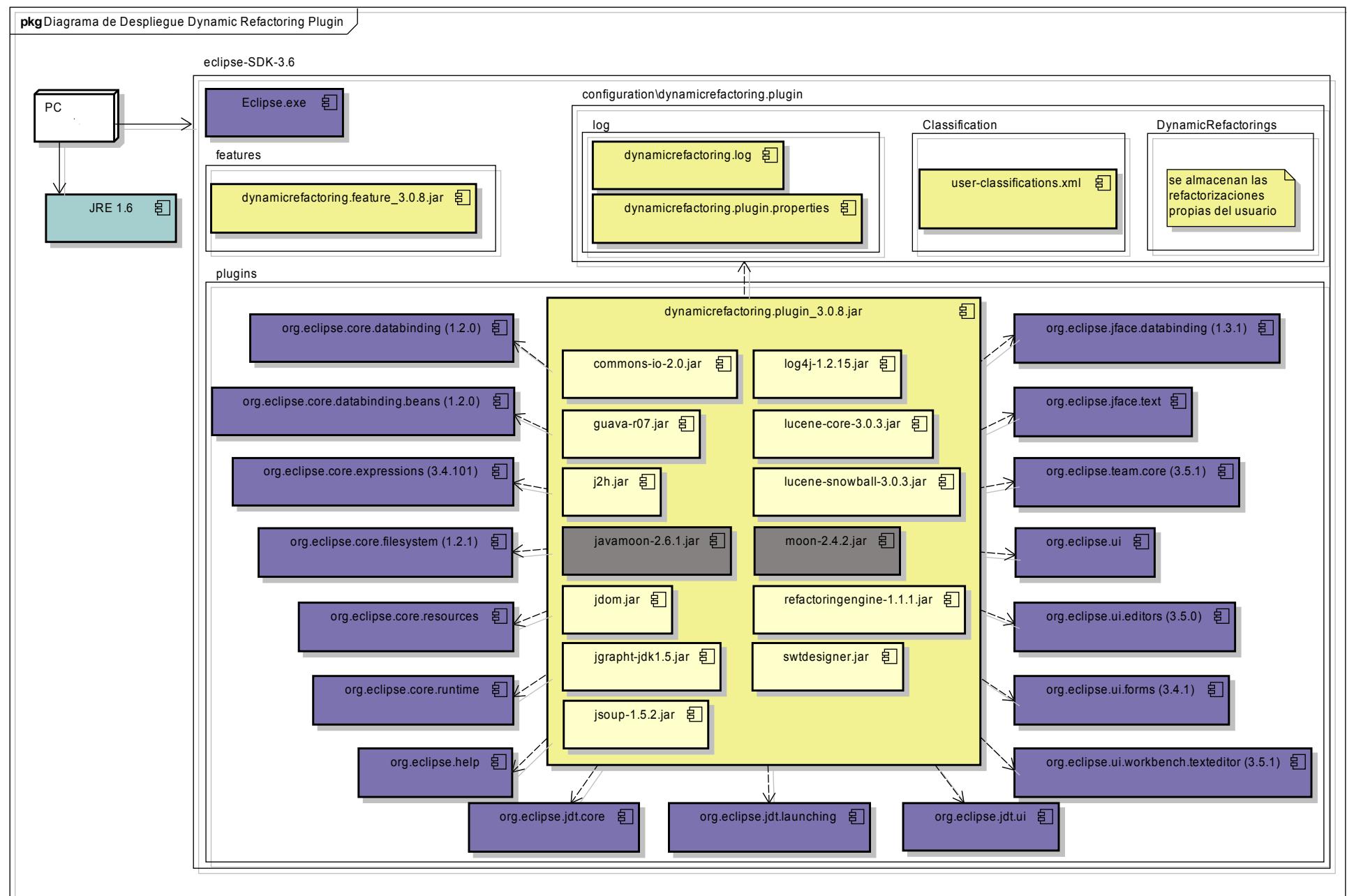


Ilustración 87: Diagrama de despliegue

Universidad de Burgos
ESCUELA POLITÉCNICA SUPERIOR
INGENIERÍA INFORMÁTICA



Anexo 4. Documentación Técnica del Programador

Clasificación de Refactorizaciones
Dynamic Refactoring Plugin 3.0

Alumnos:
Míryam Gómez San Martín
Íñigo Mediavilla Saiz

Tutor:
Raúl Marticorena Sánchez

Burgos, Julio de 2011

ANEXO IV
DOCUMENTACIÓN TÉCNICA DEL
PROGRAMADOR

ÍNDICE DE CONTENIDO

Anexo IV	
Documentación Técnica del Programador.....	317
Lista de cambios.....	323
1. INTRODUCCIÓN.....	325
2. MANUAL DEL PROGRAMADOR.....	325
2.1. Contenido del CD.....	325
2.2. Documentación de bibliotecas.....	328
2.2.1. Dependencia con plugins Eclipse.....	328
2.2.2. Bibliotecas auxiliares.....	336
2.3. Instalación de herramientas.....	344
2.3.1. Instalación Java Development Kit.....	344
2.3.2. Instalación Eclipse.....	345
2.3.3. Instalación Maven 3.....	347
2.3.4. Instalación Git.....	349
2.3.5. Instalación del plugin de Eclipse para Fogbugz -FogLyn.....	351
2.3.6. Instalación SWTBot.....	351
2.3.7. Instalación Sonar.....	353
2.3.8. Instalación Hudson.....	354
2.4. Desarrollo plugin Eclipse.....	355
2.4.1. Establecer la plataforma de desarrollo del plugin.....	355
2.4.2. Proyectos existentes en el código fuente.....	357
2.4.3. Ficheros POM.....	359
2.5. Maven.....	361
2.5.1. Pasos previos a la construcción.....	361
2.5.2. Paso 1 – Limpiar.....	362
2.5.3. Paso 2 – Compilar.....	362
2.5.4. Paso 4 – Generar empaquetado.....	363
2.5.5. Paso 5 – Ejecutar Test de Integración.....	364
2.5.6. Paso 6 – Instalar.....	364
2.5.7. Ciclo completo.....	364
2.5.8. Artefactos generados tras la construcción del proyecto.....	367
2.6. Integración bibliotecas MOON y JavaMOON.....	369
3. ASIGNAR INFORMACIÓN DE MARCA AL PLUGIN.....	373
4. FIRMA PLUGIN.....	376
4.1. Pasos.....	376
5. PRUEBAS.....	379
5.1. Pruebas Unitarias.....	379
5.2. Pruebas de interfaz.....	383
5.3. Pruebas de Cobertura.....	384

5.3.1. Paquetes del dominio.....	385
5.3.2. Paquetes de persistencia XML.....	386
5.3.3. Paquetes de la interfaz.....	386
5.3.4. Resto de paquetes.....	387
5.3.5. Análisis global.....	388
6. MÉTRICAS.....	389
6.1. Evaluación de métricas.....	389
6.2. Resultados obtenidos.....	390
7. INTERNACIONALIZACIÓN.....	401
8. AYUDA EN ECLIPSE.....	402

ÍNDICE DE ILUSTRACIONES

Ilustración 88: Estructura de carpetas del CD.....	327
Ilustración 89: Estructura de carpetas del CD II.....	328
Ilustración 90: MANIFEST.MF.....	329
Ilustración 91: Diagrama de clases de refactoring.engine.....	342
Ilustración 92: Selección espacio de trabajo Eclipse.....	346
Ilustración 93: Bienvenida Eclipse.....	346
Ilustración 94: Instalación de SWTBot para Eclipse.....	352
Ilustración 95: Ejecutar como tests de interfaz gráfica con SWTBot.....	353
Ilustración 96: Establecer la plataforma objetivo para iniciar el desarrollo.....	356
Ilustración 97: Portada de las métricas de un proyecto con Sonar.....	365
Ilustración 98: Página de HotSpots.....	366
Ilustración 99: Pantalla Time Machine en Sonar.....	367
Ilustración 100: Añadir nuevas bibliotecas a Java Build Path.....	370
Ilustración 101: Modificación Classpath plugin.xml.....	371
Ilustración 102: MANIFEST.MF.....	371
Ilustración 103: build.properties.....	372
Ilustración 104: Constantes de dynamicrefactoring.RefactoringConstants.....	372
Ilustración 105: Diálogo About Features.....	374
Ilustración 106: Diálogo About Eclipse.....	374
Ilustración 107: Asignar descripción de la característica, licencia y copyright.....	375
Ilustración 108: Tabla métricas.....	391
Ilustración 109: Comparativa para los subsistemas Completo y Plugin.....	391
Ilustración 110: Comparativa para los subsistemas Parte Operativa y Interfaz Gráfica	391
Ilustración 111: Gráfica comparativa de métricas de la aplicación completa.....	393
Ilustración 112: Grafo de Kiviat para el proyecto completo.....	394
Ilustración 113: Autoevaluación métricas para proyecto completo.....	394
Ilustración 114: Gráfica comparativa de métricas del plugin.....	395
Ilustración 115: Grafo de Kiviat para el plugin.....	395
Ilustración 116: Autoevaluación métricas para el plugin.....	396
Ilustración 117: Gráfica comparativa de métricas de la parte operativa.....	397
Ilustración 118: Grafo de Kiviat para la parte operativa.....	397
Ilustración 119: Autoevaluación métricas para la parte operativa.....	398
Ilustración 120: Gráfica comparativa de métricas de la interfaz gráfica.....	399
Ilustración 121: Grafo de Kiviat para la interfaz gráfica.....	399
Ilustración 122: Autoevaluación métricas para interfaz gráfica.....	400
Ilustración 123: Asistente de externacionalización de cadenas.....	401

ÍNDICE DE TABLAS

Tabla 57: Parámetros necesarios para la herramienta keytool.....	377
Tabla 58: Coberturas de los nuevos paquetes añadidos al dominio.....	385
Tabla 59: Cobertura de los paquetes de persistencia XML.....	386
Tabla 60: Cobertura de los nuevos paquetes de la interfaz.....	387
Tabla 61: Evolución de la cobertura de los paquetes de la interfaz.....	387
Tabla 62: Evolución de la cobertura del resto de paquetes.....	388

LISTA DE CAMBIOS

Número	Fecha	Descripción	Autor/es
0	08/02/11	Creada sección del manual del programador.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
1	22/02/11	Correcciones de formato. Explicación de instalación de Maven.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
2	28/02/11	Añadidos prerequisitos para la construcción y lista de artefactos generados por el proceso.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
3	28/02/11	Agregado explicación de desde dónde y cómo arrancar el proceso de construcción.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
4	23/02/11	Añadida instalación JDK, Eclipse Helios. Añadida integración bibliotecas MOON y JavaMOON	Míryam Gómez San Martín Íñigo Mediavilla Saiz
5	23/05/11	Añadida configuración de target platform, descripción de proyectos y de ficheros pom.xml	Míryam Gómez San Martín Íñigo Mediavilla Saiz
6	30/05/11	Agregadas firma del plugin, pruebas, instalación de herramientas e internacionalización del plugin.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
7	07/06/11	Incluido apartado referente a métricas y contenido del CD.	Míryam Gómez San Martín Íñigo Mediavilla Saiz

1. INTRODUCCIÓN

Este documento se ha realizado con el objetivo de servir de ayuda al programador en las distintas tareas relacionadas con el desarrollo del proyecto.

Entre los aspectos a tratar se encuentra el contenido del CD, las bibliotecas de las que depende el plugin así como otras bibliotecas utilizadas, el manual del programador y la ejecución de pruebas que comprueban el correcto funcionamiento de la aplicación. Además, se incluye la cobertura de las mismas y los valores obtenidos para las diferentes métricas que han sido objeto de estudio.

2. MANUAL DEL PROGRAMADOR

Este apartado esta dedicado a explicar el contenido del CD, las bibliotecas utilizadas así como la instalación de aquellas herramientas necesarias para el desarrollo del producto final. Además se explicará la forma de compilar y ejecutar tanto la aplicación como las pruebas realizadas para comprobar su correcto funcionamiento, así como el procedimiento para crear el ejecutable de la aplicación.

2.1. Contenido del CD

En este apartado se hará una breve descripción de cada una de las partes que conforman la estructura del CD, así como de su contenido. Para mayor detalle acuda al propio CD donde podrá observar su contenido por si mismo.

A continuación detallaremos cada uno de los directorios que componen la estructura del CD, y para finalizar mostraremos de forma gráfica la estructura del mismo.

Documentación

Contiene toda la documentación necesaria para la comprensión del proyecto.

- Documentación Adicional

Se ha considerado oportuno añadir documentación adicional, entre esta se encuentra un fichero con la *URL* del repositorio de descarga del plugin, y documentación acerca de la metodología ágil *SCRUM* y las técnicas *eXtreme*.

Programming para ampliar conocimientos. Además también se ha añadido un documento que ha servido de guía para la selección de métricas.

- Documentación Técnica

Contiene toda la documentación técnica del proyecto.

- cobertura: informes de cobertura obtenidos mediante la herramienta JoCoCo.
- CTTE: fichero que contiene los diagramas *CTTE*, Concur Task Trees Environment.
- doccheck: documentos generados del chequeo de la documentación del código fuente, en este caso la herramienta utilizada ha sido doclet DocCheck.
- javadoc: documentación del código fuente en formato `HTML`, para su realización ha sido utilizada la herramienta JavaDoc.
- métricas: informes de métricas, los cuales han sido generados por la herramienta SourceMonitor.
- sonar: informes que se han considerado oportunos incluir, los cuales han sido generados por la herramienta Sonar.

- Memoria

Contiene la documentación relativa a la memoria del proyecto.

- odt: documentación formato `odt`, OpenOffice.
- pdf: documentación formato `pdf`.

- Presentación

Contiene el conjunto de diapositivas utilizadas para la presentación del proyecto.

- Videos

Contiene video-tutoriales que servirán de guía al usuario.

Ejecutable

- Dynamic Refactoring Plugin

Contiene la versión final del plugin distribuible e instalable de forma manual, junto con el paquete de internacionalización al castellano.

- Tarea Ant RefactoringPlan

Contiene los recursos necesarios para la ejecución de la tarea Ant.

Producto

- dynamicrefactoring.plugin

Contiene todos los ficheros del proyecto Eclipse con que se ha desarrollado el plugin y el paquete de idioma. Además del contenido del plugin instalable, incluye ficheros fuente y ficheros de test. Es el directorio que se encuentra más orientado a los programadores.

- RefactoringPlanTask

Contiene los ficheros fuentes de la tarea Ant desarrollada, aunque esta no ha sido modificada en el presente proyecto se ha querido incluir para que quede recogida en el mismo.

Software

Contiene todas las herramientas gratuitas que se han utilizado para generar y desarrollar el producto final. Estas herramientas están distribuidas por las diferentes subcarpetas que contiene dependiendo del ámbito en el que se utilicen, desarrollo, diagramas, edición, java, métricas y test.

Vista general del CD:

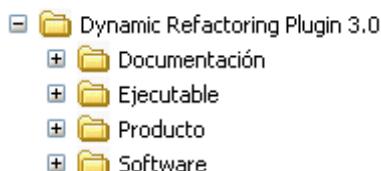


Ilustración 88: Estructura de carpetas del CD

Vista expandida del CD:

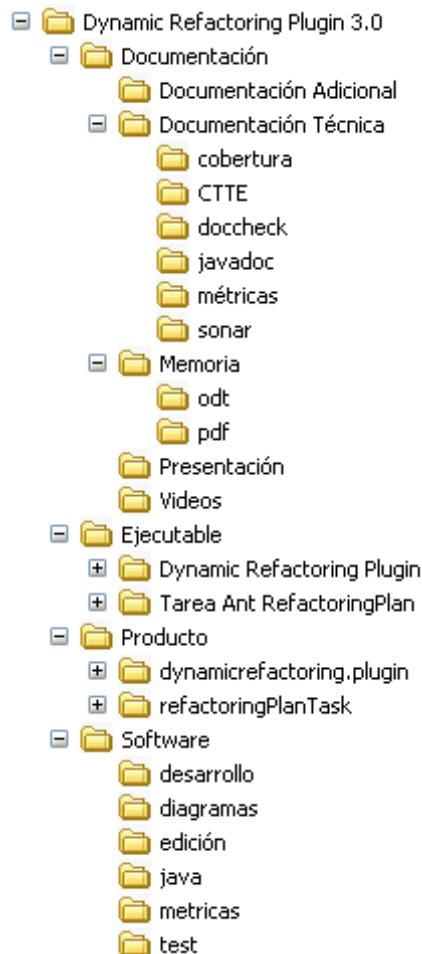


Ilustración 89: Estructura de carpetas del CD II

2.2. Documentación de bibliotecas

A continuación se detallan las librerías y componentes utilizados en el proyecto que han sido desarrolladas por terceros.

2.2.1. Dependencia con plugins Eclipse

Los *plugins* de Eclipse son módulos de software que proporcionan cierta funcionalidad que otros a su vez pueden usar posteriormente. Si el *plugin* A requiere del *plugin* B para su funcionamiento, se dice que A depende de B. Esta dependencia implica que hasta que no se

haya cargado correctamente el *plugin* B, A no funcionará. También puede ocurrir que B dependa de otros *plugins* y éstos a su vez de otros, así sucesivamente. Habrá por tanto una cadena de dependencias que llevará a un amplio conjunto de *plugins* relacionados. No hay duda de que, si la carga de algún *plugin* de la cadena fallara, los que dependen de él tendrían problemas en su funcionamiento. Aquí entra en juego el fichero MANIFEST.MF que a continuación será detallada su importancia. Este es el del presente proyecto:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Dynamic Refactoring Plug-in
Bundle-SymbolicName: dynamicrefactoring.plugin;singleton:=true
Bundle-Version: 3.0.8
Bundle-Activator: dynamicrefactoring.RefactoringPlugin
Require-Bundle: org.eclipse.core.resources,
    org.eclipse.ui,
    org.eclipse.core.runtime,
    org.eclipse.jface.text,
    org.eclipse.jdt.core,
    org.eclipse.jdt.ui,
    org.eclipse.jdt.launching,
    org.eclipse.help,
    org.eclipse.ui.forms;bundle-version="3.4.1",
    org.eclipse.ui.workbench.texteditor;bundle-version="3.5.1",
    org.eclipse.ui.editors;bundle-version="3.5.0",
    org.eclipse.core.expressions;bundle-version="3.4.101",
    org.eclipse.core.databinding;bundle-version="1.2.0",
    org.eclipse.core.databinding.beans;bundle-version="1.2.0",
    org.eclipse.jface.databinding;bundle-version="1.3.1",
    org.eclipse.team.core;bundle-version="3.5.1",
    org.eclipse.core.filesystem;bundle-version="1.2.1"
Bundle-ClassPath: ./temp,
    lib/log4j-1.2.15.jar,
    lib/jdom.jar,
    ,
    lib/swtdesigner.jar,
    lib/junit-4.4.jar,
    lib/refactoringengine-1.1.1.jar,
    lib/j2h.jar,
    lib/commons-io-2.0.jar,
    lib/javamoon-2.6.1.jar,
    lib/moon-2.4.2.jar,
    lib/guava-r07.jar,
    lib/jsoup-1.5.2.jar,
    lib/lucene-core-3.0.3.jar,
    lib/lucene-snowball-3.0.3.jar,
    lib/jgraphht-jdk1.5.jar
```

Ilustración 90: MANIFEST.MF

En el fichero `MANIFEST.MF` de cada *plugin* se encuentra una sección en la que se declaran dependencias o requerimientos de otros *plugins*. La cláusula `Require-Bundle` es la parte referente a las dependencias que tiene el *plugin* con respecto a otros *plugins*.

A continuación se describen brevemente cada una de estas dependencias.

org.eclipse.core.databinding (1.2.0)

Proporciona clases para enlazar objetos observables, por ejemplo los componentes de interfaz de usuario y los objetos del modelo.

Este paquete proporciona clases que se pueden utilizar para sincronizar el estado entre pares de objetos observables con conversión opcional de tipos de datos y validación.

Se dispone de un `DataBindingContext` que es utilizado para administrar una lista de enlaces, `Bindings`, con los resultados de su validación. Para cada enlace existen dos objetos, uno para cada sentido, `UpdateValueStrategy` o `UpdateListStrategy` que se utilizan para controlar cómo los enlaces se deben sincronizar, también pueden ser utilizados para especificar la conversión de tipos de datos y la validación.

org.eclipse.core.databinding.beans (1.2.0)

Proporciona clases que se pueden utilizar para observar los objetos que se ajusten a la especificación JavaBean. Mediante la clase `BeansObservables`, ofrece una fábrica para la creación de objetos observables de objetos Java que se ajustan a la especificación JavaBean.

org.eclipse.core.expressions (3.4.101)

Proporciona la *API* y las clases de implementación para definir un lenguaje unificado XML de expresión, el cual consiste en un conjunto de etiquetas predefinidas que representan expresiones. Este conjunto está abierto por lo que se puede ampliar con el fin de personalizar el lenguaje de expresión.

El lenguaje de expresión no es sólo para los puntos de extensión, sino que se puede

utilizar en cualquier documento XML que admitan expresiones.

org.eclipse.core.filesystem (1.2.1)

Especifica la *API* para interactuar con el sistema de archivos de plugins, pudiendo ser utilizado este para la consulta y la manipulación de los archivos almacenados asociados a los mismos.

org.eclipse.core.resources

Proporciona el soporte básico para manejar el *workspace* y sus *recursos*. Este *plugin* define las nociones de *workspace* (espacio de trabajo) y *resources* (recursos) en Eclipse. El modelo de recursos del *workspace* es muy similar a un sistema de ficheros. Todos los recursos se apoyan en un fichero o directorio real en el sistema de ficheros nativo. Se almacenan en su forma nativa usando sus nombres habituales.

Como un añadido al manejo básico de recursos, soporta varias operaciones relacionadas con el ciclo de vida del *workspace*, como la que permite guardar los cambios realizados sobre los recursos.

org.eclipse.core.runtime

Proporciona el núcleo que soporta la ejecución de los *plugins*, el registro de *plugins* y todas las operaciones relacionadas con la ejecución de la plataforma en general.

Este paquete especifica el *API* relacionado con la ejecución de la plataforma en sí misma. Incluye varias áreas, como la definición y el manejo de los *plugins* y su ciclo de vida, es decir, desde que la plataforma los arranca cuando son necesarios hasta que los detiene cuando dejan de serlo, o el mantenimiento de la plataforma. Como añadido, este *API* proporciona varios tipos de utilidades y varios tipos de monitores de progreso que permiten a *plugins* terceros integrar su funcionalidad con el núcleo de operaciones de Eclipse.

org.eclipse.help

Proporciona el soporte central para el sistema de ayuda. Gracias a este módulo es posible que *plugins* individuales generen ayuda en línea o módulos de ayuda contextual para sus componentes.

org.eclipse.jdt.core

El modelo Java es el conjunto de clases que modelan los objetos asociados con la creación, edición y construcción de un programa Java.

Este paquete contiene las clases del modelo Java, las cuales implementan los comportamientos específicos Java para los recursos y la descomposición de recursos Java en elementos del modelo. Se trata del equivalente para Eclipse del modelo MOON utilizado por nuestro *plugin*. Constituye un metamodelo que permite representar como una abstracción todos los elementos que constituyen un proyecto Java en Eclipse.

org.eclipse.jdt.launching

Módulo para la programación de interfaces que permite la interacción con el soporte de despliegue de Eclipse. Proporciona soporte para describir los entornos de ejecución de Java instalados y desplegar sus máquinas virtuales de Java.

org.eclipse.jdt.ui

Módulo para la programación de interfaces para la interacción con la interfaz de usuario de Eclipse en relación con proyectos Java. Este paquete proporciona clases de soporte para presentar elementos Java en la interfaz de usuario.

La clase `JavaUI` es el punto de acceso principal para los componentes de la interfaz de usuario de Java. Permite mediante programación la apertura de elementos Java en editores, así como otros tipos de operaciones relacionadas. El paquete contiene constantes para recuperar partes Java de la interfaz de usuario del registro del *workbench*. `JavaUI` proporciona acceso al `IWorkingCopyManager`, que gestiona las copias que están en uso de

unidades de compilación Java, es decir, una abstracción de un fichero de clase Java.

Las interfaces `ITypeHierarchyViewPart` e `IPackagesViewPart` definen la interfaz de programación para interactuar con el tipo jerarquía y los paquetes de vistas. Las clases de apoyo `JavaElementContentProvider` y `JavaElementLabelProvider` ayudan a presentar elementos de Java en vistas Jface.

org.eclipse.jface.databinding (1.3.1)

JFace Data Binding es un *framework* que conecta el modelo de dominio y la interfaz de usuario y proporciona una *API* para su correcta utilización. Ofrece la posibilidad de disponer de fábricas para crear objetos de la interfaz `IObservableValue`, la cual permite observar las propiedades de la interfaz de usuario y los objetos del modelo. Por ejemplo, la fábrica `WidgetProperties` permite crear `IObservableValues` de las propiedades de *widgets* `SWT` y la fábrica `BeanProperties` permite observar las propiedades de los `JavaBean`. En la interfaz de usuario se suele observar campos de texto, pero también se podría realizar lo propio por ejemplo con una propiedad de habilitado correspondiendo con un valor booleano del modelo.

Con el fin de convertir los tipos de datos utilizados en la interfaz de usuario a los utilizados en el modelo de dominio y viceversa, se utilizan los llamados convertidores. La validación de los datos de entradas introducidos se realiza por los llamados validadores.

La clase `DataBindingUtil` proporciona la funcionalidad para sincronizar la interfaz de usuario y los elementos del modelo. Cada interfaz de usuario y propiedad de modelo tiene que ser observado a través de un elemento `IObservableValue`. A través del método `DataBindingUtil.bindValue()` las propiedades observadas están conectadas. Además se ofrece la posibilidad de utilizar los validadores y convertidores, que convierte los datos no informados en valores predeterminados.

Además, JFace Data Binding permite utilizar iconos que se encarguen de decorar la interfaz de usuario permitiendo reflejar con ello el estado de la validación de un campo, con lo que el usuario obtiene información inmediata respecto a si la entrada para el campo es correcta o no.

org.eclipse.jface.text

Proporciona un *framework* para la creación, manipulación, visualización y edición de documentos de texto. `IDocument` es el mayor modelo de abstracción de texto. Proporciona manejadores para el contenido, manejadores de la posición usando categorías de posiciones, manejo de particiones de documentos y notificaciones de cambios. Para poder recibir notificaciones sobre los cambios de los documentos, un objeto debe implementar `IDocumentListener` y debe estar registrado con el documento. La actualización de la posición en respuesta a un cambio en el documento es llevada a cabo por implementaciones de `IDocumentPositionUpdater`. La actualización de las particiones en respuesta a un cambio en el documento es realizada por implementaciones de `IdocumentPartitioner`. Para recibir notificaciones sobre cambios en las particiones del documento, los objetos deben implementar `IDocumentParititonningListener` y deben estar registrados con el documento.

El paquete contiene implementaciones por defecto de los actualizadores de posiciones de los documentos y para los propios documentos. `AbstractDocument` utiliza `ITextStorage` para almacenar y manejar su contenido e `ILineTracker` para mantener la estructura de su contenido.

org.eclipse.team.core (3.5.1)

Proporciona la *API* para definir y trabajar con proveedores de repositorio. Un proveedor de repositorio ofrece la puesta en común de los proyectos contenidos en un área de trabajo local o *workspace* de Eclipse con lugares remotos gestionados por un tipo particular de sistema de repositorios.

Otras funcionalidades adicionales de interés que incluye esta *API* son la asignación del proveedor de repositorio para un proyecto, la gestión del contenido de los archivos y la gestión de patrones globales para el filtrado de estos.

org.eclipse.ui

Contiene el soporte para la programación y ejecución de interfaces gráficas y permite extender la interfaz de usuario de la plataforma Eclipse.

Las clases de la interfaz de usuario de la plataforma acceden a un único *workbench*. Un *workbench* es el objeto raíz de la interfaz de usuario, a partir del cual cuelgan todos los recursos del espacio de trabajo. Tiene una o más ventanas de *workbench*, cada una de las cuales tiene a su vez una colección de páginas, llamadas *workbench pages*. Solo una de ellas está activa en cada momento y es visible al usuario final.

Cada página del *workbench* contiene una colección de partes, llamadas *workbench parts*. Las partes de una página están preparadas para ser mostradas por pantalla. El usuario puede interaccionar con las páginas y sus partes y modificar un modelo a través de ellas, los recursos del *workspace*. Hay dos tipos de partes del *workbench*: vistas y editores. Un editor se usa típicamente para editar un documento o un objeto de entrada, o para navegar a través de él. Una vista se usa para navegar por una jerarquía de información (como el *workspace*), abrir un editor, o mostrar propiedades del editor activo.

La plataforma crea un *workbench* cuando este *plugin* es activado. Como esto sólo ocurre como mucho una vez durante el ciclo de ejecución de la plataforma, sólo hay una instancia del *workbench*. Esto permite a los *plugins* instalados acceder de forma estática al él, lo que les permite integrarse con los elementos que lo componen, intercambiando información y acciones con ellos.

Dentro del *workbench* el usuario interacciona con diferentes tipos de recursos. Como se requieren distintas herramientas para cada uno de ellos, el *workbench* define una serie de puntos de extensión que permiten la integración de nuevas herramientas. Hay puntos de extensión para vistas, editores, conjuntos de acciones, asistentes, etc.

org.eclipse.ui.editors (3.5.0)

Se trata de un punto de extensión para el *workbench* ya que permite a los *plugins* diseñar nuevos editores con el fin de añadir a estos al mismo.

Un editor es una parte del *workbench* que permite al usuario editar un objeto, normalmente un archivo. La forma en la que trabajan los editores es similar a la forma en la que lo hacen las herramientas de archivo para edición, salvo que los editores se encuentran estrechamente integrados en la plataforma de interfaz de usuario del *workbench*. Un editor está siempre asociado a un objeto de entrada, *IEditorInput*, que corresponde con un documento o archivo que se está editando. Los cambios realizados en un editor no se confirman hasta que el usuario los salva.

Es por tanto que, la interfaz para los editores se define en `IEditorPart`, pero los *plugins* pueden optar por extender la clase `EditorPart` en lugar de implementar un `IEditorPart` desde cero.

org.eclipse.ui.forms (3.4.1)

Eclipse Forms es un *plugin* que proporciona un conjunto de *widgets* basados en formularios para su uso en vistas, editores y asistentes, así como otras clases de utilidad. Con ello se amplia de forma considerable el *SWT (Standard Widget Toolkit)* y se permite modificar el estilo, los colores, las fuentes y otras propiedades con el fin de obtener el comportamiento deseado.

org.eclipse.ui.workbench.texteditor (3.5.1)

Proporciona un *framework* para los editores de texto. `ITextEditor` extiende de `IEditorPart` incorporando la funcionalidad específica de un editor de texto.

Los editores de texto utilizan visualizadores de fuente, `ISourceViewer`, para mostrar y editar la entrada dada al editor. Con el fin de traducir la entrada del editor en un documento y viceversa (por ejemplo, para guardar un cambio o para dar formato a un texto), un editor de texto usa proveedores de documentos, `IDocumentProvider`. Además, el paquete contiene una serie de acciones predefinidas y configurables.

2.2.2. Bibliotecas auxiliares

En este apartado se describen las bibliotecas usadas en el desarrollo del proyecto y que no se incluyen como dependencias del *plugin* con respecto a otros *plugins* de Eclipse. En la cláusula `Require-Classpath` del fichero `MANIFEST.MF` se encuentran incluidas estas bibliotecas.

commons-io-2.0.jar

Librería de Apache Software Foundation proporciona una extensión a la funcionalidad de los paquetes de Entrada/Salida que nos ofrece la plataforma *J2SE*.

Commons IO se compone de varios paquetes que nos proporcionan numerosas funcionalidades, entre estas podemos destacar; clases de utilidad para realizar tareas comunes, nuevos `InputStream` y `OutputStream`, comparadores para ficheros que implementan `java.util.Comparator`, varias implementaciones para filtros o búsquedas sobre ficheros y un componente para la monitorización de eventos en ficheros del sistema.

Algunas de las clases de esta librería que más se han utilizado a lo largo del proyecto son: del paquete `org.apache.commons.io` las clases `FileUtils` y `FilenameUtils` para la lectura y escritura de ficheros y para el tratamiento de rutas de forma independiente al sistema operativo y del paquete `org.apache.commons.io.filefilter` la clase `FileFilterUtils` para la búsqueda parametrizada sobre archivos.

doccheck.jar

Con esta biblioteca se puede verificar que los comentarios del código fuente son adecuados al *estándar de facto* utilizado para la documentación de clases Java. Identifica y señala de forma clara todos aquellos puntos en los que faltan comentarios, o en los que se ha utilizado un formato incorrecto para los mismos. Con los resultados de su exploración genera un listado en formato `HTML` en el que aparecen clasificados por paquetes y por tipo los errores de documentación cometidos.

El *doclet* encargado de la generación de documentación por defecto utilizado por la herramienta Javadoc de Java genera documentación del *API* en forma de páginas con formato `HTML`. Sin embargo este otro *doclet*, el de DocCheck, se encarga de identificar, clasificar y hacer un recuento de errores en la documentación.

Además de identificar los fallos, genera plantillas para los comentarios pendientes, sirviendo como ayuda para completar la documentación del código. Su comprobación es de completitud, es decir, comprueba si todo lo que se debe comentar está comentado, pero no analiza semánticamente los comentarios para ver si realmente tienen sentido.

guava-r07.jar

Guava es la librería de Google para Java que fue lanzada al público después ser usada durante años en los proyectos internos de Google.

Google Guava está compuesto por seis paquetes; dos de ellos son el núcleo de la librería y son empleados por los demás paquetes. Un tercer paquete es lo equivalente a la Google Collections Library, aunque contiene algunas clases adicionales. Un cuarto paquete contiene distintas clases de utilidad para trabajar con entrada y salida, otro para trabajar con datos primitivos y el último para la programación concurrente.

El propósito de la librería es reducir el código necesario para realizar ciertas tareas que son muy comunes dentro de Java y que no están soportadas por la librería estándar, así como eliminar código que es propenso a errores.

Google Guava se distribuye bajo licencia Apache 2.0, por lo que puede emplearse en cualquier proyecto comercial u opensource.

A lo largo del proyecto se ha utilizado principalmente esta librería por la flexibilidad que nos aportaba en ciertas tareas como son el uso de colecciones inmutables, los filtros o búsquedas, para evaluar precondiciones, funciones o predicados, y también con el fin de realizar operaciones sobre conjuntos, como son: intersección, unión o diferencia.

j2h.jar

Esta biblioteca permite convertir ficheros fuente Java a ficheros HTML. Dentro del presente proyecto se ha utilizado para mostrar los ficheros de los ejemplos de las distintas refactorizaciones. De forma que en esta visualización el texto se encuentre formateado con diferentes colores permitiendo una visión más clara del mismo.

javamoon-2.6.1.jar

Esta biblioteca es fundamental para el trabajo con la aplicación y para el desarrollo del proyecto en su conjunto. En particular, incluye las clases que permiten generar los modelos a partir de clases Java (paquete javamoon.construct y sus subpaquetes). También contiene el paquete javamoon.regenerate, cuyas clases son necesarias para la recuperación de código a partir de un modelo MOON. Constituye, en definitiva, la extensión de MOON para el lenguaje Java.

jdom.jar

Es una biblioteca para leer, escribir, crear y manipular XML cómodamente desde Java. Al contrario que SAX o DOM, JDOM es una biblioteca específica para Java, por lo que su uso es mucho más intuitivo y sencillo.

Un aspecto importante es que JDOM no es un parser en sí mismo, sino una biblioteca que aporta una capa de abstracción para la manipulación de documentos XML. De hecho, para realizar su trabajo utiliza un parser SAX o DOM, entre los que se puede elegir.

En el presente proyecto se ha utilizado para el manejo de los ficheros XML donde están definidas las refactorizaciones, los planes de refactorizaciones o el conjunto de refactorizaciones disponibles para cada ámbito.

jgraph-jdk1.5.jar

JGraphT es una biblioteca de clases de Java que se centra en las estructuras de datos proporcionando objetos de teoría de grafos y algoritmos. JGraphT soporta varios tipos de grafos, incluyendo: grafos dirigidos y no dirigidos, grafos ponderados y no ponderados, grafos simples, multigrafos y pseudografs. También permite grafos no modificables, así como registrar oyentes externos para el seguimiento de modificaciones en grafos.

Ademas ofrece la posibilidad de crear grafos de cualquier tipo de objetos, como pueden ser cadenas, direcciones URL, documentos XML, etc. o representaciones del modelo minimal como es el caso del uso dado en el proyecto.

jsoup-1.5.2.jar

JSoup es una librería para Java que permite la extracción y manipulación de documentos HTML. Proporciona una API muy conveniente para la extracción y manipulación de datos, utilizando lo mejor de los métodos como DOM, CSS y jQuery.

Esta librería está diseñado para hacer frente a todas las variedades de HTML que se encuentran en la web; desde los HTMLs perfectos y válidos a los marcados formateados inválidos, mediante la utilización de un árbol de análisis sensible.

Algunas de las características más importantes son el análisis `HTML` de una `URL`, un archivo o cadena, la búsqueda y la extracción de datos utilizando los selectores `DOM` transversales o `CSS`, así como la manipulación de los elementos, atributos y texto de `HTML`.

En el presente proyecto Jsoup se ha utilizado como *parser* para la extracción de la descripción de la documentación JavaDoc asociada a cada una de las entradas, acciones y predicados disponibles, la cual se encuentra en formato `HTML`.

log4j-1.2.15.jar

Se trata de un *API* para manejar el registro -log- de operaciones en los programas, algo que resulta necesario en los periodos de depuración. Es uno de los componentes del proyecto Jakarta.

Una de las mayores ventajas de cualquier *API* de *logging* frente al uso simple de la salida estándar mediante llamadas del tipo `System.out.println()`, es la capacidad de habilitar y deshabilitar ciertos logs, mientras otros no sufren ninguna alteración. Esto se realiza categorizando los mensajes de log de acuerdo al criterio del programador. Además, Log4j permite que los mensajes se impriman en múltiples destinos.

lucene-core-3.0.3.jar

Lucene es una *API* open source para la recuperación de información que permite agregarle funcionalidades de búsqueda e indexación a las aplicaciones. Esta es miembro de la familia de proyectos de Apache Jakarta bajo la licencia de Apache Software License. Actualmente, se considera como una de las librerías de Java para *IR (Information Retrieval)* más populares.

Las funciones se centran en la indexación de texto y en la búsqueda de forma eficiente, de manera que la aplicación sólo tenga que enfrentarse con los problemas propios de su dominio mientras la complejidad de la implementación de indexación o búsquedas queda oculta tras la *API* que ofrece, de uso sencillo.

Lucene, en el proceso de indexación analiza la información y construye el índice para poder identificar de forma rápida el lugar donde se encuentra algún dato en particular. Cabe destacar que para realizar esta tarea, previamente se habrán tenido que realizar un

preprocesado de la información, en el cual se pueden distinguir estas fases:

- Normalización: consiste en eliminar puntuación, eliminar acentos, y convertir todo el texto en minúsculas.
- Eliminación de *StopWords*: en esta etapa las palabras mas comunes, como son artículos, preposiciones, disyunciones y conjunciones son eliminadas del texto, así como otras que se consideren adecuadas.
- Stemming: Cada uno de las palabras del texto es reducida a su forma raíz. Por ejemplo, se eliminan la conjugación de los verbos, y se deja la raíz de este.
- Lematización: Las palabras con el mismo significado son cambiadas por una única forma, por ejemplo un sinónimo.

El proceso de búsqueda hace uso de este índice para encontrar los lugares donde aparece los datos buscados y en base a esto devolver los documentos que cumplen con la búsqueda realizada, pudiendo ser presentados los mejores resultados primero.

Además, Lucene define un lenguaje de consulta para realizar las búsquedas en los documentos indexados.

En el presente proyecto se ha utilizado para el manejo realizar la búsqueda sobre la documentación de los elementos disponibles para la definición de las refactorizaciones, es decir, para las entradas, acciones y predicados.

lucene-snowball-3.0.3.jar

Lucene mediante esta biblioteca ofrece una buena colección por defecto de *stemmers* del algoritmo Snowball para varios idiomas, encargados de la extracción de raíces de las palabras en la fase de *stemming* del preprocesado de la información para realizar posteriormente la indexación de la misma.

moon-2.4.2.jar

Esta biblioteca contiene las clases que componen un modelo MOON minimal. Sus

elementos mantienen la independencia del lenguaje y son los que permiten representar las clases, los métodos, etc. en un modelo orientado a objetos genérico. Se referencia por tanto a trabajos relacionados como [Crespo 2000] y [Crespo, López, & Marticorena n.d.] para obtener más información sobre esta biblioteca y los trabajos que se están realizando sobre ella.

refactoringengine-1.1.1.jar

Es la biblioteca que contiene las clases fundamentales del motor de refactorizaciones. Contiene el núcleo del motor, que se define en el paquete `refactoring.engine`. Las clases del paquete establecen la plantilla común para la definición de refactorizaciones.

En esta biblioteca se incluyen las clases: `AbstractRefactoring`, `Action`, `Function`, `Postcondition`, `PostconditionException`, `Precondition`, `PreconditionException`, `Predicate` y `Refactoring`.

A continuación se presenta un diagrama ilustrativo de las clases abstractas, para posteriormente pasar a explicar cada uno de los elementos que lo conforman:

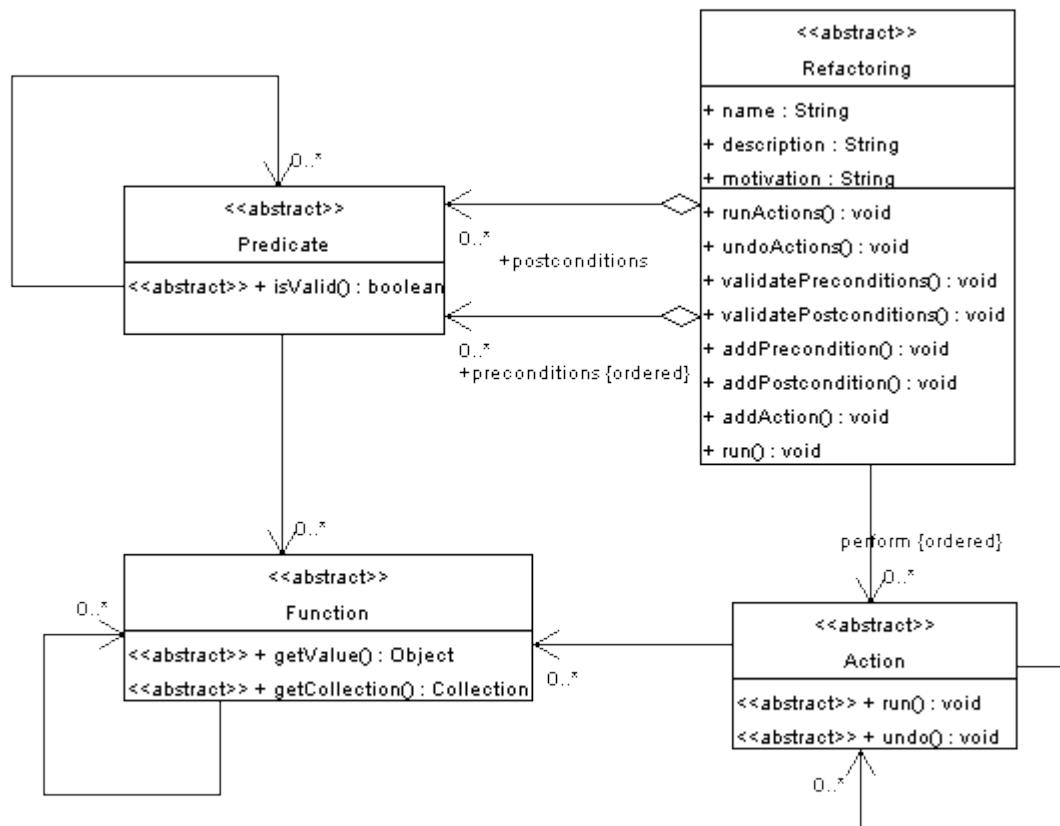


Ilustración 91: Diagrama de clases de `refactoring.engine`

- **Action:** define la interfaz para las clases de los paquetes `concreteaction` del repositorio (puesto que a partir de ahora habrá paquetes independientes para cada uno de los lenguajes orientados a objetos concretos sobre los que se definan elementos). Estas clases son las que realizan los cambios en el modelo `MOON`, es decir, las que ejecutan las modificaciones que componen las refactorizaciones.
- **Predicate:** define la interfaz para las clases de los paquetes del repositorio `concretepredicate`. Estas clases se encargan de evaluar el cumplimiento o no de una determinada condición por parte de un modelo `MOON` o alguno de sus elementos. Dependiendo de cuándo se evalúe esa condición se habla de *precondiciones* (se aplican antes que las acciones para comprobar que la refactorización puede llevarse a cabo) o *postcondiciones* (se aplican después de las acciones, comprobando que los cambios realizados son correctos); pero se trata de una distinción meramente semántica o conceptual, puesto que la operativa sigue siendo la misma.
- **Function:** define la interfaz para las clases de los paquetes del repositorio `concretefunction`. Estas clases son utilizadas por los *predicados* y las *acciones* para acceder a colecciones de objetos o a objetos concretos de un modelo `MOON` para los que no existen métodos de acceso directo.
- **Refactoring:** define la interfaz común para todas las refactorizaciones. Como se ha podido observar en el diagrama, las refactorizaciones están formadas por un conjunto de *predicados* y *acciones*, y es ésta la clase que dispone de una serie de métodos para añadir estos elementos a las refactorizaciones: `addPrecondition()`, `addAction()` y `addPostcondition()`. También tiene el método plantilla `run()`, heredado por todas las refactorizaciones para su ejecución.

swtdesigner.jar

Se trata de una biblioteca que consta de la clase `ResourceManager` la cual es una clase de utilidad para manejar recursos del sistema operativo asociados a controles `SWT` como colores, fuentes, imágenes, etc. El código de la aplicación debe llamar explícitamente al método `dispose()` para liberar los recursos del sistema operativo manejados por objetos

caché cuando estos objetos y los recursos del sistema operativo ya no son necesarios.

2.3. Instalación de herramientas

2.3.1. Instalación Java Development Kit

Java Software Development Kit es una herramienta destinada a la implementación y ejecución de aplicaciones en lenguaje Java. Su instalación es muy sencilla, para ello nos dirigiremos a la carpeta Software\java del CD que se adjunta con el presente proyecto, donde encontraremos el fichero ejecutable jdk-6u23-windows-i586.exe, haremos doble clic sobre él e iremos siguiendo los pasos para su correcta instalación.

A continuación se deberán ajustar las dos variables necesarias, estas son PATH y CLASSPATH. Para ello se tienen dos alternativas, que son las siguientes:

- establecer los valores deseados para estas variables sólo mientras duran los procesos de compilación y ejecución.
- establecer persistentemente los valores deseados como variables de entorno para todo el sistema operativo.

Optaremos por la segunda alternativa, quizá la más cómoda.

PATH

Esta variable es utilizada para invocar a los ficheros ejecutables de J2SDK (javac.exe para la compilación, java.exe para la ejecución, javadoc.exe para la generar la documentación, etc.) desde cualquier directorio donde nos encontremos sin necesidad de tener que teclear la ruta completa del comando cada vez que le necesitemos utilizar.

Para establecer su valor, sitúese sobre el icono de *Mi PC* que se encuentra en su escritorio, pulse botón derecho del ratón sobre él. Aparecerá un menú contextual, entre las opciones disponibles escoja *Propiedades*. A continuación se mostrará una pantalla donde deberá seleccionar la pestaña *Opciones avanzadas* y dentro de ella *Variables de entorno*. Dentro de esta nueva pantalla seleccione en *Variables del sistema* la variable Path y pulse *Modificar*. Aparecerá una ventana donde deberá añadir en *Valor de variable* la ruta de la carpeta bin del JDK y pulsar *Aceptar*. En

nuestro caso: ;C:\jdk1.6.0_23\bin.

Para comprobar que la instalación se ha realizado satisfactoriamente diríjase a *Inicio>Ejecutar* y escriba cmd, acto seguido aparecerá la consola de comandos, en ella deberá teclear el comando java -version. Si en ella aparece la versión que se acaba de instalar querrá decir que el proceso de instalación ha sido el correcto y por lo tanto ya es posible la compilación y ejecución de aplicaciones Java.

CLASSPATH

Esta variable es utilizada para indicar a las aplicaciones implementadas en Java dónde encontrar las clases necesarias para su ejecución.

Para establecer su valor, procederemos de la misma forma pero en este caso crearemos la variable CLASSPATH y le asignaremos la ruta de la carpeta lib del JDK y el directorio actual (representado por un punto), por lo tanto el valor de la variable quedaría de la siguiente forma: .;C:\jdk1.6.0_23\lib, así mediante la opción -classpath bastará con indicar las rutas relativas a los archivos .class propios.

2.3.2. Instalación Eclipse

La versión del entorno de desarrollo que hemos estado utilizando durante el desarrollo del plugin es la versión 3.6, a la que han denominado Helios.

La instalación del entorno de desarrollo Eclipse es muy sencilla, nos dirigiremos a la carpeta Software\desarrollo del CD que se adjunta con el proyecto donde encontraremos el fichero comprimido eclipse-rpc-helios-SR1-win32.zip, a continuación procederemos a su descompresión eligiendo el directorio deseado ya que será en ese donde permanezca instalado.

Una vez acabada la descompresión, en el directorio elegido encontraremos el fichero eclipse.exe, haremos doble clic en él para arrancar el entorno de desarrollo y nos aparecerá la siguiente ventana donde podemos establecer el espacio de trabajo.

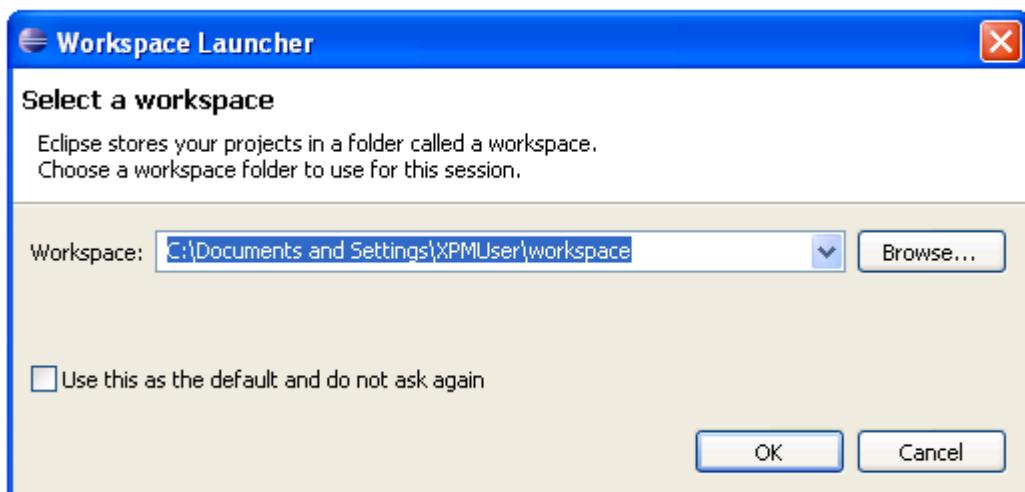


Ilustración 92: Selección espacio de trabajo Eclipse

A continuación nos dará la bienvenida y deberíamos tener configurado de manera automática el directorio donde se encuentra instalado el *JDK*, de no ser así lo configuraríamos. En este momento ya estaría todo listo para crear un proyecto donde comenzar a desarrollar.

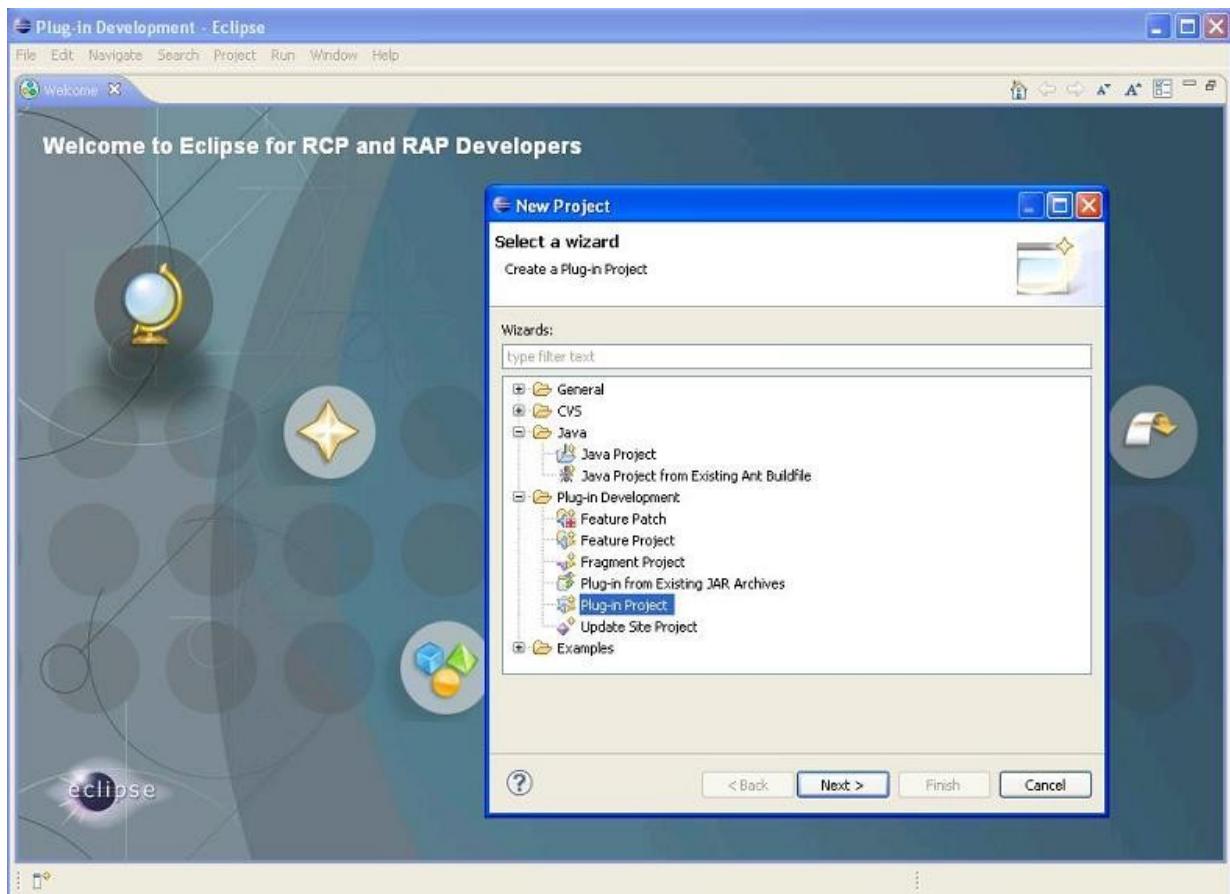


Ilustración 93: Bienvenida Eclipse

2.3.3. *Instalación Maven 3*

La construcción del producto se lleva a cabo mediante la herramienta Apache Maven, ya descrita en el apartado de *Técnicas y herramientas de la Memoria*. Es por tanto que es necesaria su instalación para llevar a cabo el proceso de construcción del software.

Posteriormente se ha incluido un apartado dedicado a Maven donde se ha profundizado en el conjunto de comandos necesarios para la construcción del software, detallando las funciones de cada uno de ellos.

Prerrequisito a la instalación de Maven 3

Para instalar Maven 3 es necesario disponer de un Java Runtime Environment (*JRE*) instalado en el equipo. La variable `JAVA_HOME` debe estar correctamente configurada apuntando al directorio en el que el *JRE* está instalado y `JAVA_HOME/bin` debe pertenecer a la variable `Path` del sistema para hacer posible la ejecución de Java.

Pasos comunes a la instalación de Maven 3

Para comenzar nos dirigiremos a la página de downloads del propio Apache Maven (<http://maven.apache.org/download.html>) [Maven Download n.d.] y nos descargaremos cualquiera de los paquetes binarios de entre los distintos formatos de ficheros comprimidos que aparecen. Otra opción es dirigirse a la carpeta `Software\desarrollo` del CD que se adjunta con el proyecto donde encontraremos el fichero comprimido `apache-maven-3.0.1-bin.zip` que deberemos copiar. Una vez completada la descarga o copia, según la opción por la que nos hayamos decantado, deberemos descomprimir el fichero en el directorio deseado, el cual a partir de este momento nos referiremos a él como `M2_HOME`, que corresponde con el nombre de la variable que se utiliza para referenciar dicho directorio y que tendremos que crear ahora.

A continuación se explican los pasos específicos para la instalación de Maven 3 en sistemas operativos UNIX y Windows, basados en los propios pasos explicados en la web [Maven Download n.d.], ya que son dependientes del *SO* en el que nos encontramos.

Instalación en Windows 2000/XP

Para añadir la variable `M2_HOME` a las propiedades del sistema se debe acceder a la configuración de las variables de entorno del sistema. Para ello, podemos proceder de la siguiente forma, pulsando la tecla de *Windows + Pausa*, a continuación seleccionamos la pestaña *Avanzadas* y se pulsa en el botón *Variables de entorno*. Cabe destacar que se debe omitir cualquier comilla incluso si la ruta contiene espacios.

En el mismo diálogo se debe añadir al variable de entorno `M2` con el valor `%M2_HOME%\bin` y también se debe actualizar (o crear si no existe) la variable de entorno `Path` dentro de las variables de usuario y añadir el valor `%M2%` para hacer que el comando `mvn` sea accesible desde línea de comandos.

Para comprobar que el proceso que acabamos de realizar es correcto, abriremos una ventana de comandos pulsando la tecla de *Windows + R* y luego teclearemos `cmd` para ejecutar el comando `mvn -version` y con él poder así verificar que Maven está instalado correctamente.

Instalación en SO basados en Unix (Linux, Solaris, Mac OS X)

En ese caso, para añadir la variable de entorno `M2_HOME`, abriremos una terminal de comandos desde la cual realizar el `export` de la variable.

Por ejemplo:

```
export M2_HOME=/usr/local/apache-maven/apache-maven-3.0.2
```

Agregamos la variable de entorno `M2` de la misma forma que acabamos de ver:

```
export M2=$M2_HOME/bin
```

Por último, tendremos que añadir la variable que acabamos de crear `M2` al `Path`, procediendo a ejecutar lo siguiente:

```
export PATH=$M2:$PATH
```

De la misma forma que para Windows comprobaremos que el proceso que acabamos de realizar es correcto, para ello ejecutamos el comando `mvn -version` verificando así que Maven está correctamente instalado.

2.3.4. Instalación Git

La instalación de Git es muy sencilla. Si el sistema operativo en que se quiere instalar Git es Linux, Git esta disponible en los repositorios de paquetes de los sistemas Linux más utilizados como Ubuntu o Fedora por ejemplo. Así la instalación en Ubuntu consiste en ejecutar el siguiente comando desde la línea de comandos:

```
$ apt-get install git-core
```

En el caso de Windows la instalación consiste en instalar el paquete msysgit disponible desde (<http://code.google.com/p/msysgit/downloads/list>) [msysgit - Git for Windows - Google Project Hosting n.d.]. Una vez instalado este paquete si se quiere trabajar con el servicio de repositorios online Github es necesario configurar un par de claves *SSH* para que dicho servicio sea capaz de reconocernos y de este modo evitar que nuestra identidad sea suplantada.

Generar claves SSH para interactuar con Github

El propio Github proporciona en su guía de usuario una página web con todas las instrucciones de cómo configurar Git para poder trabajar con Github desde Windows (<http://help.github.com/win-set-up-git/>) [Github n.d.]. De esta página han sido tomadas las instrucciones que se van a dar a continuación:

En primer lugar accedemos a la línea de comandos que hemos instalado con el paquete msysgit desde *Menú Inicio > Programas > Git > Git bash*. Ahora se tiene que crear el conjunto de claves *SSH*. Si el usuario ya cuenta con otro par de claves *SSH* previas es necesario que haga copias de seguridad y las elimine de su directorio actual. Para ello desde la línea de comandos ejecutamos:

```
$ cd ~/.ssh  
$ mkdir key_backup  
$ cp id_rsa* key_backup  
$ rm id_rsa*
```

A continuación para generar las claves *SSH* utilizamos el comando `ssh-keygen`, especificando el tipo de clave a generar (rsa) y nuestro correo electrónico que va a servir para generar la clave:

```
$ ssh-keygen -t rsa -C "your_email@youremail.com"
```

Ahora es necesario introducir una frase de paso, lo que debería generar una salida cuya última frase debería ser similar a la siguiente:

```
The key fingerprint is:  
e4:cd:ad:33:a0:d4:42:18:54:cd:e2:b2:dc:b2:68:13  
your_email@youremail.com
```

Agregar la clave SSH a nuestra cuenta de Github

Una vez logueados con nuestro usuario en Github accedemos a *Account Settings > SSH Public Keys > Add another public key*.

Abrimos el fichero `id_rsa.pub` que hemos generado en el apartado anterior con nuestro editor de textos favorito. Puede ser necesario activar la opción de mostrar ficheros ocultos en la carpeta dado que el directorio `.ssh` es un directorio oculto. Ahora copiamos exactamente el contenido del fichero y lo pegamos en el campo `Key` de la página web de Github. Tras haberlo copiado pulsamos el botón `Add Key`.

Ahora para comprobar que todos los pasos se han llevado a cabo de forma correcta desde la línea de comandos de Git ejecutamos el siguiente comando:

```
$ ssh git@github.com
```

Que debería en primer lugar preguntar si se quiere aceptar la conexión al servidor y posteriormente devolver un mensaje similar al siguiente:

```
PTY allocation request failed on channel 0  
Hi username! You've successfully authenticated, but GitHub does not  
provide shell access.  
Connection to github.com closed.
```

Ahora que Git está instalado y se han configurado las claves *SSH* para Github el último paso consiste en agregar nuestra información personal en Git. Git permite registrar un nombre de usuario y un correo electrónico que se asociará a todas las modificaciones del usuario. Para hacer esto se ejecutan los siguientes comandos:

```
$ git config --global user.name "Firstname Lastname"  
$ git config --global user.email "your_email@youremail.com"
```

Instalación y Configuración del plugin de Git para Eclipse - EGit

La instalación del plugin es muy sencilla. De forma similar a la instalación del plugin de refactorización, consiste simplemente en ir al gestor de actualizaciones de Eclipse utilizando la siguiente dirección como URL del repositorio:

<http://download.eclipse.org/egit/updates>

Tras realizar la instalación es necesario realizar la configuración del plugin para que tome las claves SSH de los ficheros adecuados y pueda conectarse a Github. Este paso se ejecuta desde Eclipse yendo a la ventana de preferencias pulsando *Window > Preferences*. Una vez allí vamos a *General > Network Connections > SSH2* y desde esa ventana escoger el directorio en el que hemos almacenado las claves en el campo *SSH2 Home*. Con estos pasos ya se ha conseguido instalar EGit y configurarlo para que pueda conectarse a un repositorio de Github.

2.3.5. Instalación del plugin de Eclipse para Fogbugz -FogLyn

La instalación del plugin de Eclipse para Fogbugz en Eclipse 3.5 es muy sencilla, consiste en seguir los mismos pasos para la instalación del plugin de refactorización pero utilizando un repositorio de actualizaciones distinto.

Desde el menú *Help* es necesario pinchar en *Install New Software*. A continuación se debe introducir la *URL* del repositorio de Foglyn <http://update.foglyn.com/stable> en el cuadro de texto *Work With* y pulsar intro.

Después de pulsar intro, Eclipse muestra una lista de software disponible desde el repositorio. Se debe marcar Foglyn y pulsar el botón de siguiente.

La siguiente pantalla muestra la lista de todo el software que va a ser instalado. Después de pulsar el botón siguiente, se muestra la licencia de Foglyn. Es necesario aceptar la licencia si se quiere finalizar la instalación.

Finalmente, después de que la instalación haya terminado es necesario reiniciar Eclipse para poder empezar a utilizar el plugin instalado.

2.3.6. Instalación SWTBot

SWTBot es la biblioteca utilizada para la creación de las pruebas de interfaz gráfica del

plugin. Esta configurada como parte del proceso de construcción con Maven luego no es necesaria ninguna configuración adicional para ejecutar las pruebas de interfaz gráfica desde línea de comandos. Sin embargo, si se quieren ejecutar las pruebas desde Eclipse sí es necesario cierta configuración adicional dado que éstas necesitan ejecutarse en un hilo independiente y esto no es posible ejecutándolas como pruebas de plugin de JUnit.

La configuración adicional consiste en instalar el plugin de SWTBot para Eclipse. Este plugin se instala agregando en el instalador de plugins de Eclipse el siguiente repositorio:

<http://download.eclipse.org/technology/swtbot/galileo/dev-build/update-site/>

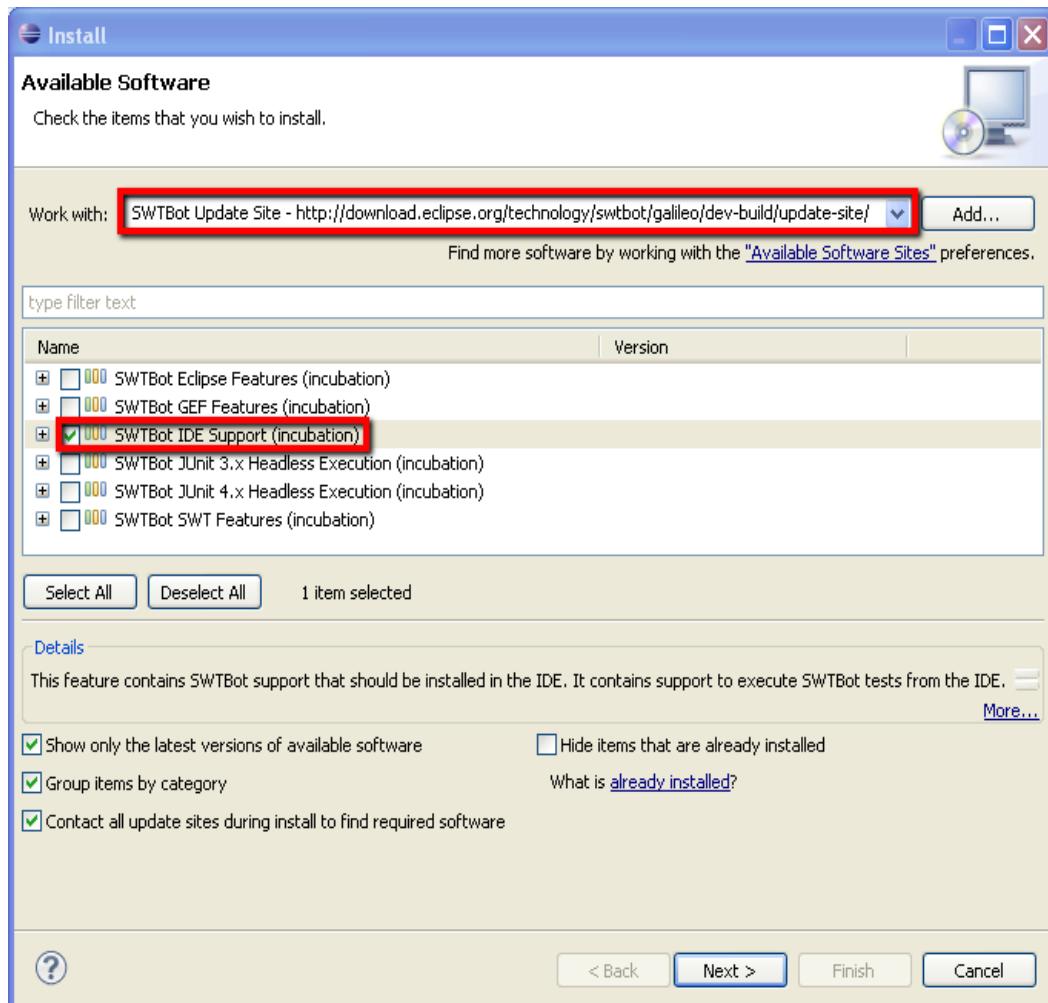


Ilustración 94: Instalación de SWTBot para Eclipse

Una vez agregado se selecciona dicho repositorio y de entre todas las opciones de plugins instalables escogemos la de *SWTBot IDE Support*.

Con la instalación terminada y tras reiniciar Eclipse al seleccionar un test sea o no de interfaz gráfica y pulsar el botón derecho del ratón sobre él aparecerá la opción *Run As* > *SWTBot Test*.

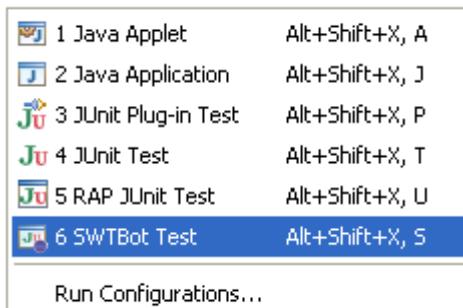


Ilustración 95: Ejecutar como tests de interfaz gráfica con SWTBot

Para más información sobre SWTBot lo más recomendable es acceder a la guía de usuario de la biblioteca disponible desde:

<http://wiki.eclipse.org/SWTBot/UsersGuide> [SWTBot - User Guide n.d.].

2.3.7. Instalación Sonar

En primer lugar es necesario indicar que Sonar es una aplicación web que se ejecuta bajo la plataforma Java. Por lo tanto los requisitos para la instalación de Sonar son exclusivamente el JDK de Java en su versión 1.5 o posterior y un navegador web para poder acceder a la aplicación. Cualquier navegador web moderno es soportado: Firefox, Chrome, Opera e Internet Explorer a partir de su versión 7 son opciones válidas.

Sonar viene configurado para ser sencillo de instalar. Tanto en Windows como en Linux la instalación consiste en los siguientes pasos:

- Descargar y descomprimir la distribución de la aplicación desde la página de descargas:

<http://www.sonarsource.org/downloads/> [Sonar» Download n.d.]

- Ejecutar la aplicación:
 - ◆ Desde Windows esto se hace ejecutando el comando:

```
bin\windows-x86-32\StartSonar.bat
```

- ◆ Desde linux el comando es muy similar:

```
bin/[OS]/sonar.sh console
```

- Finalmente para genera las métricas de nuestro proyecto sobre Sonar ejecutamos con Maven lo siguiente:

```
mvn clean install sonar:sonar
```

Como nota adicional se debe destacar que con esta configuración Sonar se ejecutaría con una base de datos Derby empotrada que no es adecuada para entornos de producción. Si se quiere utilizar Sonar con otra base de datos la siguiente página del manual de Sonar explica como se haría (<http://docs.codehaus.org/display/SONAR/Install+Sonar>) [Install Sonar - Sonar - Codehaus n.d.]

2.3.8. *Instalación Hudson*

Lo más habitual cuando se trata de instalar Hudson en Windows es instalarlo como un servicio, de este modo el servicio se ejecutará automáticamente cuando Windows arranque sin necesidad de que ningún usuario inicie sesión. Para instalar Hudson de esta manera es necesario descargar Hudson desde:

<http://hudson-ci.org/download/war/> [Download - Hudson.war n.d.]

A continuación se inicia la aplicación web antes de instalarla. Esto se consigue ejecutando:

```
java -jar hudson.war
```

Una vez que Hudson esta iniciado nos dirigiremos a <http://localhost:8080/> que es la dirección local desde la que la web es accesible. Desde ella a la página *Manage Hudson > Install as Windows Service* que requiere tener instalada una versión igual o superior a la 2.0 del framework .NET. Haciendo clic en dicho enlace se muestra la pantalla de instalación. En ella se debe seleccionar el directorio en el que se va a instalar el servicio. Dicho directorio debe existir en caso contrario la instalación fallará. Este directorio se convertirá en el directorio `HUDSON_HOME` y se utilizará para almacenar ficheros y plugins por parte de Hudson.

Una vez terminada la instalación completamente, se mostrará una página pidiendo reiniciar Hudson. Al aceptar la opción de reiniciar, Hudson se volverá a lanzar como un nuevo

servicio de Windows.

Llegado este punto se puede utilizar el gestor de servicios de Windows para confirmar que Hudson se está ejecutando como un servicio.

Instalación para Ubuntu

La instalación para Ubuntu es más sencilla y sólo consiste en la ejecución de la siguiente secuencia de comandos:

```
sudo su  
wget -q -O - http://hudson-ci.org/debian/hudson-ci.org.key | apt-key add -  
  
echo "deb http://hudson-ci.org/debian binary/" >  
/etc/apt/sources.list.d/hudson.list  
  
aptitude update  
  
aptitude install hudson  
  
exit
```

2.4. Desarrollo plugin Eclipse

2.4.1. Establecer la plataforma de desarrollo del plugin

Antes de comenzar a desarrollar el plugin es necesario establecer su plataforma objetivo o *target platform*. La *target platform* es el conjunto de plugins en los que el plugin se basa, proporcionando así el entorno en el que el plugin se ejecuta. Nuestro plugin puede declarar sus dependencias con cualquier otro plugin de entre los de su *target platform*. A partir de entonces estos plugins de los que el nuestro depende formarán parte de su classpath y pudiendo con ello utilizar cualquiera de las clases públicas que estos hayan decidido exportar.

Por defecto Eclipse define como plataforma objetivo el conjunto de plugins instalados en el propio *IDE*. Sin embargo, para el desarrollo del plugin de refactorizaciones se han incluido una serie de plugins que no vienen instalados por defecto en las distribuciones estándar de Eclipse. Es por esta razón que, es necesario establecer nuestra plataforma propia para que no aparezcan errores de compilación al incorporar los proyectos del plugin. Además, afortunadamente este es un proceso sencillo debido a que Eclipse permite definir

plataformas propias en ficheros con la extensión .target. En el caso de nuestro plugin de refactorizaciones esta misión la cumple el fichero dynamicrefactoring.target contenido en el proyecto dynamicrefactoring.targetplatform. Si se quiere empezar a desarrollar el plugin, la primera vez que se hace es necesario abrir dicho fichero y esperar durante unos minutos mientras Eclipse se descarga todas las dependencias necesarias. Este proceso puede prolongarse debido a que el tamaño del *SDK* de Eclipse supera los 150MB. Puede comprobarse que el proceso ha terminado cuando la tarea *Resolving Target Platform* marcada en el indicador de progreso de la barra inferior derecha (marcado con un 1 en Ilustración 97) de Eclipse llega al 100%. Terminado este proceso se pulsa en el enlace *Set as Target Platform* (marcado con un 2). Con esto ya se establecería la plataforma objetivo requerida y se podría empezar a desarrollar el plugin. Este proceso en ocasiones da error y se tiene que repetir el proceso nuevamente, esto es debido a las dependencias internas de unas con otras.

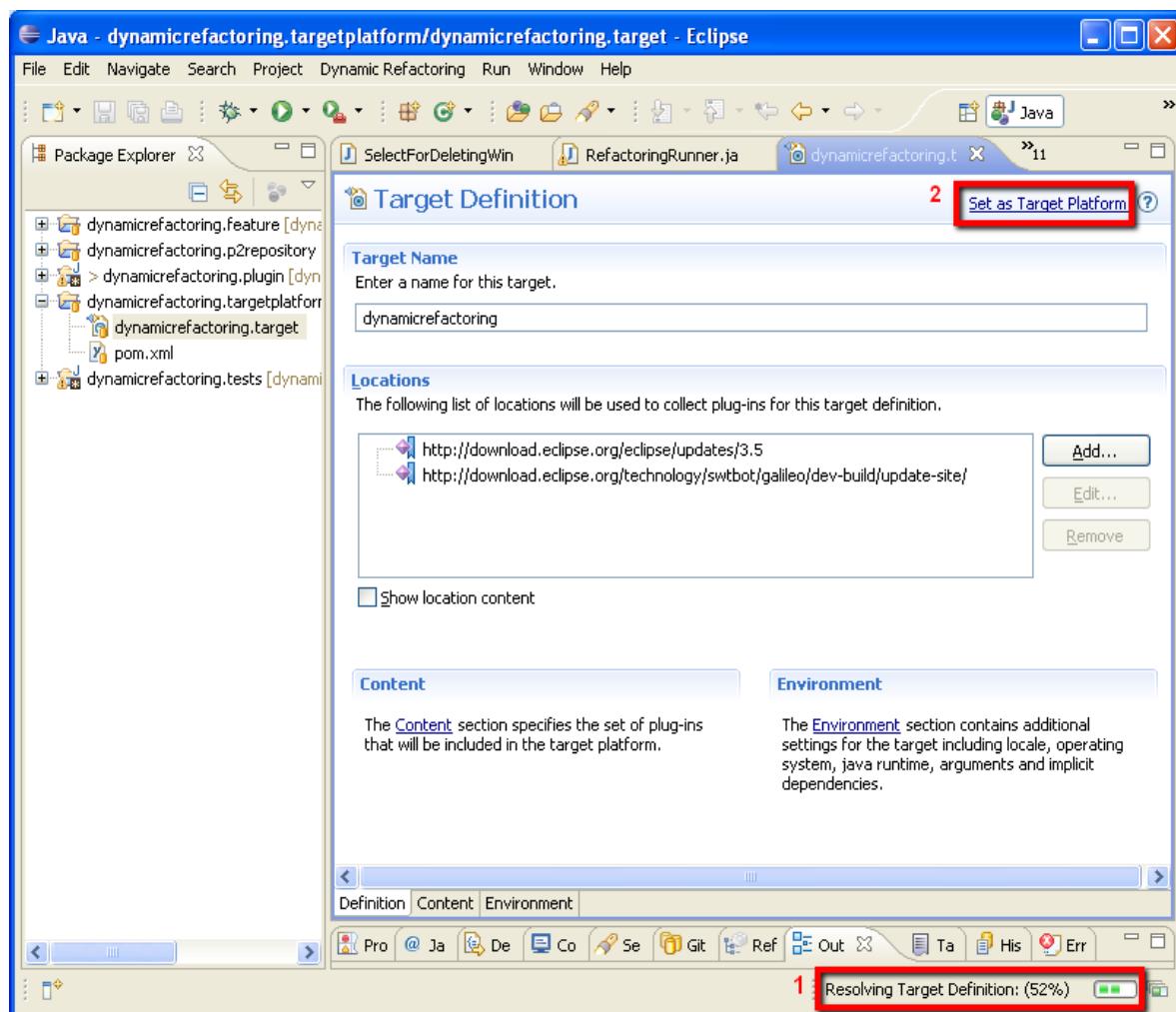


Ilustración 96: Establecer la plataforma objetivo para iniciar el desarrollo

2.4.2. Proyectos existentes en el código fuente

Si se tiene la intención de realizar modificaciones sobre el propio proyecto es necesario conocer para que sirven cada uno de los proyectos de los que éste a su vez se compone. A continuación se dará una explicación de la funcionalidad aportada por cada uno de ellos.

dynamicrefactoring.plugin

Este es el proyecto principal que contiene el código fuente del plugin, las bibliotecas JAR utilizadas, la documentación, los ficheros XML de configuración por defecto, las imágenes, la ayuda y otros ficheros secundarios.

Uno de los ficheros más importantes de este proyecto es el fichero con nombre META-INF/MANIFEST.MF. Como ya se ha citado en apartados anteriores de la documentación los plugins de Eclipse se ejecutan dentro de un entorno especial denominado *OSGI*. El fichero MANIFEST.MF es el que define cada uno de los parámetros de ejecución del plugin dentro de este entorno como por ejemplo los paquetes de que depende. Para más información sobre *OSGI* y la estructura de los ficheros MANIFEST.MF ver [OSGi with Eclipse Equinox - Tutorial n.d.] y [Help - Eclipse SDK - OSGI bundle manifest n.d.].

Otro fichero importante es el fichero con nombre plugin.xml. Este fichero es un fichero de configuración específico de los plugins de Eclipse. En él se van a definir las extensiones que se han utilizado y los puntos de extensión que se han definido.

Los puntos de extensión definen contratos que permiten a otros plugins añadir funcionalidad al plugin original. Por poner un ejemplo un punto de extensión que Eclipse define son las vistas. Al definir las vistas como puntos de extensión obliga a cualquier plugin que quiera definir una vista a que cumpla unos requisitos: por ejemplo toda vista debe tener un nombre, una clase principal que la defina o un icono.

Las extensiones por su parte son las realizaciones de las contribuciones de los plugins a los puntos de extensión. Para el ejemplo anterior una extensión sería un plugin en el que se define una vista concreta. Ver [Eclipse Extension Points and Extensions - Tutorial n.d.] para una explicación más extensa sobre los puntos de extensión y las vistas.

Eclipse proporciona un editor especial agrupado para ambos ficheros citados que evita tener que realizar modificaciones a mano del contenido de ambos.

dynamicrefactoring.plugin.nl1

Es la internacionalización del plugin al español. Permite que cuando el plugin se instala se tenga la posibilidad de visualizarlo en español o en inglés. Sus ficheros más importantes son los ficheros llamados `messages_es.properties`. Para que el plugin se muestre correctamente en castellano las traducciones de todas las cadenas definidas en el plugin principal tienen que estar traducidas al español en estos ficheros.

dynamicrefactoring.tests

Este es el proyecto en el que se definen todos los tests del plugin. Esta separado del plugin para incrementar la modularidad, sin embargo al estar definido como un fragmento [FAQ What is a plug-in fragment? - Eclipsepedia n.d.] del plugin principal los tests pueden acceder a las clases de éste sin ninguna restricción. De hecho el ser un fragmento significa que cuando los tests se van a ejecutar el contenido del proyecto de los tests se mezcla con el del proyecto del plugin en un *JAR* único.

Algo a destacar de este proyecto es que en su propio fichero `MANIFEST.MF` define un conjunto de dependencias adicionales necesarias para la ejecución de las pruebas de interfaz gráfica con *SWTBot*.

dynamicrefactoring.targetplatform

Tal y como se ha definido en la sección 2.4.1 Establecer la plataforma de desarrollo del plugin este proyecto contiene un fichero `dynamicrefactoring.target` que define el entorno desde el que el plugin debe obtener sus dependencias a la hora de ejecutarse. La ya citada sección explica qué se debe hacer con este fichero para poder comenzar a desarrollar modificaciones en el plugin.

dynamicrefactoring.feature

Una *feature* o *característica* de Eclipse es un proyecto en el que se definen un conjunto de metadatos sobre una agrupación de plugins en un fichero `feature.xml`. Sirve para hacer posible que uno o varios plugin puedan ser utilizados dentro del entorno del Eclipse Update Manager, el sistema encargado de las instalaciones y actualizaciones de plugins en Eclipse. El definir un proyecto con una característica para un plugin permite:

- Listar los requisitos en forma de dependencias del plugin. Esto es necesario para que el gestor de instalación de Eclipse conozca de qué plugins depende el plugin

a instalar. Cuando se utiliza el editor de Eclipse para ficheros `feature.xml` no es necesario indicar las dependencias de todos los plugins definidos, sino que el propio editor incorpora las dependencias de un plugin cuando este se incorpora a la característica.

- Proporcionar información sobre la característica. En los ficheros correspondientes a una característica se puede definir la licencia de los plugins. También se permite definir tanto una descripción como una página de bienvenida explicando lo que la característica ofrece a los usuarios y que se muestra tras finalizar la instalación.

dynamicrofactoring.p2repository

Un *repositorio P2* es un conjunto de características de Eclipse agrupadas por categorías. En el caso del plugin de refactorización el proyecto del repositorio es el que marca el paso final para que el proceso de construcción con Maven permita generar un archivo que el sistema de instalación de Eclipse entienda. Es posible generar este tipo de archivo a partir de una característica pero Eclipse recomienda utilizar repositorios P2.

El fichero más importante del proyecto del repositorio P2 es `category.xml`. En él se pueden definir una lista de categorías para posteriormente definir que características pertenecen a cada una de ellas.

2.4.3. Ficheros POM

Cada uno de los proyectos descritos en la sección anterior, consta de un fichero `pom.xml`. Estos ficheros junto con el fichero del mismo nombre que se encuentra en el directorio raíz del proyecto son los que definen las configuración del proceso de construcción del plugin. Su configuración consiste en lo siguiente:

- **Fichero del directorio raíz del proyecto:** Es el fichero con la configuración global del proyecto. Contiene enlaces al resto de ficheros `pom`, indicando de esa manera que el resto de proyectos también deben de ser incluidos en el proceso de construcción. Define variables globales que el resto de ficheros de configuración hereda y tareas comunes que se deben de aplicar al resto de proyectos como por ejemplo el firmado de los ficheros *JAR* generados. También define la tarea de generación de la documentación.

Además incluye las dependencias Maven generales del proyecto y define el

repositorio P2 desde el que los plugins de la aplicación y de los tests se descargan las librerías OSGI de las que dependen. Es decir, define la plataforma objetivo del proceso de construcción en Maven, de forma similar a como se había hecho en la sección 2.4.1 Establecer la plataforma de desarrollo del plugin para Eclipse. Esta configuración se muestra en el siguiente fragmento del fichero pom.xml.

```
<plugin>
  <groupId>org.sonatype.tycho</groupId>
  <artifactId>target-platform-configuration</artifactId>
  <version>${tycho-version}</version>
  <configuration>
    <resolver>p2</resolver>
    <target>
      <artifact>
        <groupId>dynamicrefactoring</groupId>
        <artifactId>dynamicrefactoring.targetplatform</artifactId>
        <version>3.0.9</version>
        <classifier>dynamicrefactoring</classifier>
      </artifact>
    </target>
    <ignoreTychoRepositories>false</ignoreTychoRepositories>
  </configuration>
</plugin>
```

- **Fichero del proyecto dynamicrefactoring.plugin:** Es un proyecto de tipo eclipse-plugin de tycho. Este tipo de proyectos generan ficheros que se pueden ejecutar como plugins dentro de Eclipse.

Tiene la particularidad de que define una tarea con Ant, en la que tras compilarse las clases del repositorio se copian los ficheros .class a una carpeta y esa carpeta se empaqueta en el fichero JAR del plugin. Esto permite disponer de los ficheros de código compilado de las refactorizaciones, lo que permite ejecutar éstas cuando el usuario lo solicita.

- **Fichero del proyecto dynamicrefactoring.test:** Es un proyecto de tipo eclipse-test-plugin que permite ejecutar tests sobre un entorno similar al propio entorno de producción que los tests se pueden encontrar en Eclipse.

La configuración especial de este fichero es necesaria para generar la página web con el informe de la cobertura de los tests mediante JaCoCo y para la ejecución de los tests de interfaz gráfica con SWTBot.

- **Fichero del proyecto dynamicrefactoring.targetplatform:** Indica que el fichero en el que se definen los repositorios desde los que se deben descargar

las dependencias *OSGI* para los proyectos de pruebas y del plugin es el fichero `dynamicrefactoring.target`, permitiendo así la compilación del plugin y la ejecución de los tests.

- **Fichero del proyecto `dynamicrefactoring.feature`:** Proyecto de tipo `eclipse-feature` que permite definir una característica de Eclipse para poder ser generada con Maven. No consta de ninguna configuración específica ya que toma por defecto el nombre del fichero `feature.xml` a partir del que debe generar la característica.
- Fichero del proyecto `dynamicrefactoring.p2repository`: Proyecto de tipo `eclipse-repository`. Toma los datos del fichero `category.xml` para generar el *repositorio p2* que permite la instalación del plugin desde Internet en Eclipse.

2.5. Maven

En este apartado se va a explicar la lista de pasos de los que consta el proceso de construcción del software y se detallará las funciones de cada uno de ellos.

2.5.1. Pasos previos a la construcción

Para poder ejecutar este proceso es necesario, obviamente, disponer de los fuentes del proyecto. Estos fuentes se pueden obtener desde un equipo que disponga de Git realizando un clonado del repositorio del proyecto que se encuentra en Github mediante el siguiente comando:

```
git clone git@github.com:txominpelu/dynamicrefactoring.plugin.git
```

Sin embargo, hay que tener en cuenta que para poder ejecutar este paso es necesario tener instalado Git y configuradas las claves *ssh* de acceso a Github. Para los interesados en llevar a cabo la instalación, ésta aparece detallada en [Github n.d.] y [Github n.d.] para sistemas operativos Linux y Windows respectivamente.

Otra opción es descargar los fuentes en formato comprimido desde la página web del propio repositorio de Github (<https://github.com/txominpelu/dynamicrefactoring.plugin>) y descomprimirlos.

La última opción consiste en copiar en un directorio del disco duro del equipo el

directorio de fuentes que contiene el CD que se proporciona junto a la documentación del proyecto.

Una vez que se dispone de los fuentes para ejecutar el proceso de construcción es necesario colocarse desde una ventana de comandos en la carpeta principal. La carpeta principal contendrá los directorios de los proyectos del plugin y tests entre otros, además de un fichero `pom.xml`. Desde esta carpeta es desde la que se deberán ejecutar todos los comandos de construcción del proyecto.

A continuación se detallan cada uno de los pasos a realizar para la construcción del proyecto.

2.5.2. *Paso 1 – Limpiar*

El primer paso se encarga de borrar los productos generados para poder construir el software desde cero. Se queda exclusivamente con lo necesario para construir el producto borrando los restos de procesos de construcción previos.

Comando a ejecutar:

```
mvn clean
```

2.5.3. *Paso 2 – Compilar*

El segundo paso realiza la compilación del proyecto y únicamente en el caso de que haya habido cambios en el código fuente es necesario. Si es así, este comando compila el código fuente de todos los paquetes.

Este paso es especial para un proyecto como el nuestro, un plugin de Eclipse, porque el plugin a construir no sólo tiene dependencias en bibliotecas disponibles en ficheros *JAR* en el propio proyecto, sino que también tiene dependencias en otros paquetes *OSGI*, que son un tipo especial de fichero *JAR*. Para poder compilar las clases es necesario recoger esas dependencias de algún sitio. El lugar del que esas dependencias se recogen es conocido como *target platform* (plataforma objetivo) que es el almacén de plugins sobre el que ese plugin necesita correr para poder encontrar sus dependencias.

La plataforma objetivo se puede definir apuntando a un directorio local (habitualmente una instalación de Eclipse) o gracias al plugin para Maven conocido como *tycho* [Eclipse Updates 3.5 n.d.] como un repositorio de Internet desde el que se

pueden descargar los paquetes. En el fichero de configuración del plugin la mayoría de los paquetes los vamos a descargar del sitio (<http://download.eclipse.org/eclipse/updates/3.5>) [Sonar n.d.]. Esta configuración está definida en el fichero `dynamicrefactoring.target` de dentro de la carpeta `dynamicrefactoring.targetplatform` que se puede editar desde la interfaz gráfica con Eclipse.

Esto nos permite hacer el proceso de construcción totalmente independiente del equipo en el que se ejecuta. Cualquier equipo con conexión a Internet sólo necesita tener Maven 3 instalado y ejecutar la fase de compilación para compilar el proyecto. El propio Maven se ocupa de descargar las dependencias la primera vez que se ejecuta y almacenarlas en el repositorio local. Esas dependencias son utilizadas posteriormente para compilar los fuentes y ejecutar de las pruebas.

Comando a ejecutar:

```
mvn compile
```

Generar documentación

Esta fase de compilación también se encarga de generar la documentación, es decir el *API*, del proyecto sobre la carpeta `doc/javadoc` de `dynamicrefactoring.plugin`.

El fichero `pom.xml` también está configurado para ejecutar el `doccheck` que genera un informe en el que identifica las clases o métodos sin comentarios y otras omisiones o irregularidades detectadas en la documentación de las propias clases sobre la carpeta `doc/doccheck` y una página web con el código fuente del proyecto del plugin sobre `doc/src_html`.

2.5.4. Paso 4 – Generar empaquetado

Este paso a partir del código fuente debidamente compilado realiza el empaquetado del mismo en el formato de fichero distribuible más adecuado, basándose en el tipo de proyecto que se trate.

En el caso de nuestro proyecto el formato de fichero distribuible será el formato *JAR*, en cambio en otros tipos de proyectos este formato cambiará. Por ejemplo, para proyectos de páginas web dinámicas el formato por defecto es *WAR*.

Comando a ejecutar:

```
mvn package
```

2.5.5. Paso 5 – Ejecutar Test de Integración

Se trata de una fase posterior al compilado y empaquetado y por tanto ejecuta ambas previamente a ejecutarse a sí misma. Por sí sola se encarga de ejecutar las pruebas de integración, es decir, los incluidos en el fragmento de pruebas del plugin.

Este comando genera un entorno *OSGI* con los paquetes de *Eclipse* y desde este entorno se ejecutan las pruebas definidas en el fragmento. De este modo, se permite hacer las pruebas en un entorno igual al de producción tal y como indica uno de los principios de la integración continua [Fowler 2000]. Si alguno de los tests falla provoca el fin del proceso de construcción para asegurar que fases posteriores, como podría ser el despliegue, no se lleven a cabo.

Comando a ejecutar:

```
mvn integration-tests
```

2.5.6. Paso 6 – Instalar

Se encarga de ejecutar todos los pasos anteriormente descritos excepto el paso javadoc que genera la documentación y además copia el paquete al repositorio local de Maven.

Comando a ejecutar:

```
mvn install
```

2.5.7. Ciclo completo

Con este comando lo que se pretende es ejecutar el proceso de construcción al completo. Las dos primeras fases, clean e install, ya han sido explicadas. El último objetivo, sonar:sonar, se encarga de generar mediante [PMD n.d.] una página web dinámica en forma de informe en el que se agrupan comprobaciones estáticas de indicadores de errores en el código generadas con [Checkstyle n.d.], [FindBugs™ n.d.] y [JaCoCo n.d.], métricas de complejidad del código, informes de la cobertura de las pruebas, número de tests ejecutados, número de líneas de código del proyecto, estadísticas de la

ANEXO IV – DOCUMENTACIÓN TÉCNICA DEL PROGRAMADOR

documentación y un pequeño grafo resumen con una estimación orientativa de la mantenibilidad del proyecto.

Comando a ejecutar:

```
mvn clean install sonar:sonar
```

A continuación se muestran unas imágenes del informe de un proyecto con Sonar.

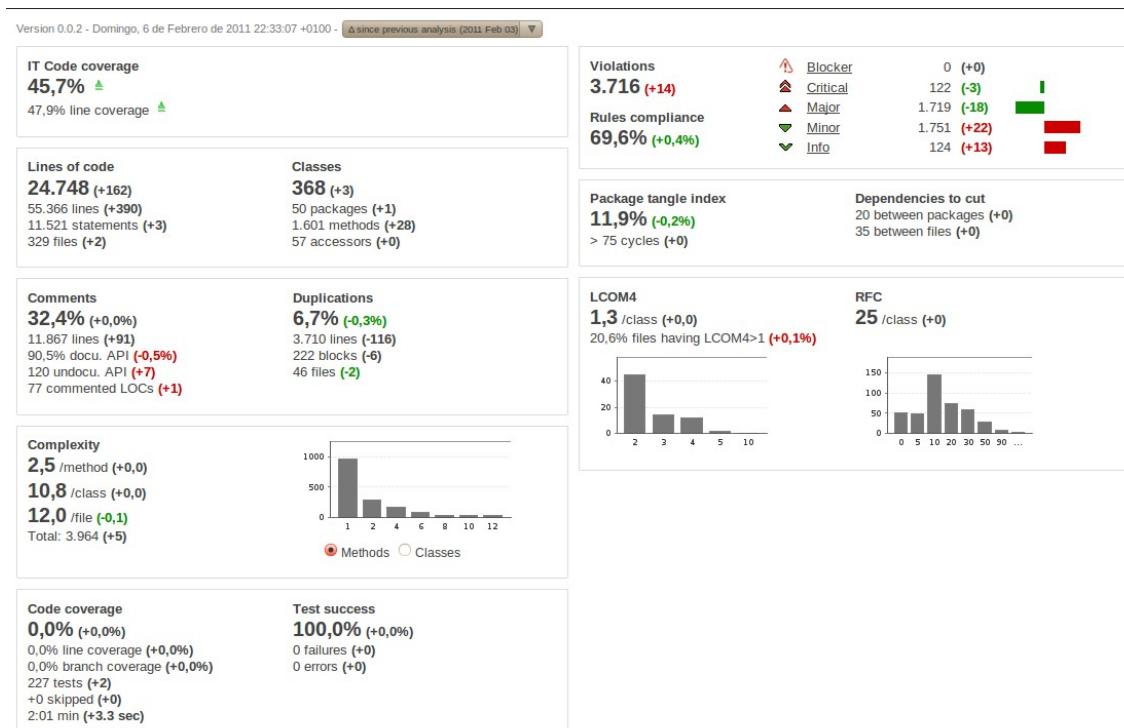


Ilustración 97: Portada de las métricas de un proyecto con Sonar

Además genera una página con el nombre de *Hotspots* (puntos calientes) en la que se muestra una lista de los defectos más importantes que presenta el proyecto agrupados por categorías tales como: lista de reglas más violadas, tests que tardan más en ejecutarse, clases más complejas, etc.

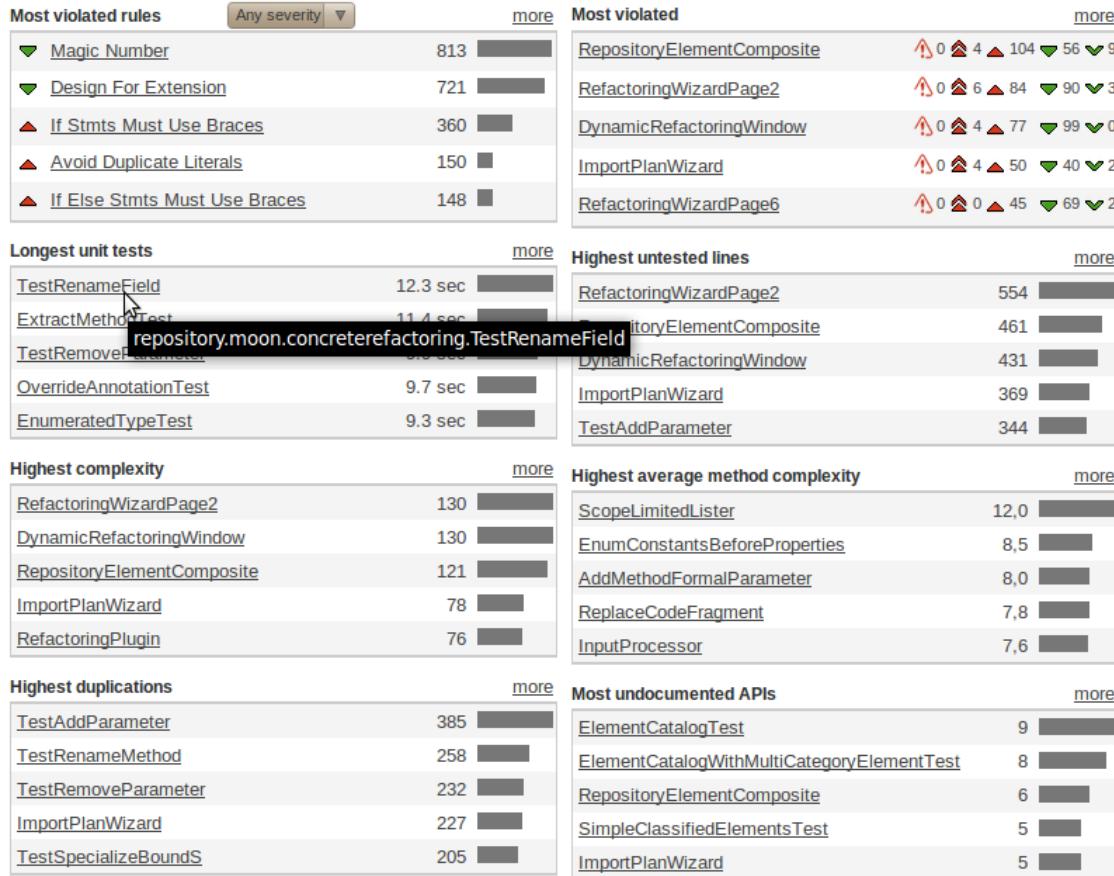


Ilustración 98: Página de HotSpots

También permite hacer retrospectivas de la evolución de las métricas en el proyecto tanto en la página principal como en el apartado *Time machine* (máquina del tiempo). En la página principal la última versión dispone de un combo que permite mostrar junto con cada medida la diferencia con los valores tomados para el proyecto en la última medición o en las mediciones realizadas hace una semana o hace un mes. Las variaciones positivas son marcadas con flechas verdes y las negativas con flechas rojas para indicar desviaciones en el código.

La máquina del tiempo por su parte permite mostrar la evolución en el tiempo de cualquiera de las métricas que se deseen en un gráfico. Ésta es una herramienta muy útil para comprobar la evolución de los valores de ciertas métricas a lo largo de la evolución del proyecto.

Además, el gráfico puede ser configurado para reflejar las medidas que el usuario considere de mayor interés.

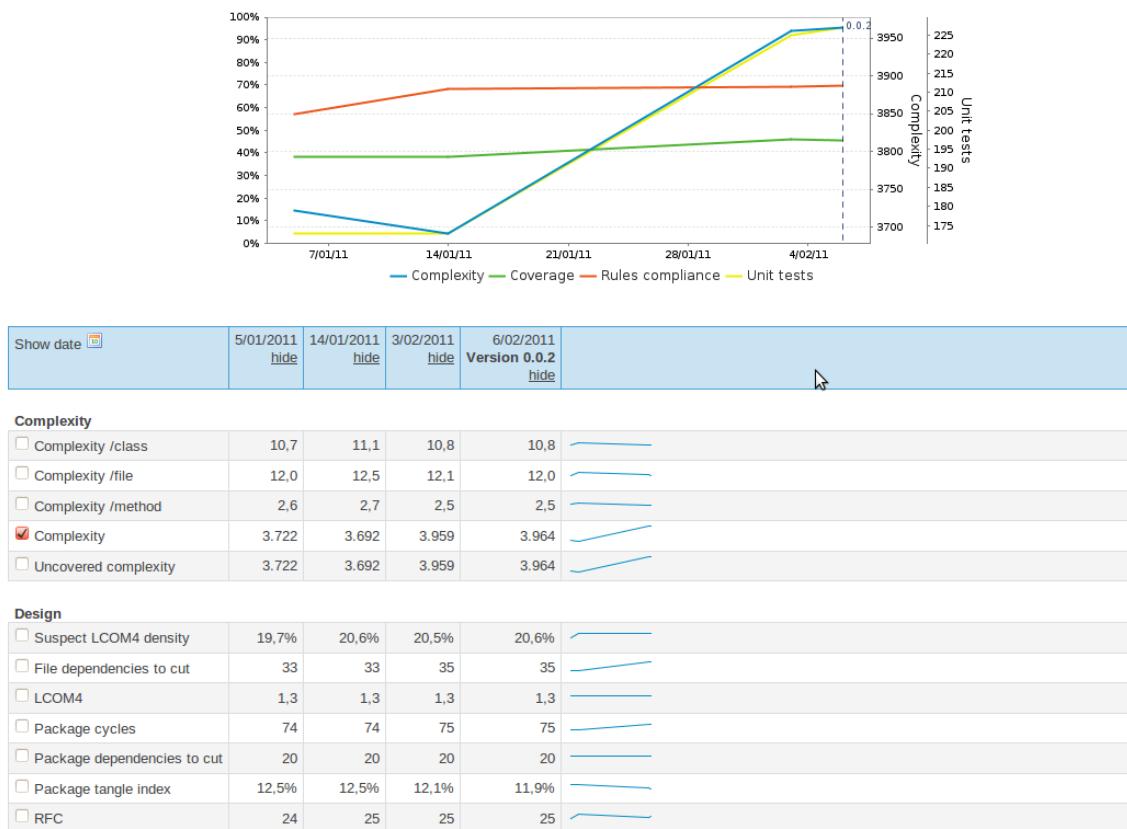


Ilustración 99: Pantalla Time Machine en Sonar

2.5.8. Artefactos generados tras la construcción del proyecto

Como consecuencia del proceso de construcción del plugin se genera una gran cantidad de ficheros. A continuación se mencionarán aquellos que son de especial interés para el programador.

Cuando se ejecuta el proceso de generación de la documentación dentro de la carpeta doc/javadoc del directorio del plugin dynamicrefactoring.plugin se encontrará la documentación en formato JavaDoc del código fuente del plugin.

Cuando se ejecuta el proceso normal de construcción del plugin marcado en el ciclo completo se generará los siguientes artefactos:

Artefactos generados en el directorio del plugin

En `dynamicrefactoring.plugin` dentro de la carpeta `target` se genera un empaquetado en forma de *JAR* que contiene el plugin. Este *JAR* puede ser utilizado para instalar el plugin directamente copiándolo en una instalación de Eclipse en el directorio `dropins`. Al arrancar Eclipse lo reconocerá y lo instalará. Esta es sólo una más de las distintas maneras de instalar el plugin, para obtener más información sobre la forma de instalación recomendada acudir a la sección de instalación del plugin en el anexo 6 del manual de usuario.

Artefactos generados en el directorio de tests

En `dynamicrefactoring.tests` dentro de la carpeta `target/sonar` se genera una lista de ficheros `XML` con los resultados de las comprobaciones de estilo y de métricas del código. Estos ficheros son utilizados por Sonar para generar la página web en la que se muestran todas las medidas obtenidas del proceso de construcción del proyecto. Los ficheros `XML` pueden ser de interés en caso de que el desarrollador quisiera obtener informes personalizados de dichas mediciones más allá de las proporcionadas por la página web de Sonar.

En la carpeta `target/surefire-reports` se genera dos ficheros por cada test ejecutado en el proceso de construcción. Ambos ficheros son una representación en texto plano y en formato `XML` respectivamente de la información sobre la ejecución de la prueba. Si la clase de prueba se ha ejecutado sin fallo la información que aparece se corresponde con los nombres de los tests ejecutados en la clase y el tiempo de ejecución de los mismos. Por el contrario, si existen errores los tests que fallaron aparecen indicados.

En la carpeta `target/cobertura` se genera un informe `HTML` con las líneas de código y porcentaje de ramas cubiertas por la ejecución de los tests. Este informe es generado al ejecutar los tests bajo el control de la librería JaCoCo [Amazon Elastic Compute Cloud (Amazon EC2) n.d.] que se encarga de monitorizar cada una de las instrucciones del proyecto ejecutadas al correr los tests.

Generación del repositorio de instalación del plugin

El módulo de Maven del repositorio *P2* que se encuentra contenido en la carpeta `dynamicrefactoring.p2repository` genera un repositorio que el usuario del plugin puede configurar en Eclipse para instalar el plugin y para obtener actualizaciones del mismo.

Instalación del plugin en el repositorio local de Maven

Al ejecutar la fase de instalación el *JAR* del plugin es almacenado en el repositorio local de Maven en el equipo. La razón de ser de este procedimiento es que de esta manera se hace disponible el plugin a otros proyectos que pudieran tener dependencias sobre dicho plugin.

De esta forma, si para otro proyecto que se estuviese desarrollando en el mismo equipo de forma independiente se decidiese utilizar alguna de las funcionalidades proporcionadas por nuestro plugin, todo lo que se tendría que hacer sería incluir nuestro plugin como una de sus dependencias en su fichero *POM* [Apache Maven n.d.]. Con dicha mínima configuración Maven se encargaría de tomar el *JAR* de nuestro plugin del repositorio local y añadirlo al `classpath` cuando ese otro proyecto se fuese a compilar.

2.6. Integración bibliotecas MOON y JavaMOON

Debido a que nuestro plugin basa la ejecución de sus refactorizaciones en el metamodelo MOON y en su correspondiente extensión para el lenguaje Java, JavaMOON, y que estos se encuentran en mejora continua con la finalidad de soportar más aspectos del lenguaje Java, así como recoger nuevos aspectos que hayan sido incluidos en las nuevas versiones del mismo, se ha dedicado este apartado para detallar el procedimiento a llevar a cabo cuando se desee realizar la integración de nuevas versiones de bibliotecas MOON y JavaMOON en el plugin.

La integración de nuevas versiones de bibliotecas MOON y JavaMOON en el plugin va a afectar tanto al directorio del plugin como al directorio de tests que corresponden con `dynamicrefactoring.plugin` y `dynamicrefactoring.tests` respectivamente.

A continuación se detalla el procedimiento a seguir:

- El primer paso es la copia de las nuevas bibliotecas MOON y JavaMOON en los directorios `dynamicrefactoring.plugin/lib` y `dynamicrefactoring.tests/lib`.
- El segundo paso es la modificación del *Java Build Path* de los subproyectos plugin y tests. Para ello, seleccione el subproyecto `dynamicrefactoring.plugin` y pulse botón derecho del ratón sobre él, aparecerá un menú contextual, entre las opciones disponibles escoja *Properties*, la cual dará paso a una nueva ventana que contendrá una lista en su parte izquierda donde deberemos elegir *Java Build Path*. A continuación seleccione la pestaña *Libraries* y pulse el botón *Add JARs...* y marque

las bibliotecas que acaba de copiar en `dynamicrefactoring.plugin/lib`, pulse OK.

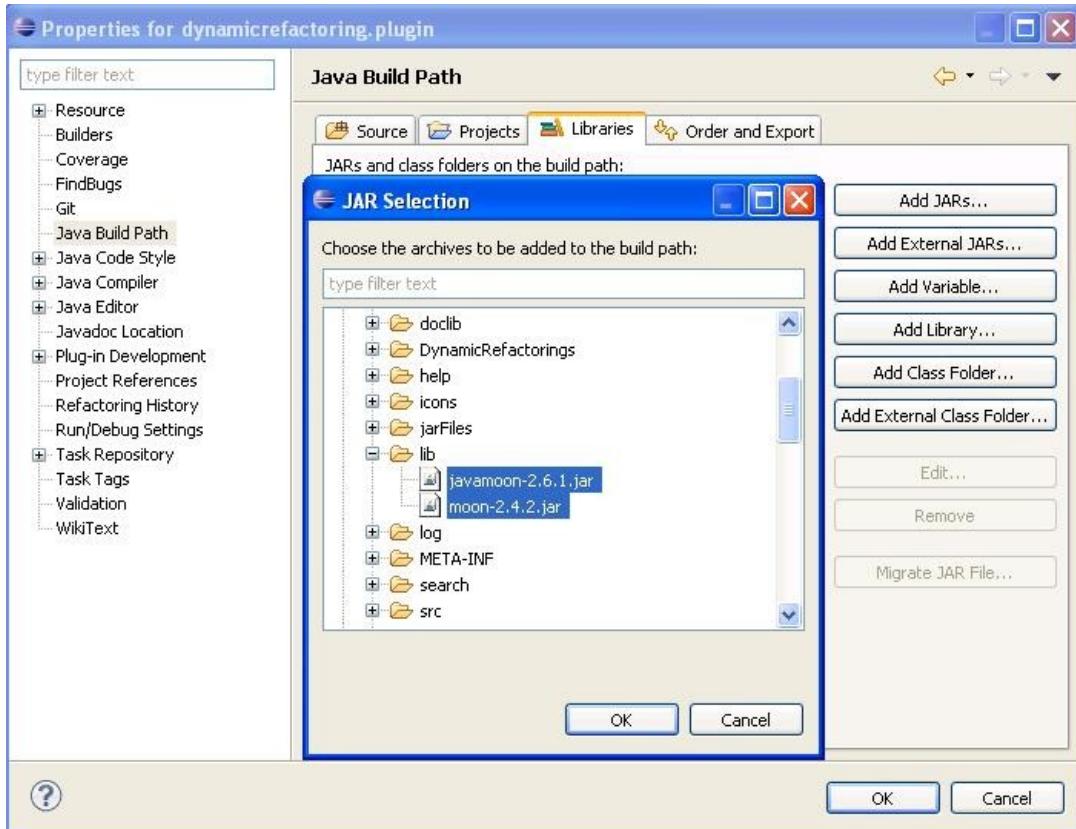


Ilustración 100: Añadir nuevas bibliotecas a Java Build Path

Con ello tendríamos añadidas las nuevas bibliotecas, ahora quedaría eliminar las obsoletas. Para ello, de la lista de bibliotecas seleccione las que corresponden con la versión antigua de las mismas y pulse el botón *Remove*.

- Seguir un procedimiento análogo al del paso anterior para la sustitución de bibliotecas en el subproyecto `tests`, `dynamicrefactoring.tests`.
- En `dynamicrefactoring.plugin` modificar el fichero `plugin.xml`, el cual contiene el `classpath` que va a ser utilizado cuando se ejecute en `plugin`. Para ello, seleccione la pestaña `Runtime` del fichero y diríjase a la sección `Classpath`, en ella deberá añadir las nuevas bibliotecas y eliminar las antiguas utilizando los botones `Add...` y `Remove` respectivamente.

Classpath

Specify the libraries and folders that constitute the plug-in classpath. If unspecified, the classes and resources are assumed to be at the root of the plug-in.

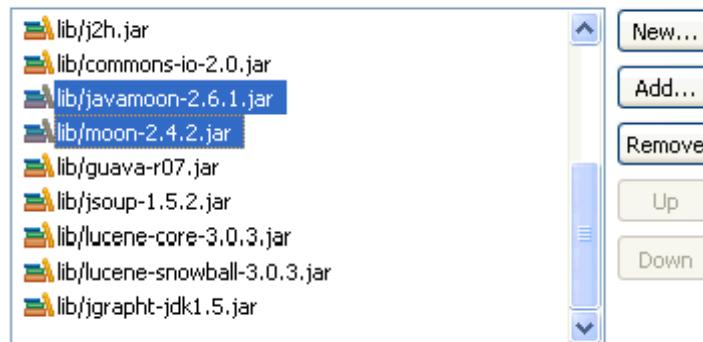


Ilustración 101: Modificación Classpath plugin.xml

- En dynamicrefactoring.tests/META-INF modificar el fichero MANIFEST.MF de la misma forma que se ha detallado en el paso anterior.
- Para comprobar que el proceso se ha realizado correctamente en el subproyecto del plugin se puede observar que el fichero plugin.xml de dynamicrefactoring.plugin en las pestañas *MANIFEST.MF* y *build.properties*, en las secciones de *Bundle-ClassPath* y *bin.includes* respectivamente, efectivamente aparecen las nuevas bibliotecas y las viejas no.

```
Bundle-ClassPath: ./temp,
lib/log4j-1.2.15.jar,
lib/jdom.jar,
.,
lib/swtdesigner.jar,
lib/junit-4.4.jar,
lib/refactoringengine-1.1.1.jar,
lib/j2h.jar,
lib/commons-io-2.0.jar,
lib/javamoon-2.6.1.jar,
lib/moon-2.4.2.jar,
lib/guava-r07.jar,
lib/jsoup-1.5.2.jar,
lib/lucene-core-3.0.3.jar,
lib/lucene-snowball-3.0.3.jar,
lib/jgraphit-jdk1.5.jar
```

Ilustración 102: MANIFEST.MF

```
bin.includes = META-INF,\  
..,\  
bin/,\  
icons/,\  
lib/,\  
log/,\  
plugin.xml,\  
refactoring.properties,\  
lib/jdom.jar,\  
DynamicRefactorings/,\  
lib/swtdesigner.jar,\  
lib/javymoon-2.6.1.jar,\  
lib/moon-2.4.2.jar,\
```

Ilustración 103: *build.properties*

- Comprobar, de la misma forma, que en el subproyecto de tests en dynamicrefactoring.tests/META-INF en la pestaña *build.properties* del fichero MANIFEST.MF que el proceso de sustitución de bibliotecas se ha realizado correctamente.
- Modificar en la clase dynamicrefactoring.RefactoringConstants que se encuentra en dynamicrefactoring.plugin/src las constantes BIBLIOTECA_MOON y BIBLIOTECA_JAVA para que queden actualizadas sus valores a las nuevas versiones de las bibliotecas. A continuación se muestra un ejemplo para las nuevas bibliotecas moon-2.4.2.jar y javymoon-2.6.1.jar.

```
/**  
 * Nombre del fichero .jar que almacena las clases que  
 * componen el modelo moon.  
 */  
public static final String BIBLIOTECA_MOON = "moon-2.4.2.jar";  
  
/**  
 * Nombre del fichero .jar que almacena las clases que  
 * componen el modelo javymoon.  
 */  
public static final String BIBLIOTECA_JAVA = "javymoon-2.6.1.jar";
```

Ilustración 104: Constantes de *dynamicrefactoring.RefactoringConstants*

- Por último, eliminar físicamente las viejas bibliotecas MOON y JavaMOON de los directorios dynamicrefactoring.plugin/lib y dynamicrefactoring.tests/lib.

Una vez realizado el proceso de sustitución el plugin queda preparado para trabajar con las nuevas versiones de las bibliotecas.

3. ASIGNAR INFORMACIÓN DE MARCA AL PLUGIN

La información de marca de una *característica* la constituyen principalmente: su licencia y su copyright, una lista de elementos descriptivos como la página de introducción o la página descriptiva de la funcionalidad del plugin y el icono que identifica al plugin. A excepción de las dos primeras, el resto de propiedades no se definen en la propia característica, sino en un plugin. Lo único que hay que configurar en la característica es cual de todos los plugins que contiene será el que define su información de marca. Esto se define en el fichero `feature.xml`.

A continuación se va a describir como establecer cada uno de los elementos para poder definir la imagen de producto del proyecto.

- Página descriptiva de la funcionalidad del plugin: esta se pueden definir en el fichero `about.ini` del plugin que aporta la información de marca. Este fichero permite definir las tres propiedades siguientes: `aboutText`, `featureImage` y `tipsAndTricksHref`.

La primera propiedad `aboutText`, es una descripción multilínea de la propia característica, que debería aportar nombre, número de versión, información relevante sobre su construcción y otra información de interés. Este texto será visible en el diálogo *About Features* accesible haciendo clic desde el diálogo *About Eclipse* al que se accede desde el menú *Help* y pulsando en *Installation Details*.

La propiedad `featureImage`, es utilizada para referenciar una imagen de 32x32 pixeles que se muestra al representar la característica en el diálogo principal *About* y en el ya citado *About Features*.

Si la documentación de la característica incluye una sección de trucos y consejos, se puede referenciar con la propiedad `tipsAndTricksHref`. A los trucos y consejos se puede acceder desde: *Help > Tips and tricks* .

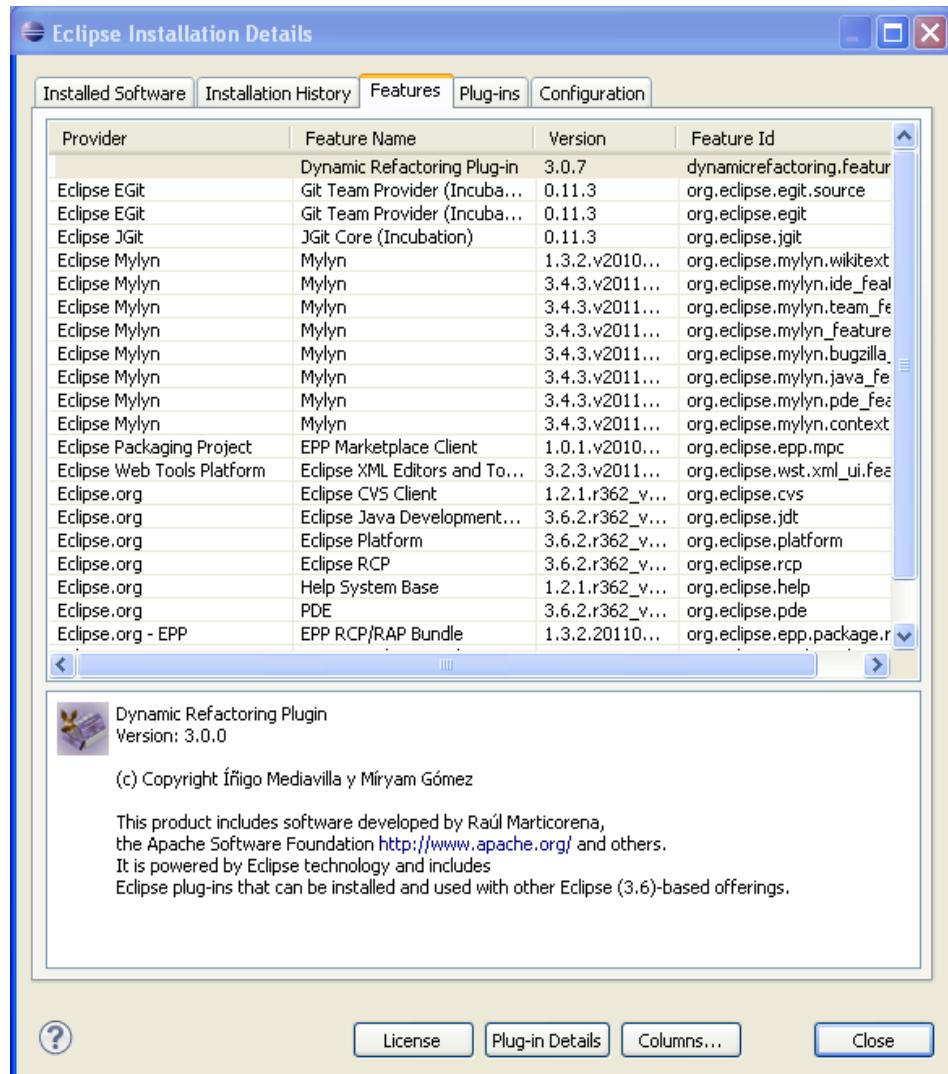


Ilustración 105: Diálogo About Features

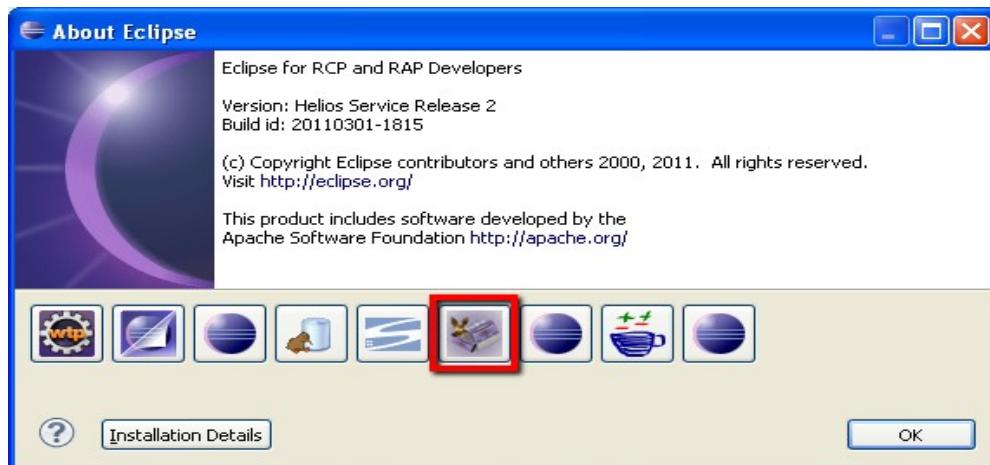


Ilustración 106: Diálogo About Eclipse

- Descripción de la característica, licencia y copyright: Todos ellos se definen en el fichero `feature.xml` del proyecto `dynamicrefactoring.feature` y son mostrados al usuario durante los procesos de actualización e instalación de la característica.

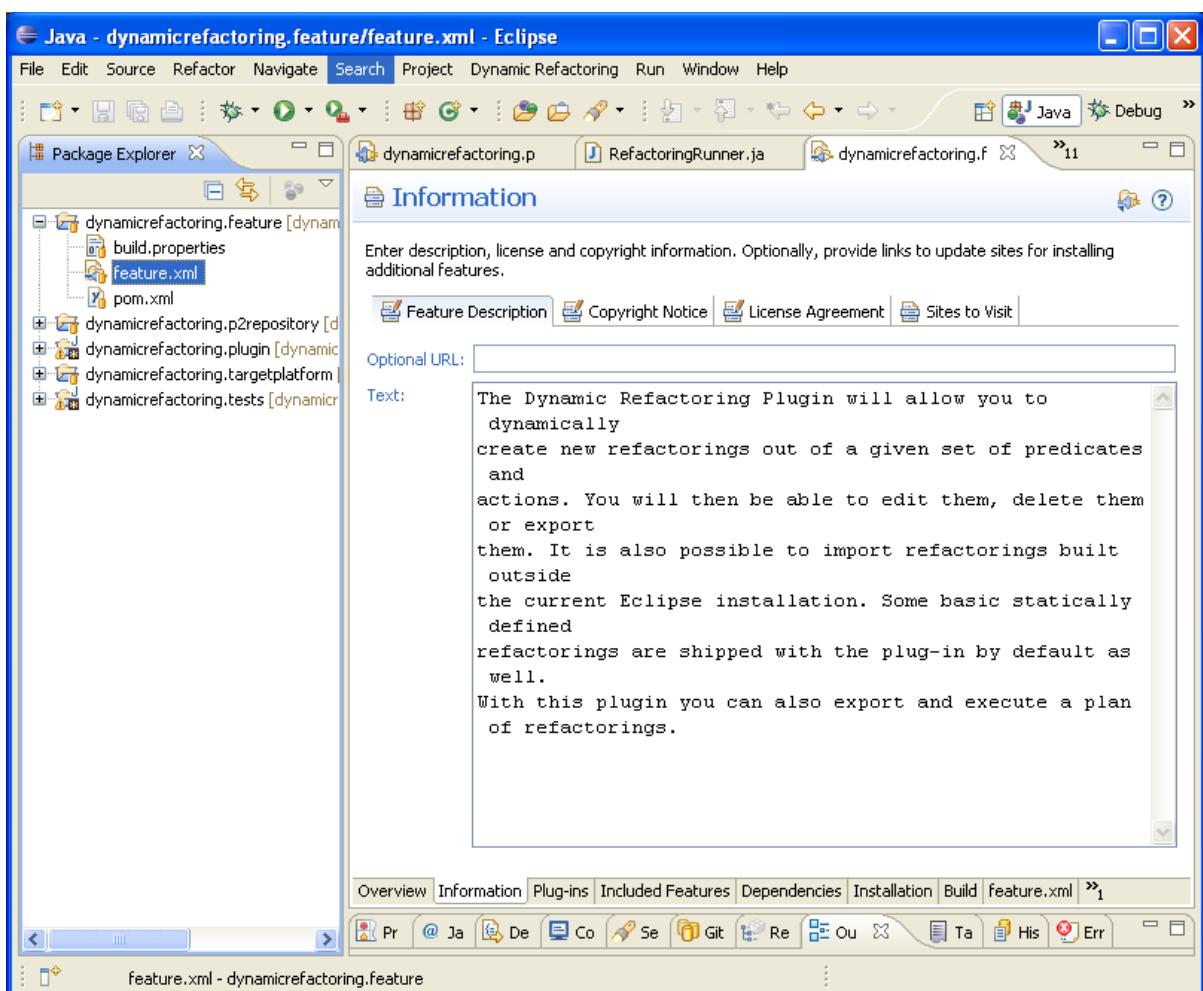


Ilustración 107: Asignar descripción de la característica, licencia y copyright

- Página de introducción: la página de introducción es una oportunidad para familiarizar a los usuarios con el uso de un plugin. Aparece de forma automática tras reiniciar Eclipse la primera vez que el usuario instala el plugin. Esta página se configura extendiendo `org.eclipse.ui.intro.configExtension` y proporcionando tu propia implementación.

Esta es la manera en que se ha definido dicha extensión en el plugin de refactorización:

```
<extension point="org.eclipse.ui.intro.configExtension">
    <configExtension
        configId="org.eclipse.ui.intro.universalConfig"
        content="intro/welcome.xml" />
</extension>
```

El fichero `intro/welcome.xml` es en el que definiremos el contenido de nuestra página de introducción. Este fichero permite definir hojas de estilo y su contenido viene definido por un DTD accesible desde [Intro Content File XML Format n.d.].

4. FIRMA PLUGIN

Firmar un plugin de Eclipse es tan sencillo como firmar cualquier fichero *JAR* de una biblioteca Java. Es necesario generar tres elementos: la clave pública, la clave privada y el certificado.

La clave privada: es una clave que sólo el creador del plugin debe conocer de modo que nadie más puede utilizar tu firma. Un fichero firmado con una clave privada sólo puede ser verificado por su correspondiente clave pública.

Sin embargo, la clave pública y la clave privada por sí solas no son suficientes para verificar una firma. Esto es debido a que tras verificar que un fichero contiene un conjunto de clave pública y privada válido, aún es necesario verificar que la clave pública realmente proviene de la entidad de la que la firma asegura provenir.

Por lo tanto el otro elemento necesario es el certificado que el firmante incluye en el fichero *JAR*. Un certificado es una declaración de una autoridad reconocida que indica quién es el propietario de una clave pública particular.

Los pasos necesarios para generar los elementos anteriores son los siguientes:

4.1. Pasos

En primer lugar se debe generar las claves pública y privada. Esto se hace utilizando la herramienta del *JDK* de Java conocida como *keytool*. Los parámetros que es necesario pasar a *keytool* para generar las claves son:

Opción	Valor
-genkey	Crea una pareja de claves (una pública y una privada asociada)
-alias	Identifica a la pareja de claves con el alias especificado.
-keypass	Contraseña para acceder a la clave privada.
-keystore	Base de datos donde se almacena la pareja de claves generadas.
-storepass	Contraseña de acceso a la base de datos donde se almacena de la pareja de claves generadas.
-dname	Especificación de los datos de la persona o entidad que firma. Estos datos son: <ul style="list-style-type: none"> • CN (CommonName): Nombre común de la persona. • OU (OrganizationUnit): Unidad organizativa o departamento. • O(OrganizationName): Nombre de la organización. • L (LocalityName): Localidad o ciudad. • C (Country): Código del país.
-validity	Número de días que se considerará válido el certificado.

Tabla 57: Parámetros necesarios para la herramienta keytool

Procedemos a generar las claves con el comando:

```
keytool -genkey -alias cjava -keypass kpi135 -keystore mykeystore
-storespass ab987c -dname "cn=Iñigo Mediavilla & Miryam Gomez, ou=LSI,
o=UBU, c=Es" -validity 365
```

A continuación vamos a exportar el certificado que no es un paso necesario, pero sí recomendado para incorporar mayor protección a la firma, dado que el certificado nos permite verificar que la clave pública realmente proviene de la entidad de la que la firma asegura provenir:

```
keytool -export -keystore myKeystore -storespass ab987c -alias cjava
-file CompanyCer.cer
```

Estos dos ficheros los vamos a guardar en la carpeta con nombre `keystore`, del proyecto `dynamicrefactoring.plugin`, que se ha creado para tal efecto y que contiene:

- `mykeystore`: fichero que almacena las claves.

- CompanyCer.cer: fichero que contiene el certificado exportado.

Ahora toca incluir el proceso de firmado del *JAR* de nuestro plugin una vez construido con las claves generadas. Normalmente este proceso se realiza directamente a través de la herramienta *jarsigner* que viene incorporada con el *JDK* de Java desde línea de comandos. Sin embargo, en el caso del plugin de refactorización se consideró útil incorporar el firmado de los *JAR* generados al proceso de construcción del proyecto para evitar de ese modo tener que firmar de forma manual el plugin cada vez que se fuera a generar. La configuración del proyecto para firmar los ficheros *JAR* se realiza desde el fichero *pom.xml* de la raíz del proyecto, lo que permite firmar los ficheros que generan varios de los proyectos. La configuración a agregar en dicho fichero es la siguiente:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jarsigner-plugin</artifactId>
  <version>1.2</version>
  <executions>
    <execution>
      <id>signing</id>
      <goals>
        <goal>sign</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <keystore>../keystore/mykeystore</keystore>
    <alias>cjava</alias>
    <storepass>ab987c</storepass>
    <keypass>kpi135</keypass>
  </configuration>
</plugin>
```

Donde con el atributo *keystore* indicamos la ruta en donde se encuentra la clave generada.

```
<keystore>../keystore/mykeystore</keystore>
```

El resto de atributos de la etiqueta *configuration* deben tener los valores con los que se ha generado la clave:

```
<alias>cjava</alias>
<storepass>ab987c</storepass>
<keypass>kpi135</keypass>
```

5. PRUEBAS

Dado que el número de métodos de test implementados es muy alto como para detenerse a explicar cada uno de ellos, en esta sección se explicará de forma breve y general qué aspectos de la aplicación prueba cada una de las clases con tests. En casos en los que haya varias clases de pruebas con tests muy similares dentro del mismo paquete se comentarán de forma agrupada las clases del paquete.

5.1. Pruebas Unitarias

En primer lugar se hablará de las pruebas unitarias, es decir, las pruebas destinadas a probar los módulos que componen el proyecto de forma individualizada.

dynamicrefactoring.domain.RefactoringMechanismTest

Un mecanismo es una precondición, acción o postcondición de las que componen una refactorización. El tipo de un mecanismo es una enumeración con cualquiera esas posibles opciones a las que puede pertenecer un mecanismo. Las pruebas que contiene esta clase comprueban que dado un tipo (por ejemplo, el tipo acción), los métodos que existen para acceder a todos los elementos en el repositorio o para comprobar si un elemento es de tipo MOON o JavaMOON son correctos.

dynamicrefactoring.domain.metadata.classifications.xml.imp.

JAXBClassificationsReaderTest

Esta clase realiza sus pruebas a través de su clase padre. Lo único que esta clase hace es asignar en su método `setUp()` un lector de clasificaciones que utiliza la biblioteca JAXB de Java. Con este lector las pruebas se ejecutan en los métodos de la clase padre. Esta arquitectura permite que si fuera necesario definir otro lector, este no tuviera que definir los tests de nuevo, sino que sólo necesitaría definir su método `setUp()`.

Las pruebas definidas para un lector de clasificaciones consisten en proporcionar al lector ficheros inválidos y comprobar que saltan las excepciones esperadas. Por otro lado se le pasan ficheros correctos y se comprueba que construye las clasificaciones deseadas.

dynamicrefactoring.domain.metadata.classifications.xml.Imp.

PluginClassificationsCatalogTest

El catálogo de clasificaciones del plugin contiene los métodos necesarios para acceder

a todas las clasificaciones disponibles y realizar modificaciones sobre ellas.

Esta clase de tests contiene pruebas para agregar una categoría, agregar una clasificación, convertir una clasificación unicategoría en multicategoría y viceversa, eliminar una clasificación o renombrarla entre otros. Además también contiene pruebas para comprobar que ciertas operaciones como intentar insertar una clasificación con el nombre de otra existente provocan que salte una excepción.

dynamicrefactoring.domain.metadata.condition

Las clases de este paquete comprueban que las condiciones se comportan correctamente. Para comprobar que las condiciones se comportan de forma válida lo que se hace es pasarlas elementos cuyo valor se conoce y asegurarse de que el resultado que se devuelve al aplicar la condición sobre dichos elementos es el esperado. Por ejemplo para validar que una condición de una palabra clave se comprueba correctamente se le pasa un objeto que contiene la palabra clave de la condición y se asegura de que devuelve verdadero indicando que el objeto cumple la condición.

dynamicrefactoring.domain.metadata.imp. ElementCatalogTest

El catálogo de elementos es un contenedor de elementos que se agrupan por categorías y que se pueden filtrar en base a criterios. El contenedor contiene dos grupos: los elementos filtrados y los no filtrados. Cuando se aplica un filtro todos los elementos que no cumplen con el filtro pasan al grupo de los elementos filtrados.

Las pruebas sobre esta clase consisten principalmente en aplicar y eliminar filtros para comprobar que la organización de los elementos dentro del contenedor es correcta y que los elementos ocupan en todo momento la categoría y grupo que deben ocupar.

dynamicrefactoring.domain.metadata.imp. ElementCatalogWithMultiCategoryElementTest

Esta clase, de igual modo que la clase anterior, realiza pruebas sobre el catálogo de elementos filtrables. La diferencia es que la clase anterior se limitaba a probar conjuntos de elementos en los que un elemento sólo podía pertenecer a una categoría de la clasificación en un momento dado (clasificaciones unicategoría). En este caso las pruebas se realizan sobre clasificaciones en las que los elementos pueden pertenecer a más de una categoría a la vez. Estos conjuntos presentan algunos casos especiales y por eso es necesario probarlos de forma separada.

De nuevo las pruebas consisten en aplicar y eliminar filtros y comprobar la organización de los elementos.

dynamicrefactoring.domain.metadata.imp. SimpleClassifiedElementsTest

`ClassifiedElements` son conjuntos de elementos organizados por categorías en base a una clasificación. Este tipo de objetos sólo permiten consultas sobre cuál es la clasificación en que se basan y qué elementos pertenecen a una categoría. Estos objetos son los que devuelven los catálogos de elementos cuando se les pregunta qué elementos están filtrados o cuales no.

Dado que los objetos de esta clase son inmutables los métodos de prueba para ella consisten en crear objetos mediante su constructor, para luego comprobar que se han creado correctamente. En concreto se crea una clasificación y un conjunto de elementos agrupados por categorías y se pasan al constructor de la clase. Luego se llama a los métodos de consulta para ver si devuelven los resultados esperados. Esto se hace para conjuntos de elementos filtrados y no filtrados.

dynamicrefactoring.domain.xml.XMLRefactoringsCatalogTest

La clase `XMLRefactoringsCatalog` define el comportamiento del repositorio único de refactorizaciones del plugin. Es por tanto la clase a la que se va a consultar sobre las refactorizaciones que existen actualmente en el plugin y también la única clase que permite realizar modificaciones sobre el catálogo de refactorizaciones. El prefijo `XML` en su nombre se debe a que todos los cambios sobre las refactorizaciones que se realizan a través de esta clase se reflejan en los ficheros `XML` de definición de las refactorizaciones.

Las pruebas consisten en consultas sobre las refactorizaciones existentes y también en modificaciones, es decir, en añadir elementos nuevos y actualizar elementos del catálogo. Dado que las refactorizaciones con imágenes y ejemplos llevan ficheros asociados y sus actualizaciones son más complejas hay pruebas específicas para añadir y actualizar refactorizaciones de estos tipos.

dynamicrefactoring.interfaz.editor.classifieditor. CategoriesSectionTest

A pesar de que la clase `CategoriesSection` del editor de clasificaciones es una clase de interfaz, se diseñó teniendo en cuenta ser testada. En primer lugar los métodos que el editor utiliza para realizar modificaciones sobre las categorías son accesibles desde la clase de tests. En segundo lugar se utilizó la inyección de dependencias para asignar un catálogo

a la clase, en lugar de que la clase se asignara uno propio. Esto permite que a la hora de realizar las pruebas se pasa un catálogo que no realiza modificaciones sobre ningún XML sino que simplemente lo hace sobre memoria, con lo que las pruebas se simplifican.

Para realizar las pruebas unitarias se invoca a métodos para realizar modificaciones sobre las categorías que ya hemos dicho que eran accesibles para la clase del test y luego se comprueba que las modificaciones sobre el catálogo han sido las adecuadas.

dynamicrefactoring.interfaz.editor.classifieditor. ClassifListSectionTest

La clase `ClassifListSection` esta diseñada con los métodos que ejecutan las modificaciones efectivas sobre el catálogo accesibles para la clase de pruebas y con un catálogo que se le pasa a través de inyección de dependencias.

Las pruebas se enfocan del mismo modo que en la clase anterior, sólo que en este caso las pruebas realizan modificaciones sobre las clasificaciones directamente en lugar de sobre las categorías.

dynamicrefactoring.interfaz.wizard.search.internal.

SearchableTypeIndexerTest

La clase `SearchableTypeIndexer` es la clase que se encarga de generar los índices a partir de los cuales se realizarán las búsquedas de los elementos en el asistente de creación de refactorizaciones.

Las pruebas que se realizan sobre esta clase consisten en generar el índice para un elemento. Entonces las pruebas se aseguran de cosas como que el número de elementos indizados haya sido el correcto o de que los elementos se han indizado de forma adecuada a través de búsquedas en el índice.

dynamicrefactoring.interfaz.wizard.search.javadoc.

EclipseBasedJavadocReaderTest

`EclipseBasedJavadocReader` es un lector de documentación de los elementos del repositorio de MOON y JavaMOON (entradas, precondiciones, acciones y postcondiciones). Sus métodos reciben el nombre completamente cualificado de una clase y devuelven la documentación asociada a la misma.

Las pruebas consisten en probar los tres tipos de situaciones que se pueden dar para las consultas de documentación sobre una clase: la clase existe y tiene documentación, la

clase existe pero no tiene documentación y finalmente la clase no existe. En el primer caso, la prueba valida que la documentación se obtiene y es la esperada. En el caso de que la clase no tenga documentación, se comprueba de que se devuelve la cadena vacía. En el último caso, cuando la clase no existe debe saltar una excepción para indicarlo.

5.2. Pruebas de interfaz

Tras haber comentado las pruebas unitarias, en este apartado se comentarán las pruebas de interfaz realizadas. Estas pruebas no comprueban que los módulos se comportan adecuadamente de forma individualizada, sino más bien que varios módulos realizan su función correctamente de forma conjunta, por lo tanto estas pruebas se pueden considerar pruebas de integración.

Estas pruebas se han realizado con la herramienta SWTBot. Para más información sobre la herramienta esta se encuentra explicada en la sección de *Técnicas y Herramientas de la Memoria*.

dynamicrefactoring.interfaz.editor.classifieditor. ClassifEditorUITest

Esta prueba consiste en utilizar las funciones del editor de clasificaciones desde la propia interfaz de forma automatizada. Para ello se hacen cosas como agregar una clasificación nueva, renombrar una clasificación existente o eliminarla y tras cada uno de los casos se comprueba que los cambios se han realizado de forma correcta y que la interfaz se ha actualizado con dichos cambios.

dynamicrefactoring.interfaz.view.RefactoringCatalogBrowserViewTest

En esta prueba de interfaz se comprueba que al escoger una clasificación el árbol del catálogo se actualiza de forma correcta. También se comprueba que el panel de resumen con toda la información de una refactorización se muestra de forma adecuada cuando se selecciona una refactorización y se hace doble clic sobre ella. Finalmente también se comprueba que si se hace doble clic sobre una refactorización con imagen asociada se muestra el panel de imagen y lo mismo para una refactorización con ejemplos.

dynamicrefactoring.interfaz.wizard.CreateRefactoringTest

Esta clase contiene un conjunto de pruebas sobre la primera página del asistente de creación de refactorizaciones. El desarrollo de las pruebas consiste en ir modificando los atributos de una refactorización a través de los campos de esta primera pantalla del

asistente. Tras cada cambio se prueba si la interfaz ha reaccionado de forma adecuada ya que el asistente está diseñado para que sólo se permita dar el paso a la página siguiente cuando todos los campos obligatorios hayan sido rellenados.

dynamicrefactoring.interfaz.wizard. CreateRefactoringWithWizardPage2Test

Esta clase contiene pruebas para la segunda de las páginas del asistente de creación y edición de refactorizaciones.

En concreto se realizan pruebas sobre la asignación de entradas a la refactorización, para comprobar que las entradas se agregan correctamente y que cuando se selecciona una entrada principal adecuada que concuerda con el ámbito escogido en la página anterior, el asistente permite continuar a la siguiente página. También hay una prueba en la que se realiza una búsqueda sobre una entrada del repositorio para validar que la búsqueda devuelve el elemento buscado.

dynamicrefactoring.interfaz.wizard. EditRefactoringCategoriesTest

Esta clase está dedicada exclusivamente a las pruebas sobre el árbol de selección de categorías del asistente de creación de refactorizaciones. Se prueba que no se permite al usuario seleccionar más de una categoría de una clasificación unicategoría y que sí se puede asignar más de una categoría en una clasificación multicategoría. También se juega con la opción que proporciona el árbol de activar todos los elementos a la vez.

5.3. Pruebas de Cobertura

La cobertura de las pruebas ha sido incrementada apreciablemente en esta última versión del plugin de refactorización. Desde un principio se consideró la calidad como un objetivo principal y las pruebas son el mejor medidor y el mejor seguro de que un sistema hace en todo momento lo que se supone que debe hacer y está en la mayor medida posible libre de errores.

En la documentación de la anterior versión de este proyecto se realizó un estudio de la cobertura exclusivamente sobre un grupo reducido de paquetes que no tomaban parte en la interfaz del plugin. Para esta versión se ha considerado que la interfaz es una parte muy crucial del producto y en ella están definidos un conjunto importante de requisitos que las pruebas deben poner en ejercicio. Por tanto el análisis de cobertura se centrará ahora en todos los paquetes que forman el proyecto dedicando especial atención a los nuevos pero sin descartar la evolución de la cobertura en los antiguos. En la parte final se realizará una

comparativa global de cual ha sido la evolución de la cobertura en el proyecto comparando un informe que se generó cuando se retomó el proyecto a partir de la versión anterior con cual es el estado final en que han quedado las pruebas.

Los informes de cobertura inicial y final en que se basa este análisis están disponibles en el CD que se adjunta con la documentación del proyecto.

5.3.1. Paquetes del dominio

Las clases de los paquetes del dominio son clases de especial importancia dado que el resto de clases de otras capas dependen de ellas, por tanto deben de estar libres de errores y deben existir pruebas que comprueben que lo están en todo momento. Estos factores se han tenido en cuenta durante el desarrollo del plugin y es por ello que las clases del dominio tienen porcentajes de cobertura especialmente altos.

Para el caso de los paquetes añadidos al proyecto y que no existían en el proyecto anterior las coberturas son las que se muestran en la siguiente tabla:

Paquete	Cobertura
dynamicrefactoring.domain.condition	100%
dynamicrefactoring.domain.metadata.classifications.xml	87%
dynamicrefactoring.domain.metadata.classifications.xml.imp	92%
dynamicrefactoring.domain.metadata.condition	88%
dynamicrefactoring.domain.metadata.imp	95%
dynamicrefactoring.domain.metadata.interfaces	100%

Tabla 58: Coberturas de los nuevos paquetes añadidos al dominio

Como se puede comprobar en todos los casos las coberturas superan el 85% lo que indica que las clases de todos estos paquetes han sido probadas de forma exhaustiva.

Además el paquete raíz del dominio dynamicrefactoring.domain ha pasado de una cobertura del 33 al 50%.

5.3.2. Paquetes de persistencia XML

Dentro de los paquetes de persistencia XML del código fuente del proyecto se ha añadido un paquete que contiene las clases relacionadas con el nuevo repositorio único de refactorizaciones que se ha implementado: el paquete `dynamicrefactoring.domain.xml`. Al igual que con el resto de clases implementadas en el proyecto se ha tenido especial atención a las pruebas definidas para las clases de este importante paquete del proyecto. Gracias a las pruebas implementadas se ha conseguido una cobertura del 81% sobre este paquete. En el resto de subpaquetes de la capa de persistencia se han conseguido coberturas incluso más altas:

Paquete	Cobertura
<code>dynamicrefactoring.domain.xml</code>	81%
<code>dynamicrefactoring.domain.xml.reader</code>	93%
<code>dynamicrefactoring.domain.xml.writer</code>	91%

Tabla 59: Cobertura de los paquetes de persistencia XML

5.3.3. Paquetes de la interfaz

Entre los paquetes de la interfaz se ha conseguido un notable incremento de la cobertura debido a la introducción de las pruebas de interfaz gráfica con SWTBot. Se partía de una situación en la que inicialmente el paquete con mayor cobertura de entre todos los paquetes de la interfaz tenía un 5% de cobertura sobre el total de sus líneas del código.

Desde el principio del proyecto se consideró que era necesario mejorar las pruebas de estos paquetes, dado que estas coberturas tan bajas hacían imposible comprobar si ciertos cambios en el código introducían defectos en la interfaz de la aplicación. La única alternativa era ejecutando el plugin y comprobándolo de forma manual, lo que traía numerosos problemas y principalmente hacía que la aplicación no se probara en muchos casos. Esta era la mejor manera de que se introdujeran defectos que en muchos casos se descubrirían mucho después de haber sido introducidos y en otros definitivamente pasaban desapercibidas llegando al producto entregado al usuario.

Teniendo en cuenta esta situación de partida las nuevas clases que se incorporaron a la interfaz se incluyeron dentro de la filosofía de pruebas. Con esta nueva filosofía los resultados de cobertura que se ha conseguido en los nuevos paquetes es muy positiva como se puede ver en la siguiente tabla en la que los nombres de los paquetes se representan sin el paquete raíz (`dynamicrefactoring.interfaz`) para evitar la redundancia:

Paquete	Cob. Final
editor.classifieditor	84%
wizard.classificationscombo	83%
wizard.search.internal	83%
wizard.search.javadoc	87%

Tabla 60: Cobertura de los nuevos paquetes de la interfaz

En el caso de los paquetes que ya existían se partía de una situación difícil porque era complejo incorporar pruebas sobre código antiguo con el que se tenía menos familiaridad y que por no tener pruebas previas no estaba diseñado para ser probado. A pesar de ello se decidió centrarse en los dos paquetes más importantes y de mayor tamaño: `view` y `wizard`. En la tabla siguiente se muestra la evolución de la cobertura de los paquetes que ya existían en la versión previa del plugin, resaltando la positiva evolución de los dos más importantes.

Paquete	Cob. Inicial	Cob. Final
<raíz>	5%	4%
dynamic	1%	1%
view	0%	47%
wizard	0%	44%
wizard.listener	0%	9%

Tabla 61: Evolución de la cobertura de los paquetes de la interfaz

El resultado se puede considerar muy positivo dado que se ha conseguido el objetivo al incrementar muy notablemente la cobertura de los dos paquetes más importantes a la vez que la cobertura del resto de paquetes o se ha mantenido o se ha incrementado aunque menos notoriamente.

5.3.4. Resto de paquetes

No se va a entrar en el análisis independizado del resto de paquetes que no han sido el foco central de las modificaciones derivadas de este proyecto. Sin embargo, de forma general se puede decir que han seguido la misma tendencia en cuanto a incremento de cobertura del código. En el peor de los casos se ha mantenido la cobertura de la que se partía. En el resto de casos hay incrementos de la cobertura, con algunos ejemplos muy positivos como el de `dynamicrofactoring.util` que ha pasado de una cobertura del 39 al

60%.

Paquete	Cob. Inicial	Cob. Final
dynamicrefactoring.action	0%	18%
dynamicrefactoring.integration	0%	0%
dynamicrefactoring.integration.selectionhandler	0%	0%
dynamicrefactoring.preferences	0%	0%
dynamicrefactoring.util	39%	60%
dynamicrefactoring.util.io	45%	45%
dynamicrefactoring.util.io.filter	6%	13%
dynamicrefactoring.util.processor	0%	0%
dynamicrefactoring.util.selection	0%	0%
repository	100%	100%
repository.java.concreteaction	54%	73%
repository.java.concretesfunction	0%	79%
repository.java.concretepredicate	75%	97%
repository.java.concreterefactoring	31%	82%
repository.moon	74%	74%
repository.moon.concreteaction	64%	92%
repository.moon.concretesfunction	48%	85%
repository.moon.concretepredicate	92%	95%
repository.moon.concreterefactoring	80%	81%

Tabla 62: Evolución de la cobertura del resto de paquetes

5.3.5. Análisis global

El análisis general que se puede obtener de los resultados de cobertura del código se puede considerar muy positivo. La cobertura es alta, superior al 80% en todos los nuevos paquetes incorporados y se ha conseguido incrementar la cobertura de la mayoría de los paquetes que ya existían. Especialmente son los casos de los paquetes view y wizard de la interfaz, dos de los paquetes más importantes y más grandes en los que se ha llegado a prácticamente el 50% de cobertura partiendo desde una cobertura inicial de cero.

Esta evolución positiva se puede resumir en un contraste entre el 38% de cobertura con que se partía al 59% que se ha alcanzado, es decir, un incremento del 20% de la cobertura total, todo ello a pesar del crecimiento experimentado por el proyecto.

6. MÉTRICAS

En este punto se describirán las técnicas empleadas para la medición de los valores obtenidos para cada una de las métricas elegidas en el apartado dedicado a esta finalidad del *Anexo 3 - Especificación del Diseño*. Asimismo, se mostrarán los resultados obtenidos para cada una de las mediciones y se valorarán de manera oportuna en relación a los resultados esperados.

6.1. Evaluación de métricas

Para la medición práctica de las métricas elegidas se ha utilizado la aplicación SourceMonitor. Esta sencilla, pero potente aplicación, permite aplicar filtros y selecciones al conjunto de clases sobre las que se van a calcular los valores de las métricas. Se ha aprovechado esta funcionalidad para ejecutar diferentes pruebas que permitan obtener evaluaciones del producto desarrollado desde diferentes puntos de vista.

Es importante destacar que las métricas obtenidas para un producto software deben interpretarse según el tipo de producto que es. Una aplicación desarrollada en un entorno o en un lenguaje concreto, puede dar lugar a resultados muy diferentes si se compara con desarrollos de otros ámbitos (piénsese nada más en el número de líneas de código generadas en un lenguaje de programación u otro en función de si es de más alto o más bajo nivel).

En el caso de esta aplicación, resulta difícil limitar el producto software a un ámbito concreto. Se trata de un desarrollo bastante complejo, que envuelve componentes que se podrían situar en distintos entornos. Así, se puede distinguir:

- *repositorio*: núcleo de paquetes que constituyen algo muy similar a una biblioteca o un framework.
- *parte operativa del plugin*: conjunto de paquetes muy interrelacionados, con jerarquías más o menos complejas y que deben implementar funcionalidades concretas bastante variadas.
- *parte gráfica del plugin*: conjunto de paquetes dependientes de muchos elementos gráficos del API gráfica de Eclipse, con el que pretenden integrarse.

Así, no sería de extrañar que cada una de estas partes del proyecto diera lugar a medidas muy diferentes entre sí y más o menos alejadas de los valores esperables. Por ejemplo, la mayoría de las clases del repositorio son elementos concretos de tipo *acción* o *predicado*, que constan fundamentalmente de solo dos métodos, más allá de que algunas cuenten con métodos auxiliares para la implementación. Por otro lado, la parte relacionada con la interfaz podría tener métodos excesivamente largos y complejos, debido a la complejidad que entraña la generación de interfaces gráficas, y más aún en un caso como este en que la interfaz debe generarse dinámicamente en algunas situaciones.

Por tanto, teniendo en cuenta todo lo que se acaba de comentar, se ha decidido llevar a cabo las mediciones ideadas sobre cuatro subsistemas del proyecto, de forma que se puedan separar posibles influencias negativas de alguna de las partes, en caso de que las hubiese. Estas cuatro variantes sobre las que se han tomado medidas son:

- ◆ El proyecto completo.
- ◆ Los paquetes correspondientes al plugin, sin el repositorio.
- ◆ Los paquetes correspondientes al área operativa del plugin.
- ◆ Los paquetes correspondientes al área gráfica del plugin.

Cabe destacar además, que se ha filtrado en todos los casos la clase *Messages*, producto del mecanismo de internacionalización de *Eclipse*, y que aparece en cada paquete como una clase de datos, lo que podría ejercer una influencia negativa en las métricas de la parte desarrollada.

6.2. Resultados obtenidos

Una vez decidido el procedimiento por el que se obtendrán las medidas de cada una de las métricas evaluadas, se pasa a calcular las métricas de la aplicación y se van a comparar con las métricas obtenidas en las anteriores versiones del plugin.

La siguiente ilustración recoge las métricas elegidas junto con el identificador que vamos a utilizar en las tablas comparativas que a continuación de esta se muestran:

M-Número métrica	Nombre
M-1	Líneas de código
M-2	Nº clases
M-3	Nº sentencias
M-4	Nº medio de métodos por clase
M-5	Nº medio de sentencias por método
M-6	Profundidad media del bloque
M-7	Complejidad media

Ilustración 108: Tabla métricas

A continuación damos paso a mostrar las tablas comparativas entre los valores obtenidos para las métricas seleccionadas de las distintas versiones del proyecto. Como se puede observar aparecen los cuatro subsistemas que en el apartado anterior se explicaron; *completo*, *plugin*, *parte operativa* e *interfaz gráfica* y para cada uno de ellos una columna por cada versión del proyecto.

M-Num	Esperado	Completo			Plugin		
		Versión 1	Versión 2	Versión 3	Versión 1	Versión 2	Versión 3
M-1	-	-	38.024	52.707	-	26.057	37.794
M-2	-	-	294	434	-	182	297
M-3	-	-	12.199	17.597	-	8.931	13.044
M-4	[4-16]	3,86	3,82	4,23	4,39	4,46	4,84
M-5	[6-12]	7,32	7,22	5,97	7,89	7,74	5,89
M-6	[1-2,2]	1,98	2,12	1,93	2,09	2,29	2,02
M-7	[2-4]	2,44	2,59	2,23	2,48	2,67	2,14

Ilustración 109: Comparativa para los subsistemas Completo y Plugin

M-Num	Esperado	Parte Operativa			Interfaz gráfica		
		Versión 1	Versión 2	Versión 3	Versión 1	Versión 2	Versión 3
M-1	-	-	13.220	19.141	-	14.827	21.321
M-2	-	-	81	130	-	108	178
M-3	-	-	3.913	5.471	-	5.641	8.380
M-4	[4-16]	4,73	5,2	6,11	4,35	4,19	4,1
M-5	[6-12]	5,13	6,13	3,96	9,87	9,06	7,98
M-6	[1-2,2]	1,81	2,04	1,73	2,21	2,43	2,2
M-7	[2-4]	2,28	2,51	1,95	2,62	2,76	2,37

Ilustración 110: Comparativa para los subsistemas Parte Operativa y Interfaz Gráfica

En estas tablas, tanto para la versión actual del proyecto como para anteriores, se ha realizado una evaluación donde se puede observar colores rojos para aquellos valores de métricas que no están entre los valores recomendados y colores verdes para aquellos que con el cambio de versión han supuesto que pasasen de no cumplir estos valores recomendados a sí hacerlo, respecto de la versión anterior.

Como se puede apreciar, prácticamente todos los valores quedan entre los márgenes mínimo y máximo esperable establecidos. Tan sólo hay cuatro medidas dentro del presente proyecto en las que el valor obtenido se desvía un poco del esperado, de estas tres pueden considerarse despreciables ya que se encuentran muy próximas al límite inferior, el más alejado a un 0,1. Por tanto, la única desviación considerable se encuentra en la *Parte Operativa* para la métrica *M-5* que corresponde con *Número medio de sentencias por método*, esta medida se ha visto influenciada y justificada por la necesidad de aplicar el patrón de diseño *Builder* tanto a *DynamicRefactoringDefinition* como a *InputParameter*, que corresponde con la definición de refactorizaciones y la de las entradas de la misma respectivamente, también se ha visto afectada por el paquete de clases que ofrecen la funcionalidad de la definición de las diferentes condiciones de filtro que se pueden crear para aplicar a las refactorizaciones, así como por las clases que representan la parte operativa de las acciones a realizar en las diferentes vistas.

También se puede observar como algunos valores de las métricas que en la versión anterior del proyecto no se encontraban entre los valores recomendados han mejorado en la presente versión entrando ya en los intervalos de referencia. El resto de valores para las métricas no han sufrido grandes alteraciones, esto indica que se ha conseguido ampliar la funcionalidad y mejorar la interfaz de la aplicación sin perjudicar el diseño interno de la misma.

Además, también se puede apreciar la evolución del tamaño del proyecto respecto a la versión anterior, observando las métricas *M-1*, *M-2* y *M-3* que hacen referencia a *Líneas de código*, *Nº clases* y *Nº sentencias* respectivamente. En este caso, tomaremos *M-3*, *Nº sentencias*, como referente del tamaño pudiendo observar para esta métrica que ha habido un incremento de un 46% aproximadamente en el tamaño del *plugin*, esto corrobora las funcionalidades que han sido incluidas tanto a nivel de *interfaz gráfica* (incremento del tamaño en un 48%) como a nivel de la *parte operativa* (incremento del tamaño en un 39%). No se observa un crecimiento tan acentuado en el subsistema correspondiente al proyecto completo ya que este incluye la parte dedicada al repositorio, recordar que en el cálculo de métricas para los demás subsistemas ha sido excluido, por lo que al no haber incorporado nuevos elementos para la construcción de refactorizaciones (acciones y

predicados) ya que este no era el objetivo de nuestro proyecto, hace que el porcentaje de crecimiento de este sea un poco inferior respecto del correspondiente al plugin, entorno al 44%.

A pesar de lo que podría haberse supuesto, la arquitectura diseñada para cada uno de los paquetes del plugin ha conseguido que las responsabilidades, la funcionalidad y la complejidad se repartan de tal modo que las cuatro métricas medidas en valores promedios se sitúen exacta o prácticamente en los intervalos deseables. Son especialmente significativas las métricas obtenidas para la complejidad, ya que como se puede observar en el histórico de proyectos de la asignatura de las medidas tomadas esta es la que más a menudo presenta valores fuera del intervalo esperado.

En la siguiente ilustración se puede ver de forma gráfica y tabular la situación de las métricas correspondientes al *projeto completo* del plugin. En esta gráfica se hace una comparativa entre las medidas obtenidas en las distintas versiones del proyecto, donde se puede observar con el numero medio de métodos por clase ahora sí entran en los límites.

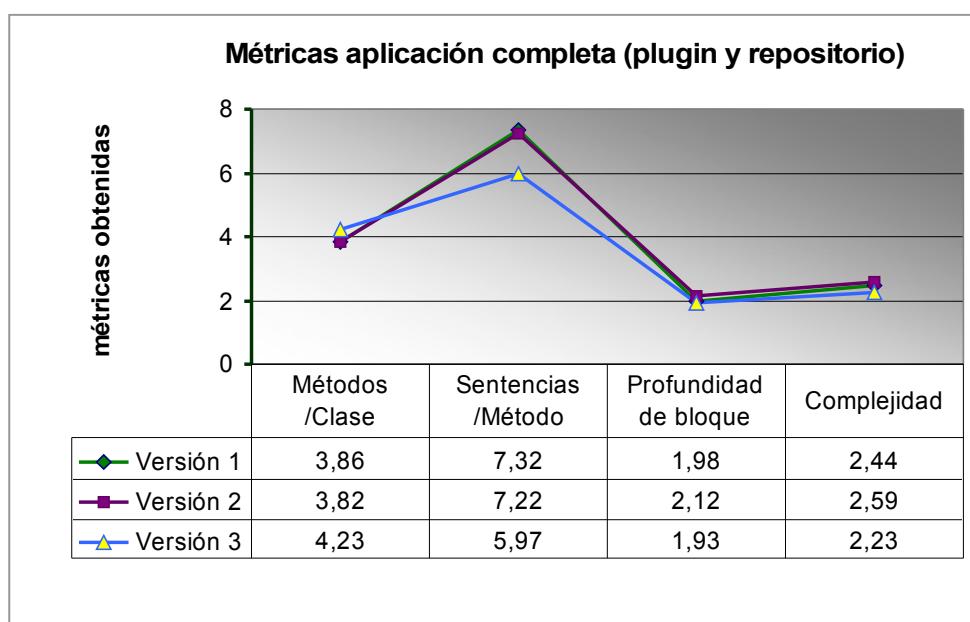


Ilustración 111: Gráfica comparativa de métricas de la aplicación completa

A continuación se muestra el grafo de Kiviat correspondiente al proyecto completo. Como se puede apreciar, las métricas correspondientes a valores medios se sitúan en el área esperada. Tan solo los valores máximos de profundidad y complejidad quedan fuera de los límites, pero estos valores se considerarían poco representativos del conjunto de la aplicación. Igualmente, la herramienta considera que el número de comentarios es excesivo, aunque esta valoración es subjetiva, máxime teniendo en cuenta que se ha tratado de

completar perfectamente la documentación JavaDoc del código. En las gráficas de Kiviat los rangos correctos para cada una de las métricas se encuentran coloreados en verde.

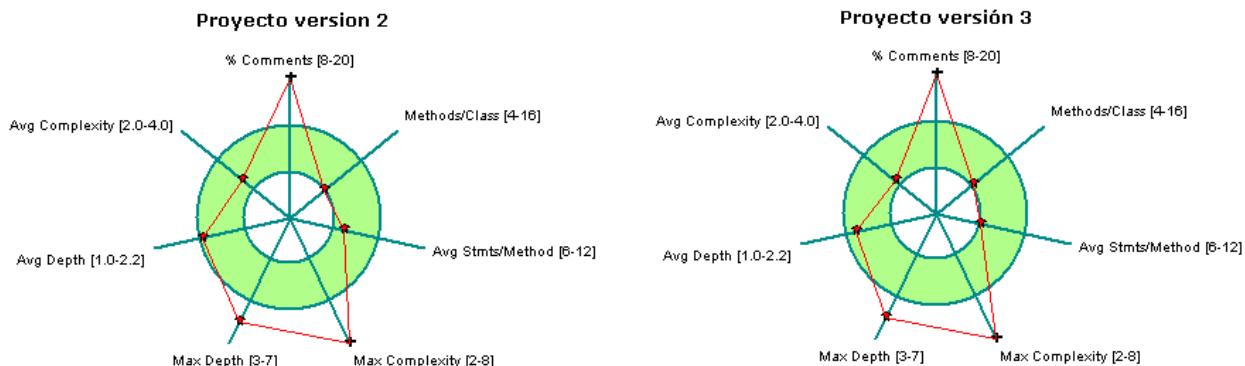


Ilustración 112: Grafo de Kiviat para el proyecto completo

Por último se presenta el resultado obtenido para el proyecto completo en la aplicación web disponible para la autoevaluación, prestaremos especial atención a los intervalos UBU los cuales han sido recogidos mediante el histórico de proyectos UBU. En este caso, todos los valores de las medidas se encuentran en el intervalo establecido.

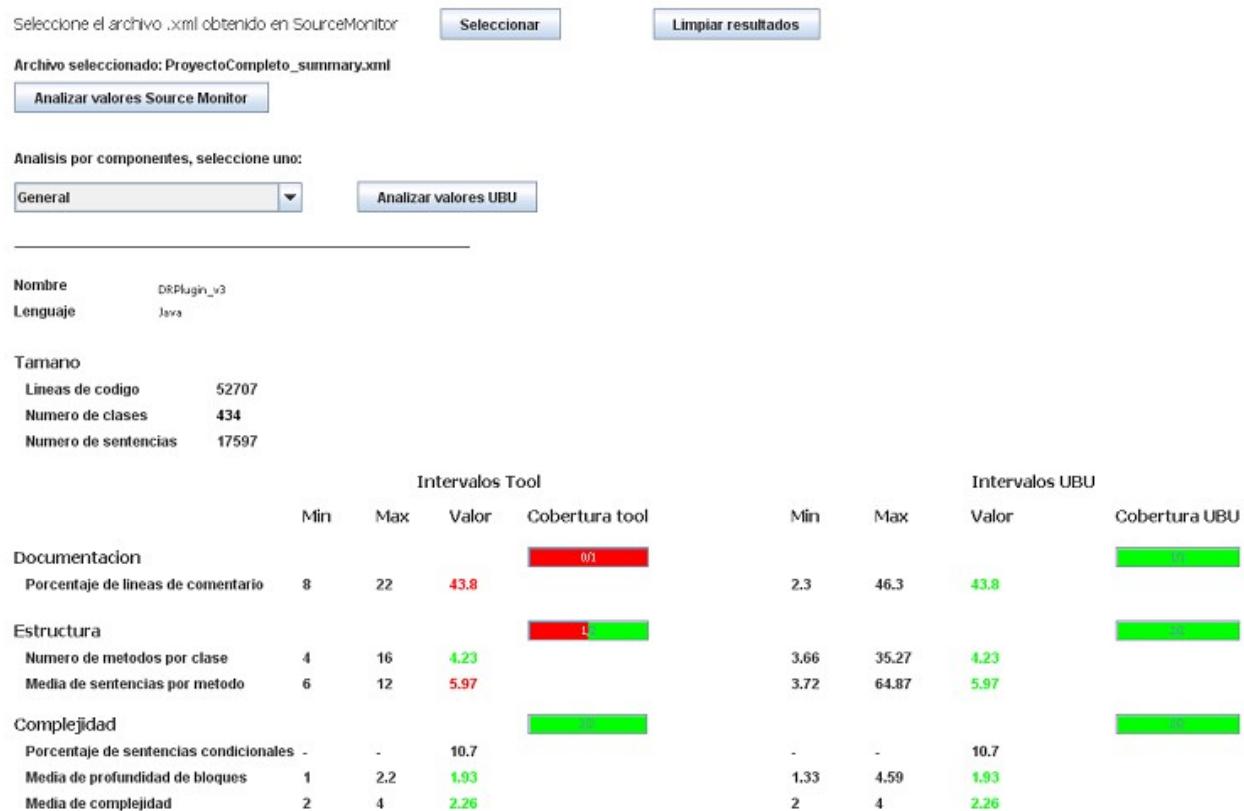


Ilustración 113: Autoevaluación métricas para proyecto completo

En la siguiente ilustración se puede ver de forma gráfica la situación de las métricas correspondientes a los paquetes del *plugin*, una vez eliminado el repositorio. Como se puede observar, se aprecia un ligero aumento en todos los valores, con respecto al proyecto completo, debido a que el repositorio cuenta con muchas clases de dos métodos, muchos de ellos cuentan sólo con unas pocas líneas. Esto hace que la complejidad de los métodos y su número haya aumentado en el subsistema del plugin respecto al del proyecto completo.

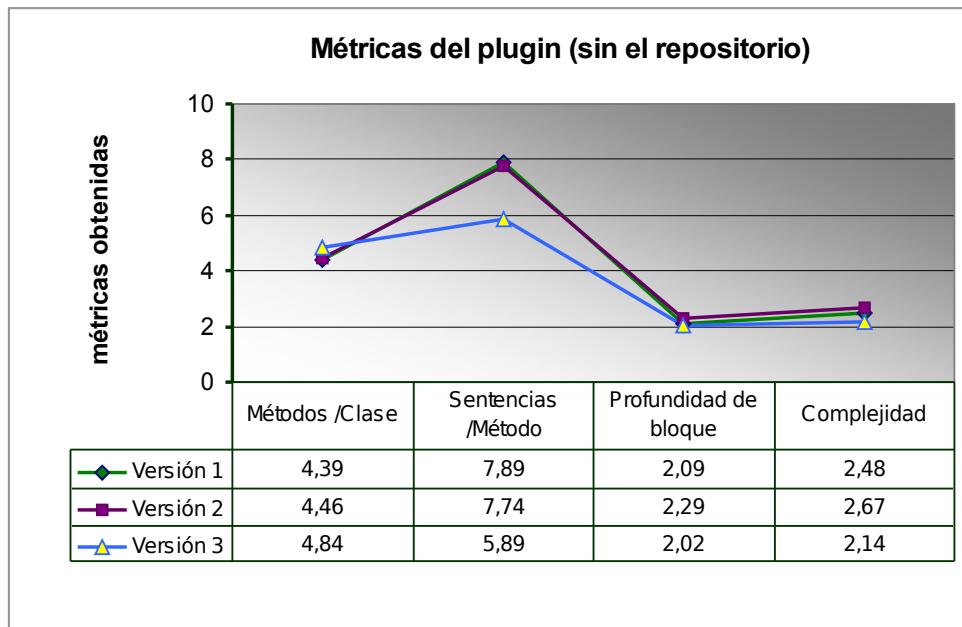


Ilustración 114: Gráfica comparativa de métricas del plugin

A continuación se muestran los grafos de Kiviat correspondiente a los paquetes exclusivamente del *plugin*, es decir, sin el repositorio. El resultado es muy similar al del proyecto completo, con las únicas variaciones comentadas debidas a la influencia del repositorio y su simplicidad en algunos puntos. También se puede apreciar una ligera mejora en la profundidad media de bloques con respecto a la versión anterior del proyecto.

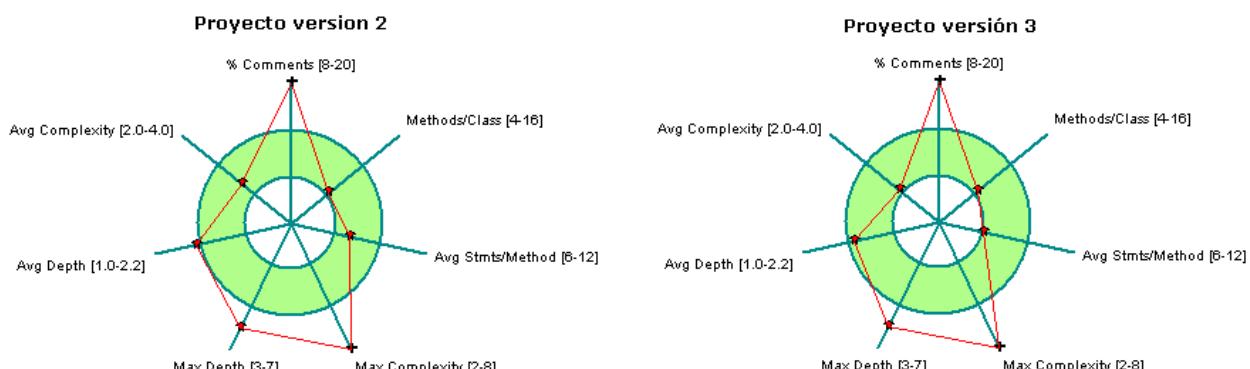


Ilustración 115: Grafo de Kiviat para el plugin

Por último se presenta el resultado obtenido para el *plugin* en la autoevaluación. Fijándonos en los intervalos UBU comprobamos como todos los valores de las medidas se encuentran en el intervalo establecido.



Ilustración 116: Autoevaluación métricas para el plugin

En la siguiente ilustración se muestra una gráfica con las métricas correspondientes a los paquetes de la *parte operativa* del plugin, es decir descartada la interfaz gráfica del mismo. La única variación observable con respecto a las otras versiones del proyecto es una disminución notable del número de sentencias por método, como ya hemos comentado anteriormente, esta medida se ha visto influenciada y justificada por la necesidad de aplicar el patrón de diseño *Builder* en distintas partes del subsistema así como por la aparición de las clases correspondientes a la definición de condiciones de filtro de refactorizaciones y por las que representan la parte operativa de las acciones a realizar en las diferentes vistas. Como se ve el resto de valores en las distintas versiones del proyecto son similares.

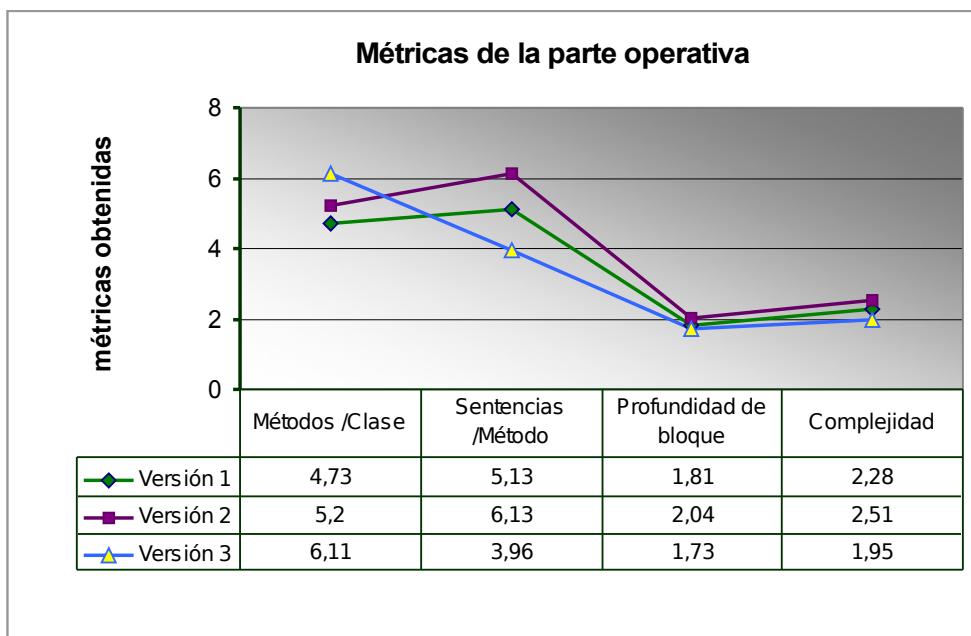


Ilustración 117: Gráfica comparativa de métricas de la parte operativa

En el siguiente grafo de Kiviat se puede observar fácilmente la desviación que acabamos de comentar referente al número medio de sentencias por método, viendo una disminución respecto a la anterior versión del proyecto, pero como ya hemos comentado esta justificada.

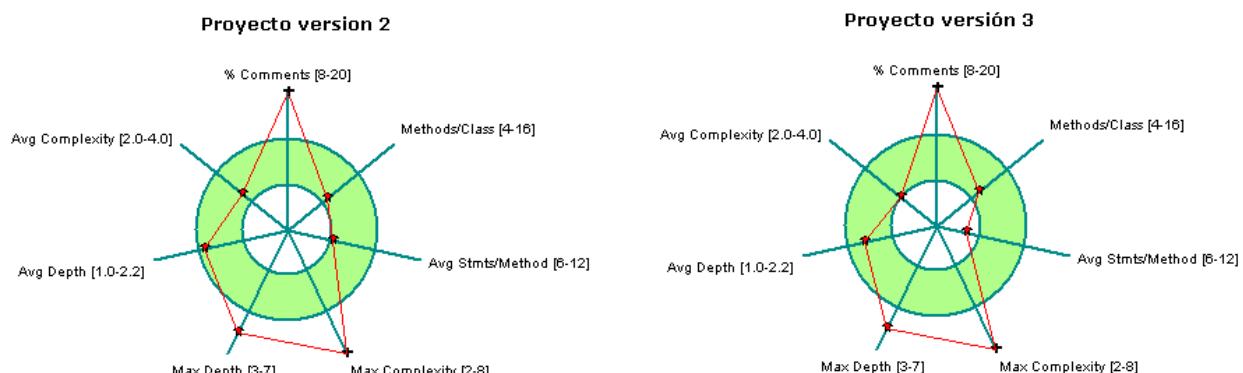


Ilustración 118: Grafo de Kiviat para la parte operativa

Por último vemos el resultado obtenido en la autoevaluación, cabe destacar que, para la evaluación se ha seleccionado el análisis de componentes *General*, como se ha hecho hasta el momento con el resto de subsistemas, ya que la parte operativa contiene tanto la de dominio como la de lógica de negocio. En este caso, si nos fijamos en los intervalos UBU aparecen dos medidas anómalas, la correspondiente a la media de complejidad que es despreciable por encontrarse casi en el límite inferior del intervalo recomendado y la correspondiente al porcentaje de líneas de comentarios, la cual se

encuentra justificada ya que como hemos comentado con anterioridad se ha tratado de completar perfectamente la documentación JavaDoc del código y además ciertas clases llevan incorporada la licencia *GNU GPL*.

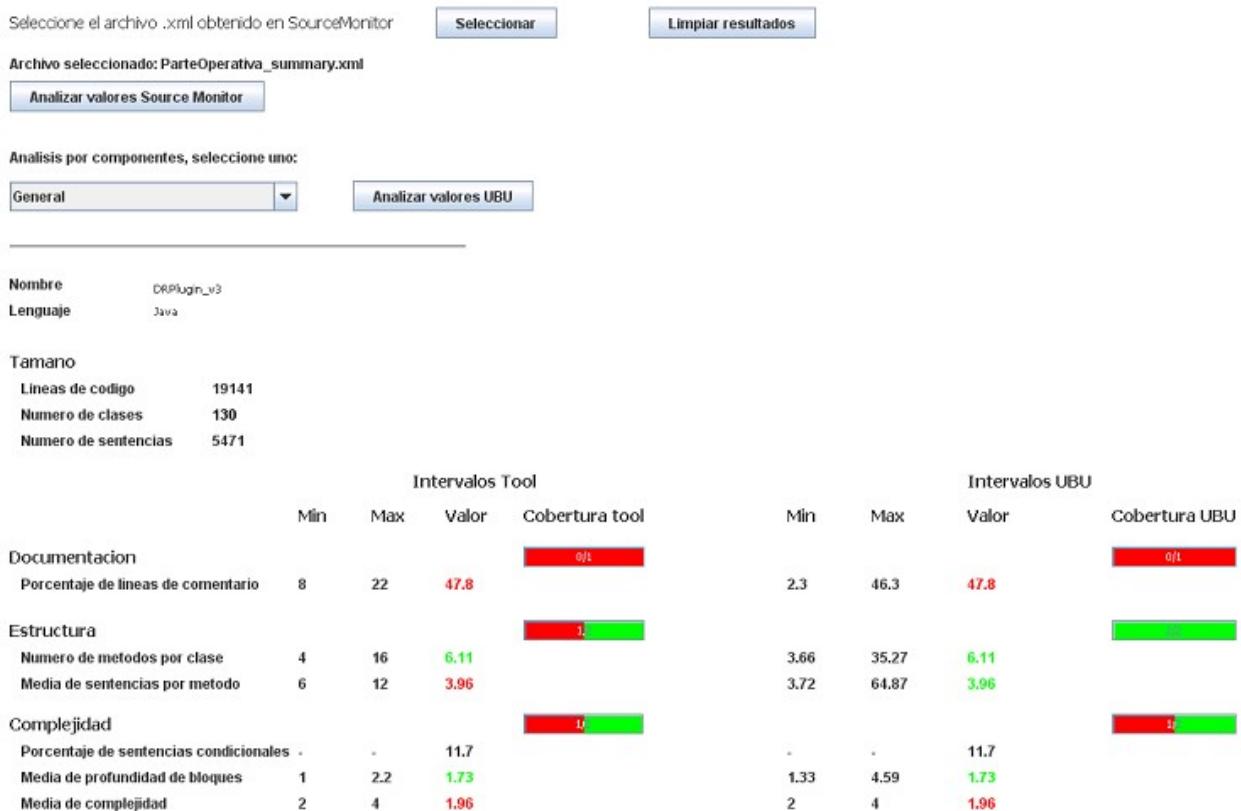


Ilustración 119: Autoevaluación métricas para la parte operativa

En la siguiente ilustración se muestra una gráfica con las métricas correspondientes a la *interfaz gráfica* del plugin, a continuación de ella se muestra su grafo de Kiviat. Como se ve todos los valores medios se encuentran en el área recomendada, destacar que los valores obtenidos para la medida correspondiente a la profundidad de bloques media ya se encuentran entre los límites recomendados, por lo que han sido corregidos respecto de la versión anterior del proyecto. En la interfaz gráfica la complejidad máxima se aleja un poco más del intervalo aconsejable. Este tipo de desviaciones se deben fundamentalmente a la complejidad que lleva asociada la interfaz gráfica del plugin, sobre todo a la hora de generar ciertas partes de forma dinámica como es en la ejecución de refactorizaciones.

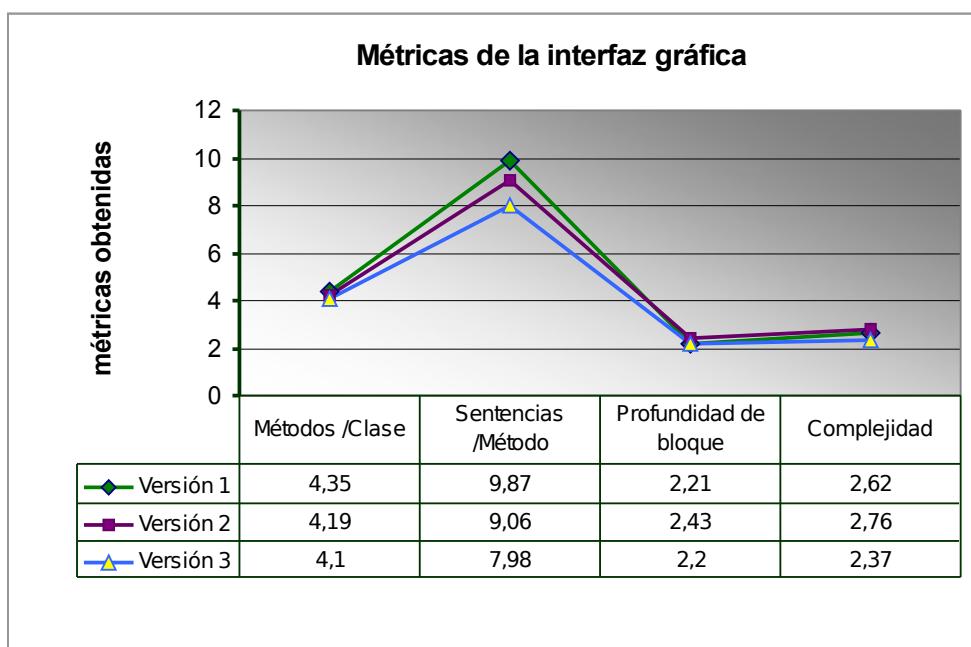


Ilustración 120: Gráfica comparativa de métricas de la interfaz gráfica

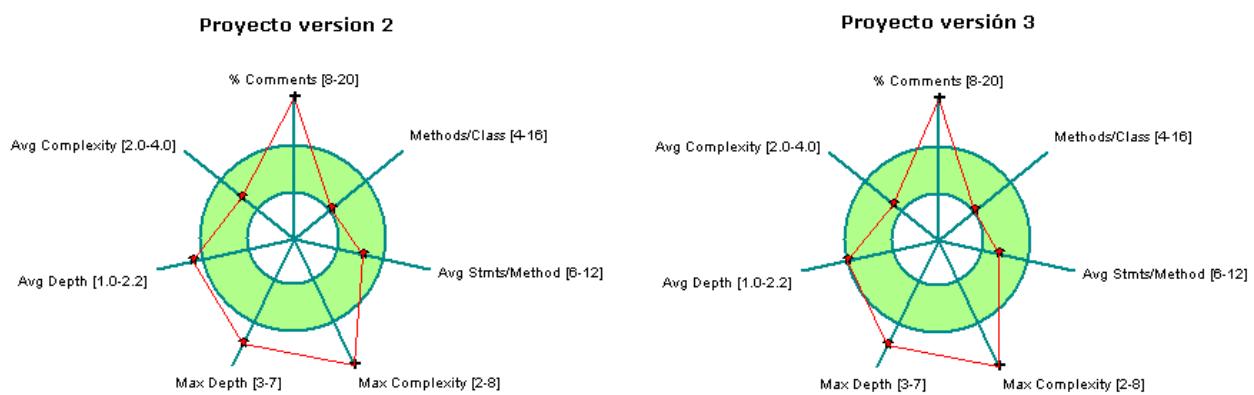


Ilustración 121: Grafo de Kiviat para la interfaz gráfica

Por último se presenta el resultado obtenido para la parte de interfaz gráfica en la aplicación web disponible para la autoevaluación, como ya hemos comentado se prestará especial atención a los intervalos UBU. Cabe destacar que, para la evaluación se ha seleccionado el análisis de componentes de tipo *Interfaz*. En este caso, únicamente aparece como medida anómala el porcentaje de líneas de comentarios, la cual se encuentra justificada ya que como hemos comentado con anterioridad se debe a la documentación JavaDoc del código y la presencia de la licencia *GNU GPL*.

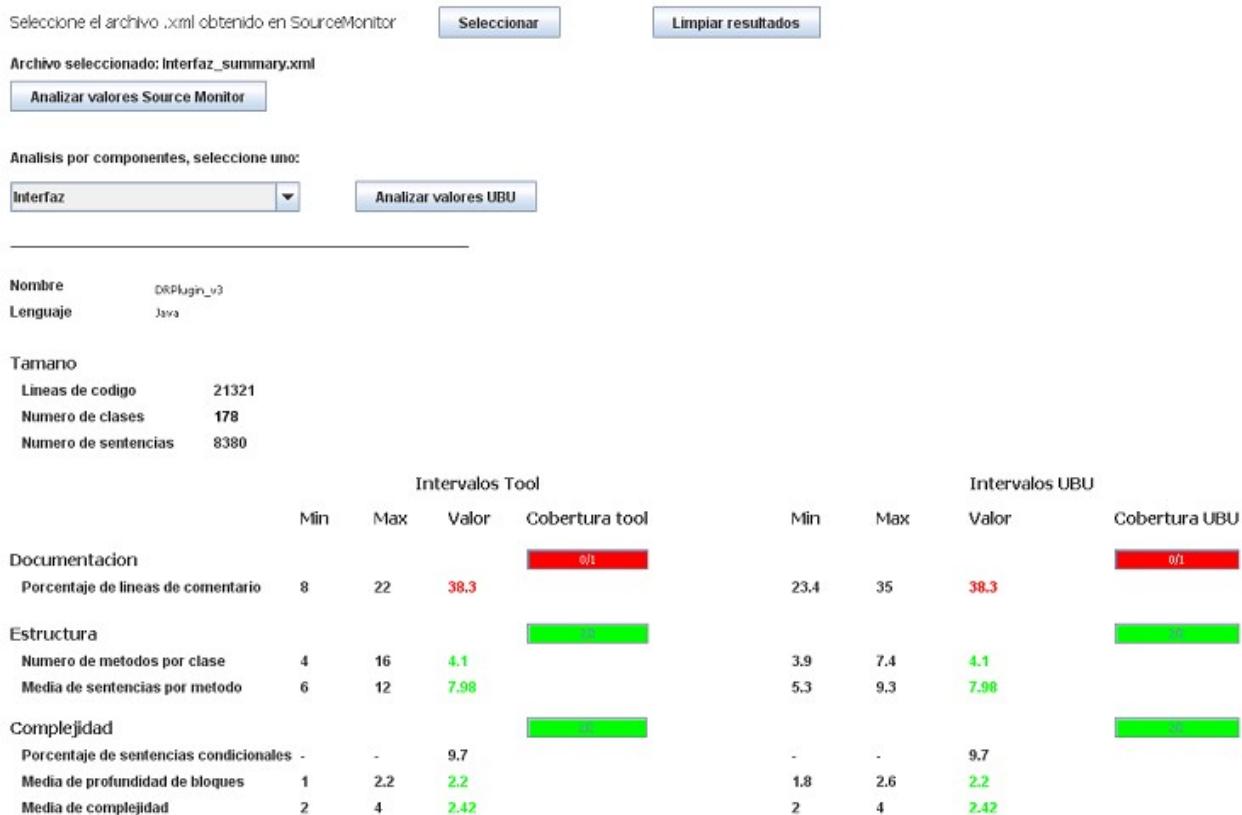


Ilustración 122: Autoevaluación métricas para interfaz gráfica

Por tanto, una vez vistos los resultados, se puede concluir que las métricas indican que detrás hay un diseño y una implementación de calidad, ya que se ha conseguido situar los valores promedios del proyecto, o la gran mayoría de los mismos, en los intervalos recomendados y corregir otros heredados de versiones anteriores del proyecto. Así pues, desde el punto de vista de estas métricas, el proyecto consigue un nivel de calidad relativamente bueno según los baremos utilizados en el análisis de la evolución de los proyectos de la asignatura.

Toda la documentación técnica de métricas que acabamos de ver se puede encontrar en el directorio Documentación\Documentación Técnica\métricas del CD que se entrega con el proyecto.

7. INTERNACIONALIZACIÓN

La internacionalización de un producto es importante cuando se quiere ampliar el número de posibles usuarios del plugin. En esta tercera versión del plugin se ha mantenido la internacionalización del plugin cuya infraestructura ya se había definido en la anterior versión. Por lo tanto la única labor necesaria para mantener la internacionalización del plugin ha sido la de mantener las cadenas de la interfaz independientes del lenguaje a través de las propias facilidades que Eclipse proporciona.

Para externalizar las cadenas de una clase se puede utilizar la función *Source > Externalize Strings* accesible pulsando con el botón derecho sobre el editor de código fuente del fichero. Esto muestra el siguiente asistente de externalización:

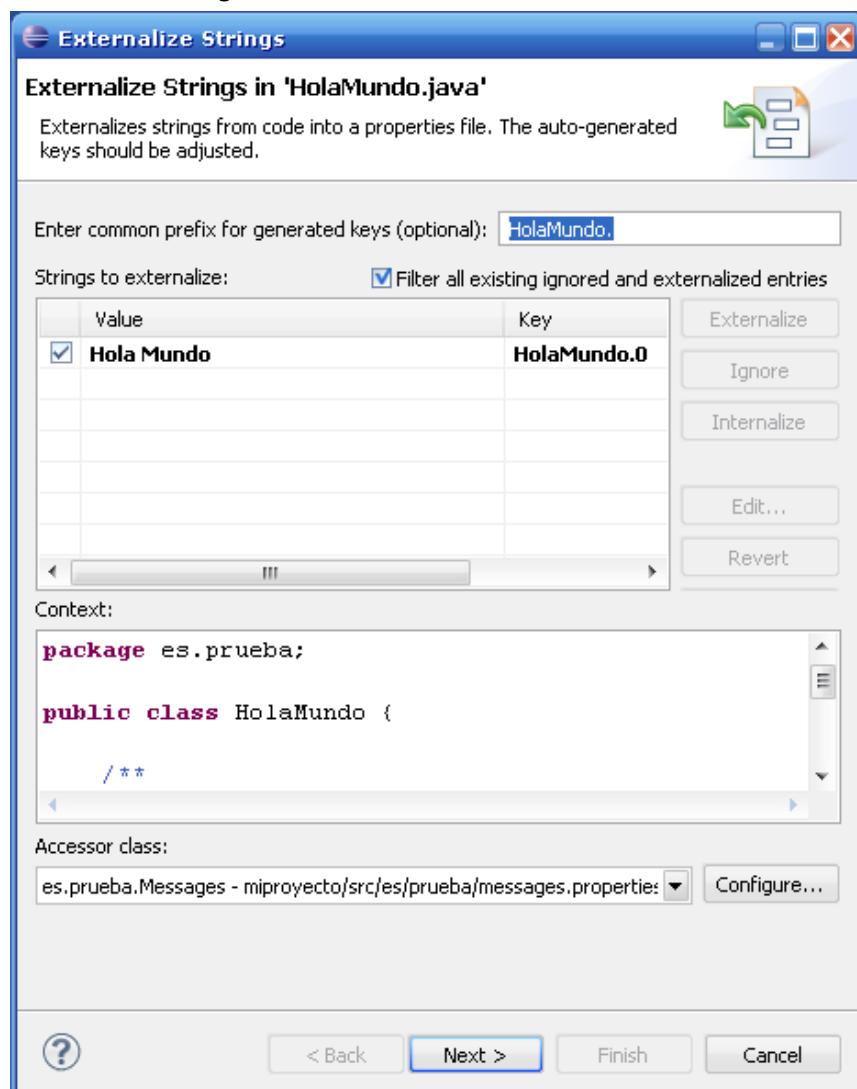


Ilustración 123: Asistente de externalización de cadenas

Al seleccionar una entrada en la tabla del asistente se puede asignar una acción a una cadena uno de los tres siguientes casos:

- Traducir: si se selecciona esta opción, se añade una entrada al fichero de propiedades; La clave generada y el código serán reemplazados por la cadena original internacionalizada cuando llegue el momento.
- No traducir nunca: la cadena se marca como no traducible y no volverá a ser tenida en cuenta en siguientes intentos de traducción de la clase debido a que se identificará con la etiqueta:

"// \$NON-NLS-1\$"

- Saltar: no modifica nada. Por lo tanto posteriores ejecuciones del asistente seguirán marcando la cadena actual como potencialmente traducible.

Para más información sobre cómo crear la infraestructura completa para realizar la internacionalización del plugin este proceso es pormenorizadamente explicado en la documentación de la versión anterior de este proyecto.

8. AYUDA EN ECLIPSE

La plataforma de Eclipse, proporciona su propio sistema de ayuda basada en una tabla de contenidos en XML que referencia a ficheros HTML. Esta plataforma de ayuda puede ser utilizada para documentar los propios plugins dentro de Eclipse pero también de forma independiente.

La documentación se divide en libros, de los que se puede tener tantos como se quiera por instancia del sistema de ayuda. Cada libro es escrito como un plugin de Eclipse, pero el trabajo que se requiere es mínimo. Para escribir un plugin de ayuda sencillo, se necesita un fichero plugin.xml que describa el plugin y que puede ser así de simple como el siguiente:

```
<plugin name="Sample Documentation Plug-in" id="com.ibm.sample.doc"
       version="1.0.0" provider-name="IBM">
  <extension point="org.eclipse.help.toc">
    <toc file="toc.xml" primary="true" />
  </extension>
</plugin>
```

Sólo es necesario cambiar el nombre del plugin y su id, versión y nombre de

proveedor con los valores asociados al proyecto en el que se está trabajando. El punto de extensión org.eclipse.help.toc identifica este plugin como un plugin del sistema de ayuda. El fichero toc.xml se comporta como la tabla de contenidos del plugin. Este fichero simplemente contendrá los datos para la información jerárquica que se muestra en el panel izquierdo de la ventana de ayuda de Eclipse. Un fichero toc simple tiene un contenido similar al siguiente:

```
<toc label="Sample Documentation">
    <topic label="My Section" href="mySection.html">
        <topic label="Foo" href="foo.html"/>
        <topic label="Bar" href="bar.html"/>
    </topic>
</toc>
```

Cada *topic* o tema se representa en la documentación final por una entrada en la lista de navegación. Estos temas pueden anidarse y cada uno apuntar a un fichero HTML o JSP. Una vez definido el fichero de tabla de contenidos y los ficheros HTML con el contenido de la ayuda sólo es necesario empaquetar el plugin para poder utilizarlo en el sistema de ayuda de Eclipse.

Universidad de Burgos
ESCUELA POLITÉCNICA SUPERIOR
INGENIERÍA INFORMÁTICA



Anexo 5. Manual de Instalación

Clasificación de Refactorizaciones
Dynamic Refactoring Plugin 3.0

Alumnos:
Míryam Gómez San Martín
Íñigo Mediavilla Saiz

Tutor:
Raúl Marticorena Sánchez

Burgos, Julio de 2011

**ANEXO V
MANUAL DE INSTALACIÓN**

ÍNDICE DE CONTENIDO

Anexo V	
<i>Manual de Instalación.....</i>	407
<i>Lista de cambios.....</i>	413
1. INTRODUCCIÓN.....	415
2. REQUISITOS SOFTWARE.....	415
3. REQUISITOS HARDWARE.....	416
4. INSTALACIÓN DE JAVA DEVELOPMENT KIT.....	417
4.1. Proceso de instalación.....	418
4.2. Configuración de variables de entorno.....	423
5. INSTALACIÓN DE ECLIPSE.....	426
6. INSTALACIÓN DE ANT.....	428
6.1. Proceso de instalación.....	429
6.2. Configuración de variables de entorno.....	430
7. INSTALACIÓN DE LA APLICACIÓN.....	431
7.1. Instalación de Dynamic Refactoring Plugin.....	431
7.1.1. Instalación manual.....	431
7.1.2. Instalación automática por red.....	432
7.1.3. Actualización automática por red.....	440
7.1.4. Desinstalación automática.....	441
7.2. Instalación de la tarea Ant RefactoringPlan.....	445
8. EJECUCIÓN DE LA APLICACIÓN.....	445
8.1. Ejecución de Dynamic Refactoring Plugin.....	445
8.1.1. Cómo mejorar el rendimiento de Eclipse.....	446
8.2. Ejecución de la tarea Ant RefactoringPlan.....	448
8.2.1. Ejecución desde consola.....	448
8.2.2. Ejecución desde Eclipse.....	450

ÍNDICE DE ILUSTRACIONES

Ilustración 124: Instalación JDK 1.6 (I).....	418
Ilustración 125: Instalación JDK 1.6 (II).....	419
Ilustración 126: Instalación JDK 1.6 (III).....	420
Ilustración 127: Instalación JDK 1.6 (IV).....	420
Ilustración 128: Instalación JDK 1.6 (V).....	421
Ilustración 129: Instalación JDK 1.6 (VI).....	422
Ilustración 130: Registro JDK 1.6.....	422
Ilustración 131: Mi PC - Propiedades.....	423
Ilustración 132: Propiedades del sistema.....	424
Ilustración 133: Modificación variables de entorno.....	425
Ilustración 134: Modificación variable del sistema.....	425
Ilustración 135: Comprobación instalación.....	426
Ilustración 136: Estructura de ficheros de la instalación de Eclipse.....	427
Ilustración 137: Selección espacio de trabajo Eclipse.....	427
Ilustración 138: Bienvenida Eclipse.....	428
Ilustración 139: Directorios Apache Ant.....	429
Ilustración 140: Menú Help en Eclipse- Install New Software.....	432
Ilustración 141: Agregar repositorio del plugin.....	433
Ilustración 142: Introducir datos repositorio.....	434
Ilustración 143: Selección plugin para su instalación.....	435
Ilustración 144: Ventana descriptiva del plugin.....	436
Ilustración 145: Ventana licencia plugin.....	437
Ilustración 146: Instalación plugin.....	437
Ilustración 147: Certificado plugin.....	438
Ilustración 148: Detalles del certificado.....	439
Ilustración 149: Ventana reinicio tras instalación del plugin.....	439
Ilustración 150: Menú Help en Eclipse- Check for Updates.....	440
Ilustración 151: Comprobar actualizaciones disponibles de plugins.....	440
Ilustración 152: Available Updates.....	441
Ilustración 153: Menú Help en Eclipse- About Eclipse SDK.....	442
Ilustración 154: About Eclipse SDK.....	442
Ilustración 155: Eclipse SDK Installation Details.....	443
Ilustración 156: Uninstall Details.....	444
Ilustración 157: Uninstalling Software.....	444
Ilustración 158: Eclipse con Dynamic Refactoring Plugin instalado.....	446
Ilustración 159: Contenido del directorio refactoringPlan.....	448
Ilustración 160: Ejecución tarea Ant RefactoringPlan desde consola.....	450
Ilustración 161: Proyecto tarea Ant RefactoringPlan.....	451
Ilustración 162: Buildfile Selection.....	451
Ilustración 163: Despliegue del fichero build.xml en la vista Ant.....	452
Ilustración 164: Ejecución tarea Ant RefactoringPlan desde Eclipse.....	452

LISTA DE CAMBIOS

Número	Fecha	Descripción	Autor/es
0	28/02/11	Añadido manual de instalación. Cambios de formato y de numeración de capítulos.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
1	04/04/11	Agregados Requisitos mínimos de instalación del plugin.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
2	14/04/11	Corregido link de instalación incorrecto y sustituida pantalla de certificado.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
3	29/04/11	Incluido requisitos SW y HW.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
4	30/04/11	Incluida instalación de herramientas	Míryam Gómez San Martín Íñigo Mediavilla Saiz
5	01/05/11	Añadido instalación y ejecución de la aplicación	Míryam Gómez San Martín Íñigo Mediavilla Saiz
6	13/05/11	Añadida desinstalación del plugin	Míryam Gómez San Martín Íñigo Mediavilla Saiz
7	05/06/11	Realizados últimos cambios.	Míryam Gómez San Martín Íñigo Mediavilla Saiz

1. INTRODUCCIÓN

Este anexo pretende servir de guía para cualquier usuario de la aplicación en el proceso de instalación de cada uno de los componentes necesarios para la ejecución de la aplicación. En él, se detallará la instalación de la máquina virtual de Java, el entorno de desarrollo Eclipse, y la propia aplicación, así como cualquier configuración adicional que fuese necesaria para aprovechar toda la funcionalidad que la aplicación ofrece.

Además se incluirá una recomendación sobre los requisitos mínimos, tanto a nivel de hardware como de software, necesarios que deberán de cumplir los equipos en los que se realice la instalación de la aplicación para el correcto funcionamiento de la misma.

2. REQUISITOS SOFTWARE

Los requerimientos software, es decir, las características que debe tener el software instalado en un equipo para poder soportar y ejecutar la aplicación son los que se describen a continuación.

En primer lugar hablaremos sobre el sistema operativo necesario. Debido a que la aplicación ha sido desarrollada en lenguaje Java, lenguaje de Sun Microsystems y adquirido posteriormente por Oracle Corporation, y que este se trata de un lenguaje multiplataforma, es decir, permite que el mismo ejecutable se pueda portar y ejecutar en cualquier sistema compatible con el software Java, al menos, se podría utilizar en sistemas operativos Windows, GNU/Linux, Solaris y Mac OS.

Para que esto sea posible, los ejecutables de Java no los puede procesar el sistema operativo directamente, sino que hace falta un programa especial, la *Máquina Virtual de Java* (en inglés *Java Virtual Machine*, *JVM*), que se encargue a su vez de ejecutar esos programas. Por lo tanto, se requiere previamente tener instalada la herramienta *JRE* (*Java Runtime Environment*). No obstante como el objetivo final es el desarrollo de aplicaciones, no olvidemos que será para esto para lo que se utilizará el plugin, requerirá la instalación de la herramienta de desarrollo *JDK* (*Java Development Kit*), la cual incluye la de ejecución, en su versión 1.6.0 o superior. Podremos acceder a ella desde el siguiente enlace:

<http://www.oracle.com/technetwork/java/index.html> [Oracle Technology Network for Java Developers n.d.].

Por otra parte es necesario tener instalado el entorno de desarrollo o *IDE* (*Integrated Development Environment*) Eclipse, para el cual ha sido desarrollado el plugin y que es imprescindible para la utilización del mismo. Se recomienda la utilización de una versión de Eclipse igual o superior a la versión 3.5, que corresponde con Eclipse Galileo, ya que no se garantiza la compatibilidad del plugin con versiones anteriores del producto por los posibles cambios que hayan podido sufrir los propios componentes de Eclipse en otras distribuciones, y de los cuales depende el plugin. Por esta razón, lo más recomendable será descargarse la última versión del paquete Eclipse IDE for Java Developers desde las respectivas páginas de descargas de las versiones 3.5 y 3.6 de la web de Eclipse, que es la siguiente: (<http://www.eclipse.org/downloads/>) [Eclipse Downloads n.d.].

Por último, destacar que se ha comprobado la correcta ejecución del plugin en los diferentes sistemas operativos mencionados; Windows, GNU/Linux, Solaris y Mac OS.

Todas las herramientas han sido incluidas en el propio CD en el que se distribuye la aplicación, en concreto en la carpeta Software destinada a tal efecto, con la finalidad de facilitar la instalación de las mismas. La instalación de cada una de estas herramientas se explicará detalladamente a continuación. Cabe reseñar que únicamente se va a explicar la instalación de las herramientas bajo el sistema operativo Windows, por lo que los usuarios de otras plataformas podrán encontrar para cada una de las herramientas una referencia a la guía de instalación que les ayudará en su instalación en el sistema operativo en el que trabajen.

3. REQUISITOS HARDWARE

La aplicación en si misma no necesita de requisitos hardware que no pueda cumplir cualquier equipo actual de nivel medio en cuanto a prestaciones y características se refiere. Aunque si bien es verdad, para un funcionamiento óptimo del plugin es recomendable disponer de buena capacidad de procesamiento y memoria RAM necesarios en los procesos internos del mismo, así como por parte del interprete de Java y del propio Eclipse.

Se deberá de tener en cuenta que las necesidades variarán en función de los proyectos con los que se trabaje así como de las refactorizaciones que se apliquen. Es decir, el requerimiento de memoria RAM, capacidad en disco y nivel de procesamiento será dependiente de la complejidad del proyecto y del costo computacional de cada una de las refactorizaciones que se ejecuten sobre el mismo.

Por lo tanto, se recomienda que los equipos en los que se realice la instalación de la aplicación cumplan con los siguientes requerimientos mínimos lo que garantizará un buen rendimiento de la misma:

- ✓ Procesador: Microprocesador de 1 Ghz, ya sea de 32 o 64 bits
Por ejemplo: Intel Pentium IV o Intel Core 2 Duo.
- ✓ Memoria: 1 GB de memoria RAM para versiones de 32 bits,
2 GB de memoria RAM para versiones de 64 bits.
- ✓ Disco: Espacio libre en disco:
 - 300 MB para *JDK* de Java.
 - 190 MB para Eclipse.
 - 45 MB para Ant.
 - 15 MB para Dynamic Refactoring Pugin.
 - Más el espacio estimado para almacenar el proyecto.
- ✓ Pantalla: Monitor con resolución de 1024 x 768, con 256 colores.
- ✓ Unidades: Dependiendo de como se desee realizar la instalación:
 - Unidad lectora de CD – ROM.
 - Tarjeta de red + conexión a Internet o conexión a red local si se instala desde un recurso compartido de red.
- ✓ Periféricos: Teclado.
Ratón o dispositivo de puntero compatible.

4. INSTALACIÓN DE JAVA DEVELOPMENT KIT

Como se ha comentado en el apartado dedicado a *Requisitos Software* se requiere de la instalación de la herramienta *JRE* (*Java Runtime Environment*) pero dado que el objetivo final es la utilización del plugin en el desarrollo de aplicaciones se va a realizar la instalación de la herramienta de desarrollo *JDK* (*Java Development Kit*), la cual incluye la de ejecución.

En este apartado se mostrará el proceso a seguir para la correcta instalación del *JDK*, así como para la configuración de las variables de entorno necesarias para los sistemas operativos Windows. Si el sistema operativo utilizado es otro se puede consultar la guía de

instalación específica para él en: <http://www.oracle.com/technetwork/java/javase/index-137561.html> [Java SE 6 Platform Installation n.d.].

Recordar que, como requisito para su instalación, debemos de tener en cuenta que se ha de disponer, como mínimo, de 300 MB de espacio libre en disco.

4.1. Proceso de instalación

En primer lugar, debemos de disponer del software que lleve a cabo la instalación del *JDK*. Para ello, se podrá obtener la distribución adecuada conforme al sistema operativo que se este utilizando en la siguiente dirección web:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>. [Java SE Downloads n.d.]

En caso de tratarse de Windows se podrá obtener el instalador a partir del CD que se suministra junto con el proyecto, se encuentra en la carpeta Software\java con el nombre jdk-6u23-windows-i586.exe, el cual será el encargado de instalar *JDK, Java Development Kit*, en el equipo.

Una vez que se dispone del mismo, para su instalación basta con hacer doble clic en el archivo y acto seguido se iniciará el asistente, el cual nos da la bienvenida. Deberemos pulsar el botón *Next >* para comenzar el proceso.



Ilustración 124: Instalación JDK 1.6 (I)

A continuación se muestra una ventana en la podremos elegir entre una instalación típica o una personalizada, en este caso realizaremos una instalación típica por lo que no añadiremos ni eliminaremos ningún componente de utilidad.

Además también nos indica la ruta en la que realizará la instalación de la herramienta de desarrollo, en este caso la modificaremos por una ruta más sencilla y manejable. Para ello pulsamos el botón *Change...* e indicamos la nueva ruta `C:\jdk1.6.0_23`.

Por último, pulsamos *Next >* para continuar con el proceso.

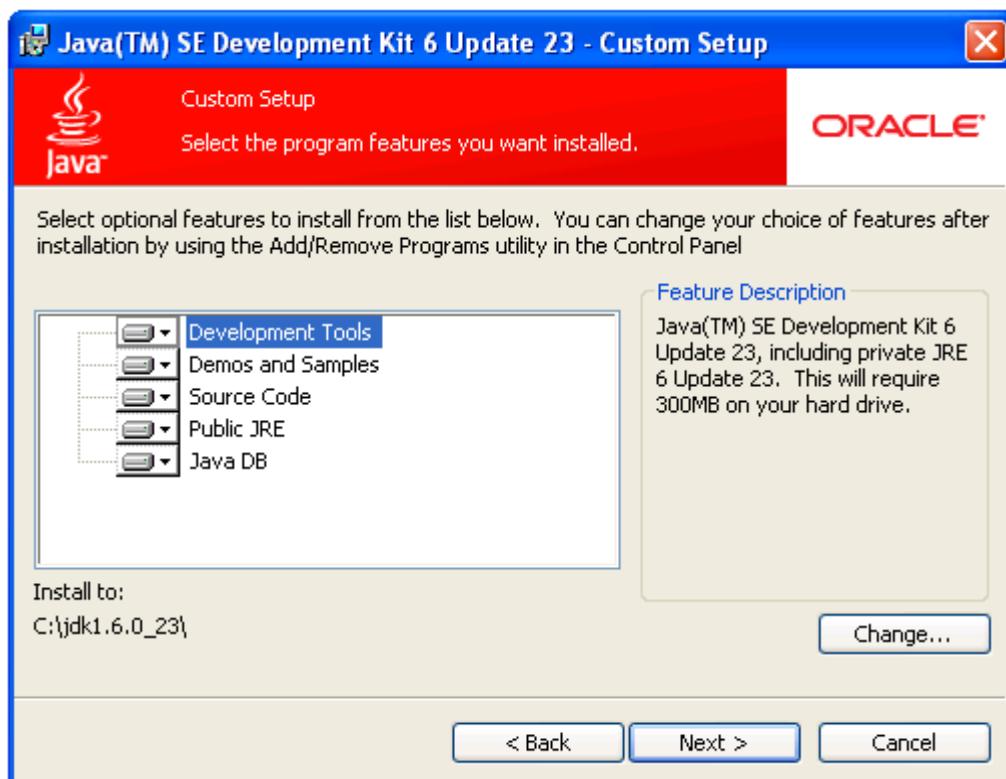


Ilustración 125: Instalación JDK 1.6 (II)

Es entonces cuando comienza la instalación de la herramienta, aparecerá una ventana con una barra de estado que marca el progreso del proceso de instalación. Este proceso puede durar varios minutos.

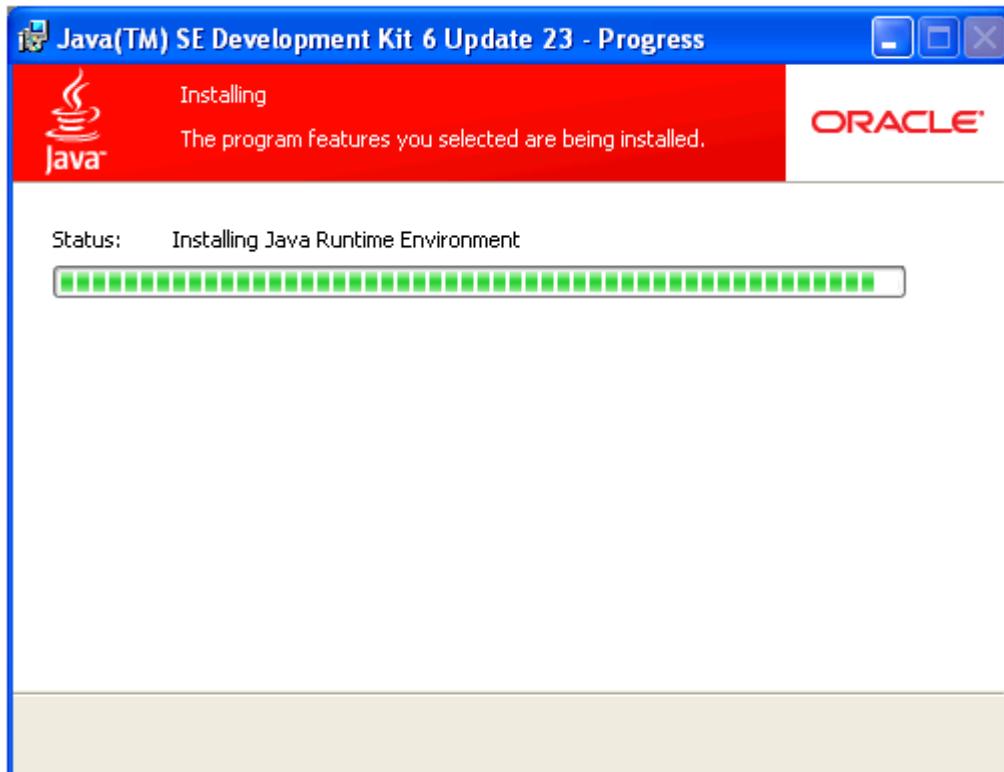


Ilustración 126: Instalación JDK 1.6 (III)

La siguiente ventana indica la ruta en la que realizará la instalación del entorno de ejecución, mantendremos la establecida por defecto y pulsaremos *Next >* para continuar.

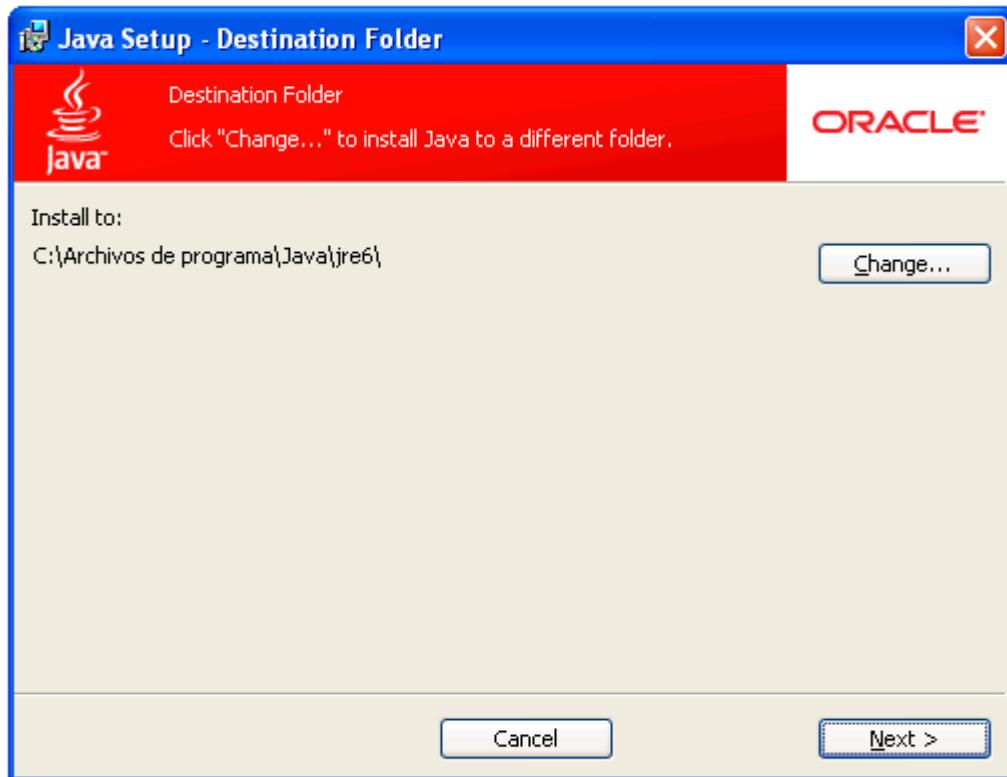


Ilustración 127: Instalación JDK 1.6 (IV)

Inmediatamente comenzará su instalación y de igual forma aparecerá una ventana con una barra de estado que marca el progreso del proceso, el cual podrá durar varios minutos.

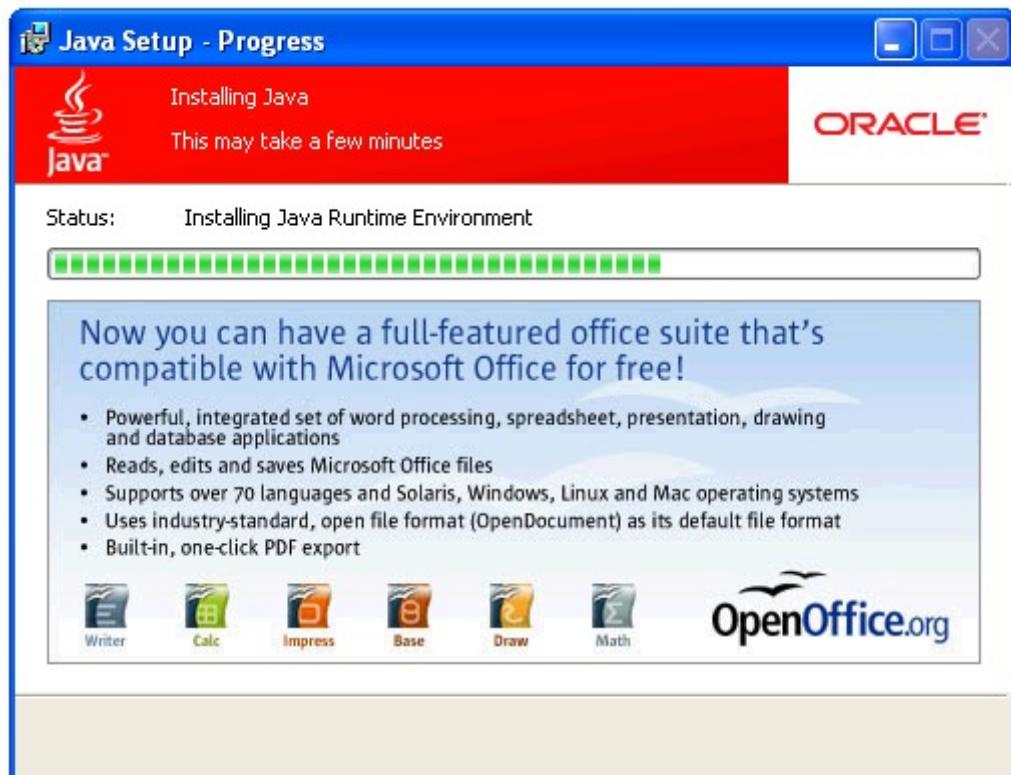


Ilustración 128: Instalación JDK 1.6 (V)

Una vez que haya finalizado la instalación nos aparecerá la siguiente pantalla donde nos informa de ello, pulsaremos el botón *Finish* para salir de este asistente y ya tendremos instalado el *JDK* en el equipo.

En esta misma ventana aparece el botón *Product Registration Information* por si quisiésemos obtener información en detalle sobre el proceso de registro del producto.

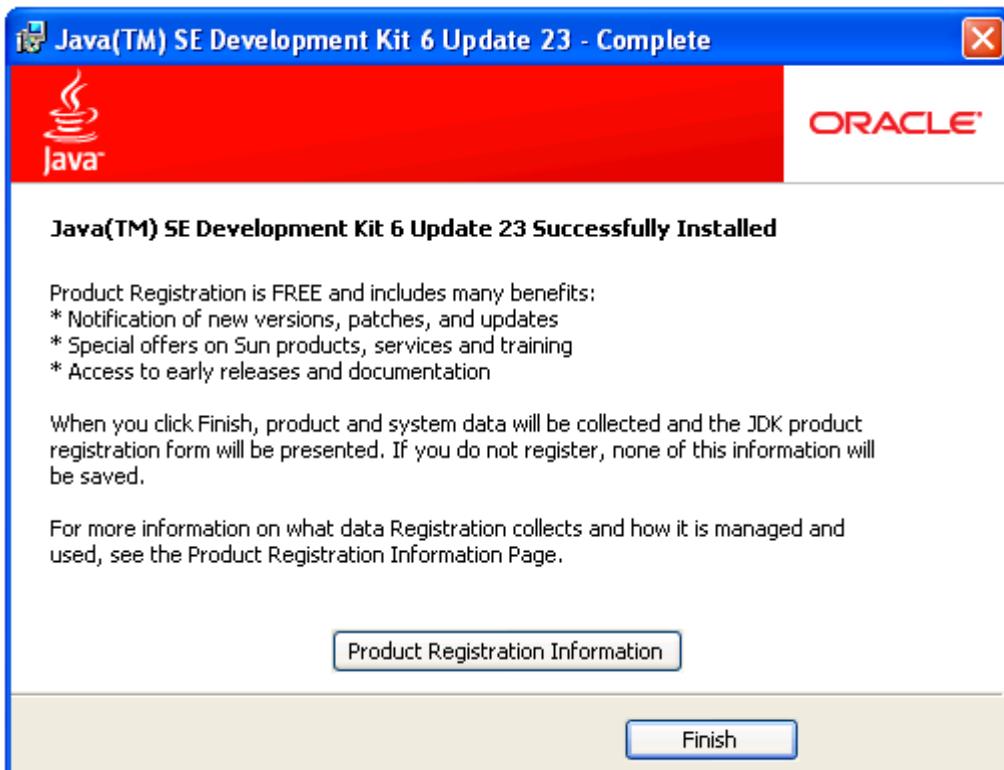


Ilustración 129: Instalación JDK 1.6 (VI)

Por último, se abre la siguiente pagina web para realizar el registro si se desea.

The screenshot shows a registration page for the Java Development Kit (JDK). The title is "Java Development Kit (JDK) Registration". A sub-instruction "Please take the time to register your software." is displayed. The Oracle logo is in the top right. Below the title, a message says: "You need an Oracle.com account to register your product. Create an account now, or if you already have one continue by registering your product below." Two buttons are shown: "Create An Account" on the left and "Use My Account" on the right. The "Create An Account" section contains the text: "Need an account? Create an Oracle.com account now." The "Use My Account" section contains the text: "Please accept the terms of use below and click \"Register Now\" to register your product." It includes a checkbox labeled "I accept the terms of use for registering Oracle programs.", a link "View terms of use", and a "Register Now" button.

Oracle Corporation respects your privacy. For more information on Oracle's Privacy Policy see <http://www.oracle.com/html/privacy.html> or contact privacy_wu@oracle.com.



Copyright 2010, Oracle Corporation and/or its affiliates

Ilustración 130: Registro JDK 1.6

4.2. Configuración de variables de entorno

Una vez tenemos instalado el *JDK*, a continuación se deberán establecer las variables de entorno correspondientes para el correcto funcionamiento del entorno de ejecución, son **PATH** y **CLASSPATH**.

PATH

Esta variable es utilizada para invocar a los ficheros ejecutables de *J2SDK*, como son entre otros *javac.exe* para la compilación, *java.exe* para la ejecución, *javadoc.exe* para la generar la documentación, desde cualquier directorio donde nos encontremos sin necesidad de tener que teclear la ruta completa del comando cada vez que le necesitemos utilizar.

CLASSPATH

Esta variable es utilizada para indicar a las aplicaciones implementadas en Java dónde encontrar las clases necesarias para su ejecución.

Para la configuración de las variables de entorno en Windows se procederá de la siguiente forma:

Sitúese sobre el ícono de *Mi PC* que se encuentra en su escritorio, pulse botón derecho del ratón sobre él. Aparecerá un menú contextual, entre las opciones disponibles escoja *Propiedades*:

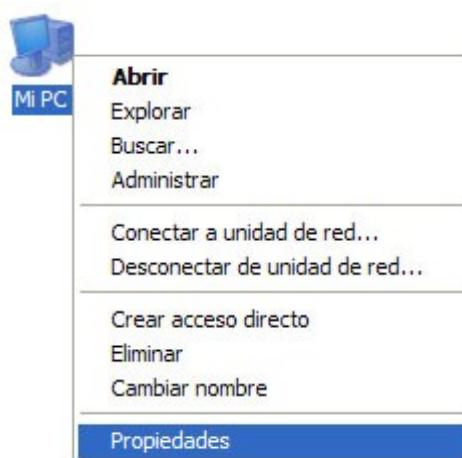


Ilustración 131: Mi PC - Propiedades

Aparecerá la siguiente pantalla, en ella seleccione la pestaña *Opciones avanzadas* y dentro de ella, en la zona inferior izquierda, se encuentra el botón *Variables de entorno* que deberá pulsar:

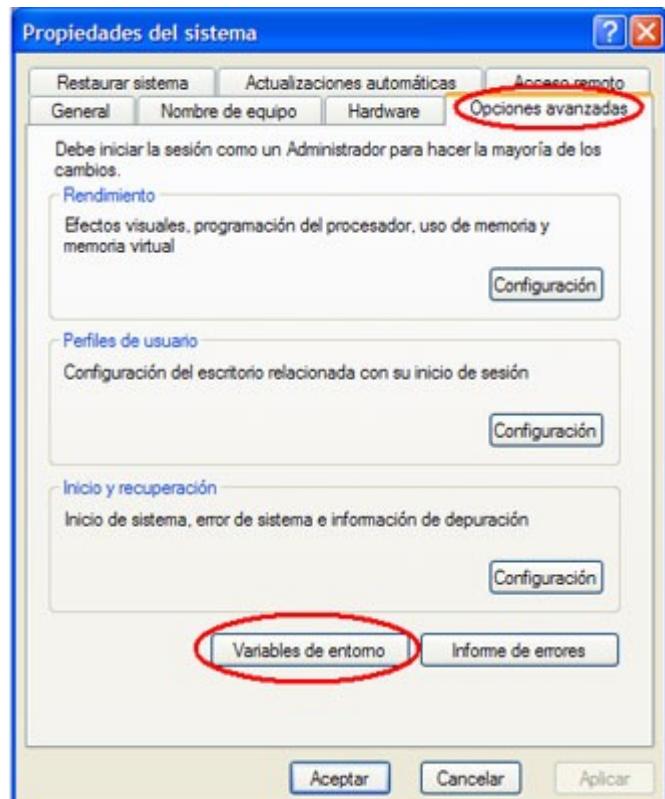


Ilustración 132: Propiedades del sistema

A continuación, en la siguiente pantalla seleccione en la zona correspondiente a *Variables del sistema* la variable *Path* que vamos a modificar y acto seguido pulse el botón *Modificar*.

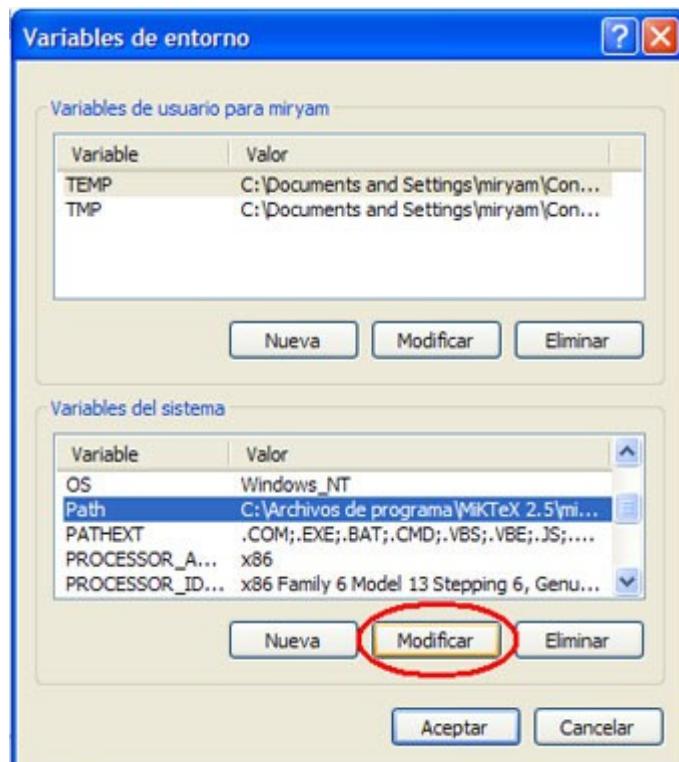


Ilustración 133: Modificación variables de entorno

Aparecerá la ventana que nos permite la modificación de la variable del sistema seleccionada, para ello deberá añadir en *Valor de variable* la ruta de la carpeta bin del JDK y pulsar Aceptar. En nuestro caso ;C:\jdk1.6.0_23\bin.

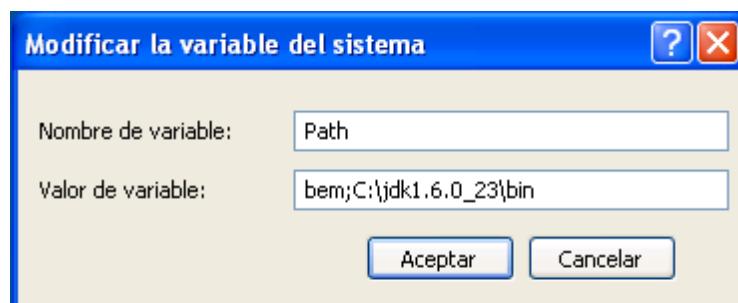
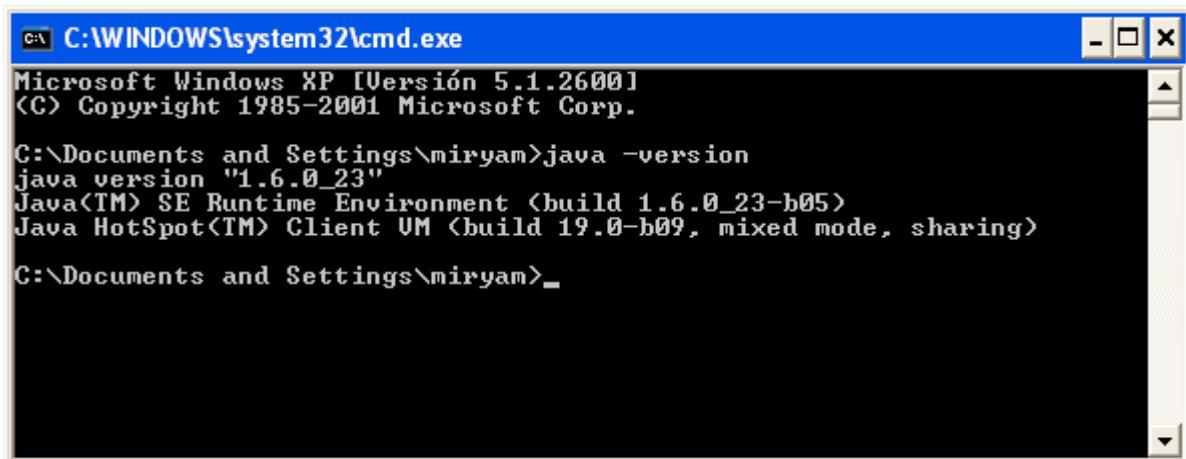


Ilustración 134: Modificación variable del sistema

Para comprobar que la instalación se ha realizado satisfactoriamente diríjase a *Inicio > Ejecutar* y escriba cmd, acto seguido aparecerá la consola de comandos, en ella deberá teclear el comando java -version. Si en ella aparece la versión que se acaba de instalar querrá decir que el proceso de instalación ha sido el correcto y por lo tanto ya es posible el desarrollo y ejecución de aplicaciones Java.



The screenshot shows a Windows XP Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window displays the following text:

```
Microsoft Windows XP [Versión 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\miryam>java -version
java version "1.6.0_23"
Java(TM) SE Runtime Environment (build 1.6.0_23-b05)
Java HotSpot(TM) Client VM (build 19.0-b09, mixed mode, sharing)

C:\Documents and Settings\miryam>
```

Ilustración 135: Comprobación instalación

5. INSTALACIÓN DE ECLIPSE

Para comenzar con la instalación del entorno de desarrollo Eclipse debemos de disponer del software que lleve a cabo la instalación. Para ello, se podrá obtener la distribución más adecuada conforme al sistema operativo que se esté utilizando y que más se adapte a las necesidades del desarrollador en la siguiente dirección web:

<http://www.eclipse.org/downloads/> [Eclipse Downloads n.d.]

En el caso de que la versión deseada sea la clásica para Windows podemos dirigirnos a la carpeta Software\desarrollo del CD que se adjunta con el proyecto donde se puede encontrar el fichero comprimido eclipse-SDK-3.6-win32.zip. Para realizar su instalación se requiere de 190 MB de espacio libre en disco.

Una vez que disponemos del software procederemos a su descompresión eligiendo el directorio deseado ya que será en ese donde permanezca instalado. Una vez acabada su descompresión, en el directorio seleccionado, se creará la siguiente estructura de ficheros de la aplicación:

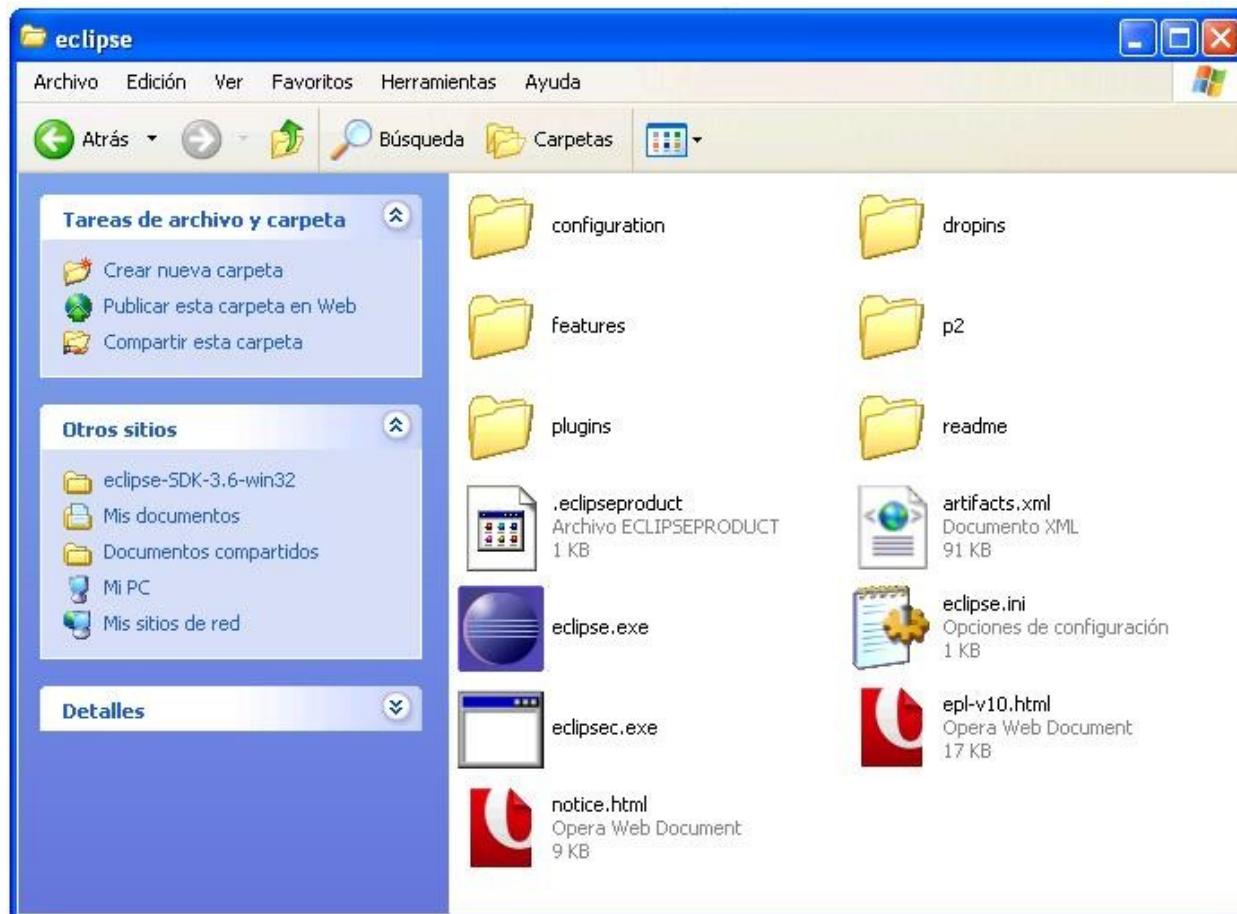


Ilustración 136: Estructura de ficheros de la instalación de Eclipse

En ella se encuentra el fichero `eclipse.exe`, haremos doble clic en él para arrancar el entorno de desarrollo y nos aparecerá la siguiente ventana donde podemos establecer el espacio de trabajo.



Ilustración 137: Selección espacio de trabajo Eclipse

A continuación nos dará la bienvenida y deberíamos tener configurado de forma automática el directorio donde esta instalado el *JDK*, de no ser así lo configuraríamos. En este momento ya estaría todo listo para crear un proyecto donde comenzar a desarrollar.

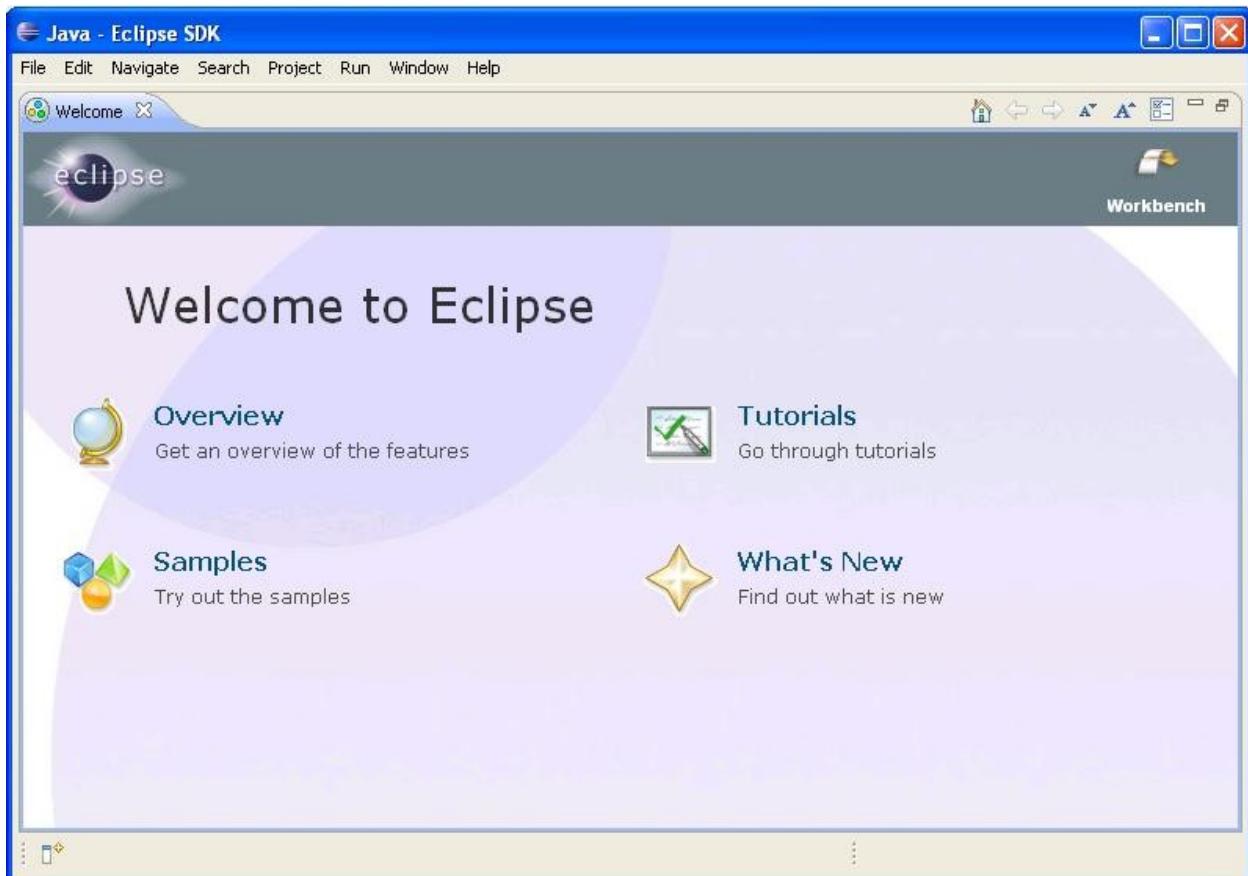


Ilustración 138: Bienvenida Eclipse

6. INSTALACIÓN DE ANT

Ant es una herramienta gratuita que facilita el proceso de desarrollo de software, especialmente en equipos grandes de desarrollo que trabajen en localizaciones distribuidas.

Ant es compatible con plataformas Linux, Unix, Windows, OS/2 Warp o MacOs. Para poder utilizar Ant es imprescindible contar con un *parser* de *XML* que cumpla con la especificación *JAXP*. Además, deberá estar disponible en el propio *CLASSPATH* del sistema. Para comodidad del usuario, la distribución binaria de Ant incluye la última versión del *parser XML* Apache Xerces2, aunque se podría sustituir por otro *parser* si así se desea.

Notas previas a la instalación

La distribución binaria de Ant consiste en la siguiente distribución de directorios:

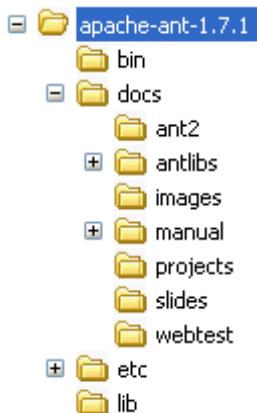


Ilustración 139: Directorios Apache Ant

Son imprescindibles los directorios `bin` (ejecutables) y `lib` (bibliotecas), el resto contienen utilidades adicionales y documentación para el usuario. Es importante advertir que no se debe instalar el fichero `ant.jar` de Ant en el directorio `lib\ext` del *JDK* o el *JRE*. Ant es una aplicación, mientras que la finalidad del directorio de extensión es albergar extensiones del *JDK*.

6.1. Proceso de instalación

Para comenzar con la instalación de Ant debemos de disponer del software que lleve a cabo la instalación. Se podrá obtener desde la dirección web siguiente:

<http://ant.apache.org/> [Apache Ant n.d.]

Otra opción es utilizar la que se adjunta con el proyecto, para ello nos dirigiremos a la carpeta `Software\desarrollo` del CD donde encontraremos el fichero comprimido `apache-ant-1.7.1.zip`. Para su instalación se requiere de 45 MB de espacio libre en disco.

Una vez que disponemos del software procederemos a su descompresión eligiendo el directorio deseado ya que será en ese donde permanezca instalado. Una vez acabada su descompresión, en el directorio seleccionado, se creará la estructura de ficheros anteriormente mostrada. A este directorio se le conocerá como `ANT_HOME`.

6.2. Configuración de variables de entorno

Para poder ejecutar Ant, es necesario configurar una serie de parámetros en el sistema. El proceso a seguir para ello será el que a continuación detalla:

- Añadir el directorio `bin` al `PATH` del sistema.
- Establecer una variable de entorno `ANT_HOME`, con el nombre del directorio en el que se instaló Ant. En algunos sistemas operativos, los scripts de Ant pueden llegar a averiguar el valor de `ANT_HOME` (dialectos de UNIX, y Windows NT/2000), pero es más fiable utilizar una variable específica para apuntar al directorio.
- Para poder utilizar algunas tareas que interactúan con clases del `JDK`, conviene definir una segunda variable adicional, `JAVA_HOME`, que apunte al directorio en el que se encuentra instalado el `JDK`.

Suponiendo que Ant esté instalado en el directorio `C:\Ant`, los siguientes comandos configurarán el necesario entorno de ejecución:

```
set ANT_HOME=C:\Ant  
set JAVA_HOME=C:\jdk1.6.0_23  
set PATH=%PATH%;%ANT_HOME%\bin
```

También se puede acudir a la opción *Propiedades* del menú contextual que se abre al pulsar el botón derecho del ratón sobre *Mi PC*; y una vez allí, crear las nuevas variables en la sección *Variables de entorno* de la pestaña *Opciones avanzadas*, como ya hemos visto con anterioridad en la instalación de otras herramientas.

7. INSTALACIÓN DE LA APLICACIÓN

La instalación de la aplicación se divide en dos partes bien diferenciadas; por una parte la instalación del plugin para Eclipse, Dynamic Refactoring Plugin, y por otra parte la instalación de la tarea Ant, RefactoringPlan. Como requisito a su instalación esta la instalación previa de las herramientas que se han ido describiendo hasta ahora.

7.1. Instalación de Dynamic Refactoring Plugin

La instalación del plugin para Eclipse, Dynamic Refactoring Plugin, se puede realizar de dos formas. A continuación presentamos las dos posibilidades que existen para llevar a cabo este proceso; la instalación manual que requiere tener disponible el CD que se suministra con el proyecto, ya que en él se encuentra el software, o bien la instalación por red que requiere de conexión a Internet, ya que previamente necesitará descargarse de forma automática el software para proceder a su instalación.

7.1.1. Instalación manual

El proceso de instalación manual del plugin se realizará a partir de software que se encuentra en el CD que se suministra con el proyecto. Para ello, se debe copiar el fichero comprimido que se encuentra en la siguiente carpeta Ejecutable\Dynamic Refactoring Plugin de dicho CD con el nombre Dynamic Refactoring.zip, copiarlo y descomprimirlo.

Una vez terminada su descompresión, se pueden observar los directorios features y plugins. A continuación de se deberá copiar el contenido de cada uno de estos directorios en el directorio del mismo nombre que se encuentra en la instalación de Eclipse que se desea utilizar.

Cuando arranquemos el entorno de desarrollo de Eclipse ya se dispondrá en él de la nueva funcionalidad que aporta el pulgin.

Desinstalación del pulgin

Para realizar la desinstalación del plugin bastará con eliminar las carpetas cuyo nombre comience con DynamicRefactoring de los directorios plugins y features que se encuentran en la instalación de Eclipse. Si se quiere desinstalar el paquete de idioma

castellano para trabajar con la versión original, se debe eliminar sólo la carpeta `DynamicRefactoring.nll` copiada.

7.1.2. Instalación automática por red

Como hemos comentado anteriormente, el proceso de instalación por red requiere disponer de conexión a Internet ya que previamente a la instalación necesitará descargarse, de forma automática, el software necesario desde el servidor web habilitado para tal efecto.

La instalación por esta vía es la más adecuada ya que, además de tratarse de un proceso más sencillo, siempre pone a disposición del usuario la última versión del plugin disponible. Además de ofrecerle la posibilidad de su actualización de forma automática.

Para comenzar la instalación arrancaremos el entorno de desarrollo Eclipse en el cual queramos realizar su instalación. Nos dirigiremos al menú *Help* de la barra de herramientas de Eclipse y pulsaremos en él para, posteriormente, seleccionar la opción *Install New Software...* que nos aparece entre las disponibles.

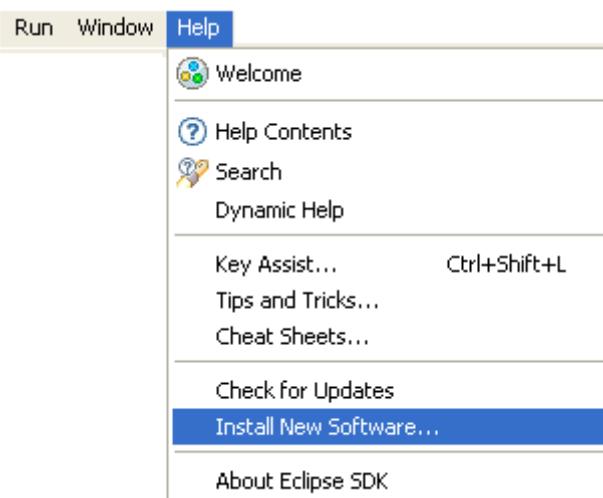


Ilustración 140: Menú Help en Eclipse- Install New Software

A continuación nos aparece una ventana donde debemos de indicar el repositorio software donde se encuentra el plugin para poder realizar su descarga.

Para ello, pulsaremos el botón *Add* que dará paso a un dialogo donde tendremos que introducir el nombre del repositorio y la ruta a través la cual es accesible este.

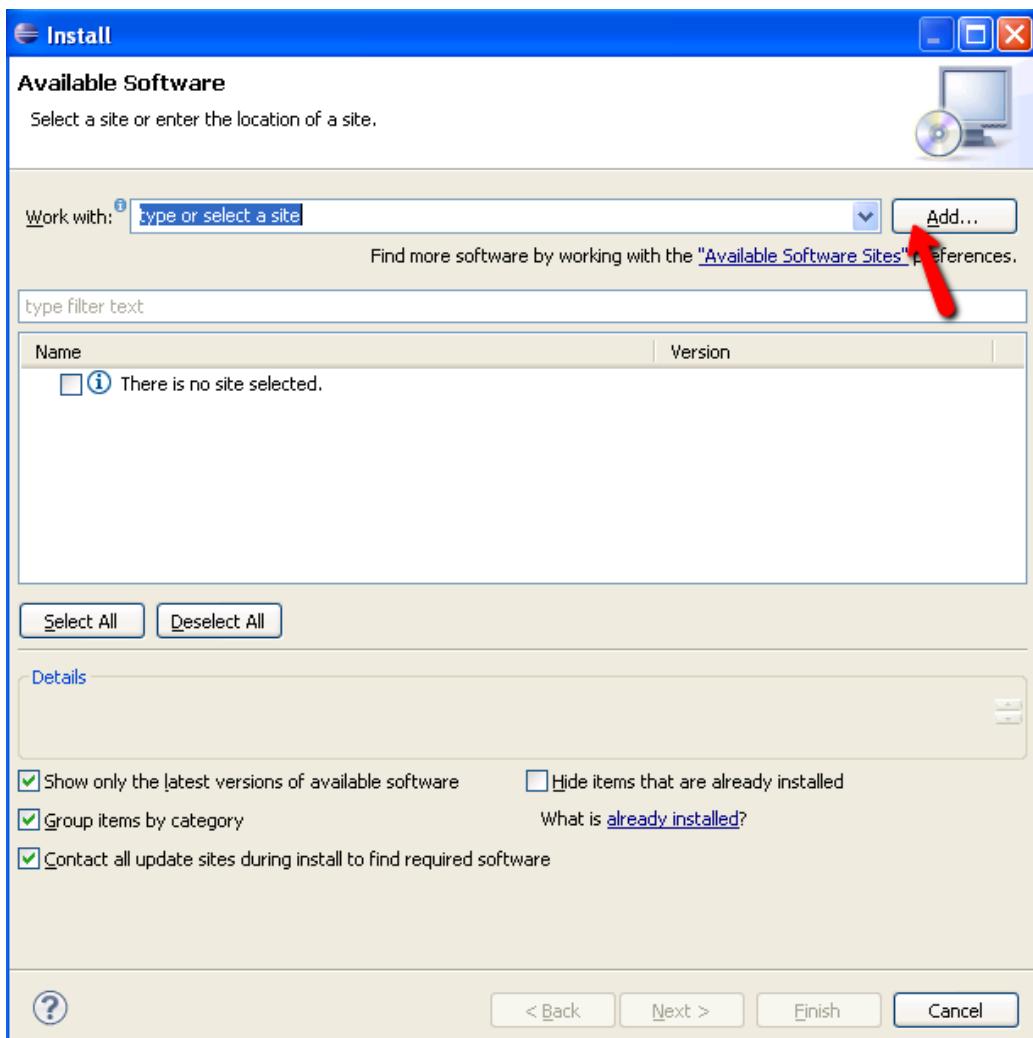


Ilustración 141: Agregar repositorio del plugin

Como nombre de repositorio se deberá escoger un nombre identificativo del repositorio del plugin a elección del usuario, por ejemplo: *DynamicRefactoring Repository* y en el apartado dedicado para la ruta del repositorio indicaremos la siguiente:

```
jar:http://refactoring-
plugin.googlecode.com/hg/dynamicrefactoring.p2repository.zip!
```

Cabe destacar que es importante prestar especial atención en no olvidarse de ninguno de los caracteres que esta contiene, ya que de lo contrario el repositorio no será accesible. Con el fin de evitar tener que teclear la ruta y con ello posibles confusiones se encuentra disponible la *URL* anterior en el fichero `url_instalacion_plugin.txt` incluido en la carpeta Documentación\Documentación Adicional del CD suministrado con el proyecto.

Como dato de interés, el hecho de que la ruta empiece por '`jar:`' y acabe en '!' es un indicador de que el repositorio se encuentra en formato de fichero comprimido. Este tipo de repositorios permiten la opción de ser descargados directamente desde la web y ser utilizados como repositorios de acceso local para realizar la instalación.

Una vez introducidos los datos, como se muestra en la siguiente imagen, pulsamos el botón *OK*.

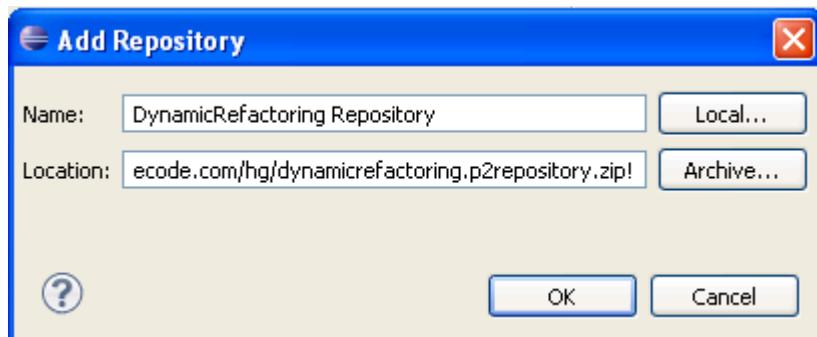


Ilustración 142: Introducir datos repositorio

A continuación Eclipse se dispondrá a descargar los datos del repositorio. Mientras se encuentre realizando esta tarea aparecerá el texto *Pending...* en la ventana de diálogo.

Una vez Eclipse haya terminado de descargarse dicha información, mostrará en pantalla el plugin con la categoría y la versión disponible. Para comenzar el proceso de instalación bastará con seleccionar el plugin pulsando los *checkbox* que aparecen a su izquierda o bien pulsaremos el botón *Select All* para conseguir el mismo efecto y por último pulsar el botón *Next >*.

Nótese que la versión que aparece en las siguientes imágenes no tiene por qué coincidir con la última versión del plugin disponible, por lo que no se deberá de tener como referencia.

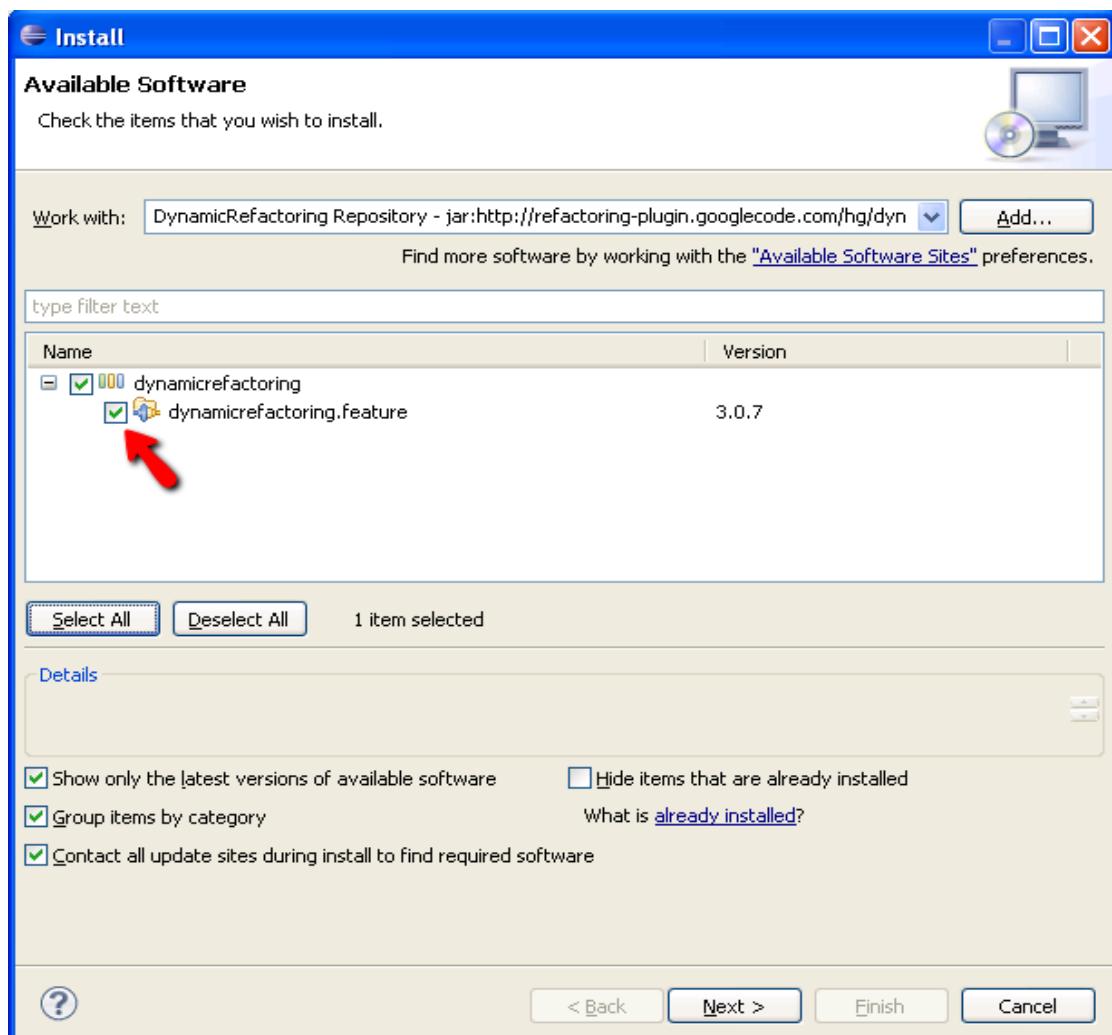


Ilustración 143: Selección plugin para su instalación

La siguiente pantalla recoge los detalles de la instalación, si seleccionamos el plugin que aparece en la lista de elementos a instalar aparecerán una breve explicación del mismo en la zona inferior de la ventana.

Si se desea ampliar esta información pulsaremos *More...* y aparecerá una nueva ventana que recoge las distintas propiedades del plugin.

Pulsaremos *Next >* para avanzar a la siguiente pantalla del asistente de instalación del plugin.

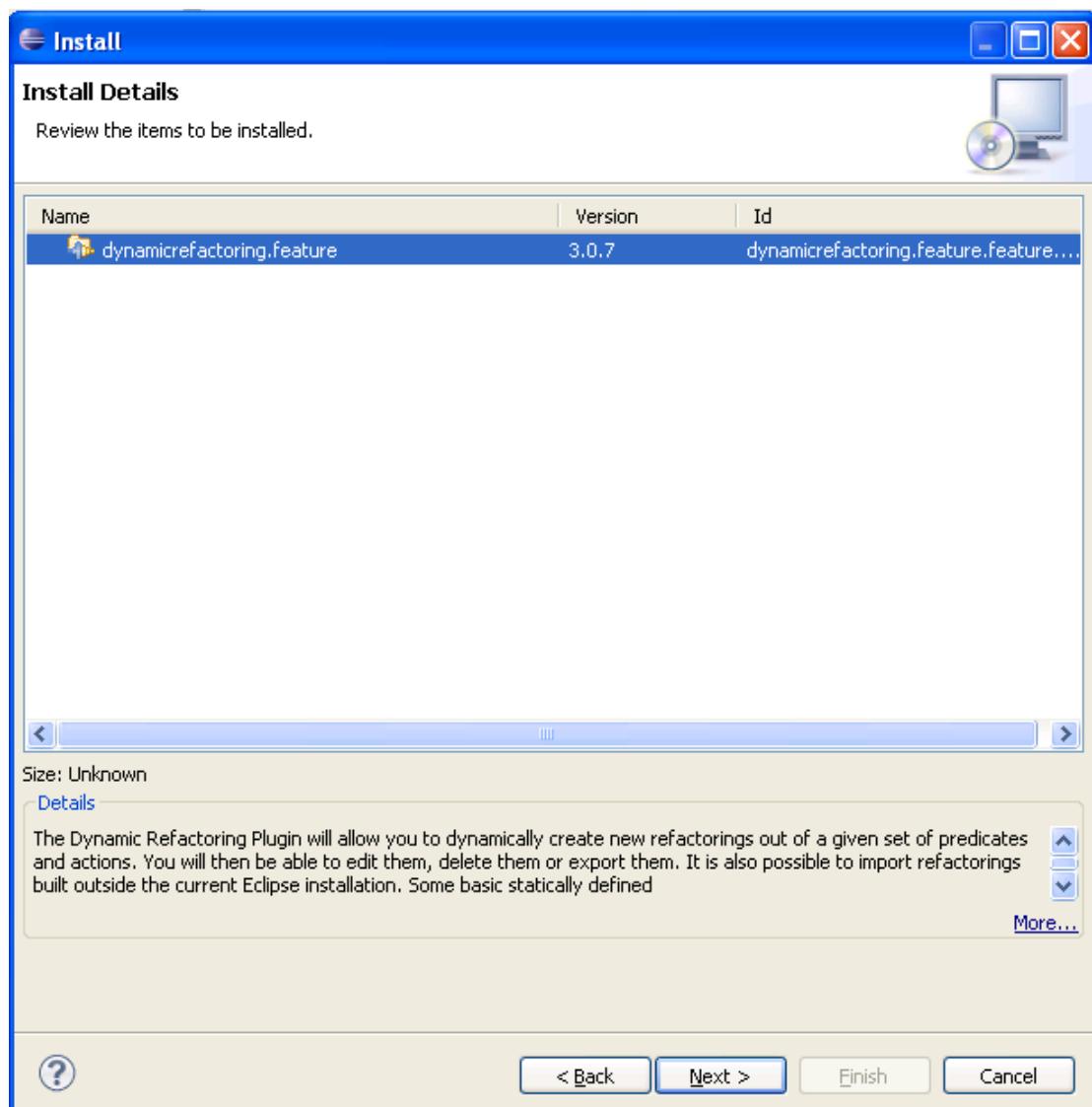


Ilustración 144: Ventana descriptiva del plugin

Esta pantalla muestra la información relativa a la licencia del plugin, podremos leerla y si estamos conformes a ella seleccionaremos el botón *I accept the terms of the license agreement* para que así quede reflejado y pulsaremos el botón *Finish* para que de comienzo el proceso de instalación. En caso contrario, se cancelará no realizando la instalación del plugin.

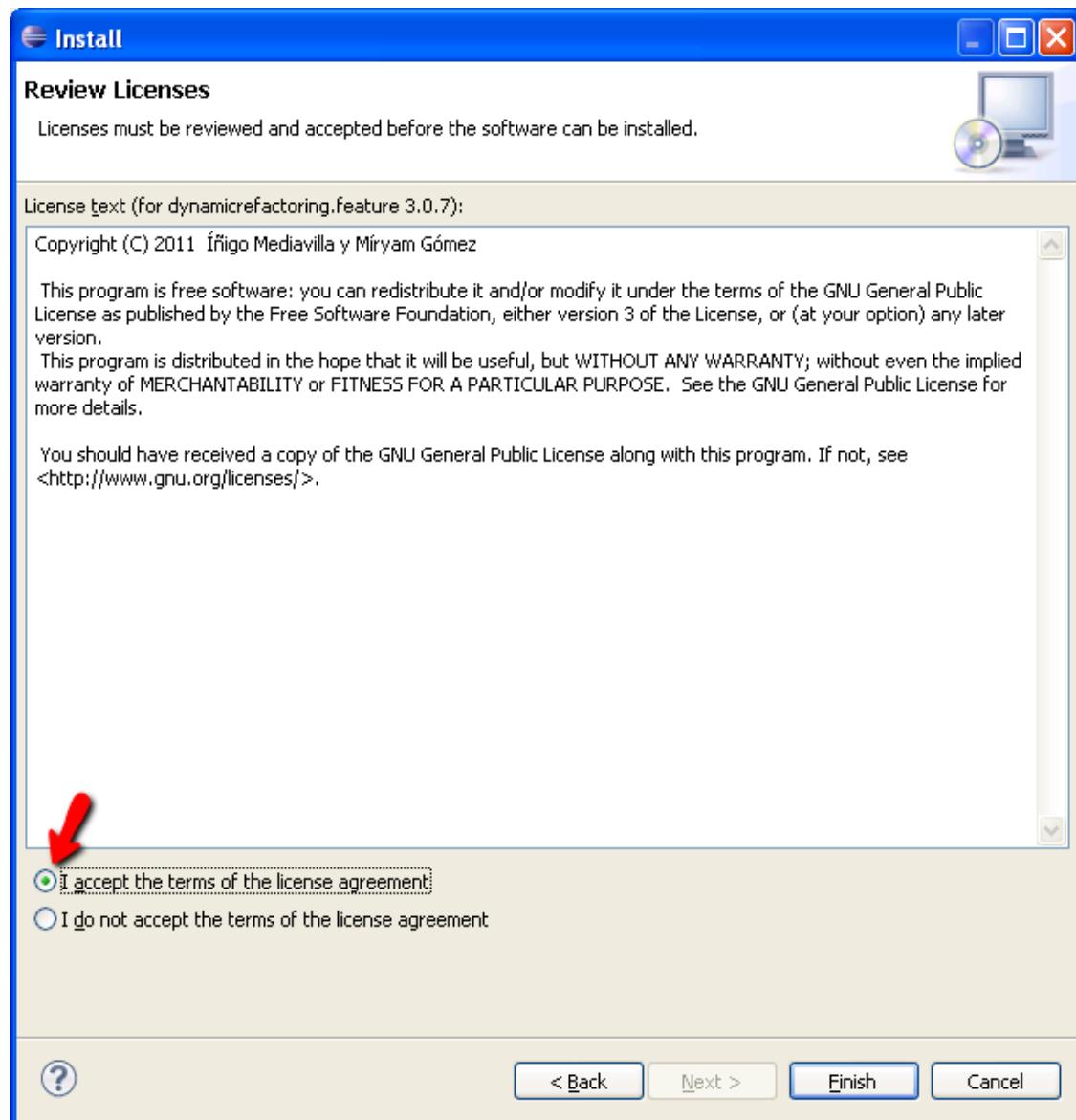


Ilustración 145: Ventana licencia plugin

Eclipse nos informará del progreso del proceso de instalación con esta ventana:

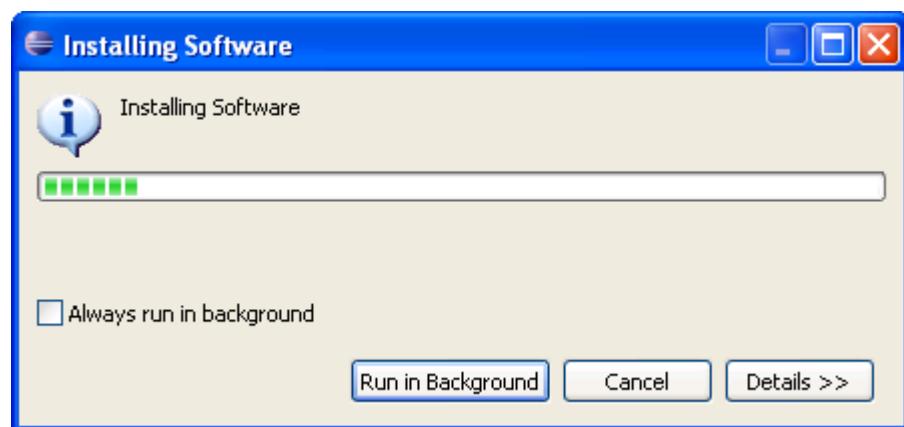


Ilustración 146: Instalación plugin

Durante el proceso de instalación, el cual durará varios minutos, aparecerá un cuadro de diálogo indicando que el software se encuentra firmado, en él se mostrarán los datos de la autoridad que lo firmó y aparecerá disponible un botón que podrá ser pulsado si se desea consultar más información acerca del mismo.

La finalidad de este dialogo es preguntar si se desea confiar en el certificado y por lo tanto seguir con el proceso de instalación. En caso de ser así, deberemos seleccionar el certificado o bien pulsar en el botón *Select All*, con el que conseguiremos el mismo efecto.

Por último, pulsaremos el botón *OK* para poder continuar.

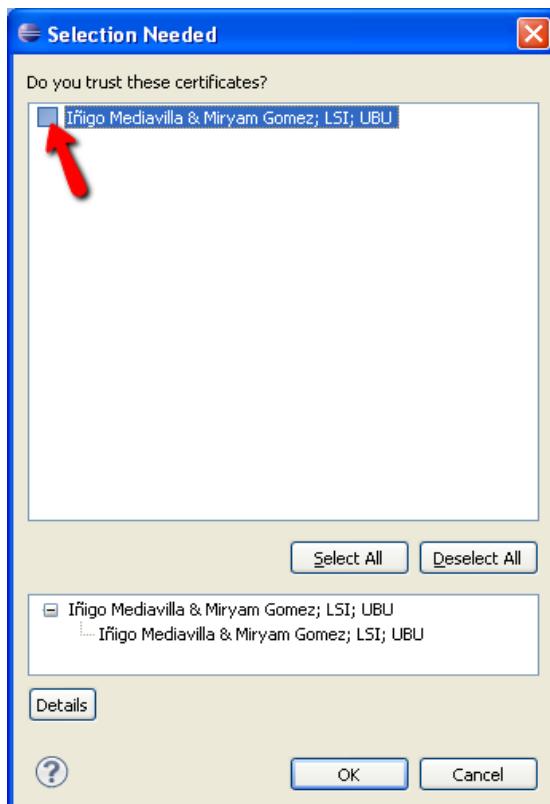


Ilustración 147: Certificado plugin



Ilustración 148: Detalles del certificado

Cuando el proceso ha terminado Eclipse preguntará si se quiere reiniciar el entorno de desarrollo para que los cambios producidos por la instalación hagan efecto y poder así empezar a utilizar las nuevas funcionalidades incorporadas por el plugin. Si así se desea se pulsará la opción de *Restart Now*, en caso contrario lo dejaremos para más tarde pulsando el botón *Not Now* o bien se pulsará *Apply Changes Now* para hacer efectivos los cambios sin reiniciar Eclipse lo que podrá provocar errores. Esta última opción no es recomendable.



Ilustración 149: Ventana reinicio tras instalación del plugin

Una vez que Eclipse se haya reiniciado ya podemos disfrutar de las funcionalidades del plugin de refactorizaciones desde nuestro espacio de trabajo.

7.1.3. Actualización automática por red

Para actualizar el plugin a su última versión disponible es muy sencillo, basta con dirigirnos al menú *Help* de la barra de herramientas de Eclipse y pulsar en él para, en este caso, seleccionar la opción *Check for Updates...* que nos aparece entre las disponibles.

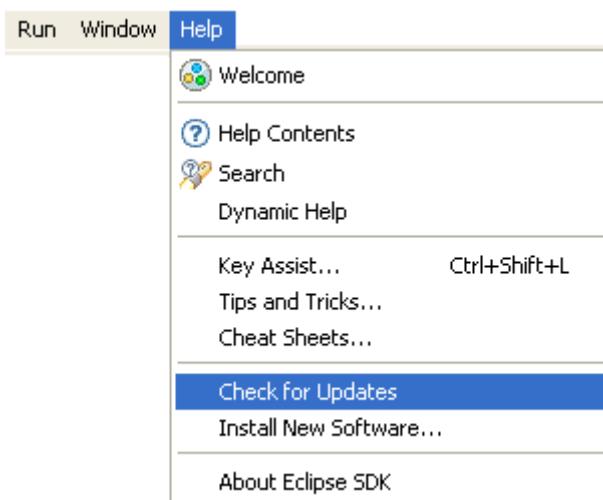


Ilustración 150: Menú Help en Eclipse- Check for Updates

Mientras Eclipse realiza la comprobación para obtener los plugins para los cuales existen actualizaciones aparecerá la siguiente ventana, que nos informará del progreso de este proceso.

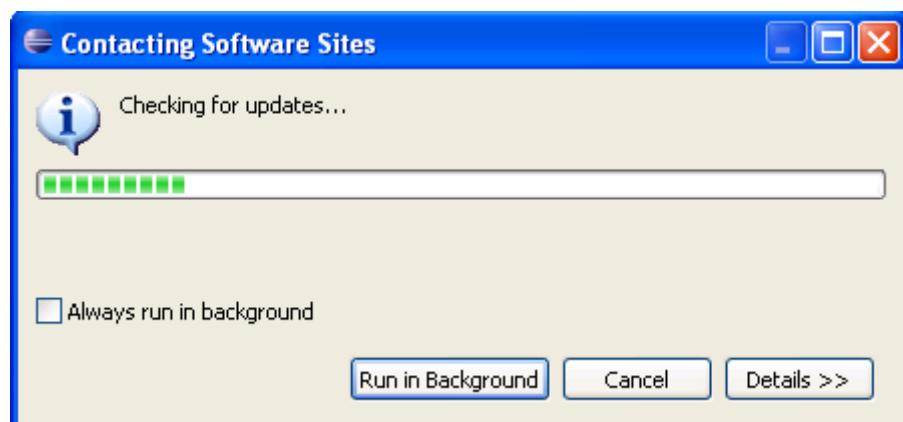


Ilustración 151: Comprobar actualizaciones disponibles de plugins

A continuación aparecerá una ventana con la lista de plugins para los que existen actualizaciones, es decir, para los que se dispone de nuevas versiones. Dentro de esa lista basta con marcar *dynamicrefactoring.feature* para iniciar el proceso de actualización a la última versión del plugin siguiendo un proceso similar al de instalación ya descrito.

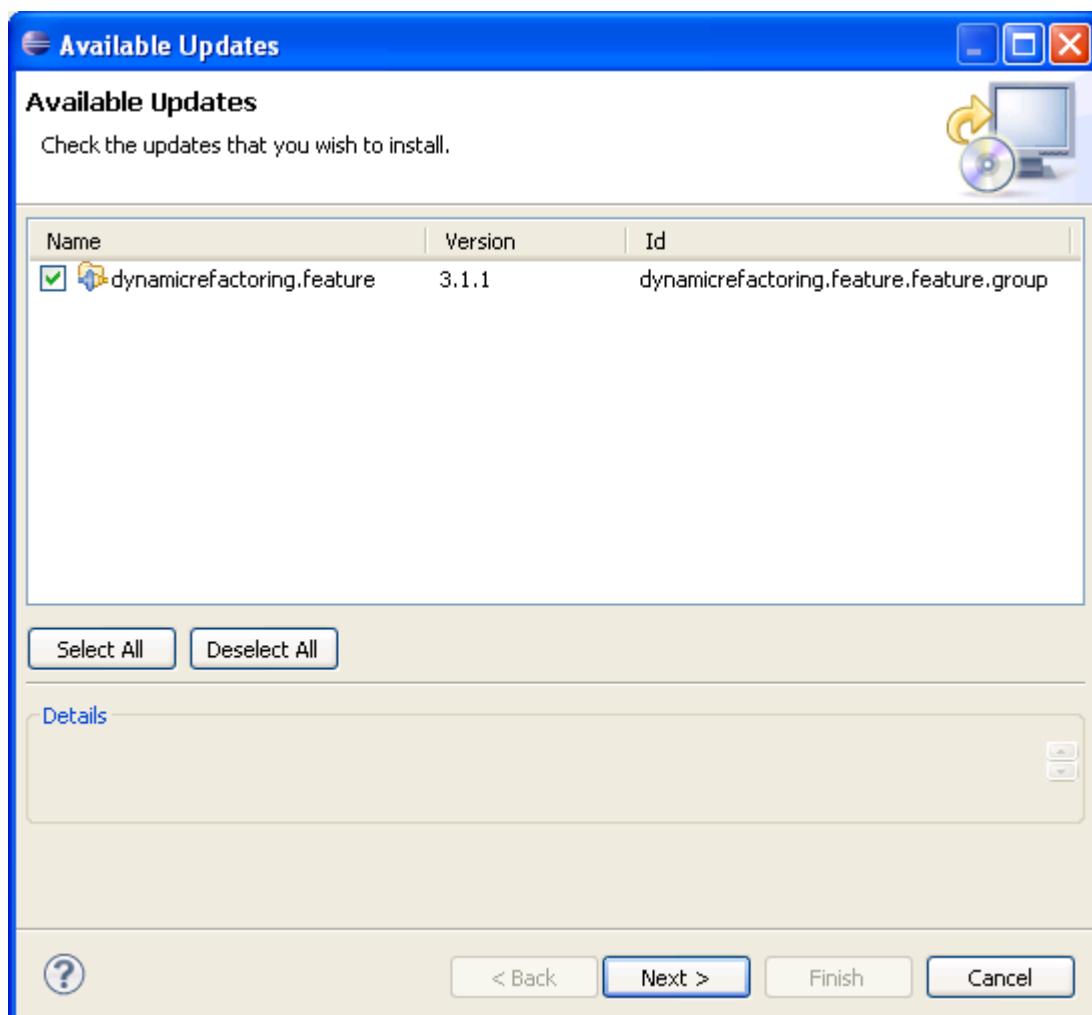


Ilustración 152: Available Updates

Indicar que para realizar la actualización del plugin de forma automática existe otra opción que es la misma que a continuación se detalla para el proceso de desinstalación pero eligiendo la opción *Update...* .

7.1.4. Desinstalación automática

Para realizar la desinstalación del plugin de forma automática nos dirigirnos al menú *Help* de la barra de herramientas de Eclipse y pulsaremos en él para, en este caso, seleccionar la opción *About Eclipse SDK* que nos aparece entre las disponibles.

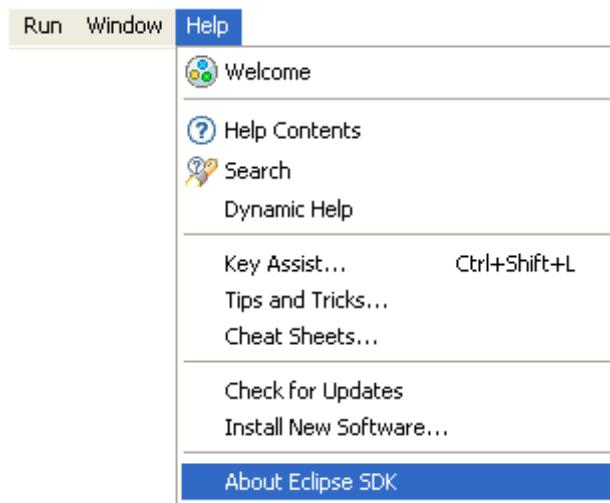


Ilustración 153: Menú Help en Eclipse- About Eclipse SDK

A continuación aparecerá una ventana en la que se muestra la información relativa a la versión de Eclipse que se encuentra instalada así como los plugin que se han instalado, entre estos se encontrará el plugin de refactorizaciones. Pulsaremos el botón *Installation Details*.



Ilustración 154: About Eclipse SDK

Esta nueva ventana contendrá una serie de pestañas, nos fijaremos en la primera pestaña, que corresponde con *Installed Software*, en ella aparecerá la relación de software instalado, de esta lista seleccionaremos el plugin pulsando `dynamicrefactoring.feature` y seguidamente pulsaremos el botón *Uninstall...* .

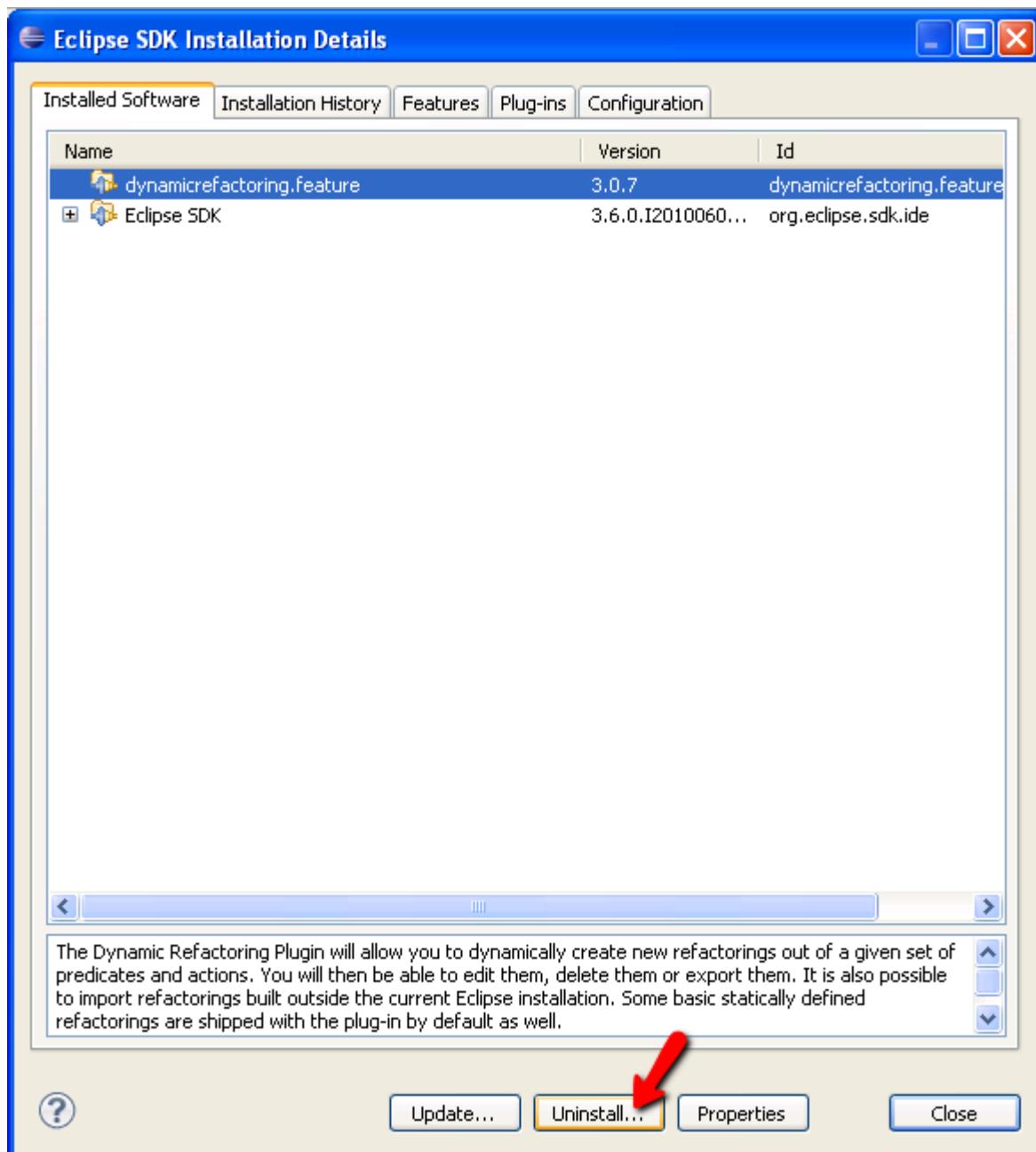


Ilustración 155: Eclipse SDK Installation Details

Es ahora cuando comenzará el proceso de desinstalación, Eclipse mediante el siguiente dialogo nos solicitará realizar la confirmación del software a desinstalar. Para ello,

seleccionaremos de nuevo el plugin y pulsaremos el botón correspondiente a *Finish*.

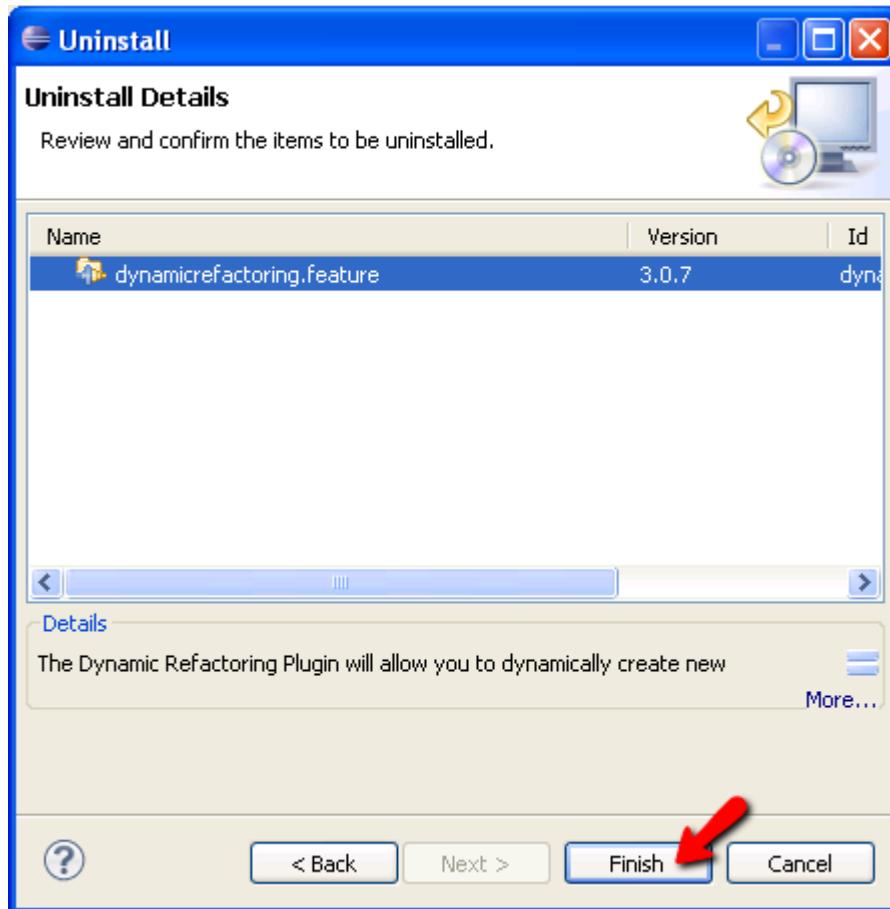


Ilustración 156: Uninstall Details

Mientras Eclipse realiza la desinstalación del plugin aparecerá la siguiente ventana, que nos informará del progreso de este proceso.

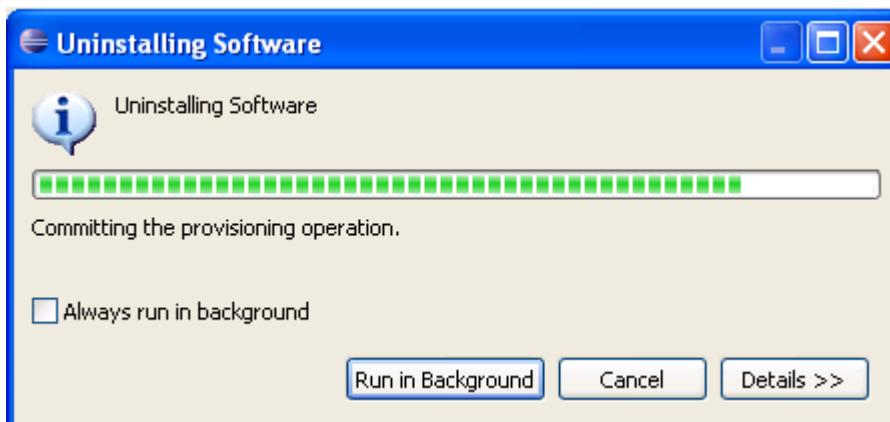


Ilustración 157: Uninstalling Software

Cuando el proceso ha terminado Eclipse preguntará si se quiere reiniciar el entorno de desarrollo para que los cambios producidos por la desinstalación hagan efecto. Si así se desea se pulsará la opción de *Restart Now*, en caso contrario lo dejaremos para más tarde pulsando el botón *Not Now* o bien se pulsará *Apply Changes Now* para hacer efectivos los cambios sin reiniciar Eclipse lo que podrá provocar errores. Esta última opción no es recomendable.

Una vez que Eclipse se haya reiniciado podemos comprobar que las funcionalidades incorporadas por el plugin de refactorizaciones ya no se encuentran disponibles.

7.2. *Instalación de la tarea Ant RefactoringPlan*

La instalación de la tarea Ant, RefactoringPlan, consiste en copiar el contenido de la carpeta Ejecutable\Tarea Ant RefactoringPlan a cualquier ubicación del equipo en la que se deseé realizar dicha instalación.

8. EJECUCIÓN DE LA APLICACIÓN

Este apartado esta dedicado a la ejecución de la aplicación, por una parte se explicará la ejecución del plugin para Eclipse, Dynamic Refactoring Plugin, y por otra parte la ejecución de la tarea Ant, RefactoringPlan.

8.1. *Ejecución de Dynamic Refactoring Plugin*

El proceso de ejecución del plugin, Dynamic Refactoring Plugin, no requiere de ningún paso adicional. Simplemente cuando arrancamos Eclipse este se encarga de la carga de los plugins que se encuentran instalados.

Para comprobar que la carga de nuestro plugin se ha realizado con éxito basta con fijarse si la interfaz gráfica del entorno de desarrollo incorpora las nuevas funcionalidades del plugin, como son:

Elemento del menú de la barra de herramientas: *Dynamic Refactoring*

Vistas: *Refactoring Progress, Refactoring History,*

Available Refactorings, Refactoring Catalog Browser.

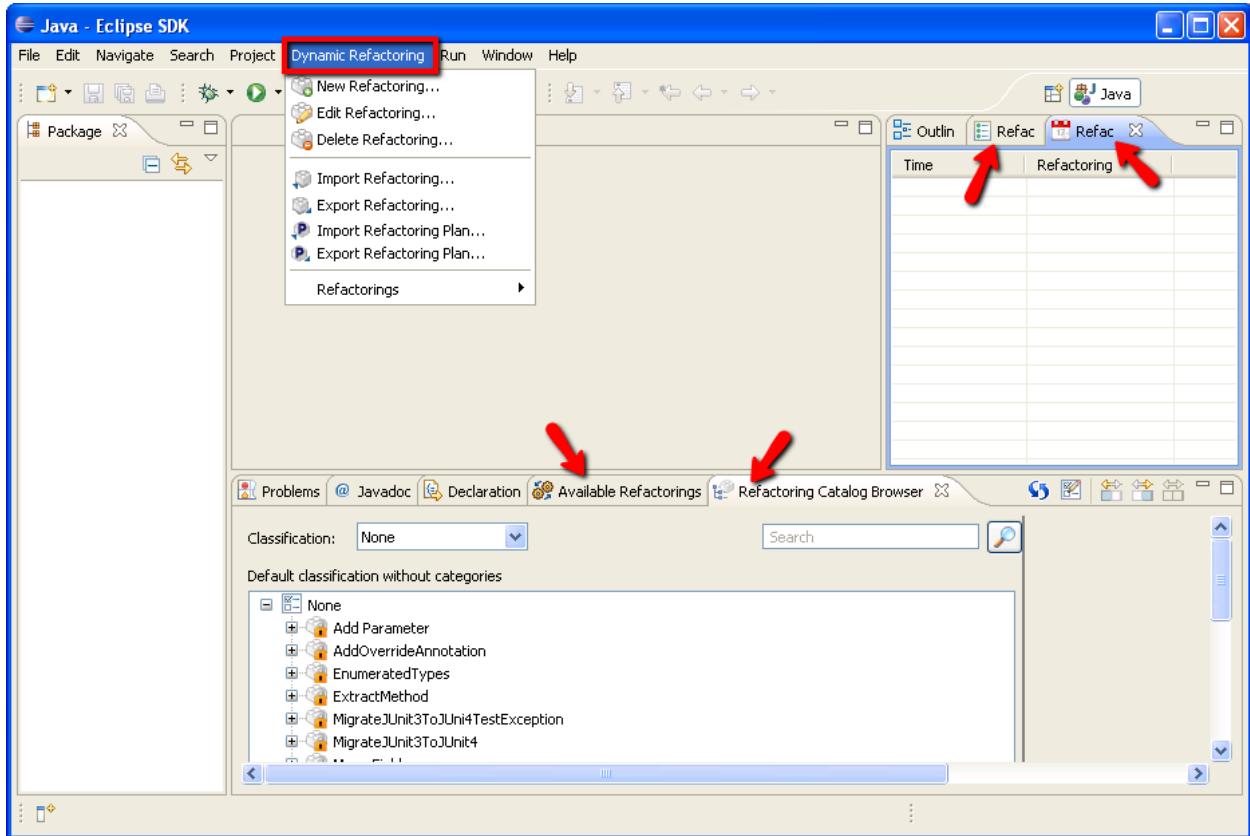


Ilustración 158: Eclipse con Dynamic Refactoring Plugin instalado

En este momento ya podemos disfrutar de todas las funcionalidades del plugin de refactorizaciones desde nuestro espacio de trabajo.

8.1.1. Cómo mejorar el rendimiento de Eclipse

Debido a que Eclipse es un *IDE* que posee la característica de que se le puede aumentar las prestaciones mediante el uso de plugins y que en nuestro caso así lo haremos instalando el plugin de refactorizaciones, esto provocará que el consumo de memoria RAM se eleve por lo que posiblemente sea necesario mejorar su rendimiento.

Para mejorar el rendimiento de Eclipse se dispone del fichero de configuración `eclipse.ini` que se encuentra en la misma carpeta que el ejecutable, `eclipse.exe`, en el cual se puede establecer entre otras opciones los valores deseados para la memoria. Este fichero de configuración se carga en el arranque de Eclipse.

A la hora de editar este fichero de configuración es importante aclarar que cada opción y cada argumento de la misma debe estar en su propia linea. Además las líneas que se encuentren después de la opción `-vmargs` son pasadas como argumentos a la *JVM* (*Java Virtual Machine*).

Virtual Machine) y las que hayan sido especificadas antes serán las opciones propias para Eclipse, al igual que cuando se utilizan argumentos en la línea de comandos.

Las opciones que en nuestro caso nos interesan son las que se encuentran después de la opción `-vmargs`, es decir, las que se pasan como argumentos a la *JVM*. En concreto, las siguientes, para las cuales se da un ejemplo:

- `vmargs` Parámetros adicionales a pasar a la *JVM*
- `Xms128m` Destinar como mínimo 128Mb de memoria para la pila (heap) de *JVM*
- `Xmx1024m` Destinar como máximo 1Gb de memoria para la pila (heap) de *JVM*
- `XX:MaxPermSize=128m` Utilizar 128Mb máximo como espacio de pila permanente de generación.
- `XX: PermSize=128m` Utilizar 128Mb espacio de pila permanente de generación

Por lo tanto, estos parámetros indican a la máquina virtual de Java las condiciones de memoria con las que puede trabajar Eclipse, si queremos que su rendimiento mejore, tendremos que aumentar los valores por defecto en el fichero de configuración. Así además de conseguir una ejecución más fluida evitaremos los molestos errores relacionados con la falta de memoria, *Out of Memory*, *Stack Overflow*, etc.

Los valores óptimos a asignar a cada opción del fichero de configuración dependerán de las prestaciones de cada equipo, en este caso, de la memoria disponible que tenga. A continuación se dan unos valores que pueden ser tomados como referencia dependiendo de la cantidad de memoria RAM disponible en el equipo:

Para 2.5GB de memoria RAM:

```
-Xms1024m -Xmx1536m -XX:PermSize=128m -XX:MaxPermSize=128
```

Para 1GB de memoria RAM:

```
-vmargs -Xms256m -Xmx512m -XX:PermSize=128m -XX:MaxPermSize=128m
```

Para 512MB de memoria RAM:

```
-vmargs -Xms128m -Xmx256m -XX:PermSize=64m -XX:MaxPermSize=64m
```

No obstante cabe destacar que aparte de tener en cuenta la memoria física de la que disponga el equipo se debe de tener en cuenta la memoria que está siendo ya utilizada, ya

que de lo contrario una asignación excesiva de memoria podría afectar negativamente al comportamiento de otras aplicaciones que estemos utilizando así como al propio sistema operativo. Es por ello, que daremos una regla simple con la que orientarnos en la elección de los valores, que se basará en la memoria libre disponible. Esta se determinará como el resultado de la resta entre la memoria física disponible y la memoria utilizada por el sistema operativo y las aplicaciones que habitualmente se utilicen. De esa memoria es recomendado tomar 2/3 para la memoria máxima ($-Xmx$) y 2/3 de la memoria máxima para memoria mínima ($-Xms$). Para obtener más información, se puede consultar la sección *Set-up: optimized Java JVM configuration* en el siguiente enlace:

http://www.ibm.com/developerworks/websphere/library/techarticles/0204_searle/searle.html

8.2. Ejecución de la tarea Ant RefactoringPlan

La tarea Ant refactoringPlan puede ser ejecutada desde diferentes entornos. En este anexo se va a explicar como ejecutarla desde consola y desde el entorno de desarrollo Eclipse. Para poder ejecutar una tarea de Ant de forma correcta es necesario conocer las entradas de la misma, por ello a continuación se va a proceder a explicar cada una de las entradas de esta tarea:

- dest: directorio donde se crearan los ficheros .java tras la ejecución del plan de refactorizaciones.
 - source: directorio donde se encuentran los ficheros .java sobre los que se quiere ejecutar el plan de refactorizaciones.
 - RefactoringPlanPath: directorio con el plan de refactorizaciones a ejecutar.
- La estructura de este directorio tiene que contener los siguientes elementos:



Ilustración 159: Contenido del directorio refactoringPlan

8.2.1. Ejecución desde consola

Para poder utilizar esta tarea es necesario la creación de un fichero .xml, por defecto

el nombre de este fichero es build.xml. Dentro de este fichero se necesita relacionar el nombre de la tarea ant con la clase que implementa su funcionalidad mediante una tarea taskdef.

Dentro de taskdef se tiene que indicar el nombre de la nueva tarea a generar, el nombre de la clase a ejecutar cuando se llama a la tarea y el classpath que permite la correcta ejecución de la tarea.

Para que la tarea funcione de manera correcta este fichero tiene que encontrarse dentro del directorio de la aplicación refactoringPlan (directorio Ejecutable\Tarea Ant RefactoringPlan del CD) que contiene los ficheros .class y las bibliotecas que permiten la ejecución de la tarea. Este directorio, previamente, debe ser copiado a una ruta cualquiera dentro del ordenador. A continuación se muestra un ejemplo del fichero build.xml:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="MyTask" basedir=". " default="execute">

    <!-- Directorio de classes del refactoringPlan -->
    <property name="lib.dir" value="${basedir}/lib"/>

    <!-- Directorio de classes del refactoringPlan -->
    <property name="bin.dir" value="${basedir}/bin"/>

    <target name="execute.init"
            description="Execute a refactoring plan">
        <taskdef name="refactoringPlan"
                classname="RefactoringPlan">
            <classpath>
                <fileset dir="${lib.dir}">
                    <include name="**/*.jar"/>
                    <exclude name="ant.jar"/>
                </fileset>
                <pathelement location="${bin.dir}"/>
            </classpath>
        </taskdef>
    </target>
</project>
```

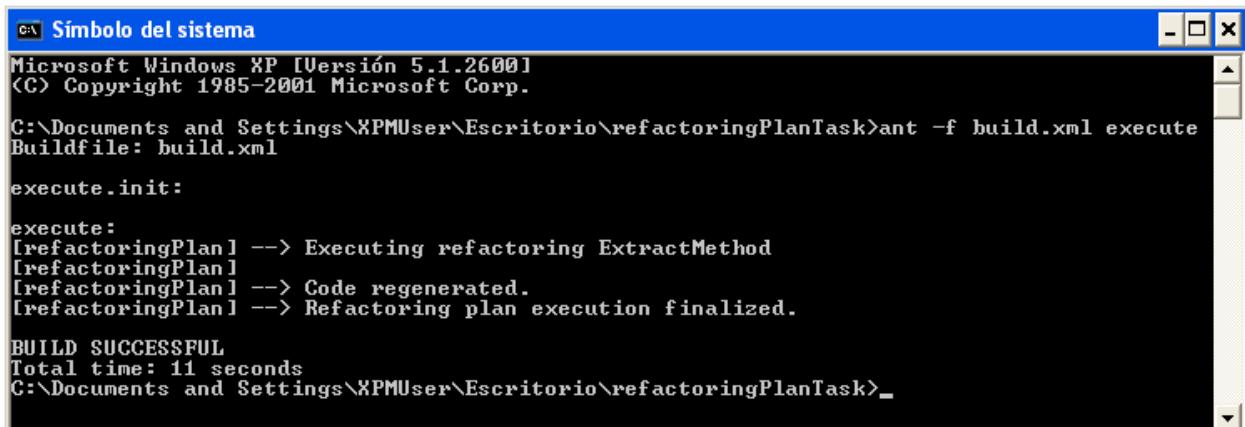
Después de definir la tarea, hay que crear una nueva etiqueta target, dentro del fichero XML anterior, dependiente de la tarea definida en el paso previo, que se encargue de ejecutar la tarea refactoringPlan y en la que se va a indicar las entradas que utilizará:

```
<target name="execute"
      description="Execute a refactoring plan"
      depends="execute.init">
  <refactoringPlan
    dest=".\\destination"
    source=".\\source"
    refactoringPlanPath=".\\refactoringPlan"/>
</target>
```

En el ejemplo anterior hay que adecuar el valor de los atributos de la tarea a los diferentes directorios en los que se encuentran los fichero fuentes, el plan de refactorizaciones y el directorio en el que se almacenará el resultado de la tarea (este directorio no tiene porque existir físicamente ya que si no está creado la tarea se encargará de generararlo).

Una vez generado el fichero .xml la tarea se puede ejecutar desde línea de comandos mediante la sentencia:

```
ant -f miFichero.xml execute
```



The screenshot shows a Windows command prompt window titled "Símbolo del sistema". The title bar also displays "Microsoft Windows XP [Versión 5.1.2600]" and "(C) Copyright 1985-2001 Microsoft Corp.". The main window contains the following text output from the Ant build process:

```
Microsoft Windows XP [Versión 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\XPMUser\Escritorio\refactoringPlanTask>ant -f build.xml execute
Buildfile: build.xml

execute.init:

execute:
[refactoringPlan] --> Executing refactoring ExtractMethod
[refactoringPlan]
[refactoringPlan] --> Code regenerated.
[refactoringPlan] --> Refactoring plan execution finalized.

BUILD SUCCESSFUL
Total time: 11 seconds
C:\Documents and Settings\XPMUser\Escritorio\refactoringPlanTask>
```

Ilustración 160: Ejecución tarea Ant RefactoringPlan desde consola

8.2.2. Ejecución desde Eclipse

Para poder ejecutar el fichero .xml que se ha creado en el apartado anterior desde Eclipse es necesario visualizar la vista *Ant*. Para ello hay que seleccionar la opción de menú *Window > Show View > Ant* de Eclipse.

También es necesario crear un proyecto en Eclipse con el código de la tarea. Para ello, nos dirigiremos al menú *File > New > Project...* y elegiremos un proyecto de tipo *General > Project*, pulsaremos *Next>*. En la siguiente ventana introduciremos el nombre del proyecto, por ejemplo `antTask`, y seleccionaremos la ruta en la que se encuentra este.

Tras crear el proyecto de Eclipse la estructura del mismo se debería corresponder con la siguiente:

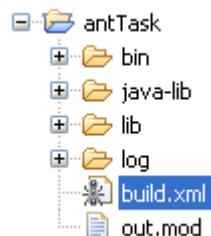


Ilustración 161: Proyecto tarea Ant RefactoringPlan

Una vez que se tiene creado el proyecto en Eclipse en el que se encuentra también el fichero `.xml` generado se debe pulsar el botón *Add BuildFields* dentro de la vista *Ant*.  Tras presionar este botón se abre un diálogo en el que hay que señalar el fichero `.xml` que se ha creado con la definición de la tarea.

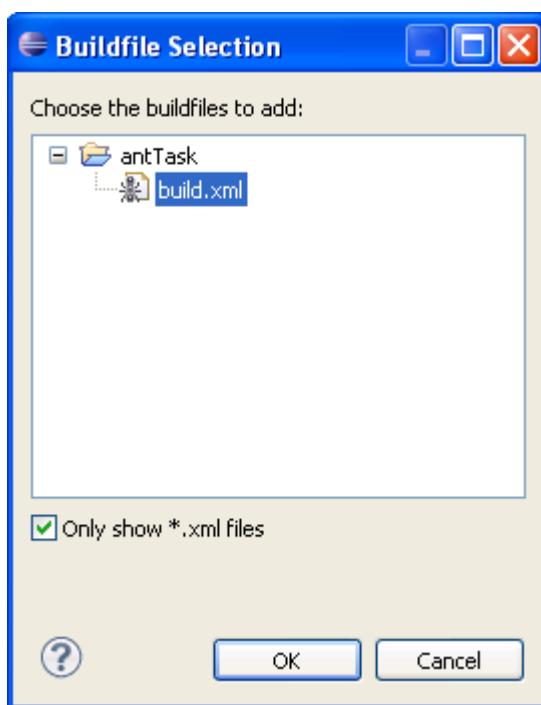


Ilustración 162: Buildfile Selection

Una vez que se le ha seleccionado se actualizará la vista con cada uno de los objetivos definidos en el fichero .xml, para poder ejecutar uno de ellos basta con dar doble click sobre el que nos interesa. En el caso del .xml que se ha puesto de ejemplo se pulsaría doble click sobre el objetivo execute que se encargaría de ejecutar la tarea RefactoringPlan.

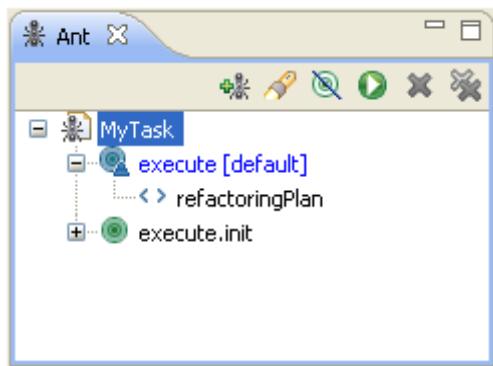


Ilustración 163: Despliegue del fichero build.xml en la vista Ant

En la consola del propio Eclipse podremos visualizar el resultado de la ejecución de la tarea correspondiente. En el caso del ejemplo que hemos visto el resultado obtenido es el siguiente:

```
<terminated> antTask build.xml [Ant Build] C:\Archivos de programa\Java\jre6\bin\javaw.exe (05/06/2011 19:50:08)
Buildfile: C:\Documents and Settings\XPMUser\Escritorio\refactoringPlanTask\build.xml
execute.init:
execute:
[refactoringPlan] --> Executing refactoring ExtractMethod
[refactoringPlan]
[refactoringPlan] --> Code regenerated.
[refactoringPlan] --> Refactoring plan execution finalized.
BUILD SUCCESSFUL
Total time: 11 seconds
```

The screenshot shows the Eclipse Console view with the title "Console". The output of the "antTask build.xml" command is displayed, showing the execution of the "execute" target which triggers the "refactoringPlan" task. The output indicates the task is executing "ExtractMethod", regenerating code, and finalizing the refactoring plan. The build is successful, and the total execution time is 11 seconds.

Ilustración 164: Ejecución tarea Ant RefactoringPlan desde Eclipse

Universidad de Burgos
ESCUELA POLITÉCNICA SUPERIOR
INGENIERÍA INFORMÁTICA



Anexo 6. Manual de Usuario

Clasificación de Refactorizaciones
Dynamic Refactoring Plugin 3.0

Alumnos:
Míryam Gómez San Martín
Íñigo Mediavilla Saiz

Tutor:
Raúl Marticorena Sánchez

Burgos, Julio de 2011

ANEXO VI
MANUAL DE USUARIO

ÍNDICE DE CONTENIDO

Anexo VI	
Manual de Usuario.....	455
Lista de cambios.....	463
1. INTRODUCCIÓN.....	465
1.1. Primeros pasos.....	465
2. MENÚ DYNAMIC REFACTORING.....	472
2.1. New Refactoring	473
2.2. Edit Refactoring.....	486
2.3. Delete Refactoring.....	491
2.4. Export Refactoring.....	493
3. VISTA: AVAILABLE REFACTORINGS.....	496
4. VISTA: REFACTORING CATALOG BROWSER.....	499
4.1. Refactoring Catalog Browser Toolbar.....	500
4.2. Classification Panel.....	503
4.2.1. Classification.....	504
4.2.2. Search.....	506
4.2.3. Visualization.....	511
4.2.4. Filters.....	513
4.3. Summary Panel.....	517
4.3.1. Overview.....	518
4.3.2. Inputs.....	521
4.3.3. Mechanism.....	522
4.3.4. Image.....	522
4.3.5. Examples.....	523
4.3.6. Source Viewer Dialog.....	524
5. EDITOR: CLASSIFICATIONS EDITOR.....	527
5.1. Classifications.....	529
5.2. Selected Classification.....	532
5.3. Categories.....	534
6. CONSULTAR AYUDA.....	538
7. CONFIGURACIÓN ADICIONAL.....	540
7.1. Catálogo de clasificaciones.....	540
7.2. El fichero XML de clasificaciones.....	540
7.3. Exportación de refactorizaciones.....	542

ÍNDICE DE ILUSTRACIONES

Ilustración 165: Eclipse con Dynamic Refactoring Plugin.....	466
Ilustración 166: Menú Dynamic Refactoring.....	467
Ilustración 167: Plugin - Vista Available Refactorings.....	467
Ilustración 168: Plugin - Vista Refactoring Catalog Browser.....	468
Ilustración 169: Plugin - Classifications Editor.....	469
Ilustración 170: Interfaz de selección de elementos del plugin a visualizar.....	470
Ilustración 171: Customize Perspective - Java.....	470
Ilustración 172: Window > Show View.....	471
Ilustración 173: Menú Dynamic Refactoring.....	472
Ilustración 174: Dynamic Refactoring - New Refactoring.....	473
Ilustración 175: New Refactoring - Step 1: Description.....	474
Ilustración 176: New Refactoring - Step 1: Keywords.....	474
Ilustración 177: New Refactoring - Step 1: Categories.....	475
Ilustración 178: New Refactoring - Step 2: Input configuration.....	476
Ilustración 179: New Refactoring - Step 2: Detalle resumen.....	477
Ilustración 180: New Refactoring - Step 2: Navegador.....	477
Ilustración 181: New Refactoring - Step 2: Detalle error.....	478
Ilustración 182: New Refactoring - Step 2: Búsqueda.....	481
Illustración 183: Step 3: Preconditions.....	482
Ilustración 184: New Refactoring - Step 3: Actions.....	483
Ilustración 185: New Refactoring - Step 3: Postconditions.....	484
Ilustración 186: New Refactoring - Step 3: Confirmation.....	485
Ilustración 187: Dynamic Refactoring - Edit Refactoring.....	486
Ilustración 188: Edit Refactoring – Dynamic Refactoring Selection.....	486
Ilustración 189: Edit Refactoring – Available refactorings.....	487
Ilustración 190: Edit Refactoring – Show only user's refactorings.....	488
Ilustración 191: Edit Refactoring – Botones.....	488
Ilustración 192: Edit Refactoring – Step 7: Confirmation.....	490
Ilustración 193: Dynamic Refactoring - Delete Refactoring.....	491
Ilustración 194: Delete Refactoring – Dynamic Refactoring Selection.....	491
Ilustración 195: Delete Refactoring – Confirmation Needed.....	492
Ilustración 196: Delete Refactoring – Dynamic Refactoring Deleted.....	492
Ilustración 197: Dynamic Refactoring - Export Refactoring.....	493
Ilustración 198: Export Refactorings.....	493
Ilustración 199: Export Refactorings - Available refactorings.....	494
Ilustración 200: Export Refactorings – Show only user's refactorings.....	494
Ilustración 201: Vista Available Refactorings - Selección código.....	496
Ilustración 202: Vista Available Refactorings.....	496
Ilustración 203: Vista Available Refactorings - Barra de herramientas.....	497
Ilustración 204: Vista Available Refactorings - Botón ref. plugin deshabilitado.....	497
Ilustración 205: Vista Available Refactorings - Botón ref. usuario deshabilitado.....	498
Ilustración 206: Vista Refactoring Catalog Browser.....	499
Ilustración 207: Vista Refactoring Catalog Browser - Barra de herramientas.....	500
Ilustración 208: Vista Refactoring Catalog Browser – Botón Editor de clasificaciones.	501
Ilustración 209: Vista Refactoring Catalog Browser – Botón Mostrar panel izquierdo..	502
Ilustración 210: Vista Refactoring Catalog Browser – Botón Mostrar panel derecho...502	
Ilustración 211: Vista Refactoring Catalog Browser – Botón Mostrar todos los paneles	503
Ilustración 212: Vista Refactoring Catalog Browser – Classification Panel.....	504

Ilustración 213: Vista Refactoring Catalog Browser – Classification.....	505
Ilustración 214: Vista Refactoring Catalog Browser – Desplegable Classification.....	505
Ilustración 215: Vista Refactoring Catalog Browser – Classification Description.....	505
Ilustración 216: Vista Refactoring Catalog Browser – Classification Scope.....	506
Ilustración 217: Vista Refactoring Catalog Browser – Search.....	506
Ilustración 218: Vista Refactoring Catalog Browser – Search Help.....	507
Ilustración 219: Vista Refactoring Catalog Browser – Search II.....	508
Ilustración 220: Vista Refactoring Catalog Browser – Search Error.....	509
Ilustración 221: Vista Refactoring Catalog Browser – Search Aviso I.....	509
Ilustración 222: Vista Refactoring Catalog Browser – Search Aviso II.....	510
Ilustración 223: Vista Refactoring Catalog Browser – Search Aviso III.....	510
Ilustración 224: Vista Refactoring Catalog Browser – Visualization.....	511
Ilustración 225: Vista Refactoring Catalog Browser – Representación refactorización.	512
Ilustración 226: Vista Refactoring Catalog Browser – Show filtered.....	513
Ilustración 227: Vista Refactoring Catalog Browser – Filters.....	513
Ilustración 228: Vista Refactoring Catalog Browser – Columna Selected.....	514
Ilustración 229: Vista Refactoring Catalog Browser – Columna Name Condition.....	514
Ilustración 230: Vista Refactoring Catalog Browser – Columna Clear.....	515
Ilustración 231: Vista Refactoring Catalog Browser – Clear All Conditions.....	515
Ilustración 232: Vista Refactoring Catalog Browser – Filter II.....	516
Ilustración 233: Vista Refactoring Catalog Browser – Summary Panel.....	517
Ilustración 234: Vista Refactoring Catalog Browser – Pestaña Overview.....	518
Ilustración 235: Vista Refactoring Catalog Browser – Description.....	518
Ilustración 236: Vista Refactoring Catalog Browser – Motivation.....	519
Ilustración 237: Vista Refactoring Catalog Browser – Categories.....	519
Ilustración 238: Vista Refactoring Catalog Browser – Categories Error.....	520
Ilustración 239: Vista Refactoring Catalog Browser – Keywords.....	520
Ilustración 240: Vista Refactoring Catalog Browser – Sin Keywords.....	520
Ilustración 241: Vista Refactoring Catalog Browser – Keywords Error.....	521
Ilustración 242: Vista Refactoring Catalog Browser – Pestaña Inputs.....	521
Ilustración 243: Vista Refactoring Catalog Browser – Pestaña Mechanism.....	522
Ilustración 244: Vista Refactoring Catalog Browser – Pestaña Image.....	523
Ilustración 245: Vista Refactoring Catalog Browser – Pestaña Examples.....	524
Ilustración 246: Vista Refactoring Catalog Browser – Source Viewer Dialog.....	525
Ilustración 247: Vista Refactoring Catalog Browser – Source Viewer Dialog Menú.....	525
Ilustración 248: Classifications Editor – Clasificación editable.....	527
Ilustración 249: Classifications Editor – Clasificación no editable.....	528
Ilustración 250: Classifications Editor - Sección Classifications.....	529
Ilustración 251: Classifications Editor - Botones sección Classifications.....	529
Ilustración 252: Classifications Editor - Add Classification.....	530
Ilustración 253: Classifications Editor - Add Classification Error.....	530
Ilustración 254: Classifications Editor - Delete Classification.....	531
Ilustración 255: Classifications Editor - Rename Classification.....	531
Ilustración 256: Classifications Editor - Rename Classification Error.....	532
Ilustración 257: Classifications Editor - Sección Selected Classification.....	532
Ilustración 258: Classifications Editor - Name.....	533
Ilustración 259: Classifications Editor - Description.....	533
Ilustración 260: Classifications Editor - Modify Description.....	534
Ilustración 261: Classifications Editor - Sección Categories.....	534
Ilustración 262: Classifications Editor - Botones sección Categories.....	535
Ilustración 263: Classifications Editor - Add Category.....	535
Ilustración 264: Classifications Editor - Add Category Error.....	536

Ilustración 265: Classifications Editor - Delete Category.....	536
Ilustración 266: Classifications Editor - Rename Category.....	537
Ilustración 267: Classifications Editor - Rename Category Error.....	537
Ilustración 268: Help > Help Contents.....	538
Ilustración 269: Help - Dynamic Refactoring Plugin.....	539
Ilustración 270: DTD de clasificaciones.....	541

LISTA DE CAMBIOS

Número	Fecha	Descripción	Autor/es
0	30/04/11	Introducción y configuración adicional.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
1	29/05/11	Incluido manual de usuario del menú <i>Dynamic Refactoring</i> , editor <i>Classifications Editor</i> y el apartado referente a consultar la ayuda.	Míryam Gómez San Martín Íñigo Mediavilla Saiz
2	31/05/11	Incluido manual de usuario de la vista <i>Available Refactorings</i> .	Míryam Gómez San Martín Íñigo Mediavilla Saiz
3	04/06/11	Incluido manual de usuario de la vista <i>Refactoring Catalog Browser</i> y modificado apartado destinado a la configuración adicional.	Míryam Gómez San Martín Íñigo Mediavilla Saiz

1. INTRODUCCIÓN

El objetivo que se quiere cumplir con el desarrollo de este anexo es servir de guía al usuario final para la correcta utilización del plugin de refactorizaciones dinámicas para Eclipse. Con ese fin, primeramente se realizará un paseo por cada una de sus partes para posteriormente entrar en detalle en las mismas, además irá acompañado todo ello de explicaciones junto con ilustraciones de cada opción disponible con el fin de facilitar al usuario su comprensión.

En este manual, exclusivamente se incluyen aquellas funcionalidades de nueva creación que han supuesto un cambio en la interfaz gráfica y aquellas que han sido modificadas para su mejora, por ello si usted se considera un usuario principiante le recomendamos la lectura previa de los manuales de usuario de versiones anteriores del plugin [Fuente & Herrero n.d.] y [Fuente de la Fuente n.d.], con el objetivo de familiarizarse con el entorno del mismo.

1.1. Primeros pasos

La aplicación arranca cuando el usuario ejecuta el entorno de desarrollo Eclipse, es decir, cuando hace doble clic sobre el ícono `eclipse.exe`, ya que este se encarga de la carga de los plugins que se encuentran instalados. Por lo que, previamente se habrá tenido que realizar la instalación del plugin siguiendo alguno de los dos procedimientos detallados en el *Anexo 5 – Manual de instalación* para tal efecto.

El plugin se encuentra internacionalizado por lo que esta disponible en los idiomas inglés y español. Por defecto, Eclipse toma el idioma de la configuración del sistema operativo. Si se quiere utilizar otro idioma bastará con arrancar Eclipse con el parámetro `-nl` y la opción de idioma correspondiente. Por ejemplo para arrancar Eclipse y el plugin en inglés se debe ejecutar desde línea de comandos:

```
eclipse.exe -nl en
```

Cuando nos encontramos en la interfaz de Eclipse con la perspectiva de Java, *Java perspective*, abierta y activa podemos observar el elemento del menú de herramientas correspondiente al plugin, *Dynamic Refactoring*, y las vistas que se muestran por defecto, que son: *Refactoring History*, *Available Refactorings*, *Refactoring Progress*, *Refactoring Catalog Browser*. Inicialmente el editor de clasificaciones, *Classifications Editor*, se

encuentra oculto.

A continuación veremos una ilustración de la interfaz, para pasar a ver cada parte de forma general, indicando en que zonas de las mismas nos centraremos más adelante.

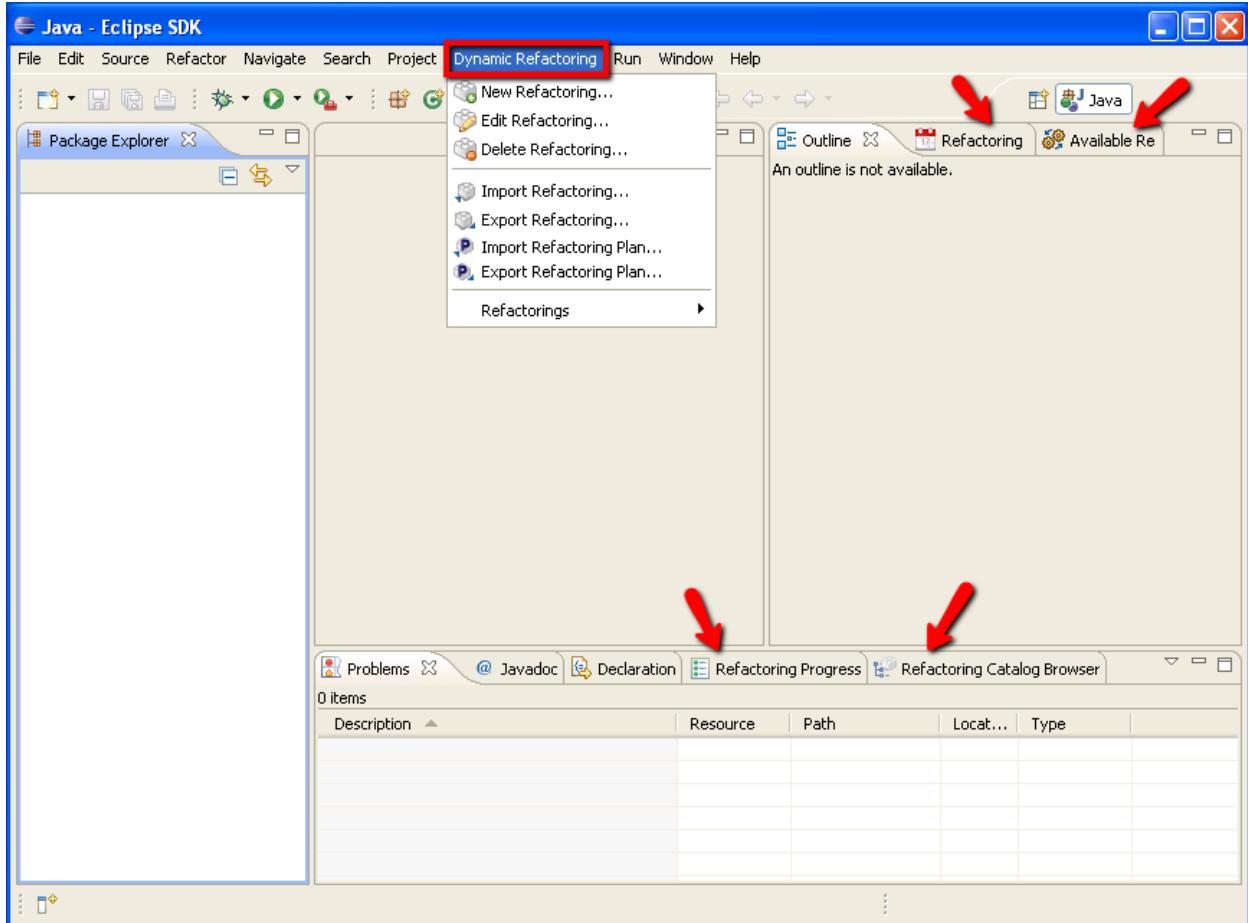


Ilustración 165: Eclipse con Dynamic Refactoring Plugin

Las vistas *Refactoring History* y *Refactoring Progress* no han sufrido modificaciones por lo que no serán objeto de estudio en el presente manual de usuario. Veamos las que si lo son:

Dynamic Refactoring: Este menú ofrece acceso a la gestión de refactorizaciones, a su ejecución, a su importación/exportación y a la importación/exportación de un plan de refactorizaciones. En esta parte, nos centraremos en las funcionalidades que han sido modificadas, estas son: *New Refactoring...* , *Edit Refactoring...* , *Delete Refactoring...* y *Export Refactoring...* .

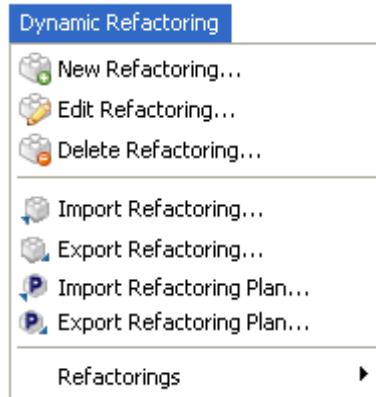


Ilustración 166: Menú Dynamic Refactoring

Available Refactorings: esta vista muestra las refactorizaciones disponibles que pueden ser ejecutadas tomando como entrada principal el elemento que este siendo seleccionado por el usuario en el editor de código fuente. En este caso, detallaremos las nuevas funcionalidades incorporadas.

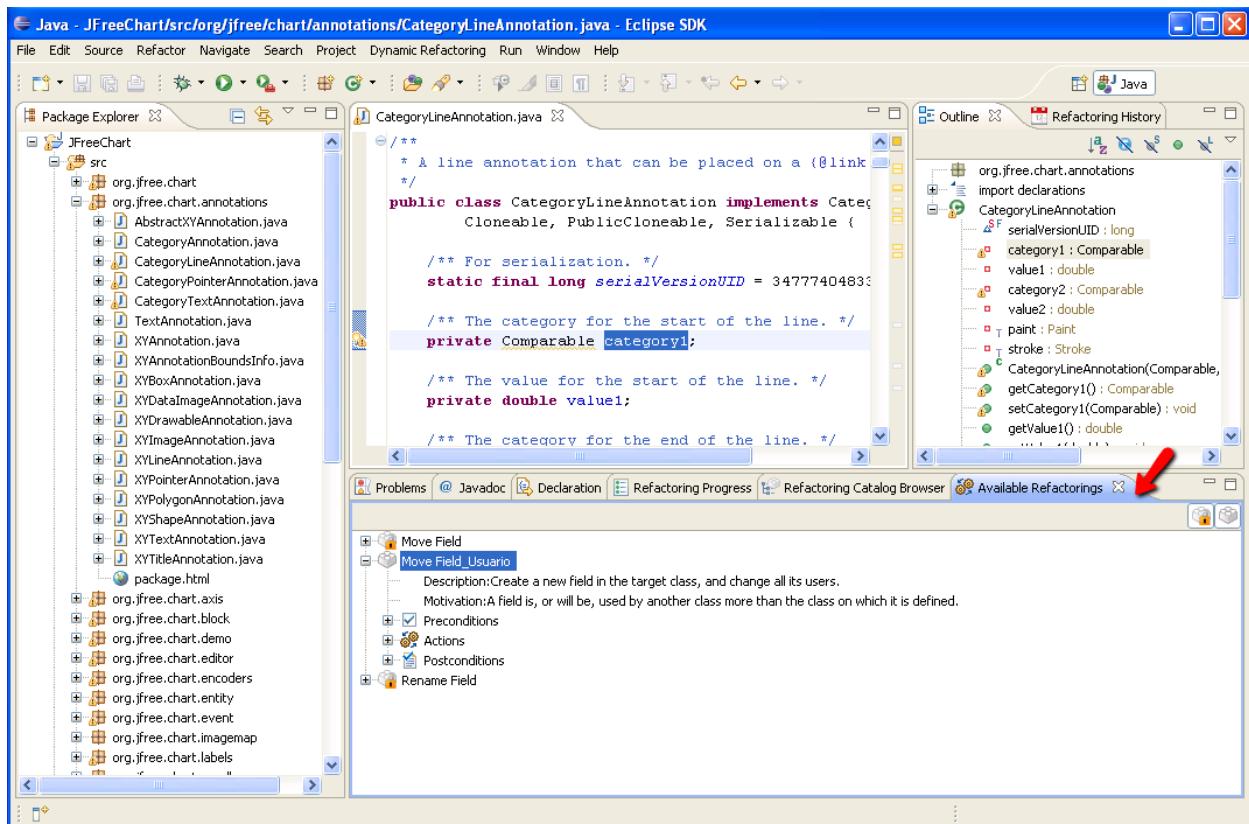


Ilustración 167: Plugin - Vista Available Refactorings

Refactoring Catalog Browser: vista que muestra el catálogo de refactorizaciones, recoge toda la información asociada a estas, permite ver las clasificaciones de las refactorizaciones y realizar búsquedas sobre estas. En este caso, al tratarse de una vista nueva se realizará un detalle al completo de ella.

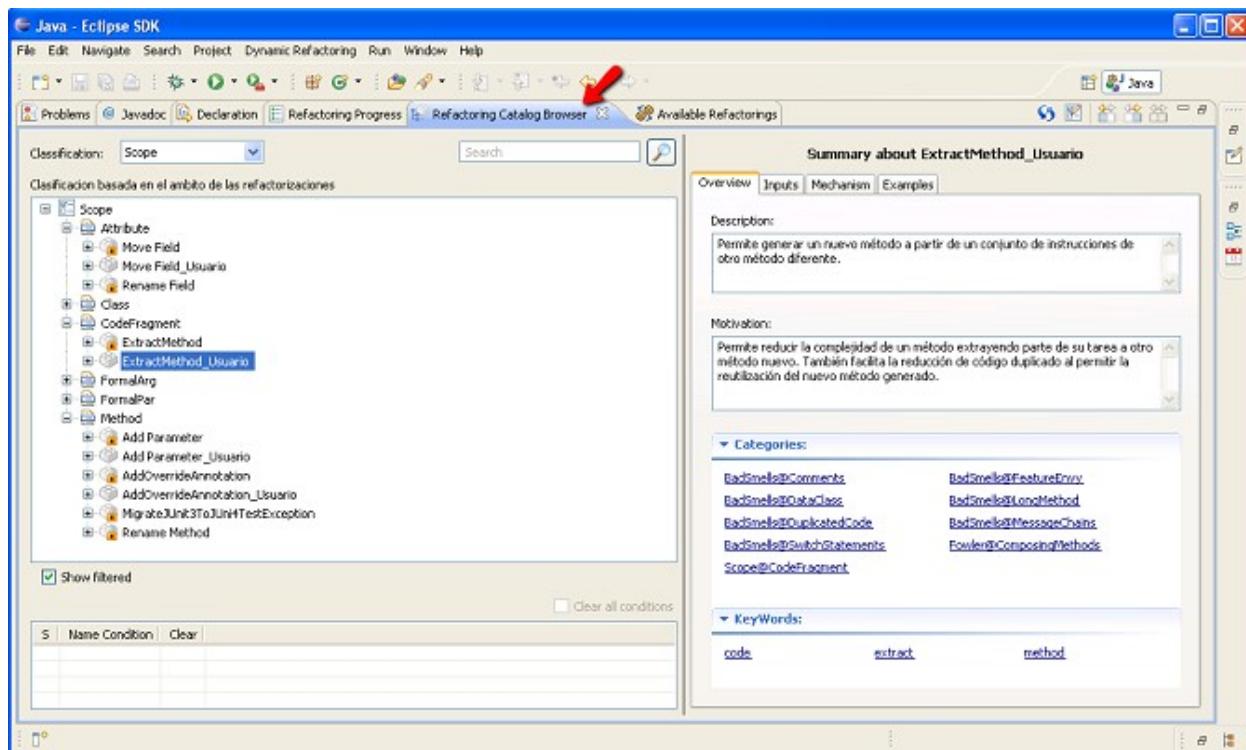


Ilustración 168: Plugin - Vista Refactoring Catalog Browser

Classifications Editor: editor que permite al usuario crear y modificar sus propias clasificaciones, así como definir las categorías que tendrá disponible cada una de ellas.

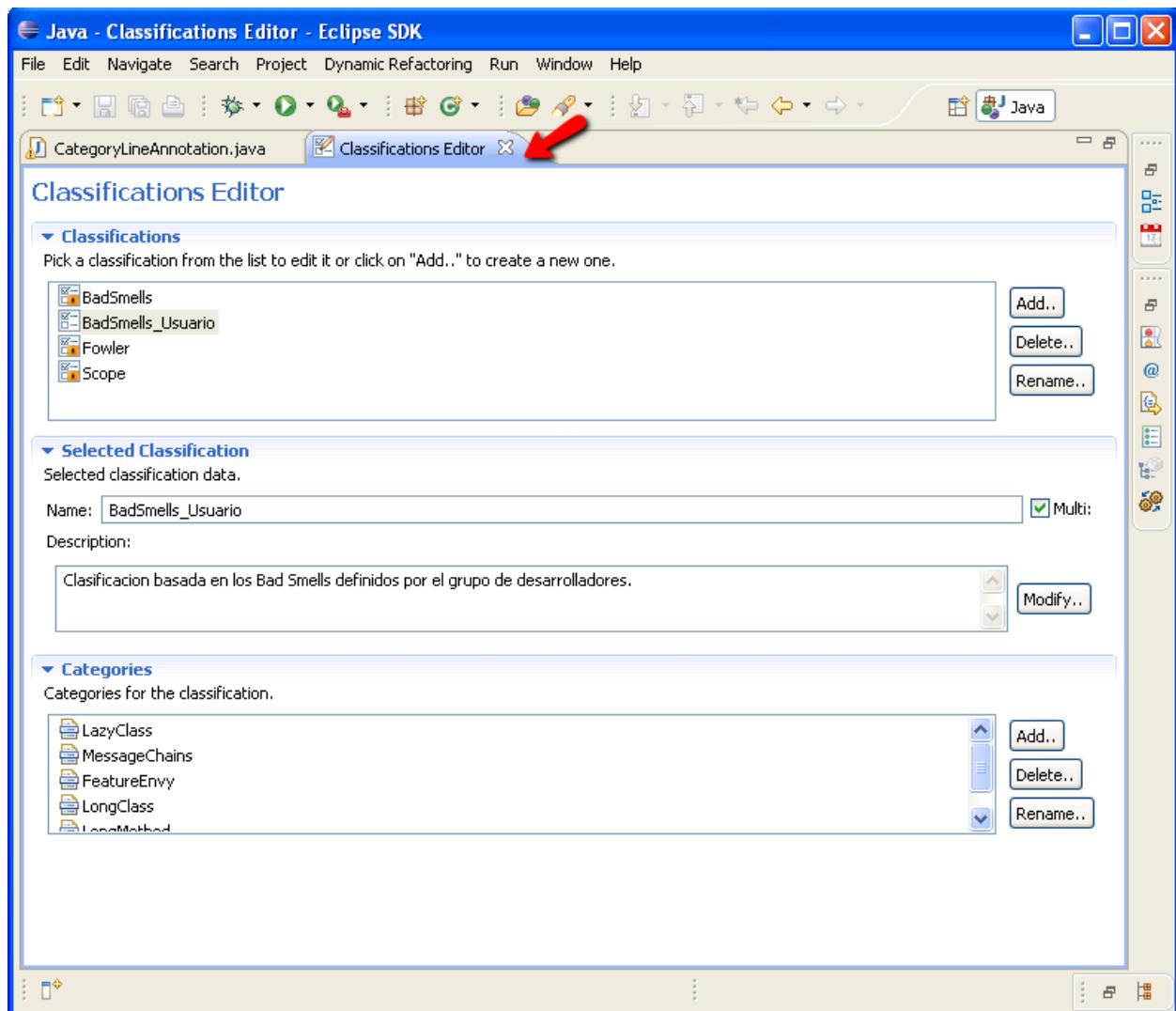


Ilustración 169: Plugin - Classifications Editor

Una vez visto cada uno de los elementos de la interfaz del plugin de forma general pasaremos a entrar en detalle en cada uno de ellos. Antes de esto, vamos a realizar una pequeña aclaración de cómo el usuario puede hacer visibles estos elementos y cómo puede ocultarlos.

El usuario podrá elegir qué elementos de la interfaz del plugin de refactorizaciones quiere visualizar, para ocultarlos bastará con cerrar el propio elemento. Para visualizarlos, seleccionaremos *Window* en el menú de herramientas y la opción *Show View > Other*. Nos dirigiremos a la entrada *Dynamic Refactoring* de la ventana que nos aparecerá y en ella seleccionaremos los elementos de la interfaz del plugin que deseemos visualizar. A continuación se muestra dicha ventana.

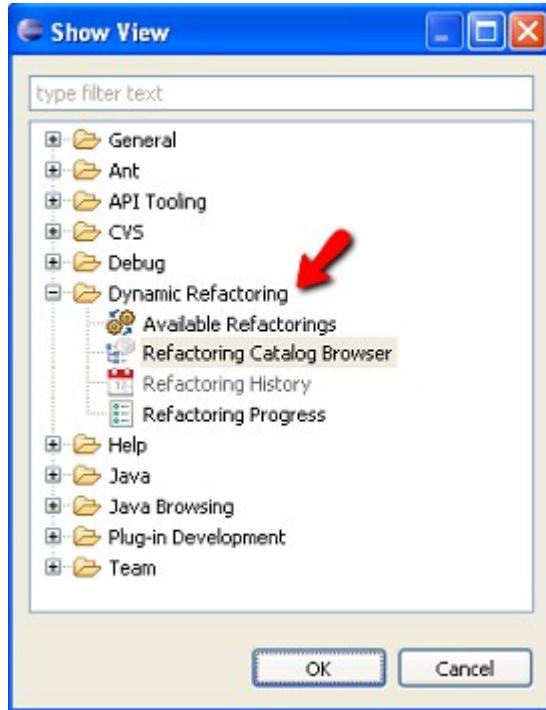


Ilustración 170: Interfaz de selección de elementos del plugin a visualizar

Además, si se desea agregar entradas para alguno de los elementos de la interfaz del plugin al menú *Window > Show View* para hacer mas sencillo el acceso a estos, nos dirigiremos a *Window > Customize Perspective...* y en la pestaña *Shortcuts* del dialogo que aparece, elegimos en el desplegable *Show View*, seleccionamos *Dynamic*

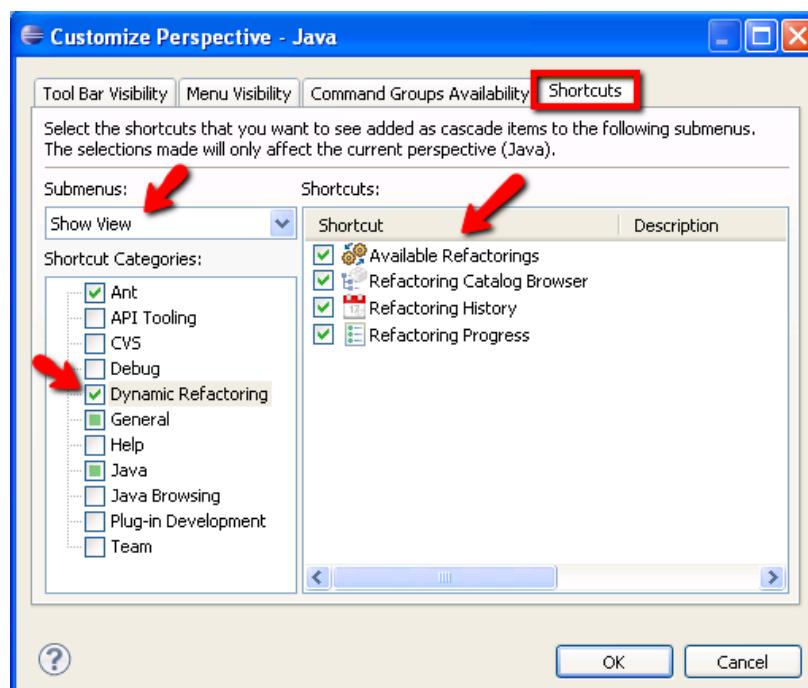


Ilustración 171: Customize Perspective - Java

Refactoring y elegiremos en la lista aquellos para los cuales se quiera agregar entradas.

Una vez seleccionados los elementos y aceptado el dialogo, comprobaremos que las entradas han sido creadas yendo al menú *Window > Show View* y viendo que estos aparece automáticamente.

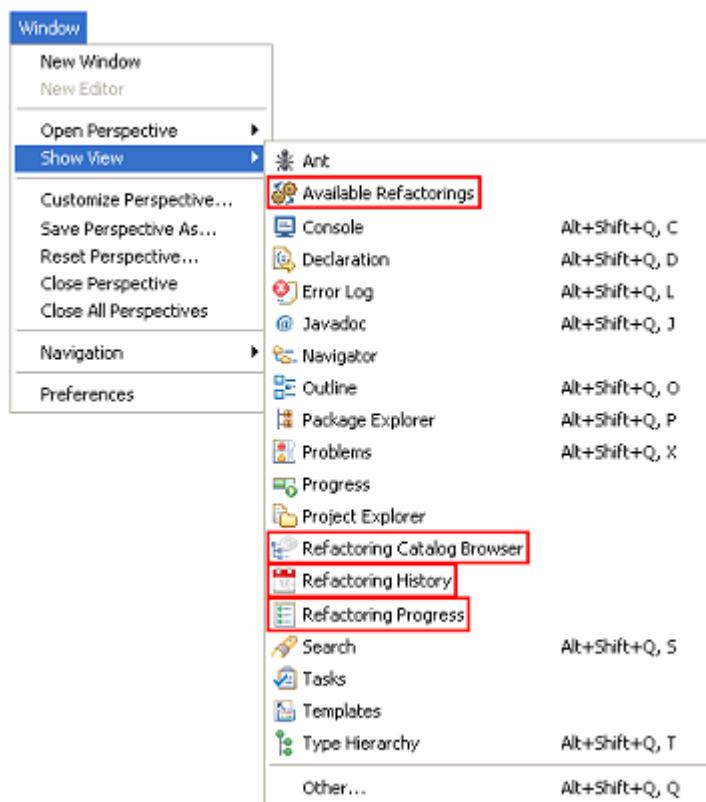


Ilustración 172: Window > Show View

2. MENÚ DYNAMIC REFACTORING

Este menú ofrece acceso a gran parte de la funcionalidad con la que el plugin cuenta. Se puede dividir en tres partes diferenciadas que son :

Gestión de refactorizaciones

- New Refactoring...*
- Edit Refactoring...*
- Delete Refactoring...*

Exportación/Importación

- Import Refactoring...*
- Export Refactoring...*
- Import Refactoring Plan...*
- Export Refactoring Plan...*

Refactorings

En este apartado nos centraremos en detallar exclusivamente aquellas funcionalidades de nueva creación que han supuesto un cambio en la interfaz gráfica y aquellas que han sido modificadas para su mejora. Nos centraremos por tanto en las opciones siguientes que a continuación pasaremos a explicar: *New Refactoring...*, *Edit Refactoring...*, *Delete Refactoring...* y *Export Refactoring...* .

A continuación se muestra el menú *Dynamic Refactoring*:

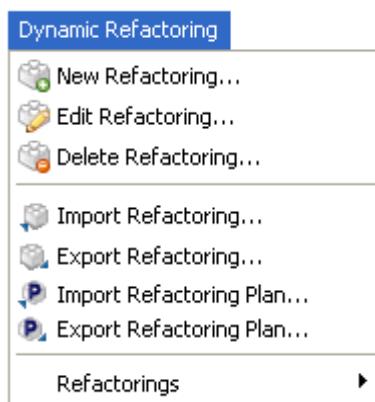


Ilustración 173: Menú Dynamic Refactoring

2.1. New Refactoring

La primera funcionalidad referente a la gestión de refactorizaciones que ofrece el menú *Dynamic Refactoring* es la creación de refactorizaciones mediante la opción *New Refactoring...* que aparece en el mismo.

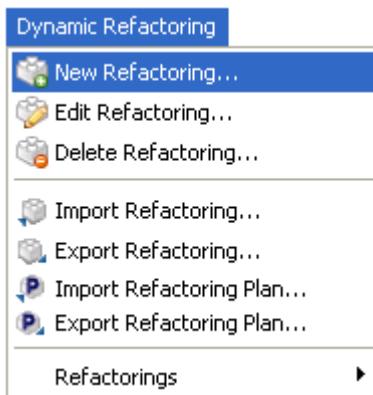


Ilustración 174: Dynamic Refactoring - New Refactoring

Nos dirigiremos a *Dynamic Refactoring* > *New Refactoring* con el fin de que de comienzo el asistente que nos guiará en el proceso de creación de una refactorización. Este asistente está formado por siete pasos. En cada paso se puede cancelar la creación, ir al paso anterior o ir al paso siguiente (una vez que todos los campos necesarios se hayan completado).

A continuación detallaremos los cambios que ha sufrido el asistente por lo que si necesitase aclaraciones respecto a partes de la interfaz que no han sufrido cambios y por tanto no se indican en el presente manual le remitimos a los manuales de usuario de versiones anteriores del plugin ([Fuente & Herrero n.d.] y [Fuente de la Fuente n.d.]).

PASO 1 – Descripción de la refactorización

En este primer paso se debe rellenar la información general de la refactorización, que corresponde con: nombre, descripción, imagen, motivación, palabras clave y categorías. Nos vamos a centrar en estas dos últimas ya que han sido incorporadas en esta versión.

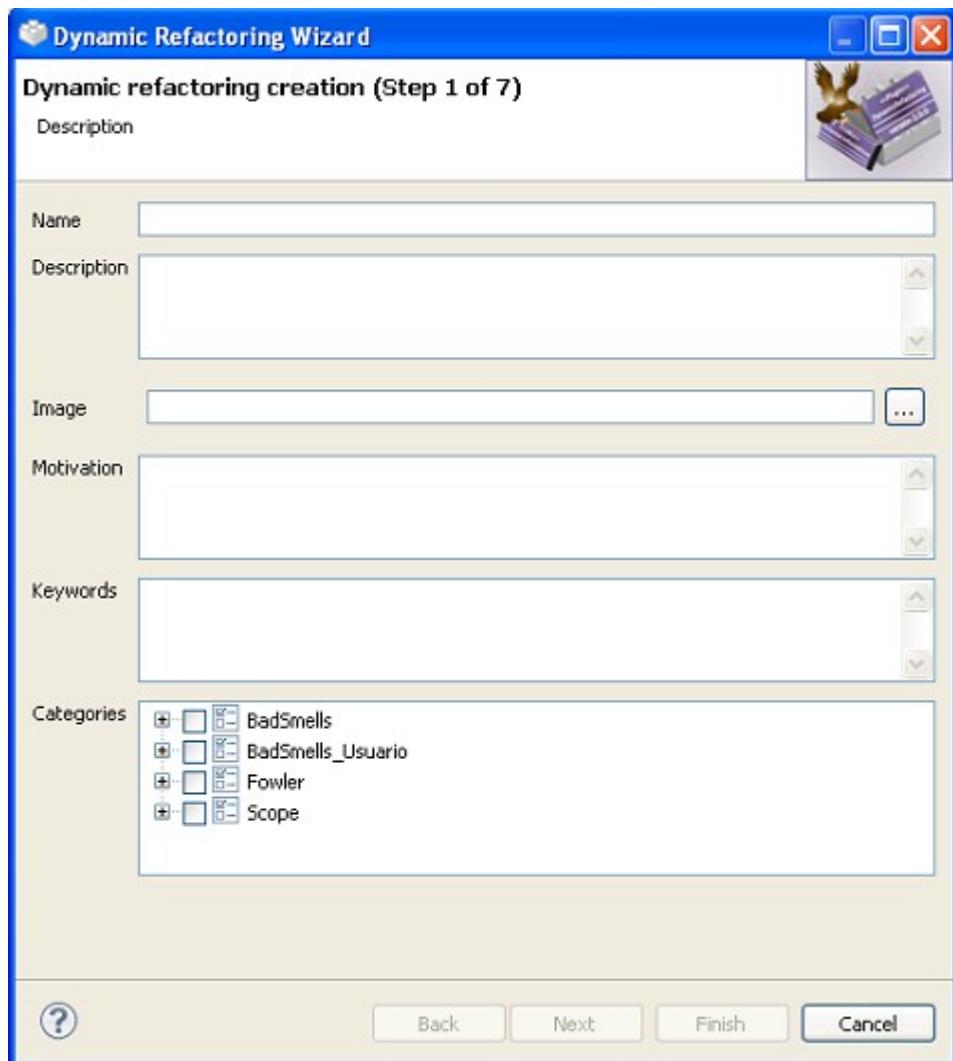


Ilustración 175: New Refactoring - Step 1: Description

Keywords

Permite indicar las palabras clave que se quieren asociar a la refactorización, el carácter de separación entre una y otra será ',' (carácter coma).

Se trata de un campo opcional.

Keywords	method, prueba, usuario Establish a set of KeyWords to identify the refactoring.
----------	---

Ilustración 176: New Refactoring - Step 1: Keywords

Categories

Permite definir las categorías a las que pertenecerá la refactorización para cada una de las clasificaciones que hay disponibles. Será opcional definir las categorías para todas las clasificaciones salvo para la clasificación *Scope*, ya que deberá coincidir con el ámbito para el cual se definirá la refactorización.

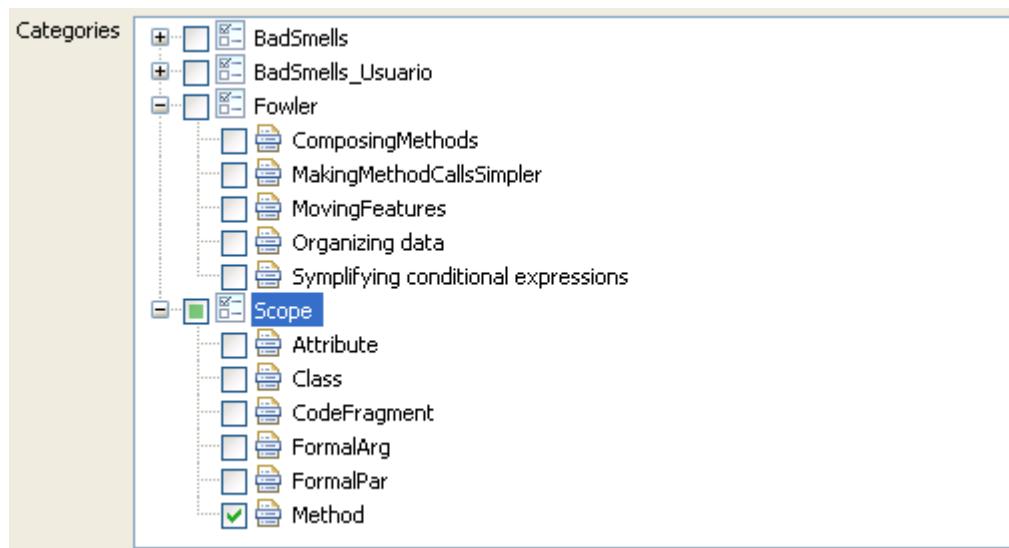


Ilustración 177: New Refactoring - Step 1: Categories

PASO 2 – Entradas de la refactorización

En el segundo paso se configurarán las entradas de la refactorización, para cada entrada se definirá los campos: *name*, *main*, *from*, *method*. La lista de la derecha *Inputs* muestra las entradas que ya han sido seleccionadas para la refactorización y la que se encuentra a la izquierda, *Types*, la lista de tipos disponibles.

Cada vez que un tipo es seleccionado dentro de la lista de izquierda *Types* se actualiza el navegador `html` de la parte inferior con la información del `javadoc` asociado a dicho tipo. En caso de no encontrarse disponible el fichero javadoc de la misma se muestra una página `html` que indica que la información solicitada no se encuentra disponible. Esto permite ayudar al usuario a conocer la funcionalidad de cada uno de los tipos disponibles.

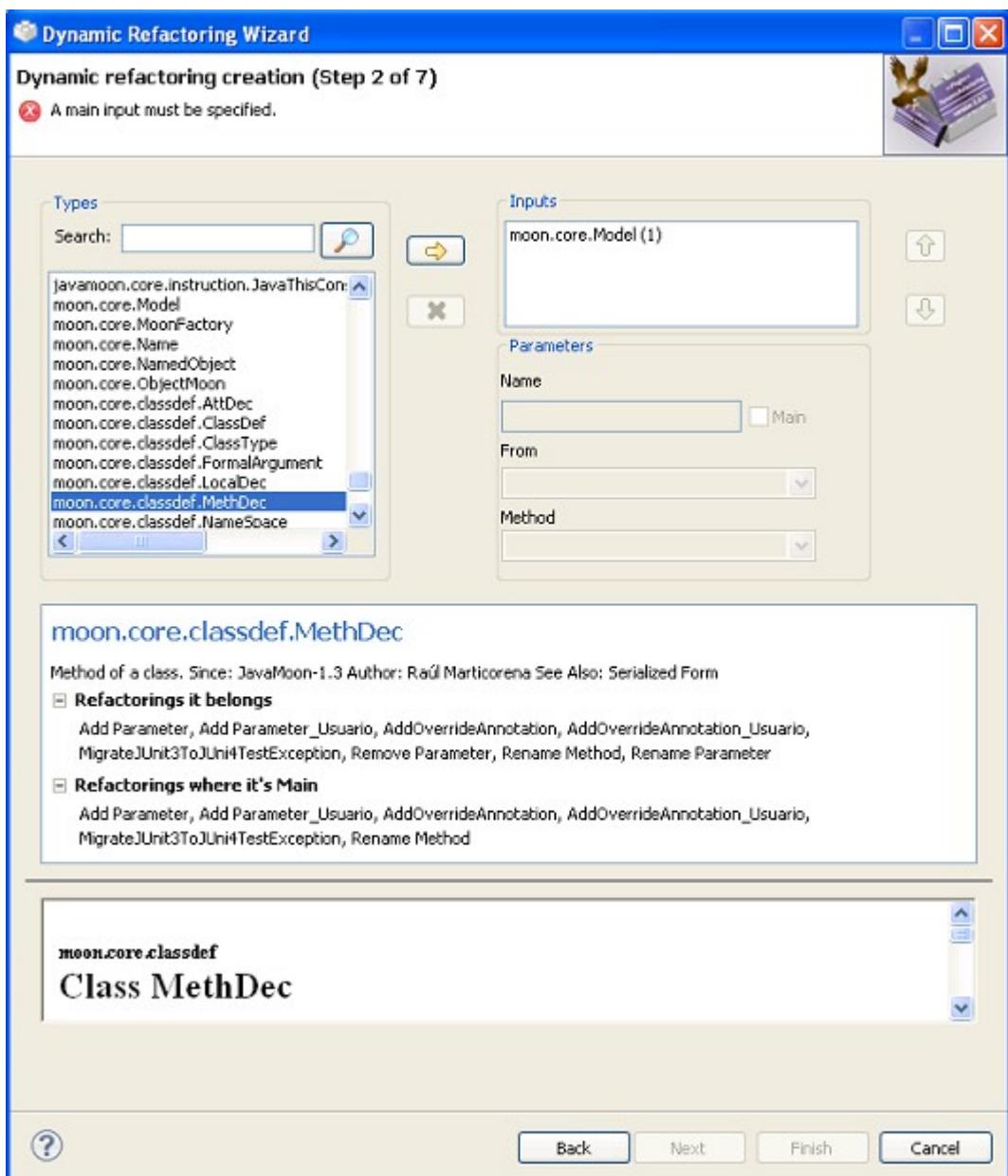


Ilustración 178: New Refactoring - Step 2: Input configuration

Además, como se puede apreciar en la ilustración anterior, aparece una nueva zona en la interfaz en la que se muestra un resumen del tipo de entrada seleccionada. En este resumen se muestra la descripción del tipo de entrada y la relación de las refactorizaciones que la están utilizando en su definición (**Refactorings it belongs**) así como las que la tienen como entrada principal (**Refactorings where it's Main**). A su vez, si situamos el ratón encima de alguna de estas refactorizaciones aparecerá un *tooltip* que recoge su información básica. En la siguiente ilustración se puede apreciar en detalle.

moon.core.classdef.MethDec

Method of a class. Since: JavaMoon-1.3 Author: Raúl Marticorena See Also: Serialized Form

Refactorings it belongs

Add Parameter, Add Parameter_Usuario, AddOverrideAnnotation, AddOverrideAnnotation_Usuario, MigrateJUnit3ToJUnit4TestException, Remove Parameter, Rename Method, Rename Parameter

Refactorings where it's Main

Add Parameter, Add Parameter_Usuario, AddOverrideAnnotation, AddOverrideAnnotation_Usuario, MigrateJUnit3ToJUnit4TestException, Rename Method, Remove Parameter

Remove Parameter

Description: Remove a given parameter from a method's signature.

Motivation: A parameter is no longer used by the method body.

Ilustración 179: New Refactoring - Step 2: Detalle resumen

Además, el navegador html se puede ampliar desplazando la parte superior del mismo hacia arriba teniendo presionado el botón izquierdo del ratón, de esta forma el navegador consigue un tamaño mayor permitiendo mejorar su visualización.

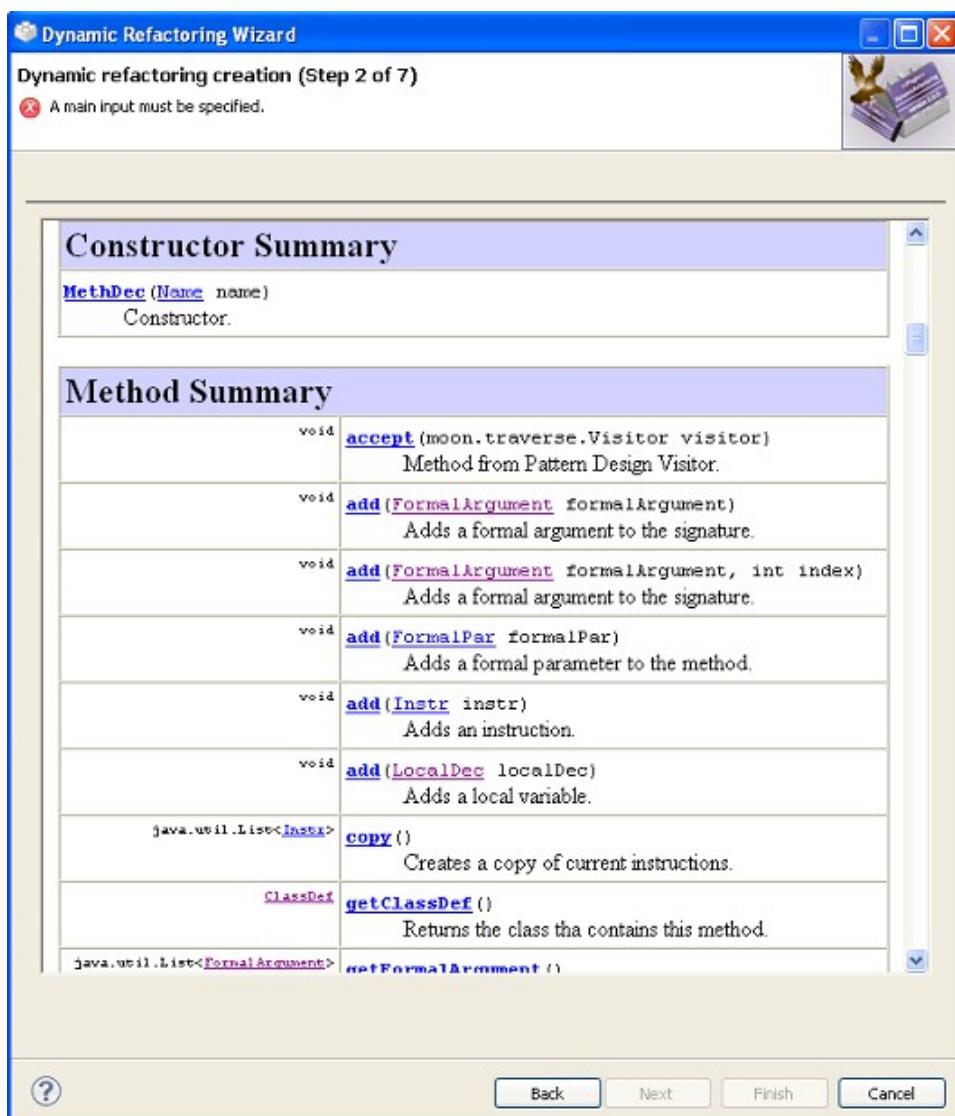


Ilustración 180: New Refactoring - Step 2: Navegador

Existen ciertas **restricciones** que si no se cumplen impedirán pasar al siguiente paso en el asistente de creación de refactorizaciones. Estas son las siguientes:

- ◆ Se debe especificar una entrada principal.
- ◆ No puede haber más de una entrada principal.
- ◆ La entrada principal debe ser una de los siguientes tipos:
 - moon.core.classdef.ClassDef
 - moon.core.classdef.MethDec
 - moon.core.classdef.AttDec
 - moon.core.classdef.FormalArgument
 - moon.core.genericity.FormalPar
 - moon.core.instruction.CodeFrgment
- ◆ Un subtipo de los anteriores.
- ◆ El tipo de la entrada principal debe concordar con el ámbito establecido para la refactorización en la primera página del asistente, es decir, con la categoría seleccionada para la clasificación *Scope*.
- ◆ Todas las entradas tienen que tener un nombre único, excepto si se trata de una instancia de *CodeFrgment* o un subtipo de la misma.
- ◆ Si se especifica un valor para el campo *From* de una entrada, se debe especificar también un valor para el campo *Method*.

En caso de que se produzca alguno de estos errores el asistente no nos permitirá avanzar al siguiente paso y nos indicará del error en la zona destinada a tal efecto, es decir, en la parte superior del asistente con un aspa roja, como se puede observar en la siguiente ilustración.



Ilustración 181: New Refactoring - Step 2: Detalle error

Por último, también se dispone de un campo de texto en el que poder realizar búsquedas sobre la lista de tipos, *Types*. Para realizar la búsqueda basta con introducir el filtro deseado en el campo de texto y pulsar el botón para que se produzca la misma, representado con una lupa.

El proceso de búsqueda ha cambiado totalmente respecto de la versión anterior del plugin por lo que a continuación se explica en detalle.

Las búsquedas se realizan sobre las descripciones de cada tipo, las cuales se encuentran disponibles en su correspondiente javadoc, basándose en la raíz de las palabras y obviando aquellas carentes de significado. Por lo tanto, en el listado de tipos de entrada disponibles originalmente aparecen todas ordenadas de forma alfabéticamente conforme al nombre (simple o cualificado) que se este mostrando. Al realizar una búsqueda se mostrarán las diez mejores concordancias obtenidas, en caso de haberlas, las cuales estarán ordenadas por relevancia.

Como acabamos de comentar, la búsqueda se realiza por la raíz de las palabras pero en caso de querer realizar búsquedas más complejas se dispone de un lenguaje de consulta específico para realizar estas. Este lenguaje lo provee Lucene ([Apache Lucene n.d.]) y a continuación detallaremos lo más destacado. No obstante se puede ampliar información en ([Query Parser Syntax n.d.] http://lucene.apache.org/java/3_0_1/queryparsersyntax.html)

Lenguaje de consulta Lucene

Una consulta se divide en términos y operadores. Hay dos tipos de términos; los términos simples y las frases. Un solo término es una palabra mientras que una frase es un grupo de palabras entre comillas dobles. Los términos se pueden combinar con operadores booleanos para formar una consulta más compleja.

Además Lucene realiza la modificación de términos de consulta para proporcionar una gama más amplia de opciones de búsqueda.

Búsquedas con comodín

Soporta búsquedas de caracteres comodín simples y múltiples dentro de consultas de términos simples, no en frases. Los caracteres comodín pueden ir al final o entre medias del término y son los siguientes:

- *: El carácter anterior 0 o más veces.
- ??: El carácter anterior una vez como máximo, es decir, carácter opcional.

Búsquedas aproximadas

Soporta búsquedas aproximadas basadas en la distancia de Levenshtein, distancia de edición, o distancia entre palabras ([Levenshtein n.d.]), para realizar búsquedas similares al término introducido. Para ello, se hace uso del carácter '~' y se pone al final del término introducido.

Búsquedas haciendo boosting del término

Permite realizar búsquedas dando importancia a términos, es decir, si desea destacar como más relevante a un término de la búsqueda lo indicará poniendo tras él el carácter '^' seguido de un numero que indicará el factor de relevancia.

Búsqueda con operadores booleanos

Permiten combinar condiciones mediante operadores lógicos. Los operadores booleanos soportados son los siguientes y *siempre se deben indicar en mayúsculas*: "AND", '+', "OR", "NOT" y '-'. El operador "OR" es el operador por defecto, esto quiere decir que si no hay operador booleano entre dos términos este será el utilizado.

- "OR" : concuerda si cumple con alguno de los términos. Equivalente a ||.
- "AND": concuerda si cumple con ambos términos. Equivale a '+'.
- "NOT": no se puede utilizar con un único término y su significado es la exclusión de aquellos que concuerdan con el término que aparece después de este operador. Equivale a '!' y '-'.

Agrupación

Admite el uso de paréntesis para agrupar clausurás para formar subconsultas, muy útil para el control de la lógica booleana.

Caracteres especiales

Soporta caracteres especiales que forman parte de la sintaxis de consulta. La lista actual de los caracteres especiales es la siguiente:

+ - && || ! () { } [] ^ " ~ * ? : \

Para escapar estos caracteres especiales se deberá utilizar el carácter '\' delante de ellos.

En la siguiente ilustración se puede observar el resultado obtenido de realizar una búsqueda con el uso de operadores booleanos:

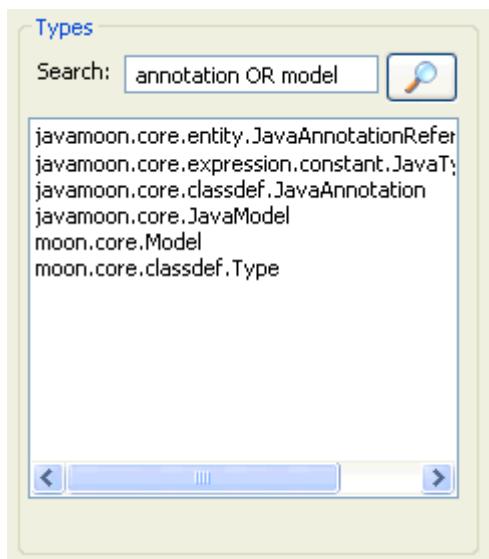


Ilustración 182: New Refactoring - Step 2: Búsqueda

Una vez realizada una búsqueda, en caso de que se desee eliminar esta y por tanto que se muestren todos los elementos disponibles bastará con realizar una búsqueda vacía o por el carácter '*' (carácter asterisco) para conseguir el efecto deseado.

PASO 3 – Precondiciones de la refactorización

En el tercer paso se deberán seleccionar las precondiciones elegidas para que formen parte de la definición de la refactorización.

La lista de la derecha muestra las precondiciones que ya han sido seleccionadas para la refactorización y la que se encuentra a la izquierda, *Preconditions*, la lista de todas las precondiciones disponibles.

Al igual que en el paso anterior, cada vez que una precondición es seleccionada dentro de la lista de izquierda *Preconditions* se actualiza el navegador `html` de la parte inferior con la información del `javadoc` asociada.

También dispone de un campo de búsqueda que facilita al usuario la localización de una precondición determinada, su funcionamiento es análogo al visto anteriormente para el

caso de los tipos. El checkbox *Qualified* ofrece la posibilidad de mostrar los nombres cualificados de las diferentes precondiciones, permitiendo diferenciar entre precondiciones propias de Java o independientes del lenguaje.

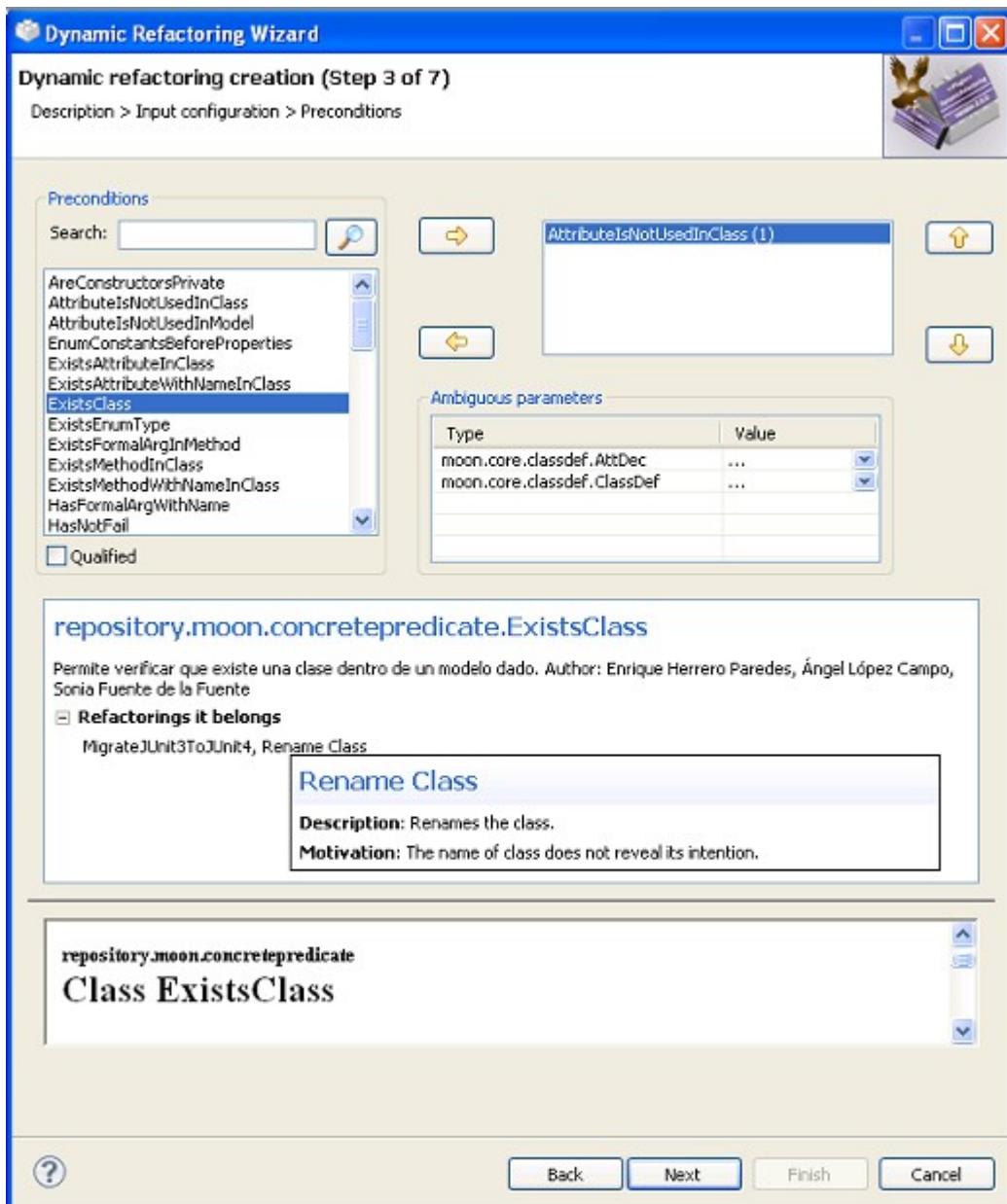


Illustration 183: Step 3: Preconditions

PASO 4 – Acciones de la refactorización

La interfaz correspondiente a este cuarto paso es idéntica a la del paso anterior diferenciándose únicamente en que en este paso se está trabajando con acciones en vez de precondiciones. Por tanto, la forma de utilización es similar a la del paso anterior.

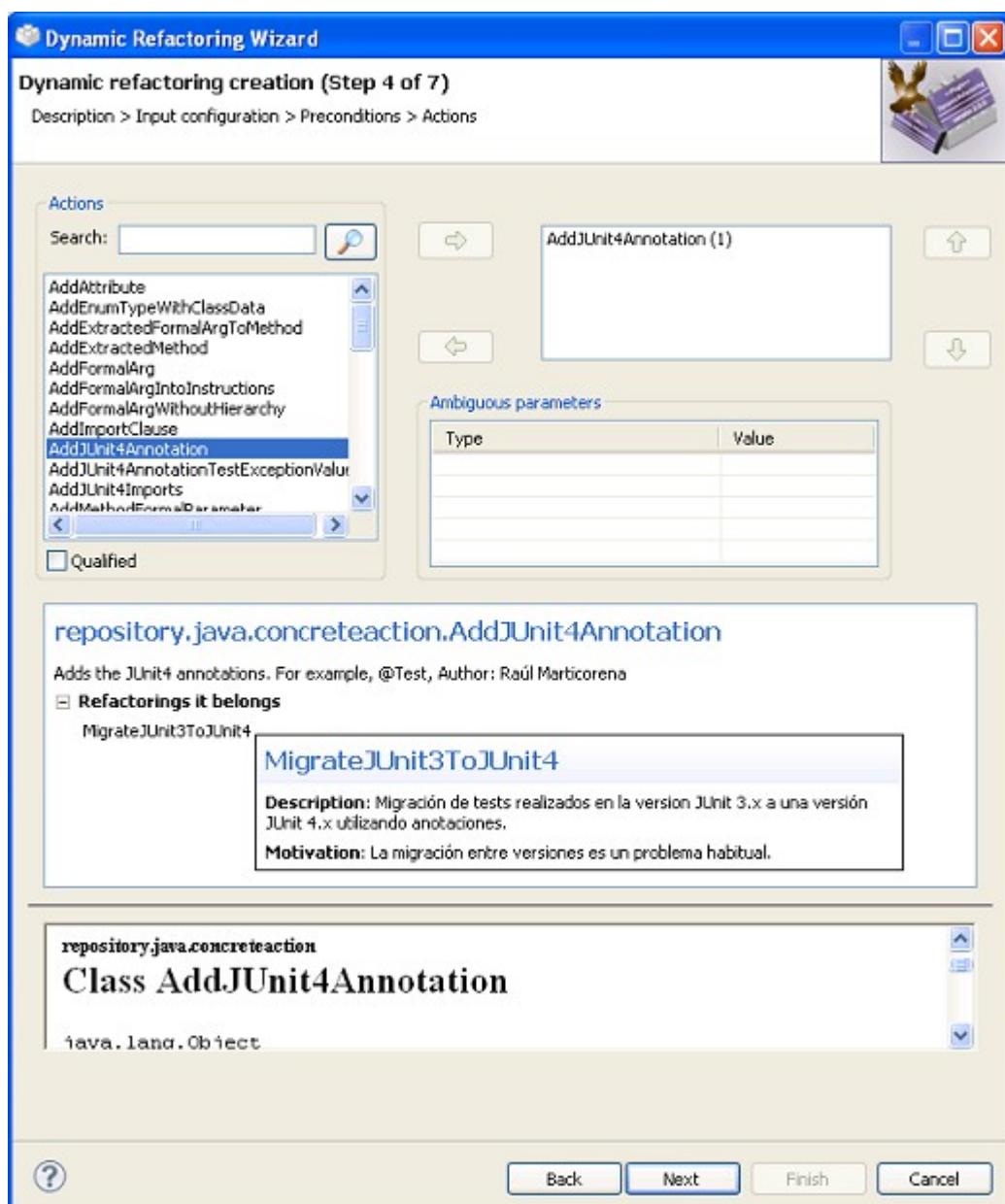


Ilustración 184: New Refactoring - Step 3: Actions

PASO 5 – Postcondiciones de la refactorización

De la misma manera que en el paso anterior, la interfaz correspondiente al quinto paso es idéntica a la del tercer paso pero en este caso se seleccionan las postcondiciones en vez de las precondiciones.

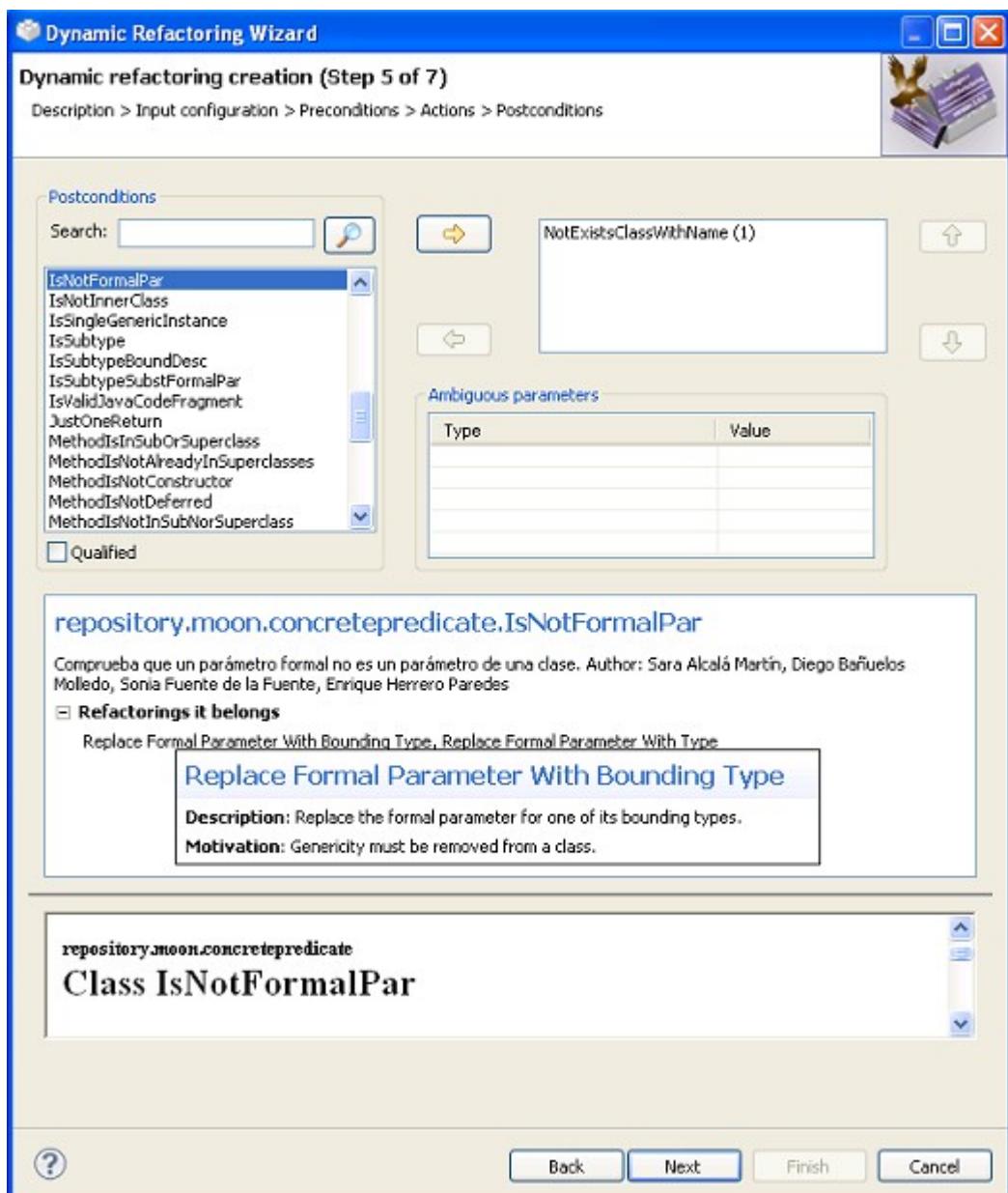


Ilustración 185: New Refactoring - Step 3: Postconditions

PASO 6 – Ejemplos de la refactorización

La sexta página del asistente permite asociar dos ejemplos a la definición de la refactorización. Este no es un paso obligatorio por lo que no es necesario incluir ejemplos. La interfaz correspondiente a este paso no ha sido modificada, respecto de la versión anterior del plugin, por lo que no se va a detallar.

PASO 7 – Resumen de la refactorización

En el último paso se muestra un resumen informativo de la definición que se ha ido construyendo para la refactorización a lo largo del asistente, con todos los datos indicados por el usuario en las páginas anteriores. Una vez que se haya revisado la configuración de la refactorización, se puede guardar pulsando el botón *Finish*.

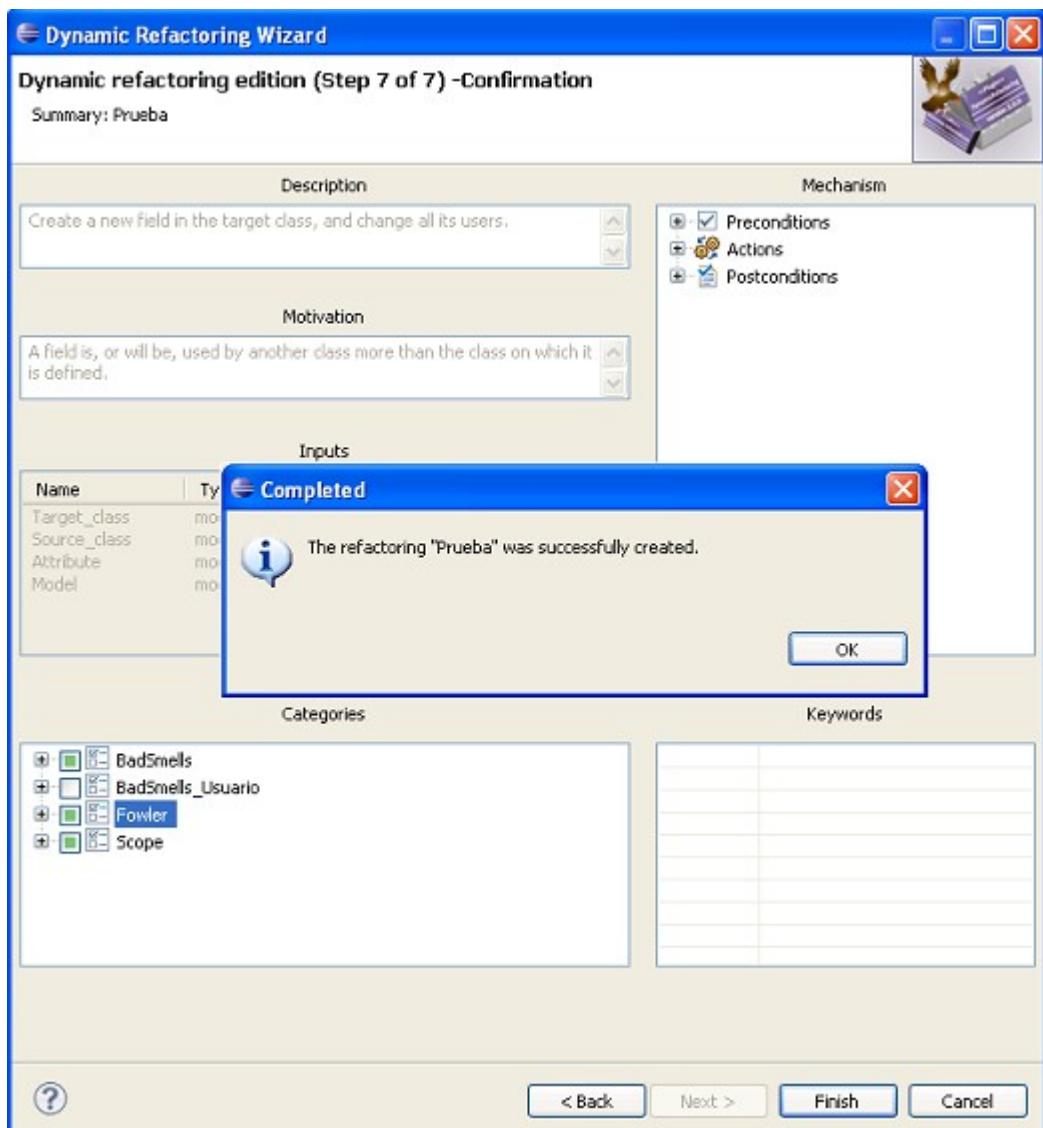


Ilustración 186: New Refactoring - Step 3: Confirmation

Una vez que la nueva refactorización ha sido creada, esta aparecerá en la lista de refactorizaciones disponibles que se muestra en las distintas partes de la interfaz gráfica del plugin como una refactorización propia del usuario, pudiéndose identificar como tal mediante el siguiente ícono.

-  Ícono refactorización definida por el usuario (editable).

2.2. Edit Refactoring

La segunda funcionalidad referente a la gestión de refactorizaciones que ofrece el menú *Dynamic Refactoring* es la edición de refactorizaciones mediante la opción *Edit Refactoring...* que aparece en el mismo.

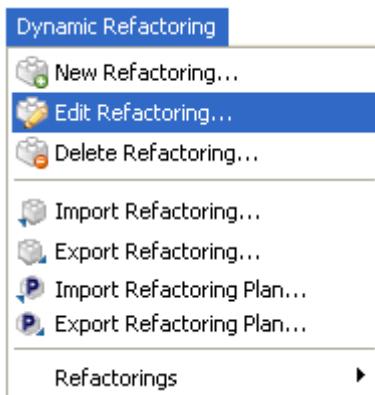


Ilustración 187: Dynamic Refactoring - Edit Refactoring

Nos dirigiremos a *Dynamic Refactoring > Edit Refactoring* para que se abra el diálogo que se muestra a continuación, el cual permite la edición de refactorizaciones.

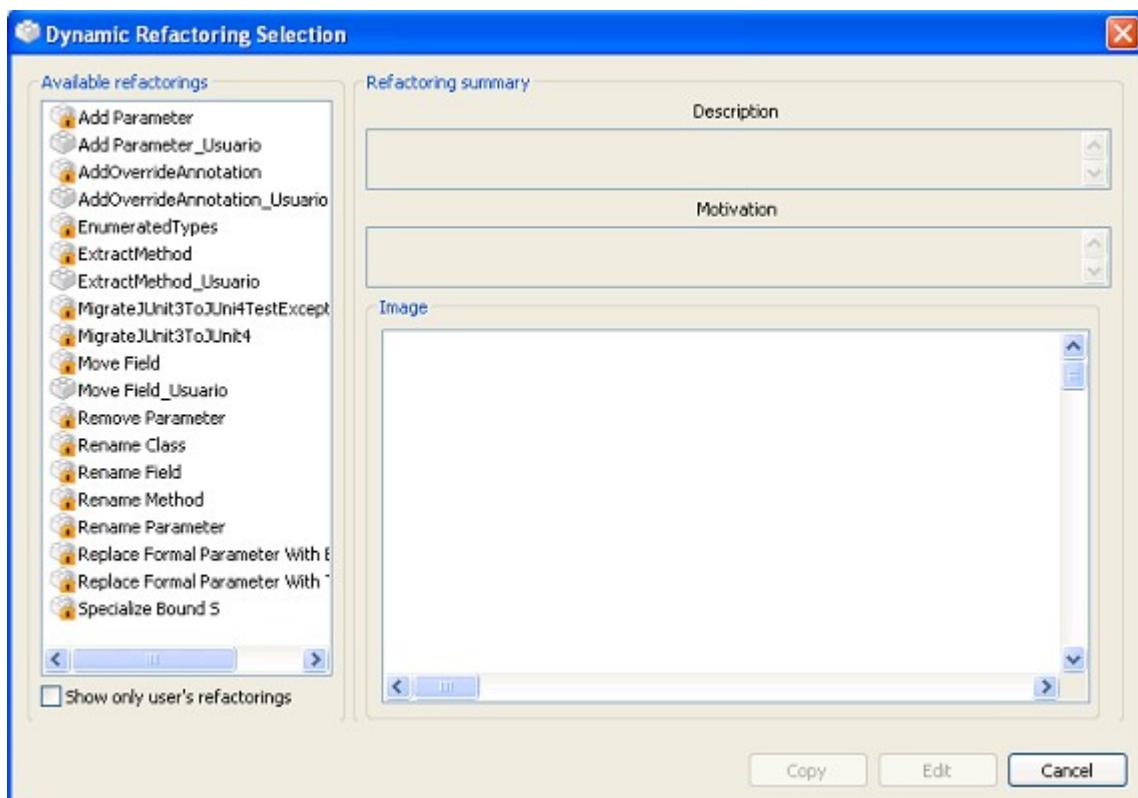


Ilustración 188: Edit Refactoring – Dynamic Refactoring Selection

A continuación se detallan los cambios que ha sufrido esta interfaz:

Available refactorings

En la parte izquierda del diálogo aparece la lista de refactorizaciones disponibles, cada una de ellas incluye el icono representativo para indicar si la refactorización viene suministrada con el plugin o si es propia del usuario.

Estos son los siguientes:

- ⌚ Icono refactorización suministrada con el plugin (no editable).
- ⌚ Icono refactorización definida por el usuario (editable).

Una vez que una refactorización ha sido seleccionada en la lista de refactorizaciones disponibles, se mostrará en la parte derecha del diálogo la información básica de la misma (descripción, motivación e imagen, esta última si la tuviese) y los botones correspondientes estarán disponibles para su selección como detallaremos más adelante.

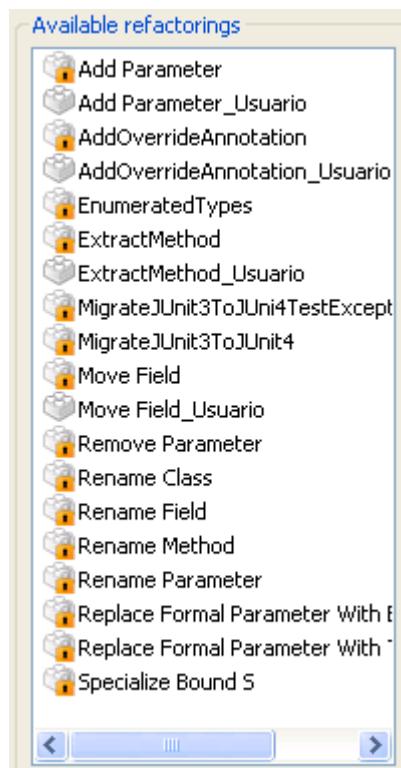


Ilustración 189: Edit Refactoring – Available refactorings

Show only user's refactorings Show only user's refactorings

Nuevo botón que permite ocultar las refactorizaciones suministradas con el plugin y por tanto sólo mostrar aquellas propias del usuario, mejorando así su visualización.

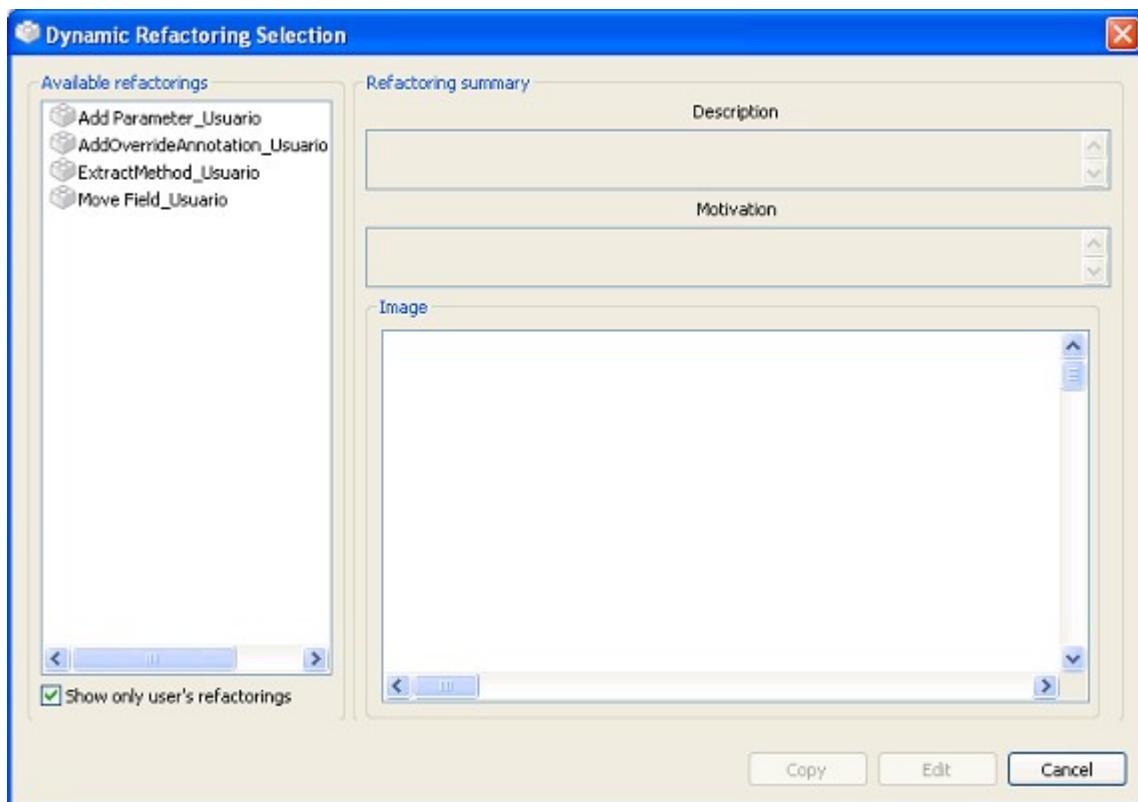


Ilustración 190: Edit Refactoring – Show only user's refactorings

A continuación se indica la función de cada uno de los botones que se encuentran disponibles cuando una refactorización es seleccionada, ya que esta ha cambiado respecto a la versión anterior del plugin salvo para el botón *Cancel*.

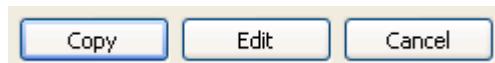
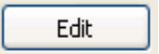


Ilustración 191: Edit Refactoring – Botones

Copy Copy

Nuevo botón que se encuentra disponible al seleccionar una refactorización independientemente de si se trata de una refactorización suministrada con el plugin o propia del usuario.

Permite crear una nueva refactorización a partir de la seleccionada, tomando su definición como definición por defecto la cual a lo largo del proceso de edición podrá ser personalizada como se deseé. El nombre por defecto dado a la refactorización será el mismo con el sufijo '_Copy', igualmente podrá ser modificado.

Edit Edit

Botón que permite la edición de la refactorización que se encuentre seleccionada y que únicamente se encuentra habilitado en el caso de tratarse de una refactorización editable, es decir, propia del usuario.

Una refactorización suministrada con el plugin no se podrá modificar por lo que para ella no estará disponible este botón, el efecto deseado se conseguirá realizando una copia de esta y sobre ella realizar la edición.

Una vez que ha sido seleccionado alguno de los dos botones anteriores comenzará el proceso de edición que será guiado a través de un asistente. El asistente de modificación es idéntico al de creación con la salvedad de que en la modificación los campos aparecerán completados con los valores de la refactorización que se está editando. Para obtener más detalles acerca de cómo cumplimentar los diferentes campos del asistente se puede consultar el apartado anterior.

Al alcanzar el último paso (paso 7), que corresponde con el paso de confirmación, se podrán guardar los cambios realizados en la refactorización pulsando el botón *Finish*. Un mensaje de confirmación indicará que la refactorización se ha guardado correctamente, como se puede observar en la siguiente ilustración.

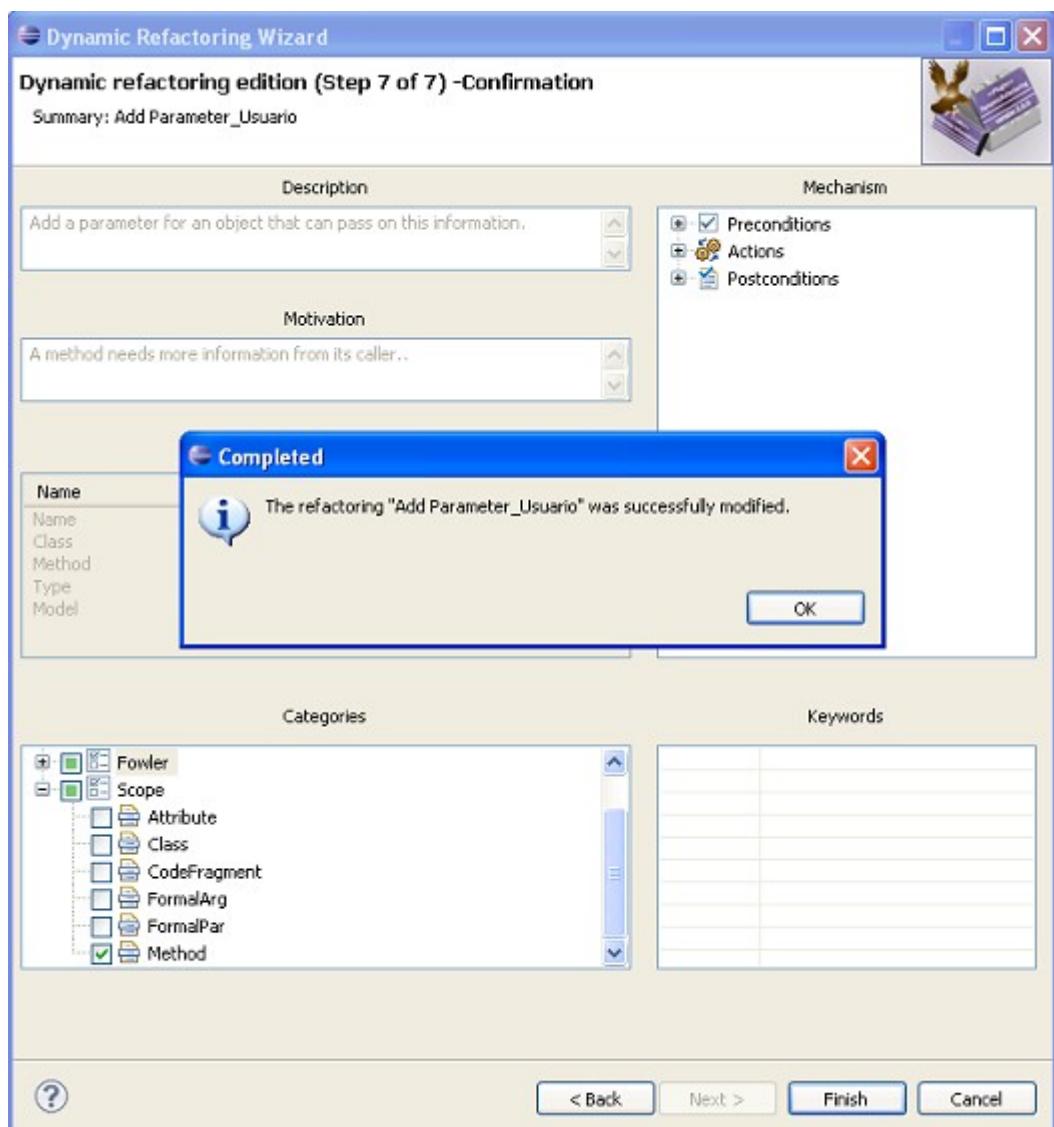


Ilustración 192: Edit Refactoring – Step 7: Confirmation

2.3. Delete Refactoring

La última funcionalidad referente a la gestión de refactorizaciones que ofrece el menú *Dynamic Refactoring* es la del borrado de refactorizaciones mediante la opción *Delete Refactoring...* que aparece en el mismo.

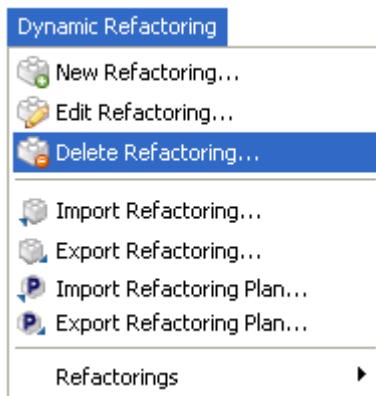


Ilustración 193: Dynamic Refactoring - Delete Refactoring

Nos dirigiremos a *Dynamic Refactoring* > *Delete Refactoring* para que se abra el siguiente diálogo, que permite la eliminación de refactorizaciones.

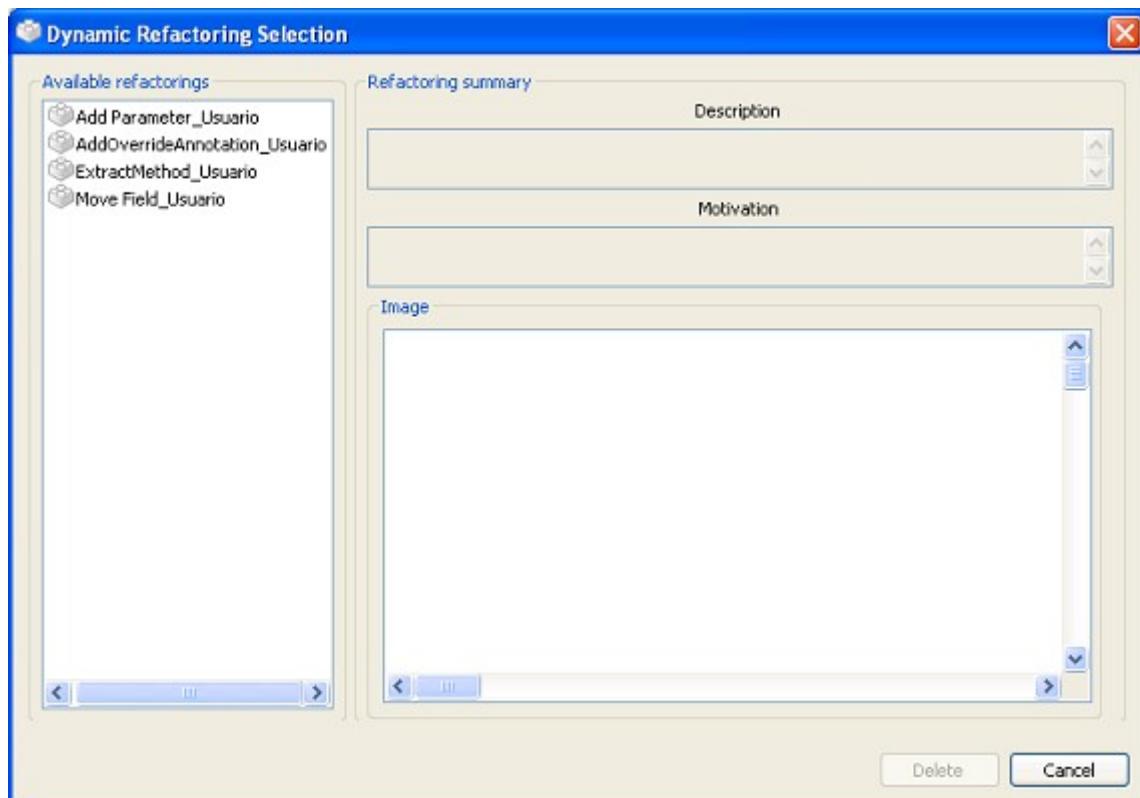


Ilustración 194: Delete Refactoring – Dynamic Refactoring Selection

En la parte izquierda del diálogo aparece la lista de refactorizaciones (*Available refactorings*) disponibles para su borrado, únicamente aparecerán las refactorizaciones que hayan sido definidas por el usuario, es decir, las suministradas con el plugin no aparecerán por no ser editables.

Una vez que haya sido seleccionada en la lista la refactorización a eliminar de entre las disponibles, se mostrará en la parte derecha del diálogo la información básica de la misma (descripción, motivación e imagen, esta última si la tuviese) y el botón *Delete* estará disponible para su selección.

Si el botón *Delete* es seleccionado, antes de proceder al borrado de la refactorización se pedirá confirmación mediante la siguiente pantalla. En caso de estar seguro de querer eliminarla de forma permanente pulsaremos el botón *OK*, en caso contrario pulsaremos *Cancel* o *Close* (*X* o *ESC*).



Ilustración 195: Delete Refactoring – Confirmation Needed

Una vez que la refactorización ha sido eliminada de forma permanente nos aparece el siguiente mensaje informativo.

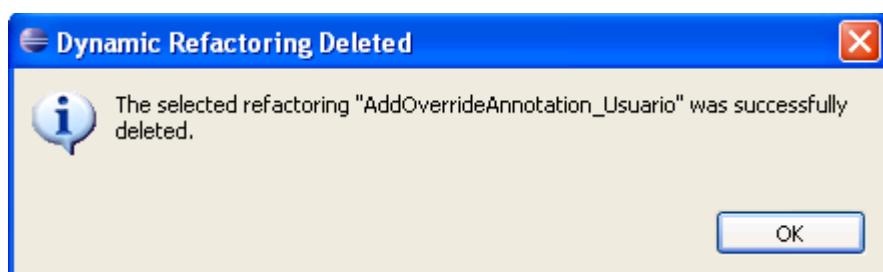


Ilustración 196: Delete Refactoring – Dynamic Refactoring Deleted

Cabe destacar que es recomendable hacer una copia de seguridad de la carpeta de refactorizaciones *DynamicRefactorings* antes de eliminar refactorizaciones, puesto que de otra manera no se podrán volver a recuperar estos ficheros. Si se guarda una copia, se pueden recuperar manualmente, o a través de la utilidad de importación.

2.4. Export Refactoring

Otra de las funcionalidades que ofrece el menú *Dynamic Refactoring*, mediante la opción *Export Refactoring...*, es la de exportar las refactorizaciones dinámicas existentes con el objetivo de que estas se puedan instalar fácilmente en otra instalación de Eclipse que cuente con el plugin de refactorizaciones o incluso que sirvan como copia de seguridad de las propias refactorizaciones.

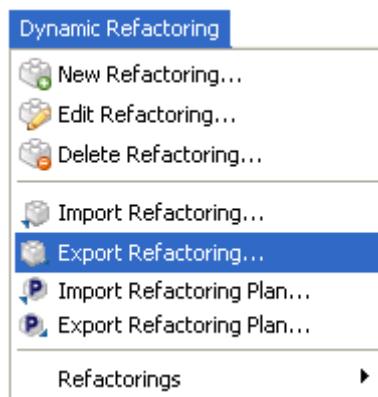


Ilustración 197: *Dynamic Refactoring - Export Refactoring*

Por lo tanto, para exportar refactorizaciones se debe dirigir a *Dynamic Refactoring > Export Refactoring* para que se abra el diálogo que se muestra a continuación, que permite la exportación de refactorizaciones.

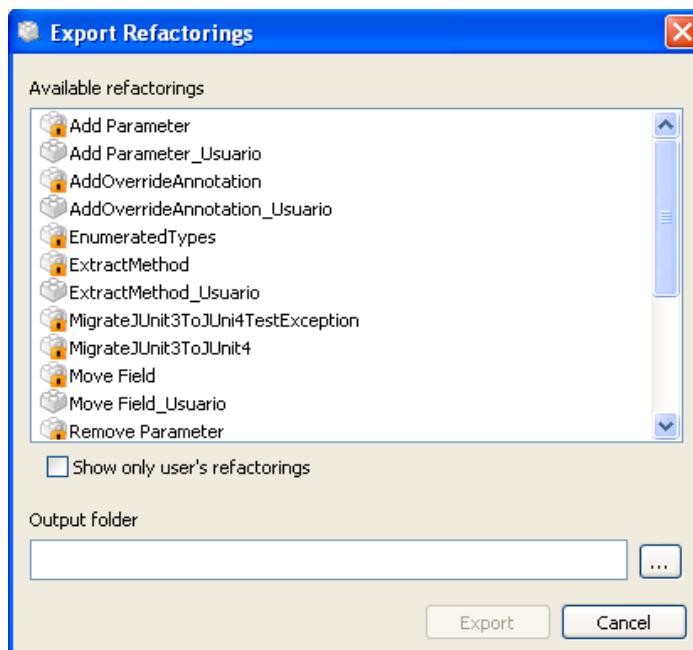


Ilustración 198: *Export Refactorings*

A continuación se detallan los cambios que ha sufrido la interfaz:

Available refactorings

En la parte superior de este dialogo aparece la lista de refactorizaciones disponibles, cada una de ellas incluye el icono representativo para indicar si la refactorización viene suministrada con el plugin o si es propia del usuario.

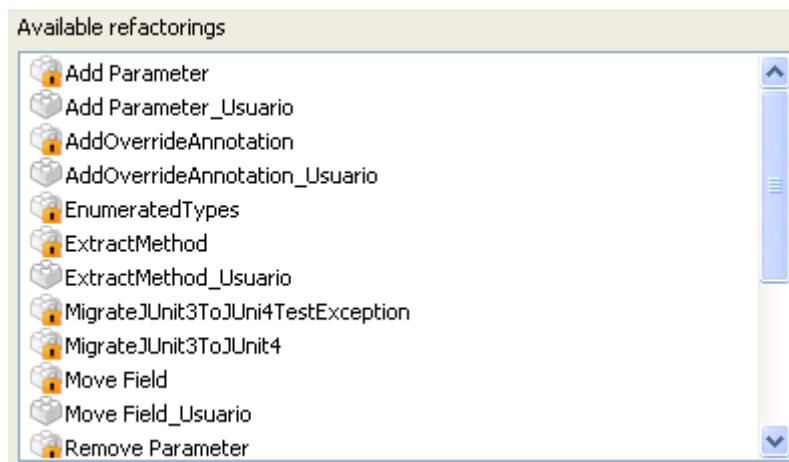


Ilustración 199: Export Refactorings - Available refactorings

Show only user's refactorings

Show only user's refactorings

Nuevo botón que permite ocultar las refactorizaciones suministradas con el plugin y por tanto sólo mostrar aquellas propias del usuario, mejorando así su visualización.

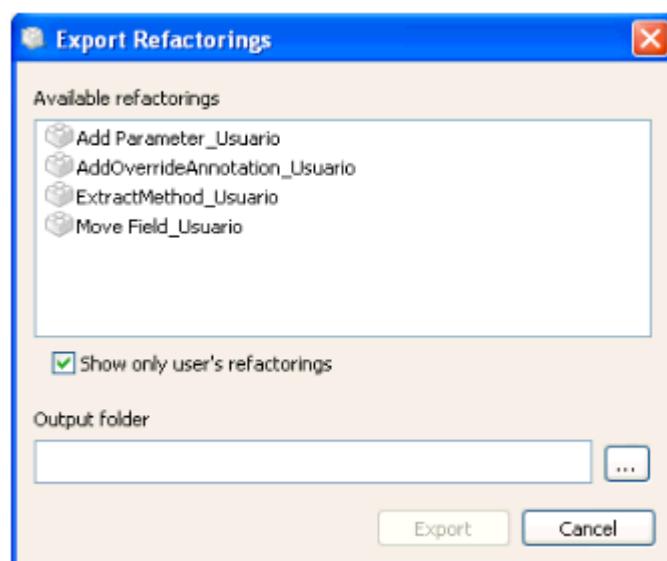


Ilustración 200: Export Refactorings – Show only user's refactorings

El resto de funcionalidades no ha sufrido ningún cambio, por lo que para realizar la exportación de refactorizaciones basta con seleccionar las refactorizaciones a exportar (permitiendo la selección múltiple de las mismas mediante el uso de la tecla CTRL) y elegir el directorio destino donde se desea realizar el proceso de exportación. Una terminado este proceso, se habrá creado en él un directorio para cada una de las refactorizaciones exportadas.

3. VISTA: AVAILABLE REFACTORINGS

La vista *Available Refactorings* se encarga de mostrar las distintas refactorizaciones disponibles que pueden ser ejecutadas tomando como entrada principal el elemento que este siendo seleccionado por el usuario en el editor de código fuente, con el objetivo de guiar al usuario hacia la tarea final evitando que sea éste el que tenga que buscar en el menú adecuado para poder ejecutar la refactorización.

```
package org.jfree.chart.annotations;

import java.awt.BasicStroke;□

/**
 * A line annotation that can be placed on a {@link CategoryPlot}.
 */
public class CategoryLineAnnotation implements CategoryAnnotation,
    Cloneable, PublicCloneable, Serializable {

    /** For serialization. */
    static final long serialVersionUID = 3477740483341587984L;

    /** The category for the start of the line. */
    private Comparable category1; □

    /** The value for the start of the line. */
    private double value1;
```

Ilustración 201: Vista Available Refactorings - Selección código

En el ejemplo, en el editor de código se ve como ha sido seleccionado un atributo y automáticamente la vista de refactorizaciones disponibles muestra lo siguiente:

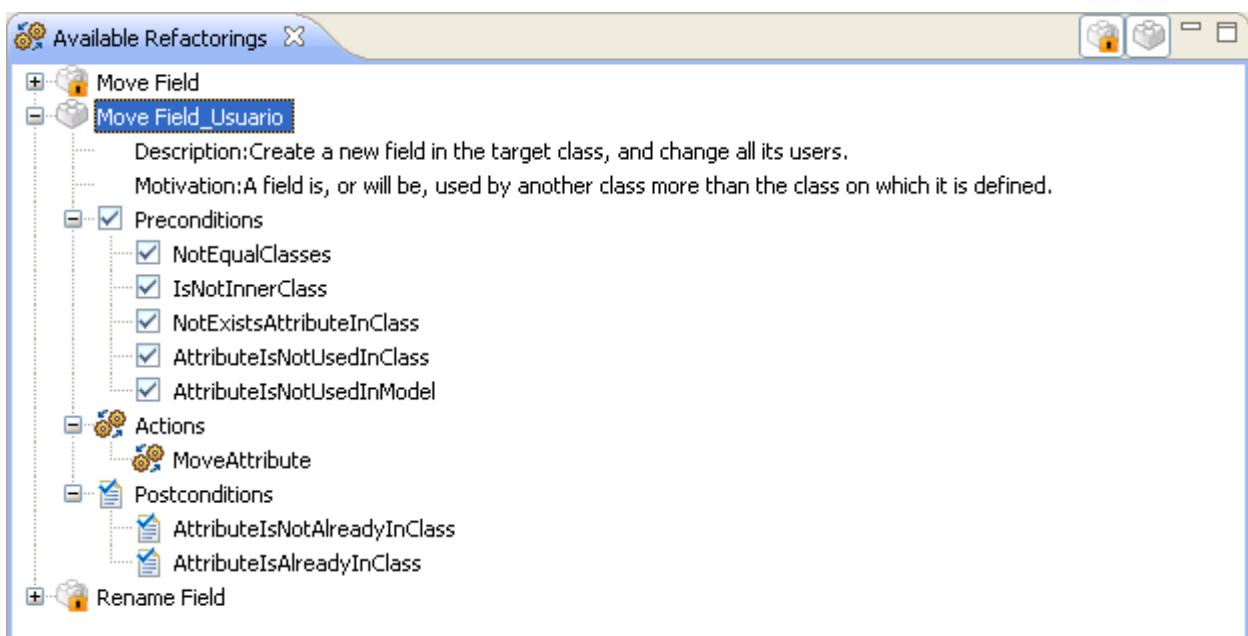


Ilustración 202: Vista Available Refactorings

En esta vista, al igual que hemos visto en otras partes de la interfaz del plugin, se puede identificar a las refactorizaciones como propias del usuario o bien como suministradas por el plugin, mediante la visualización del icono correspondiente.

 Icono refactorización suministrada con el plugin (no editable).

 Icono refactorización definida por el usuario (editable).

Además, se ha incorporado una barra de herramientas que incluye las distintas acciones que se pueden llevar a cabo sobre esta. En este caso, nos encontramos con dos botones cuya funcionalidad a continuación se va a detallar. Por defecto se encuentran ambos habilitados.

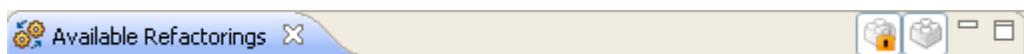


Ilustración 203: Vista Available Refactorings - Barra de herramientas

- ◆ Mostrar refactorizaciones no editables



Permite mostrar u ocultar las refactorizaciones suministradas por el plugin, es decir, las refactorizaciones no editables, mediante el habilitado o deshabilitado de este botón.

El efecto conseguido mediante el deshabilitado de este botón en el ejemplo sería el siguiente:

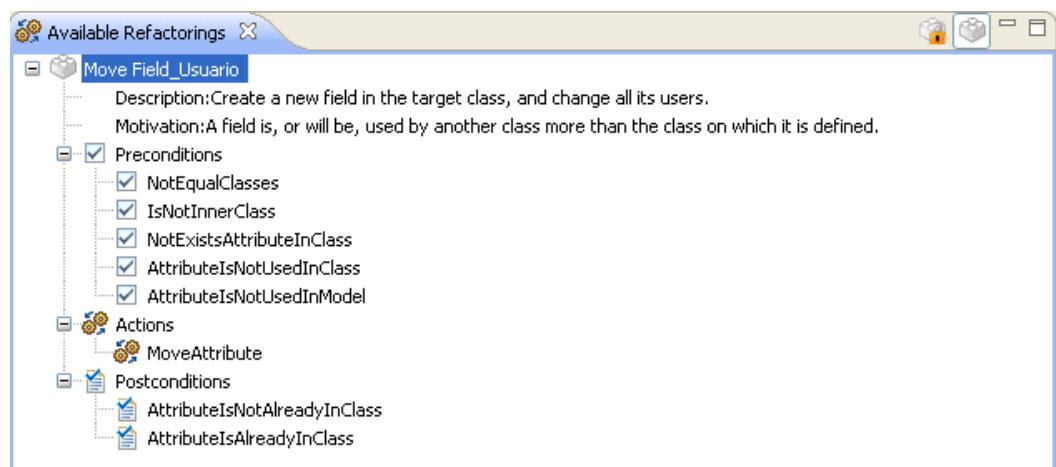
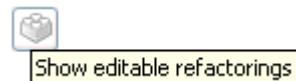


Ilustración 204: Vista Available Refactorings - Botón ref. plugin deshabilitado

- ◆ Mostrar refactorizaciones editables



Permite mostrar u ocultar refactorizaciones creadas por el usuario, es decir, las refactorizaciones editables, mediante el habilitado/deshabilitado de este botón.

El efecto conseguido mediante el deshabilitado de este botón en el ejemplo sería el siguiente:

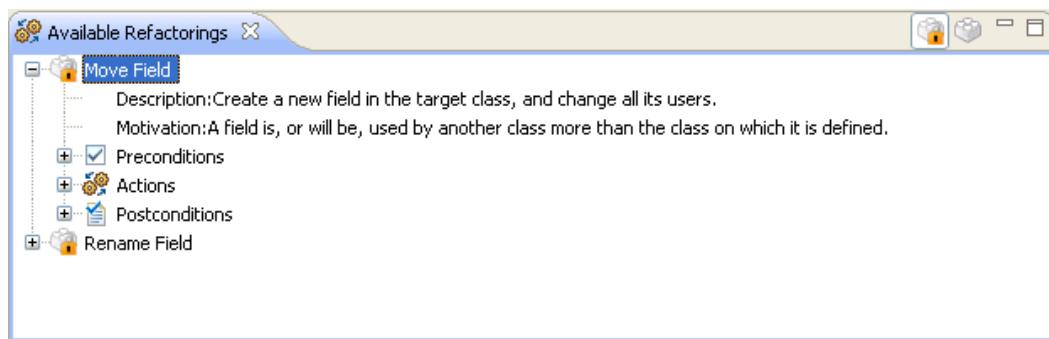


Ilustración 205: Vista Available Refactorings - Botón ref. usuario deshabilitado

4. VISTA: REFACTORING CATALOG BROWSER

La vista *Refactoring Catalog Browser* muestra el catálogo de refactorizaciones disponibles, además recoge toda la información asociada a las mismas y permite visualizar la distribución de refactorizaciones según las categorías a las que pertenecen para cada una de las clasificaciones existentes. También ofrece la posibilidad de realizar búsquedas sobre estas creando filtros que poder aplicar.

En esta vista se pueden identificar tres partes diferentes, que son las siguientes:

- La barra de herramientas de la vista, que vamos a denominar *Refactoring Catalog Browser Toolbar*.
- Un panel destinado a la clasificación de refactorizaciones, que vamos a llamar *Classification Panel*. Corresponde con el panel izquierdo de la vista.
- Un panel destinado a la visualización de la información asociada a una refactorización, que vamos a denominar *Summary Panel*. Corresponde con el panel derecho de la vista.

En la siguiente ilustración se puede apreciar cada una de las partes que acabamos de comentar. A continuación detallamos su funcionalidad y las distintas opciones que ofrecen.

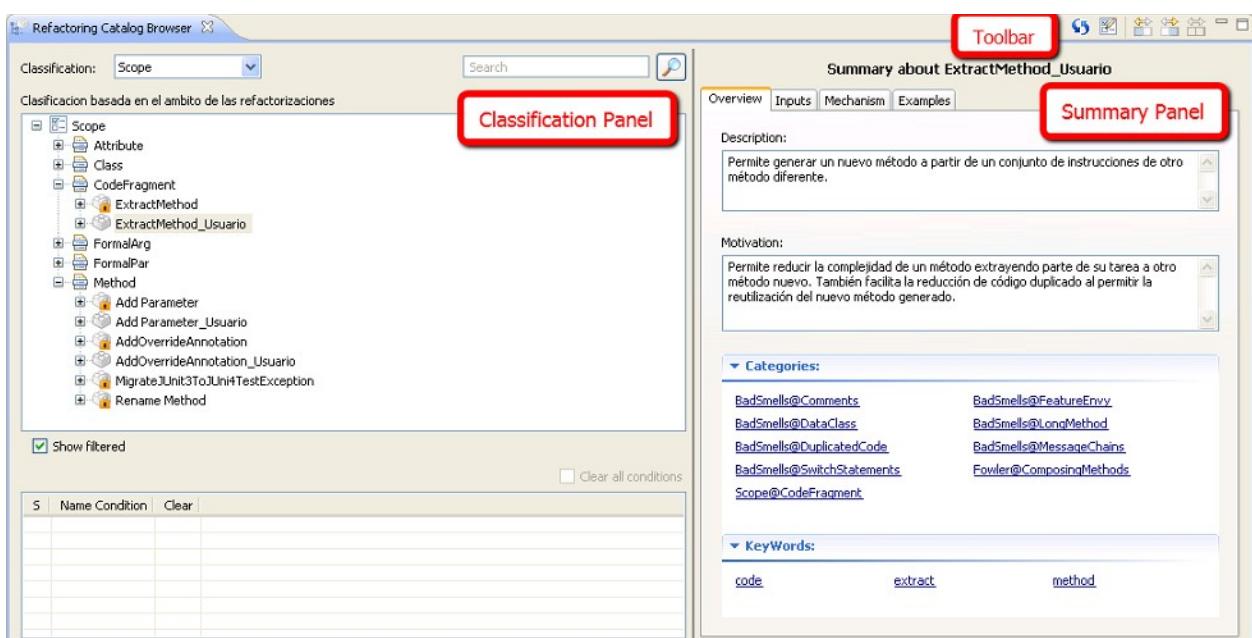


Ilustración 206: Vista Refactoring Catalog Browser

4.1. Refactoring Catalog Browser Toolbar

La vista *Refactoring Catalog Browser* dispone de una barra de herramientas que incluye las distintas acciones que se pueden llevar a cabo sobre esta. En este caso, nos encontramos con cinco botones que han sido distribuidos en dos grupos; el de la izquierda contiene dos botones independientes mientras que el de la derecha contiene tres botones con funcionalidad relacionada, estos últimos estarán siempre todos disponibles salvo uno de ellos que se encontrará deshabilitado.

En la siguiente ilustración se puede apreciar la barra de herramientas de la vista y acto seguido se detallará la funcionalidad de cada uno de los botones que contiene.



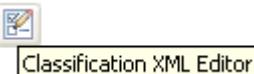
Ilustración 207: Vista Refactoring Catalog Browser - Barra de herramientas

- ◆ Refrescar vista



Permite refrescar la vista con la finalidad de visualizar actualizado el catálogo de refactorizaciones. Esta funcionalidad es importante ya que la modificación de refactorizaciones puede realizarse por varias vías, por ejemplo por un cambio en su propia definición pero también podría ser por un cambio en una categoría a la que esta pertenece. Es por ello que es importante que el usuario tenga presente que cuando realice algún cambio que afecte a estas deberá utilizar esta funcionalidad para que la información que la vista muestre se encuentre actualizada.

- ◆ Editor de clasificaciones



Permite abrir el editor de clasificaciones con el fin de visualizar la información asociada a las clasificaciones disponibles o realizar la edición de las mismas. Cuando este botón es pulsado se abrirá el editor de clasificaciones en la zona que Eclipse dispone para la visualización de editores. En caso de que este ya estuviese abierto se activará la pestaña correspondiente al editor, situándonos en él.

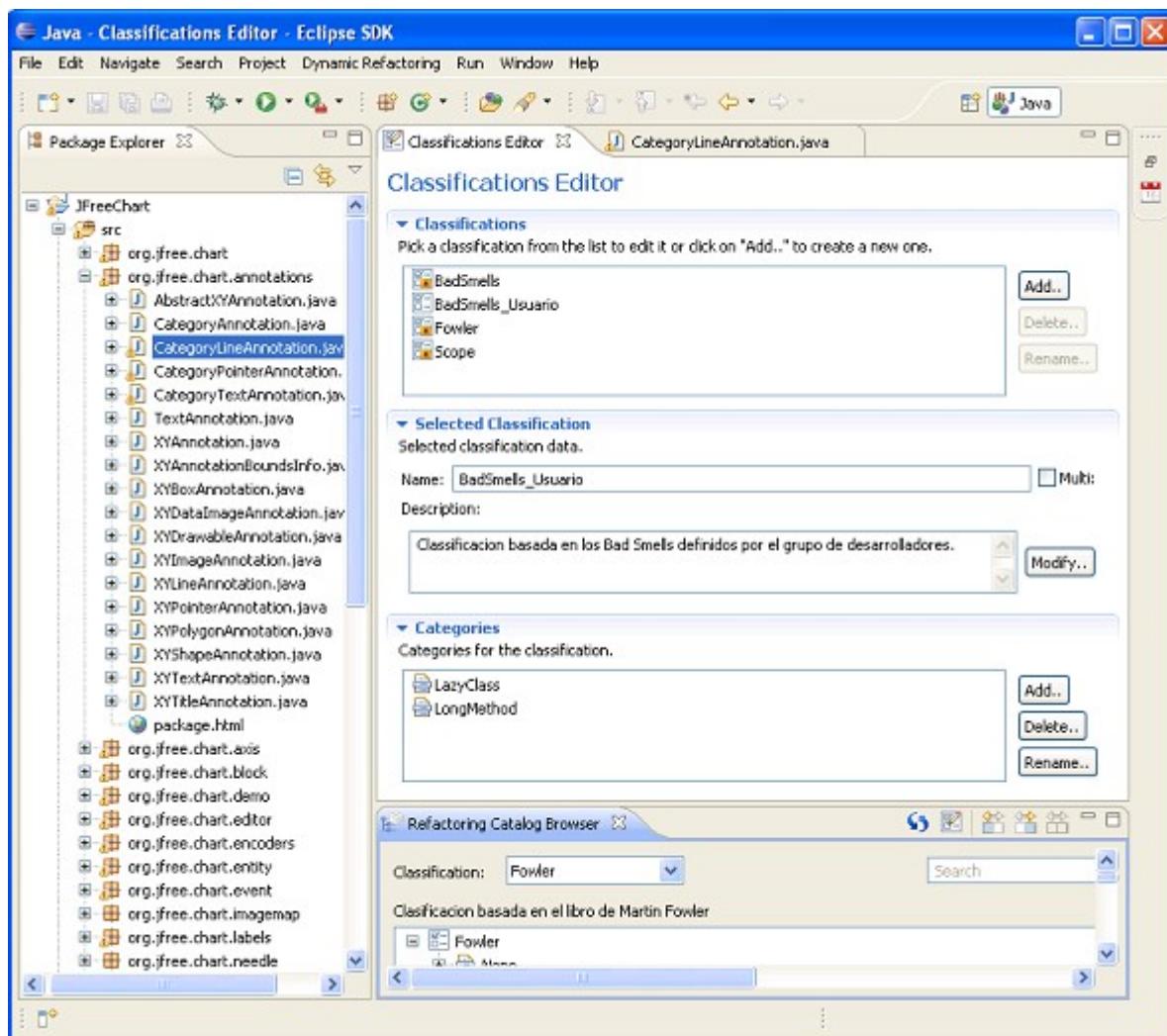


Ilustración 208: Vista Refactoring Catalog Browser – Botón Editor de clasificaciones

- ◆ Mostrar panel izquierdo



Permite mostrar únicamente el panel izquierdo *Classification Panel*, destinado a la clasificación de refactorizaciones, disponiendo de todo el tamaño de la vista para su visualización. Además cuando este botón es pulsado se deshabilita y quedan habilitados los otros dos; *Show right pane* y *Show all panes*.

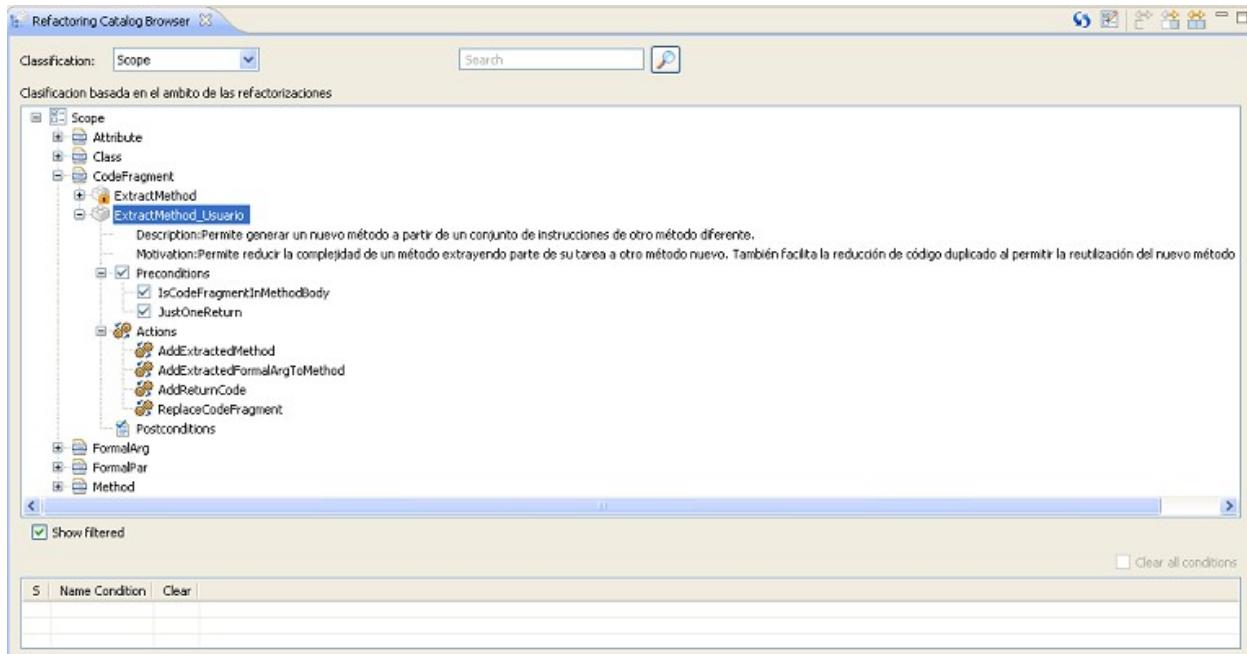


Ilustración 209: Vista Refactoring Catalog Browser – Botón Mostrar panel izquierdo

- ◆ Mostrar panel derecho  **Show right pane**

Permite mostrar únicamente el panel derecho *Summary Panel*, destinado a la visualización de la información asociada a una refactorización, disponiendo de todo el tamaño de la vista para su visualización. Además cuando este botón es pulsado se deshabilita y quedan habilitados los otros dos; *Show left pane* y *Show all panes*.



Ilustración 210: Vista Refactoring Catalog Browser – Botón Mostrar panel derecho

- ◆ Mostrar todos los paneles



Permite mostrar tanto el panel izquierdo *Classification Panel*, destinado a la clasificación de refactorizaciones, como el panel derecho *Summary Panel*, destinado a la visualización de la información asociada a una refactorización. Para separar ambos paneles se dispone de un *splitter* que ofrece la posibilidad de ampliar el tamaño de cualquiera de los dos paneles para mejorar su visualización, para ello basta con seleccionar este y manteniendo presionado el botón izquierdo del ratón desplazarlo hacia derecha o izquierda, según corresponda para conseguir el efecto deseado.

Además cuando este botón es pulsado se deshabilita y quedan habilitados los otros dos; *Show left pane* y *Show right pane*.

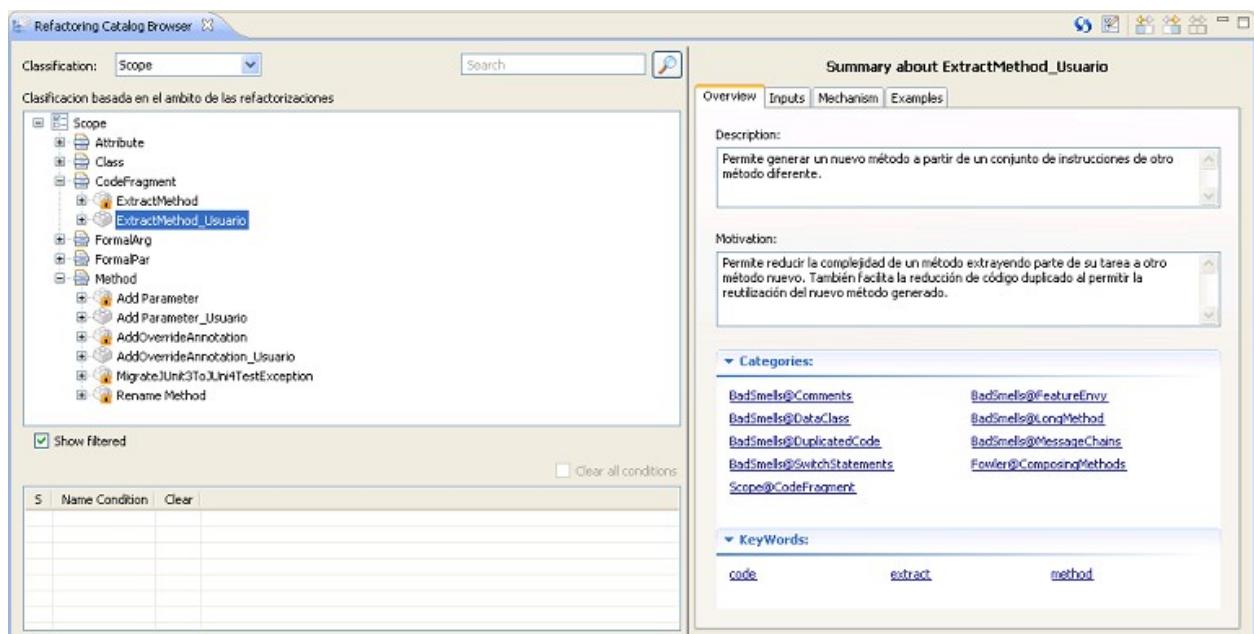


Ilustración 211: Vista Refactoring Catalog Browser – Botón Mostrar todos los paneles

4.2. Classification Panel

El panel *Classification Panel* se sitúa en la zona izquierda de la vista y esta destinado a la visualización del catálogo de refactorizaciones. En él se da la posibilidad de visualizar las refactorizaciones clasificadas según las categorías de la clasificación que previamente se haya seleccionado. Además se podrán aplicar filtros al catálogo de refactorizaciones con el objetivo de mostrar exclusivamente aquellas refactorizaciones que cumplan con ciertos

criterios que se establezcan. A continuación entraremos en detalle en cada una de las funcionalidades que acaban de ser descritas.

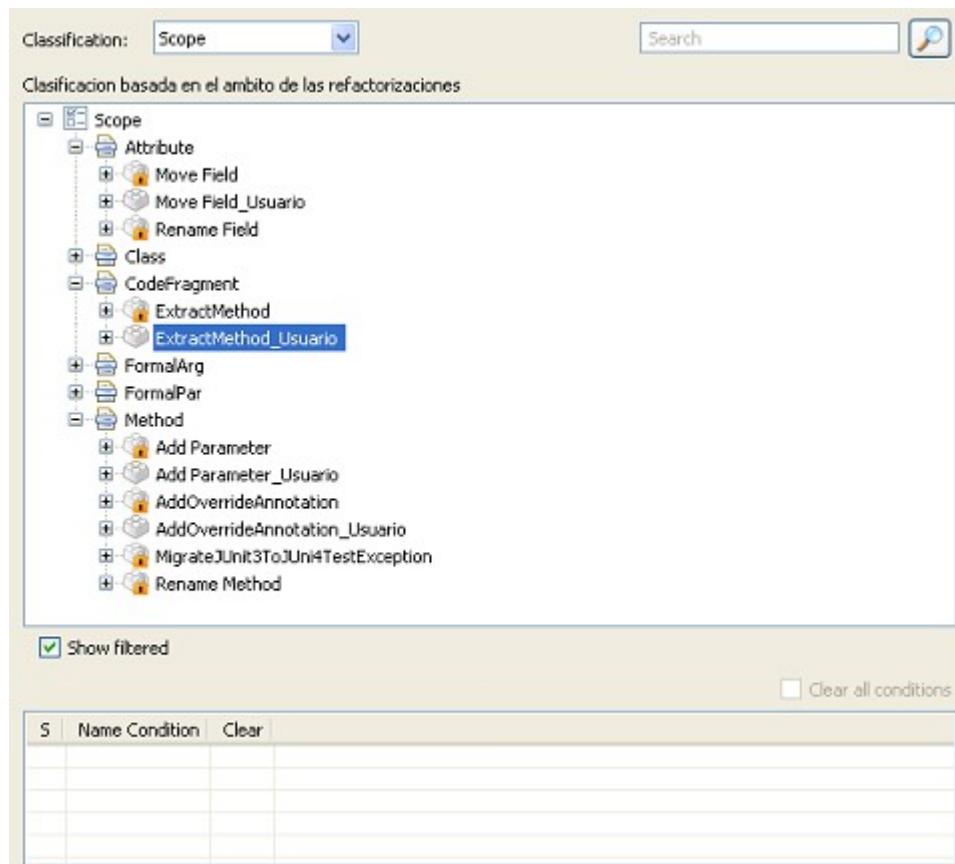


Ilustración 212: Vista Refactoring Catalog Browser – Classification Panel

En este panel se pueden diferenciar cuatro partes; zona superior izquierda destinada a la selección de la clasificación, zona superior derecha que aporta funcionalidades para la definición de filtros, zona central donde permite visualizar el catálogo de refactorizaciones clasificado y filtrado y zona inferior destinada a la visualización de los filtros creados y que permite la gestión de los mismos. Veamos cada uno de los elementos que componen cada zona.

4.2.1. Classification

Se trata de la zona superior izquierda del panel *Classifications Panel* y consta de un desplegable de clasificaciones y una descripción de la clasificación seleccionada.

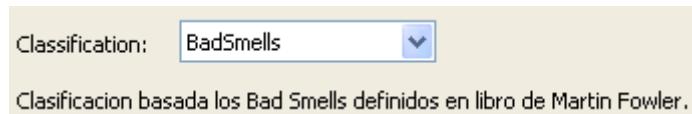


Ilustración 213: Vista Refactoring Catalog Browser – Classification

Classification

Desplegable que contiene todas las clasificaciones disponibles con la finalidad de que se pueda seleccionar aquella con la que clasificar el catálogo de refactorizaciones. En caso de no querer elegir ninguna clasificación de las disponibles seleccionaremos la correspondiente a *None* en el desplegable.

Una vez que ha sido seleccionada una clasificación la zona de visualización queda actualizada mostrando el catálogo de refactorizaciones clasificado de acuerdo a las categorías de las que dispone la clasificación seleccionada.

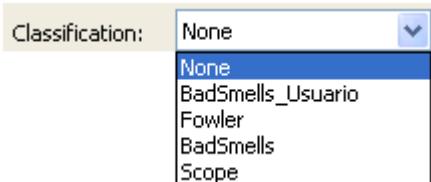


Ilustración 214: Vista Refactoring Catalog Browser – Desplegable Classification

Description

Permite visualizar la descripción que tiene asociada la clasificación que ha sido elegida en el desplegable.

Default classification without categories

Ilustración 215: Vista Refactoring Catalog Browser – Classification Description

En la siguiente ilustración podemos observar como cambia la zona destinada a la visualización del catálogo al elegir la clasificación *Scope*. También comprobaremos como la descripción de la clasificación se ha adecuado a la elegida.

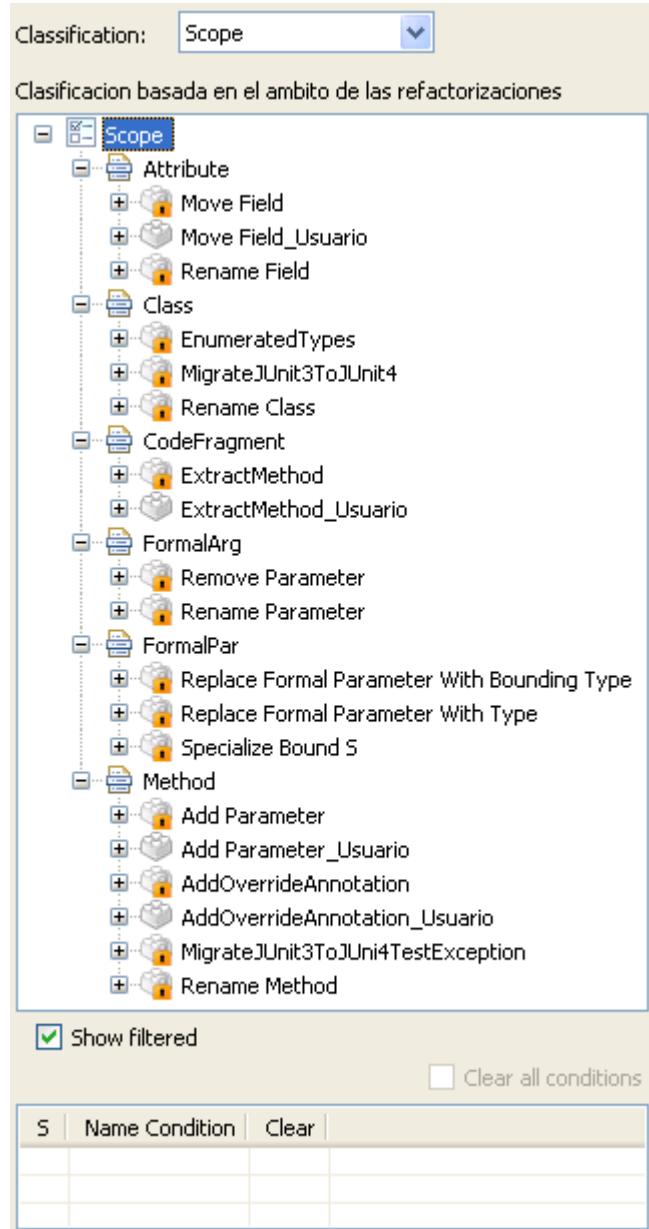


Ilustración 216: Vista Refactoring Catalog Browser – Classification Scope

4.2.2. Search

Se trata de la zona superior derecha del panel *Classifications Panel* y consta de un cuadro de texto donde introducir la búsqueda y un botón para aceptar la misma, creando así una condición de filtro. A continuación veremos en detalle cómo aprovechar esta funcionalidad con la que la vista cuenta.



Ilustración 217: Vista Refactoring Catalog Browser – Search

Este sistema de filtrado consiste en aprovechar la metainformación con que las refactorizaciones cuentan como es su nombre, descripción, motivación, sus palabras clave y las categorías a las que pertenecen con el fin de resaltar las refactorizaciones que cumplan con ciertos criterios establecidos y ocultar el resto. Las condiciones de filtro que hayan sido creadas se mostrarán para que en todo momento se tenga conocimiento de ellas, además se podrán gestionar como veremos más adelante en el apartado dedicado a estas.

Los criterios por los que se puede filtrar son: categorías, texto y palabras clave. La sintaxis a la que deberán responder será la siguiente:

- Categoría category:'clasificación'@'categoría'
Por ejemplo: category:scope@method
- Texto text:'text'
Por ejemplo: text:add
- Palabra clave key:'palabra clave'
Por ejemplo: key:annotation

Debido a que las búsquedas para crear condiciones de filtros deben de realizarse con una sintaxis fija se proporciona ayuda integrada para que facilite su construcción, evitando así que se tenga que conocer de antemano esta sintaxis. Para hacer visible esta ayuda tendremos que hacer clic sobre el cuadro de texto donde se introduce la búsqueda y acto seguido aparecerá el icono  de una interrogación a su izquierda. Si nos situamos sobre ella con el cursor o bien pulsamos la tecla F1 aparecerá la ayuda en forma de *popup*, como muestra la ilustración. Si lo hacemos de la primera forma se mostrará la ayuda hasta que decidamos dejar de situar el cursor en el ícono de interrogación, sin embargo si se realiza a través de la tecla de ayuda al cabo de unos segundos desaparecerá automáticamente.

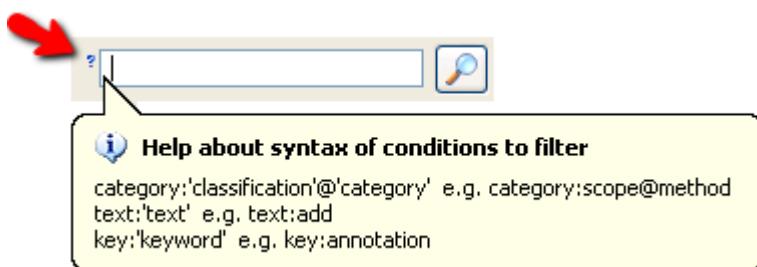


Ilustración 218: Vista Refactoring Catalog Browser – Search Help

A continuación crearemos las condiciones de filtrado del ejemplo que hemos visto y veremos como cambia la visualización del catálogo de refactorizaciones.

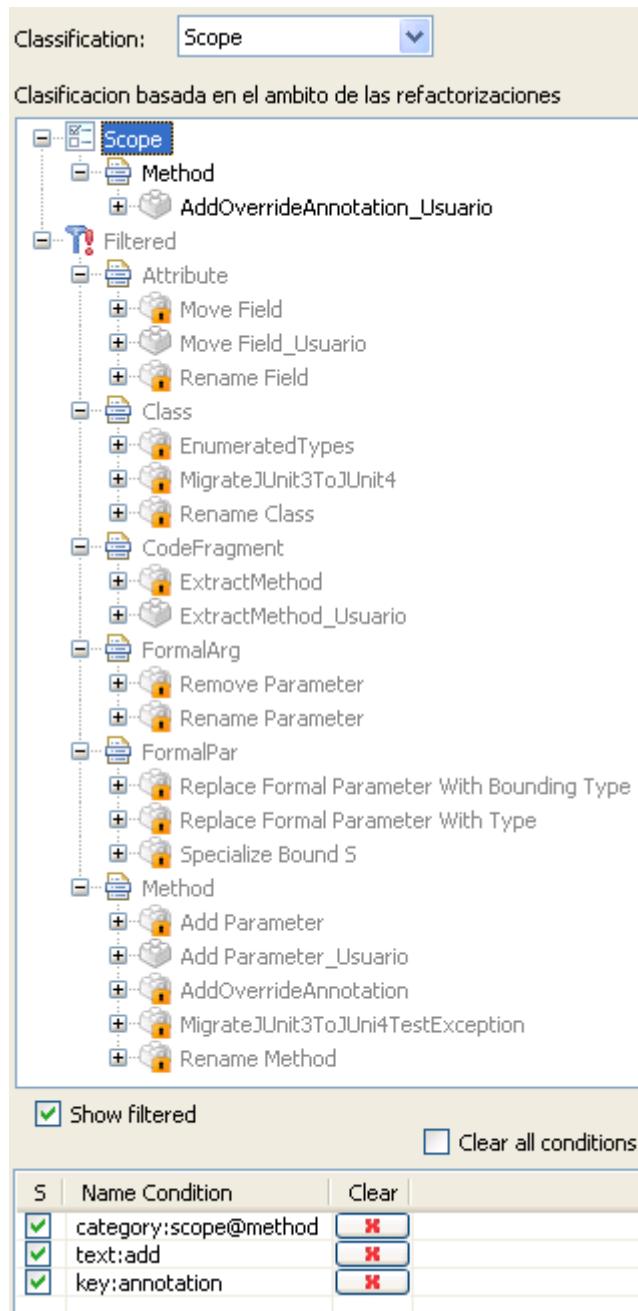


Ilustración 219: Vista Refactoring Catalog Browser – Search II

Previamente habremos añadido la palabra clave *annotation* a la refactorización *AddOverrideAnnotation_Usuario*, la cual se ha creado como copia de la refactorización *AddOverrideAnnotation*, mediante el asistente de edición de refactorizaciones disponible.

Recordar que una vez editada la refactorización se deberá refrescar la vista para recoger los cambios, para ello pulsaremos el botón *Refresh* disponible para tal efecto en la barra de herramientas de la propia vista.

Ademas, en el proceso de creación de las condiciones de filtro la aplicación nos puede advertir de errores en sintaxis, de la existencia ya de un filtro idéntico al que se esta intentando crear o de que ciertos datos introducidos parecen no ser correctos. Destacar que no es *case sensitive* por lo que el problema con mayúsculas y minúsculas esta resuelto.

A continuación vamos a ver en detalle los posibles errores que nos pueden aparecer.

Error de sintaxis en el filtro

Advierte del error de sintaxis y no crea el filtro.

Ejemplo: `textt : add`

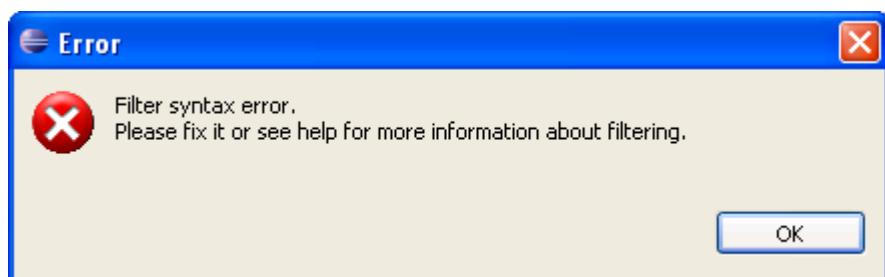


Ilustración 220: Vista Refactoring Catalog Browser – Search Error

Filtro ya existente

Avisa que el filtro que se esta intentando crear ya existe, por tanto no lo crea.

Ejemplo: `key:annotation` (previamente ya lo habíamos creado)

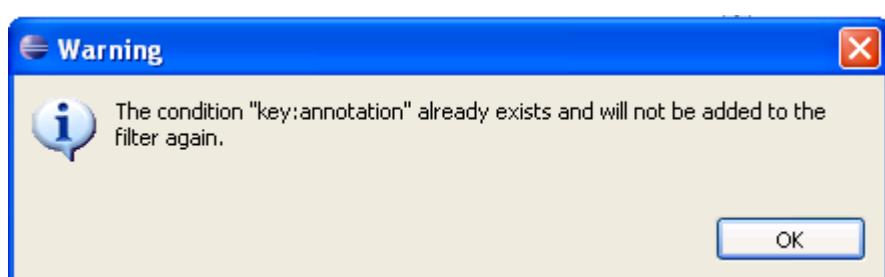
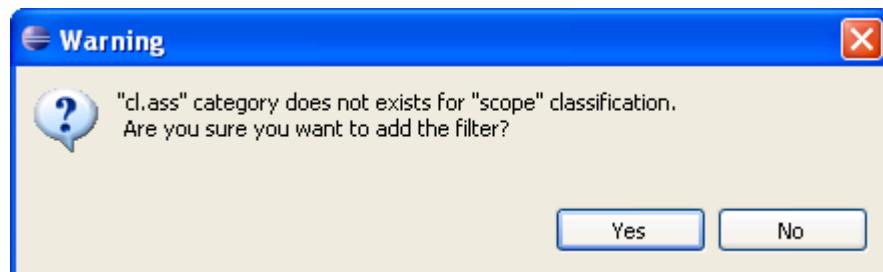


Ilustración 221: Vista Refactoring Catalog Browser – Search Aviso I

Filtro para una categoría que no existe

Avisa que se está intentando crear un filtro para una categoría que no existe, es decir, que la clasificación indicada no dispone de esa categoría. Nos preguntará si aún en ese caso queremos crear el filtro. Pulsaremos Yes o No según deseemos.

Ejemplo: category:scope@class

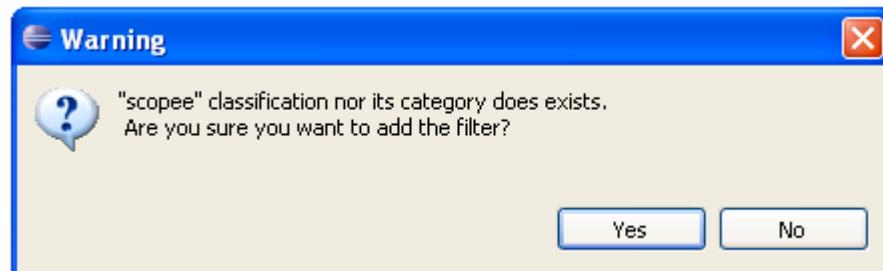


*Ilustración 222: Vista Refactoring Catalog Browser – Search Aviso
II*

Filtro para una clasificación que no existe

Avisa que se está intentando crear un filtro para una clasificación que no existe. Nos preguntará si aún en ese caso queremos crear el filtro. Pulsaremos Yes o No según deseemos.

Ejemplo: category:scopee@class



*Ilustración 223: Vista Refactoring Catalog Browser – Search Aviso
III*

4.2.3. Visualization

Se trata de la zona central del panel *Classifications Panel* y consta de un árbol que permite la representación del catálogo de refactorizaciones y un *checkbox* para elegir si se desea visualizar o no las refactorizaciones filtradas, en caso de haberlas.

Después de haber creado los filtros que acabamos de ver, la zona de visualización queda como en la siguiente ilustración se muestra. En ella se pueden observar cada uno de los elementos que la componen, posteriormente se realizará una explicación de los mismos.

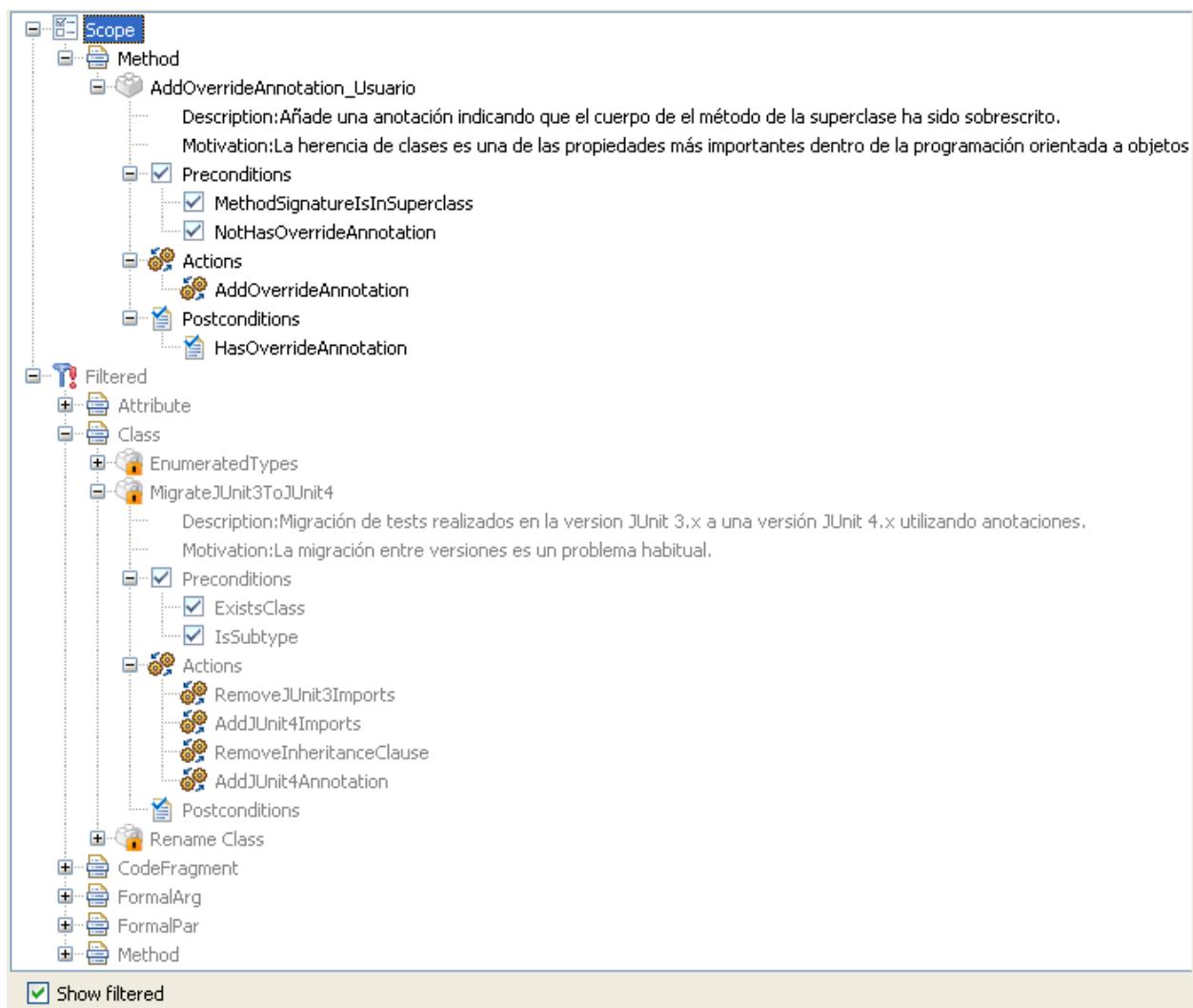


Ilustración 224: Vista Refactoring Catalog Browser – Visualization

Refactoring Catalog Tree

Árbol que representa el catálogo de refactorizaciones clasificado según las categorías de la clasificación seleccionada en el desplegable de clasificaciones que ya hemos

visto. Este árbol se divide en dos grupos; el de la propia clasificación y el de los filtrados (*Filtered*), este último aparecerá únicamente en caso de haber elementos filtrados. Cada uno de estos grupos estará a su vez dividido en las categorías de las que se compone la clasificación y cada una de estas contendrán las refactorizaciones que pertenecen a las mismas. Nótese que en el grupo correspondiente a filtrados también aparecerán las refactorizaciones clasificadas. Además, cada uno de los niveles se presentará ordenado alfabéticamente y el grupo de filtrados, en caso de aparecer, se situará al final del árbol.

A continuación se muestra la relación de iconos que aparece en el árbol y que corresponden a cada uno de los elementos que acabamos de describir.

-  Icono clasificación.
-  Icono categoría de una clasificación.
-  Icono grupo filtrados.
-  Icono refactorización suministrada con el plugin (no editable).
-  Icono refactorización definida por el usuario (editable).
-  Icono precondición de una refactorización.
-  Icono acción de una refactorización.
-  Icono postcondición de una refactorización.

Por último, vamos a ver la representación que se hace de las refactorizaciones. Si se despliega en el árbol cualquiera de las refactorizaciones aparecerá su descripción, motivación y los mecanismos que la definen, es decir, precondiciones, acciones y postcondiciones. Si de desea obtener más detalles de la misma haremos doble clic, con cualquier botón del ratón, sobre la misma y aparecerá a la derecha el panel de resumen, *Summary Panel*. No entraremos en detalle en este panel ya que será objeto de estudio posteriormente.

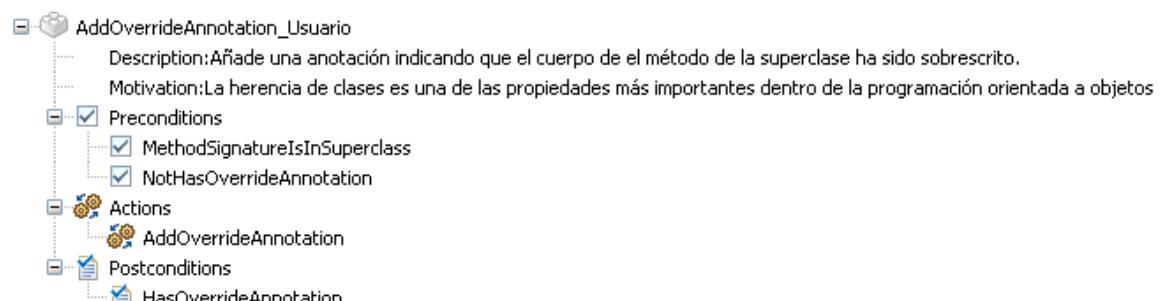


Ilustración 225: Vista Refactoring Catalog Browser – Representación refactorización

Show filtered Show filtered

Permite hacer visible u ocultar el grupo de filtrados. El grupo de filtrados muestra los elementos atenuando su visibilidad, es decir, los marca de color gris como se puede observar en la ilustración anteriormente mostrada en la que aparece este grupo.

En caso de desmarcar este *checkbox* el grupo de filtrados se ocultará, no haciéndose visible para el usuario como en la siguiente ilustración se puede comprobar.

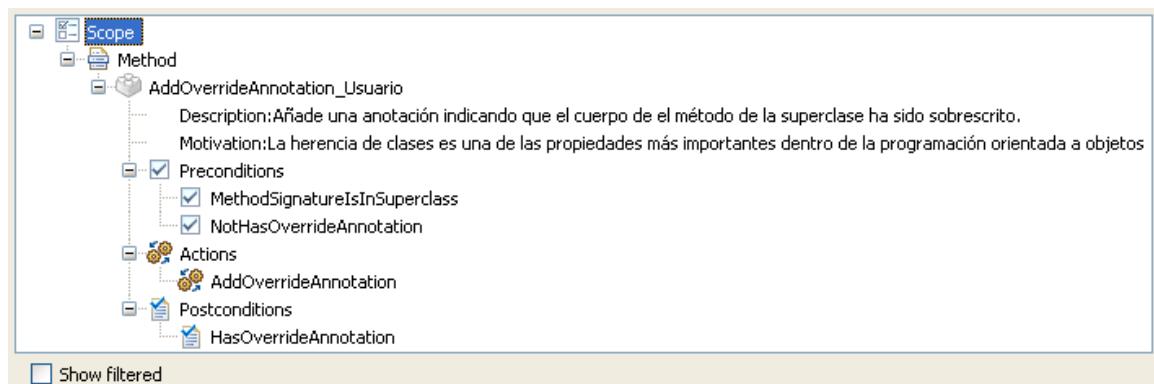


Ilustración 226: Vista Refactoring Catalog Browser – Show filtered

4.2.4. Filters

Se trata de la zona inferior del panel *Classifications Panel* y esta destinada a la visualización de las condiciones de filtro creadas, además permite la gestión de las mismas. La mayor parte de esta funcionalidad se encuentra recogida en la tabla de condiciones de filtro. A continuación veremos cómo hacer uso de esta funcionalidad que ofrece la vista.

S	Name	Condition	Clear
<input checked="" type="checkbox"/>	category:scope@method		
<input checked="" type="checkbox"/>	text:add		
<input checked="" type="checkbox"/>	key:annotation		

Ilustración 227: Vista Refactoring Catalog Browser – Filters

La tabla de condiciones de filtro contiene una condición por cada fila de la misma, que habrá sido previamente creada. Esta tabla esta formada por tres columnas que vamos a ver en profundidad, son las siguientes:

Selected

Primera columna de la tabla que contiene un *checkbox* por cada condición de filtro creada y que permite seleccionar o no esta para aplicarla o no en el filtro sobre el catálogo de refactorizaciones. Por defecto, cuando se crea la condición de filtro estará activa.

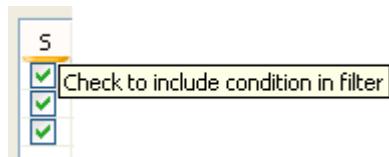


Ilustración 228: Vista Refactoring Catalog Browser – Columna Selected

Name Condition

Segunda columna de la tabla, contiene el texto correspondiente a la condición de filtro que previamente habrá sido creada. Cuando una condición es desactivada, para que esta no se aplique en el filtro sobre el catálogo de refactorizaciones, aparecerá atenuada su visibilidad mostrándola en color grisáceo.

Name Condition
category:scope@method
text:add
key:annotation

Ilustración 229: Vista Refactoring Catalog Browser – Columna Name Condition

Clear

Tercera y última columna de la tabla, contiene un botón por cada condición de filtro creada y su función es la de permitir eliminar la condición de forma persistente.



Ilustración 230: Vista Refactoring Catalog Browser – Columna Clear

Clear all conditions

Además, también se dispone de un botón para eliminar de forma persistente todas las condiciones de filtro que se han creado y por tanto se encuentran en la tabla.

De esta forma con solo pulsar este botón estaremos eliminando todas las condiciones de filtro de una única vez evitando tener que recorrernos una a una todas las condiciones y pulsando en su botón *Clear* correspondiente.

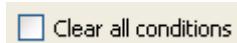


Ilustración 231: Vista Refactoring Catalog Browser – Clear All Conditions

Una vez que hemos visto las distintas posibilidades que nos ofrece esta parte de la interfaz, vamos a ver con afecta la desactivación de una de las condiciones del filtro en la visualización el catálogo de refactorizaciones.

En el ejemplo que hemos visto anteriormente desactivaremos la condición `key:annotation` y observaremos que:

- aparece la condición marcada como desactivada y el nombre de la misma aparece atenuado su visibilidad, en color grisáceo.
- en el árbol que representa el catálogo de refactorizaciones clasificado, en la categoría *Method* ya no aparece únicamente *AddOverrideAnnotation_Usuario* sino que también aparecen las refactorizaciones *AddOverrideAnnotation*, *Add Parameter_Usuario* y *Add Parameter*, las cuales no contienen la palabra clave *annotation*. Además comprobaremos que estas han desaparecido del grupo de filtrados, de la categoría *Method* del grupo *Filtered*.

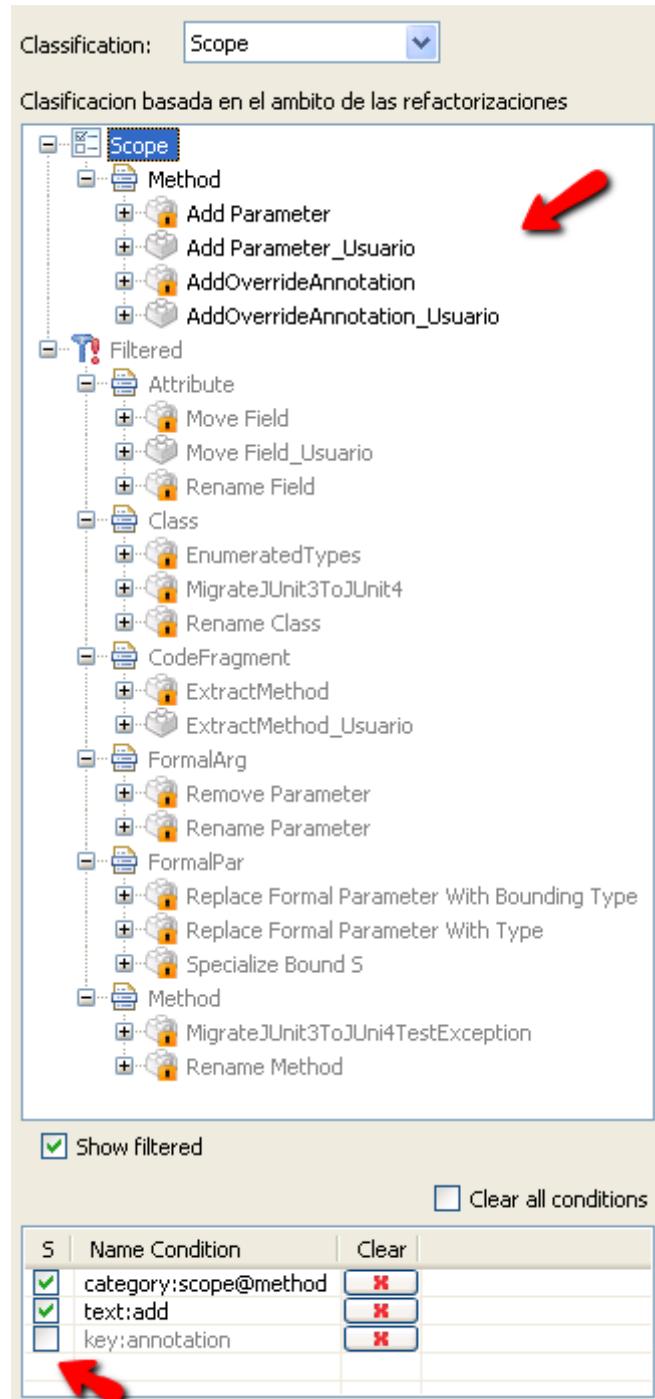


Ilustración 232: Vista Refactoring Catalog Browser – Filter
II

4.3. Summary Panel

El panel *Summary Panel* se sitúa en la zona derecha de la vista y se muestra cuando una refactorización es seleccionada y se hace doble clic sobre ella en el árbol que representa el catálogo de refactorizaciones. Este panel de resumen está destinado a la visualización de la información asociada a una refactorización y consta de una etiqueta en la que indica la refactorización de la que se muestra la información y una serie de pestañas en las que se encuentra repartida la información asociada a la misma. Siempre aparecerán tres pestañas; *Overview*, *Inputs* y *Mechanism* y también podrán aparecer hasta dos pestañas adicionales que corresponden con *Image* y *Examples*, en caso de que la definición de la refactorización disponga de esta información añadida. A continuación veremos cada una de estas pestañas en detalle.

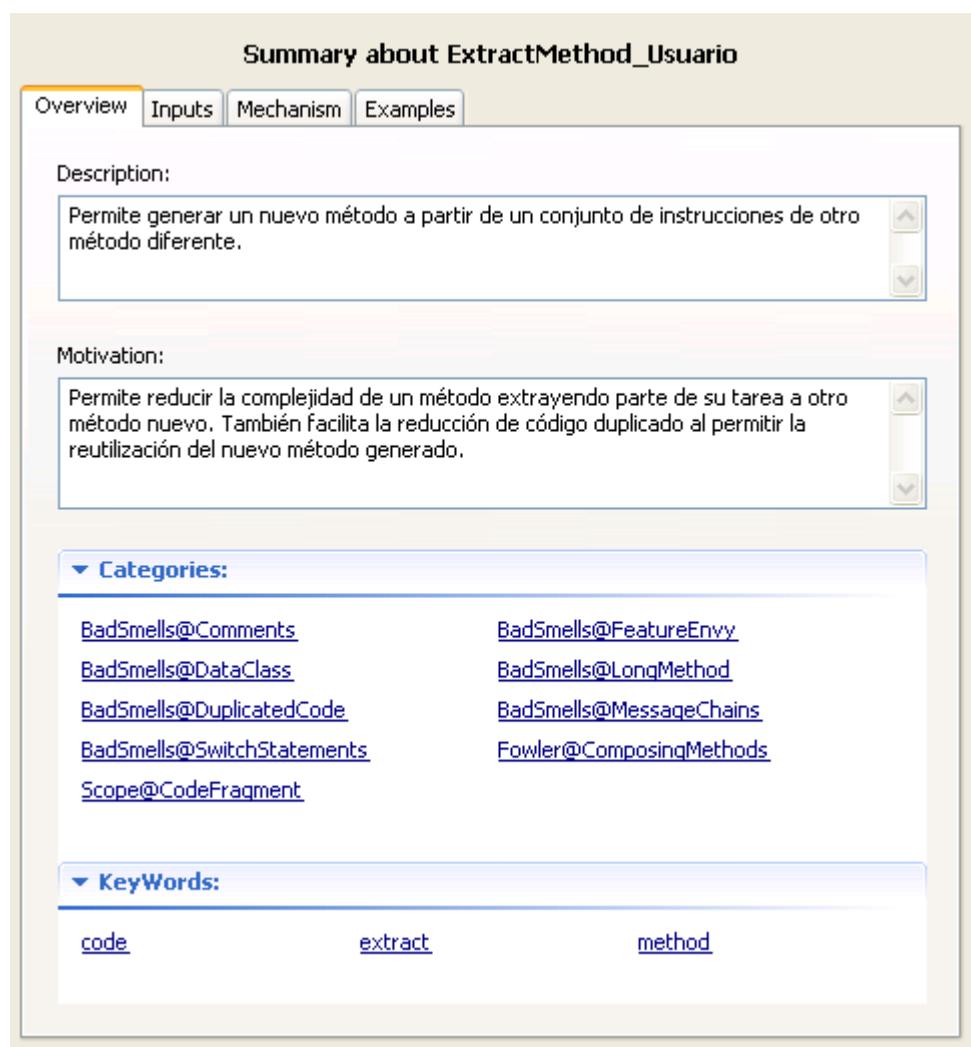


Ilustración 233: Vista Refactoring Catalog Browser – Summary Panel

4.3.1. Overview

Se trata de la primera pestaña del panel *Summary Panel* y corresponde con la vista general de la refactorización seleccionada. Esta contiene la descripción, la motivación, las categorías a las que pertenece y las palabras claves que definen a la refactorización. A continuación vamos a ver cada uno de estos elementos.

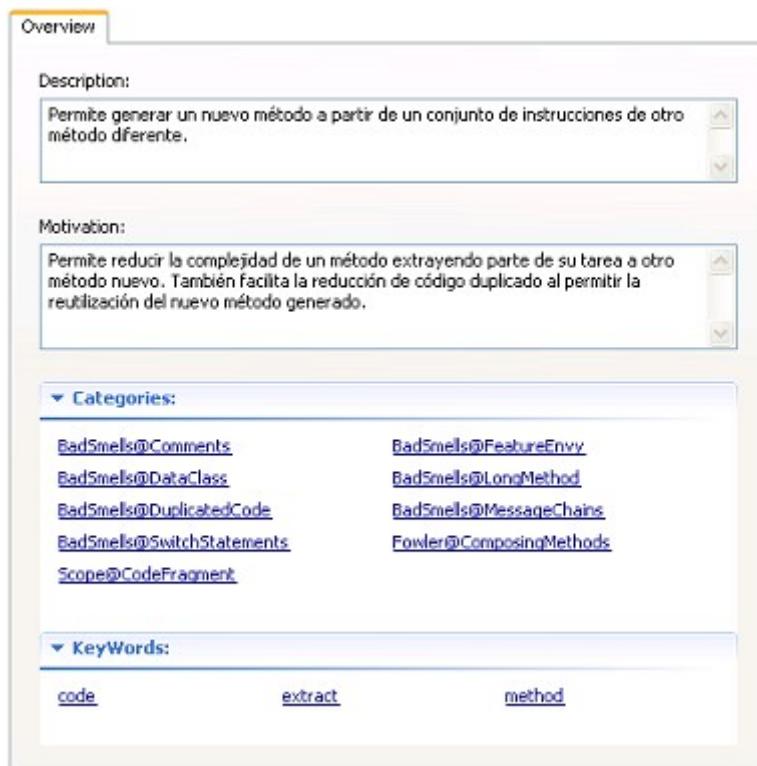


Ilustración 234: Vista Refactoring Catalog Browser – Pestaña Overview

Description

Permite visualizar la descripción asociada a la refactorización. En ella se indica para qué sirve la misma.

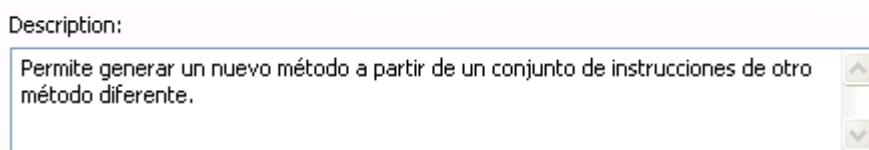


Ilustración 235: Vista Refactoring Catalog Browser – Description

Motivation

Permite visualizar la motivación asociada a la refactorización. En ella se indica la motivación que llevará a aplicar esta refactorización.

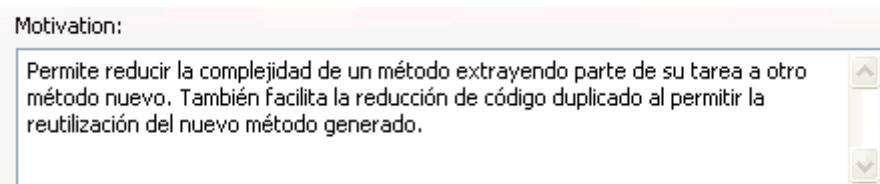


Ilustración 236: Vista Refactoring Catalog Browser – Motivation

Categories

Permite visualizar las categorías a las que pertenece la refactorización para cada una de las clasificaciones que se hayan utilizado en su definición. Siempre aparecerá la categoría a la que pertenece la refactorización para la clasificación *Scope*, debido a que es obligatoria su definición ya que deberá coincidir con el ámbito para el cual se encuentra definida la refactorización.

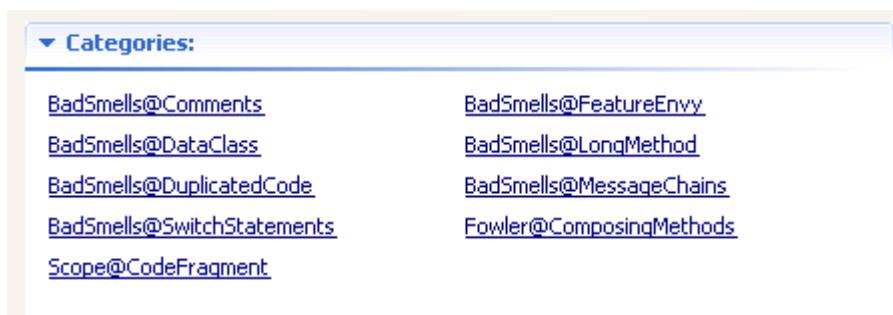


Ilustración 237: Vista Refactoring Catalog Browser – Categories

La representación que se utiliza para mostrar las categorías es la siguiente:

'clase'@'categoría'

Además las categorías aparecen visualizadas mediante enlaces o *links*, los cuales si son pulsados permiten añadir de forma automática la condición de tipo categoría al filtro aplicado al catálogo de refactorizaciones. En este caso, el único error que puede aparecer será debido a la existencia de un filtro idéntico, ya que por sintaxis o datos

incorrectos no podrá darse el caso por utilizar esta vía para su adición.

Por ejemplo, si intentamos añadir dos veces la misma categoría pulsando sobre el link Scope@CodeFragment obtendremos el siguiente aviso.

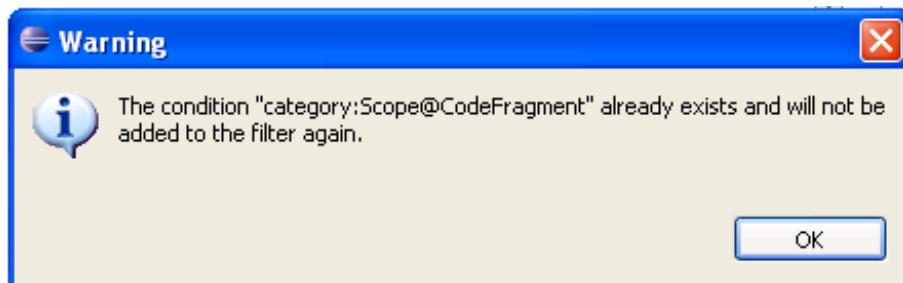


Ilustración 238: Vista Refactoring Catalog Browser – Categories Error

Keywords

Permite visualizar las palabras clave que han sido asociadas a la refactorización.



Ilustración 239: Vista Refactoring Catalog Browser – Keywords

Al tratarse de una información adicional a la hora de definir la refactorización puede que no exista información relativa a palabras clave. En tal caso aparecerá cerrada la sección de *keywords*, como en la siguiente ilustración se puede observar.

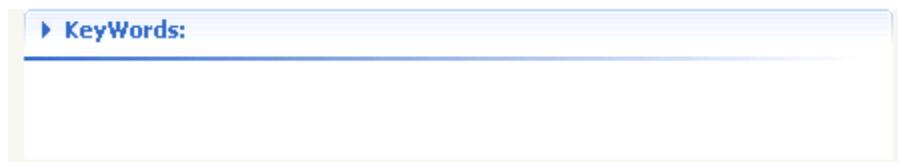


Ilustración 240: Vista Refactoring Catalog Browser – Sin Keywords

Además, también estas palabras clave aparecen representadas mediante enlaces o *links*, los cuales si son pulsados permiten añadir de forma automática la condición de tipo palabra clave al filtro aplicado al catálogo de refactorizaciones. De igual forma que ocurría con las categorías, el único error que puede aparecer será debido a la existencia de un filtro idéntico. Por ejemplo, si intentamos añadir dos veces la misma palabra clave pulsando sobre el link code obtendremos el siguiente aviso.

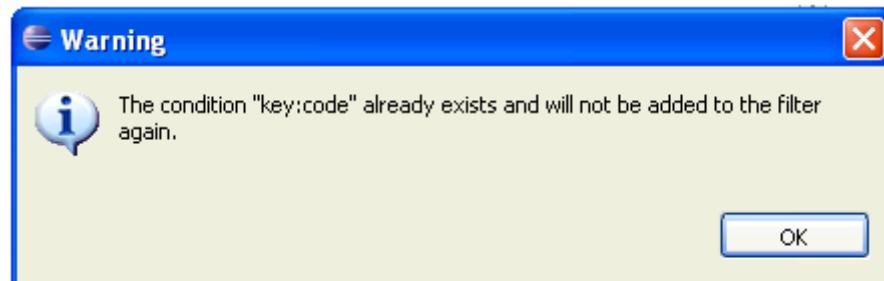


Ilustración 241: Vista Refactoring Catalog Browser – Keywords Error

4.3.2. *Inputs*

Se trata de la segunda pestaña del panel *Summary Panel* y corresponde con las entradas definidas para la refactorización que se encuentra seleccionada. Estas entradas aparecen representadas mediante filas en una tabla y se dispone de ellas el nombre, el tipo de entrada, a partir de donde se obtiene el valor o valores posibles, y un indicador de si se trata de la entrada principal.

Inputs			
Name	Type	From	M
Name	moon.core.Name		<input type="checkbox"/>
Fragment	javamoon.core.instruction.JavaCodeFragment		<input checked="" type="checkbox"/>
Model	moon.core.Model		<input type="checkbox"/>

Ilustración 242: Vista Refactoring Catalog Browser – Pestaña Inputs

4.3.3. Mechanism

Se trata de la tercera pestaña del panel *Summary Panel* y contiene los mecanismos definidos para la refactorización seleccionada, es decir, la lista de precondiciones, acciones y postcondiciones.

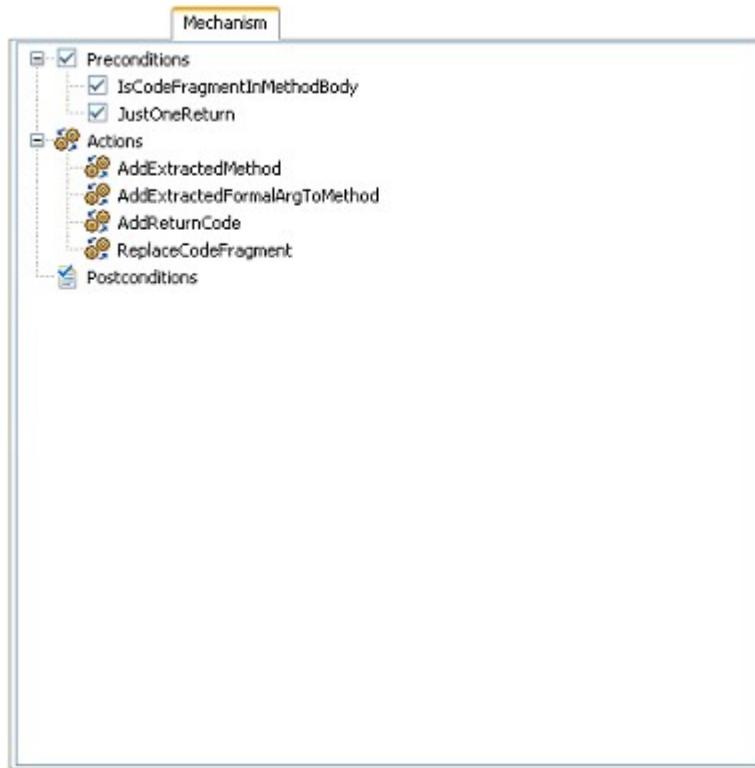


Ilustración 243: Vista Refactoring Catalog Browser – Pestaña Mechanism

4.3.4. Image

Se trata de una pestaña opcional del panel *Summary Panel* y muestra la imagen representativa de la refactorización seleccionada, la cual ha sido incluida en la definición de la misma. En caso de que no haya una imagen disponible la pestaña no aparecerá, de ahí que sea opcional.

En este caso como la refactorización que se está utilizando, *ExtractMethod_Usuario*, para visualizar las distintas pestañas no dispone de imagen representativa mostraremos la

pestaña *Image* de la refactorización *AddOverrideAnnotation_Usuario*.

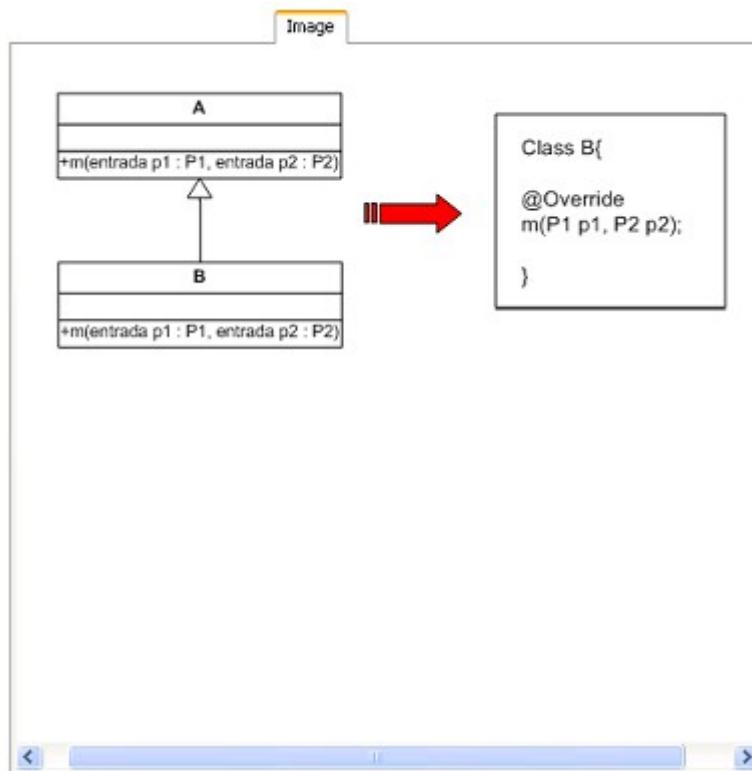


Ilustración 244: Vista Refactoring Catalog Browser – Pestaña *Image*

4.3.5. Examples

Se trata de la última pestaña que puede aparecer en el panel *Summary Panel* y también se trata de una pestaña opcional. Es la encargada de mostrar los enlaces o *links* a los ejemplos disponibles para la refactorización que ha sido seleccionada.

Al igual que ocurre con la imagen representativa de la refactorización, para que la refactorización disponga de estos ejemplos previamente habrán tenido que ser establecidos en la definición de la refactorización, en el proceso de creación o edición de la misma.

Para poder visualizar estos ejemplos bastará con pulsar sobre el *link* del ejemplo elegido y acto seguido aparecerá una ventana en la que se mostrará por una parte el código fuente del ejemplo antes de aplicar la refactorización y por otra el código fuente resultante de aplicar la refactorización.



Ilustración 245: Vista Refactoring Catalog Browser – Pestaña Examples

4.3.6. Source Viewer Dialog

El visor de código fuente *Source Viewer Dialog* aparece cuando se pulsa sobre alguno de los *links* de ejemplos que aparecen en la pestaña *Examples*. En este visor se pueden identificar claramente dos partes destinadas a la visualización de código fuente; la que se encuentra a la izquierda con el nombre *Original Source* y la que se encuentra a la derecha denominada *Refactored Source*. Ambas presentan el código formateado con sintaxis Java con el objetivo de mejorar su visualización así como facilitar su comprensión.

Original Source

Muestra el código fuente original, es decir, antes de aplicar la refactorización.

Refactored Source

Muestra el código fuente resultante de aplicar la refactorización.

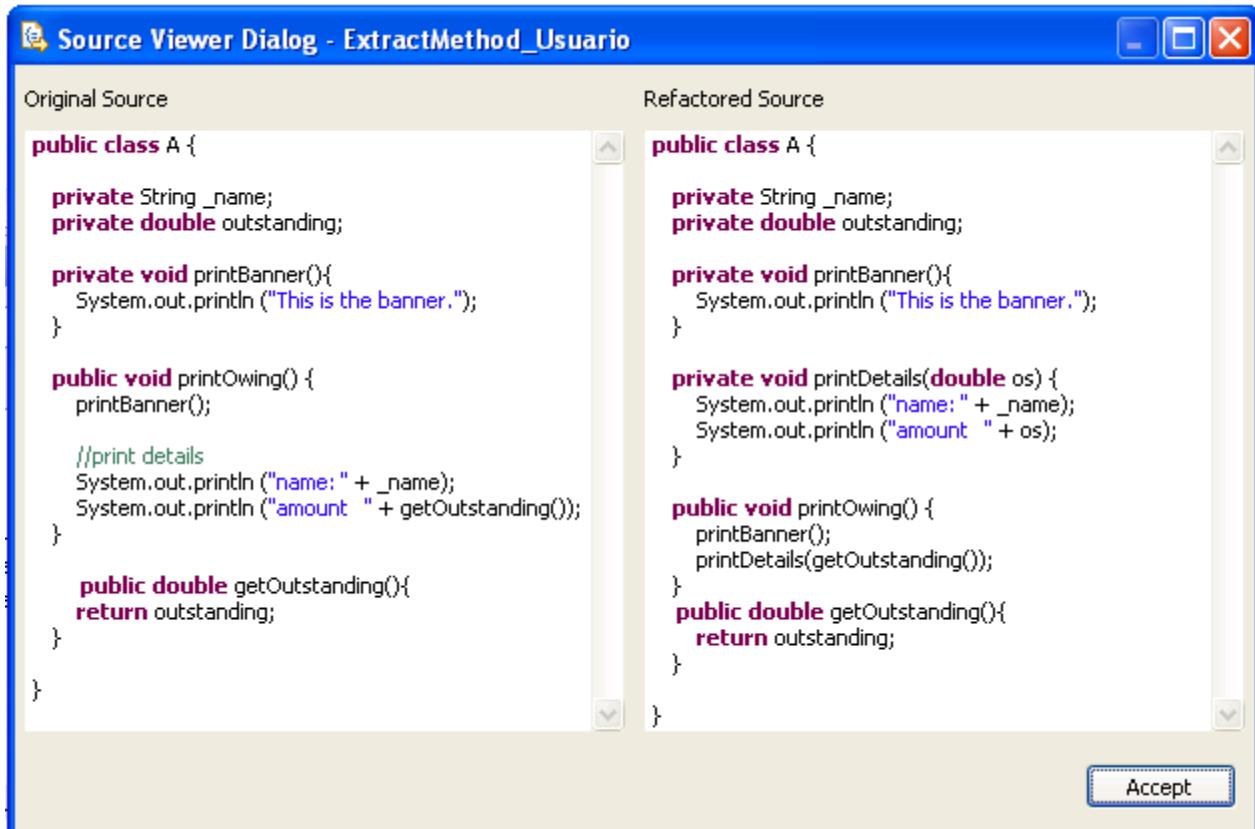


Ilustración 246: Vista Refactoring Catalog Browser – Source Viewer Dialog

Además, cada una de las partes destinadas a la visualización de código fuente con las que cuenta, *Original Source* y *Refactored Source*, dispone de menú contextual cuando se pulsa con el botón izquierdo del ratón sobre los mismos. Este menú contextual dispone de las siguientes acciones:

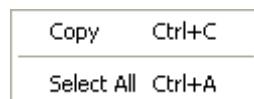


Ilustración 247: Vista Refactoring Catalog Browser – Source Viewer Dialog Menú

Copy

Permite copiar el código que se tenga seleccionado en el visor de código fuente correspondiente. Esta acción únicamente estará disponible si existe código seleccionado, sino se mostrará deshabilitada.

Select All

Permite seleccionar todo el código del visor de código fuente correspondiente.

Teclas rápidas

La relación de combinaciones de teclas rápidas para el visor de código fuente *Source Viewer Dialog* es la siguiente:

5. EDITOR: CLASSIFICATIONS EDITOR

Classifications Editor es un editor de clasificaciones diseñado, además de para poder visualizar las clasificaciones disponibles y la información relativa a las mismas, para permitir crear y modificar clasificaciones propias del usuario, así como definir las categorías que tendrá disponible cada una de ellas.

En este editor se pueden diferenciar claramente tres secciones distintas, que son:

- *Classifications*.
- *Selected Classification*.
- *Categories*.

En la siguiente ilustración se puede apreciar cada una de ellas. A continuación detallaremos su funcionalidad y las distintas opciones que nos ofrecen.

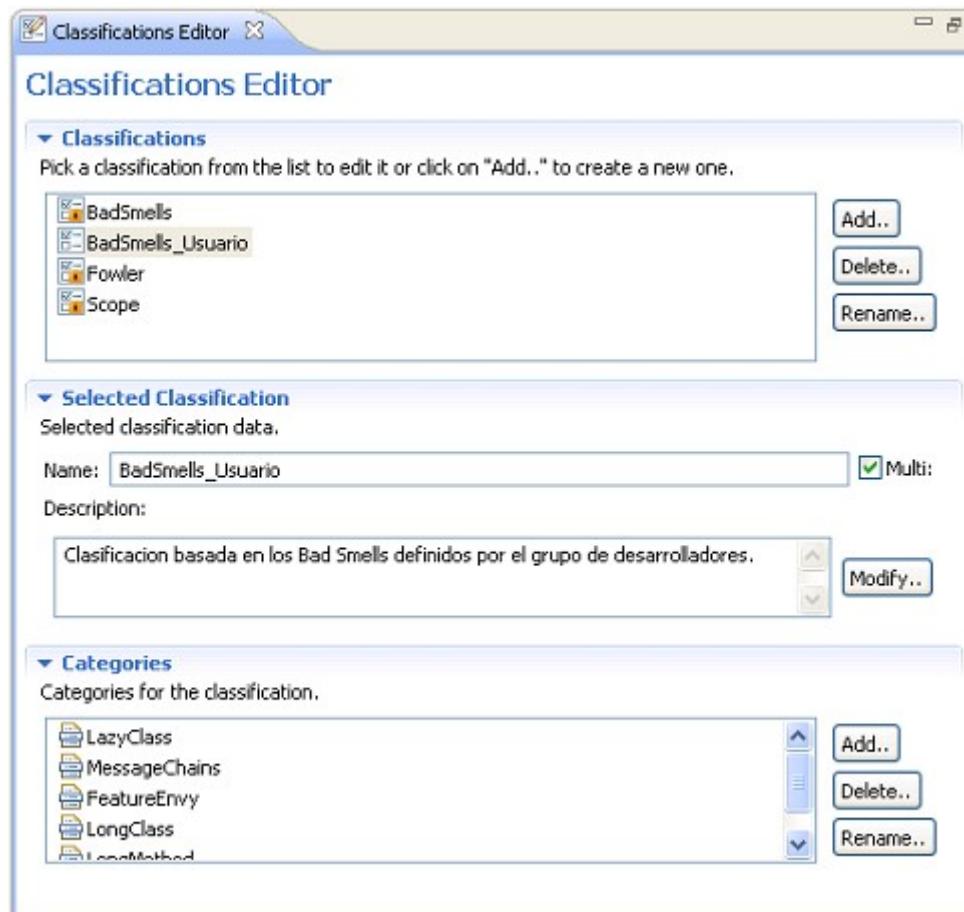


Ilustración 248: *Classifications Editor – Clasificación editable*

En caso de seleccionar una clasificación no editable, es decir, una clasificación que viene suministrada con el plugin la apariencia que tendrá el editor será la que se muestra a continuación:

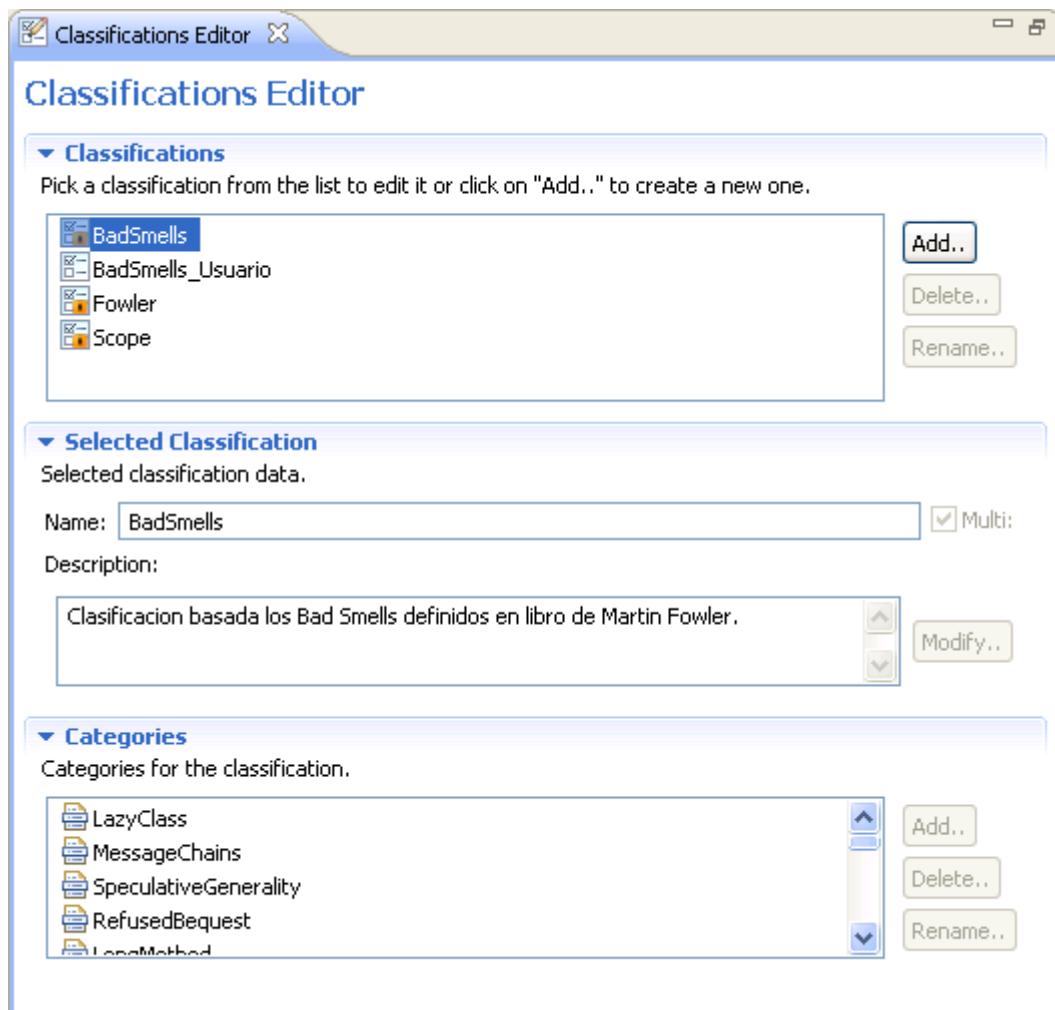


Ilustración 249: *Classifications Editor – Clasificación no editable*

El procedimiento normal a seguir para la creación de una nueva clasificación es el siguiente:

1. En la sección *Classifications*, añadir una nueva clasificación.
2. En la sección *Selected Classification*, especificar la descripción para la clasificación y determinar si se trata o no de una clasificación multicategoría.
3. En la sección *Categories*, añadir todas las categorías de las que se compone la clasificación.

5.1. *Classifications*

La sección *Classifications* se sitúa en la zona superior del editor *Classifications Editor* y esta destinada a la visualización y gestión de clasificaciones.

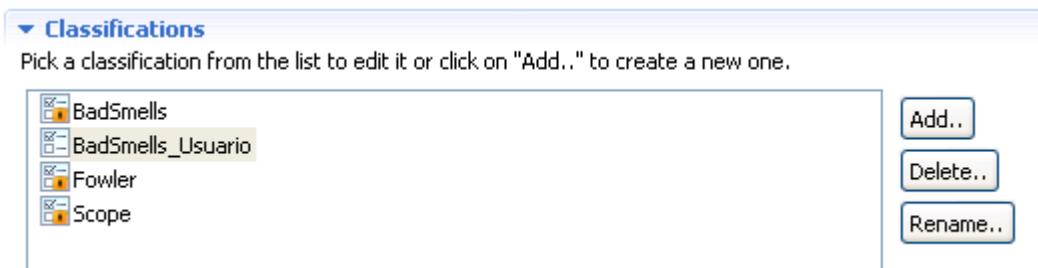


Ilustración 250: Classifications Editor - Sección Classifications

En la parte izquierda de esta sección se podrá visualizar el listado de clasificaciones que se encuentran disponibles, pudiendo identificar mediante el ícono correspondiente las que vienen suministradas con el plugin, y por tanto no editables, de las de definición propia del usuario.

Ícono clasificación suministrada con el plugin (no editable).

Ícono clasificación definida por el usuario (editable).

En la parte derecha de esta sección se puede observar un conjunto de botones que van a permitir realizar las diferentes acciones para la gestión de clasificaciones.



Ilustración 251: Classifications Editor - Botones sección Classifications

Para que puedan estar habilitados, tanto el botón *Delete...* como el *Rename...*, habrá que seleccionar previamente un clasificación que sea editable, es decir, propia del usuario. El botón *Add...* siempre aparecerá habilitado. A continuación vamos a ver cada uno de ellos.

Add **Add..**

Su función es la de añadir una nueva clasificación de usuario a las ya existentes, con la que poder clasificar a las refactorizaciones. Por lo tanto, si se desea realizar esta acción el primer paso será pulsar este botón que dará paso al siguiente dialogo, el cual solicitará el nombre que se quiere dar a la clasificación. Una vez introducido, bastará pulsar el botón *OK* para su creación.

En caso de querer abandonar sin salvar pulsaremos *Cancel* o *Close (X o ESC)*.

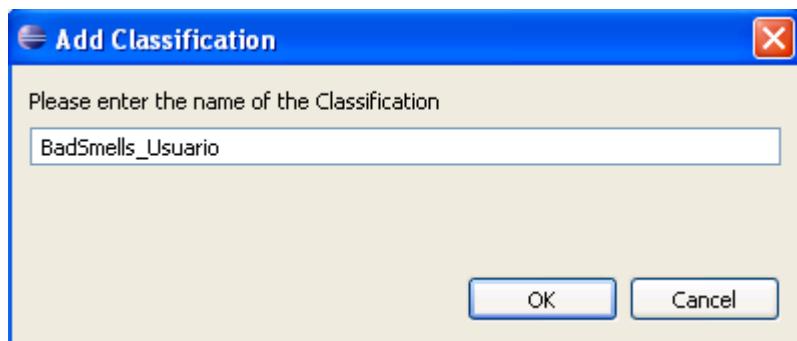


Ilustración 252: *Classifications Editor - Add Classification*

En caso de introducir un nombre que ya está siendo utilizado por otra clasificación la aplicación mostrará el siguiente aviso, no permitiendo su creación mientras no se introduzca un nombre válido.

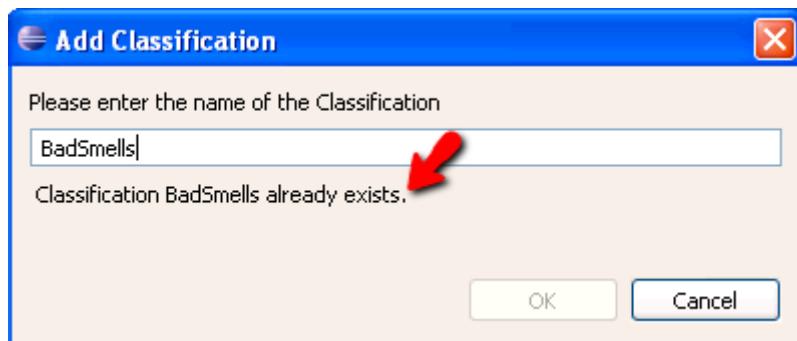


Ilustración 253: *Classifications Editor - Add Classification Error*

Delete

Su función será la de eliminar la clasificación de usuario seleccionada en la lista de clasificaciones disponibles. Se pedirá confirmación previa antes de proceder a su borrado, avisando que al eliminar la clasificación también lo hará con las categorías de esta y por tanto realizará lo propio en la definición de aquellas refactorizaciones que estuviesen clasificadas con alguna categoría de la clasificación a eliminar.

En caso de estar seguro de querer eliminarla pulsaremos el botón *Proceed*, en caso contrario pulsaremos *Cancel* o *Close (X o ESC)*.

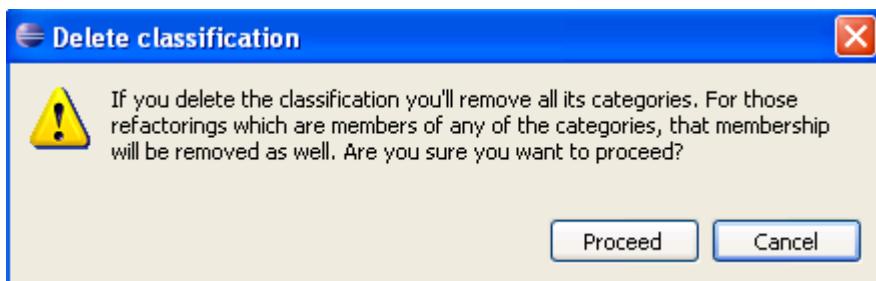


Ilustración 254: *Classifications Editor - Delete Classification*

Rename

Su función será la de renombrar la clasificación de usuario seleccionada en la lista de clasificaciones disponibles. Se introducirá el nuevo nombre que se quiera dar a la clasificación en el cuadro de texto destinado para tal fin y pulsaremos *OK*.

En caso de querer abandonar sin renombrar pulsaremos *Cancel* o *Close (X o ESC)*.

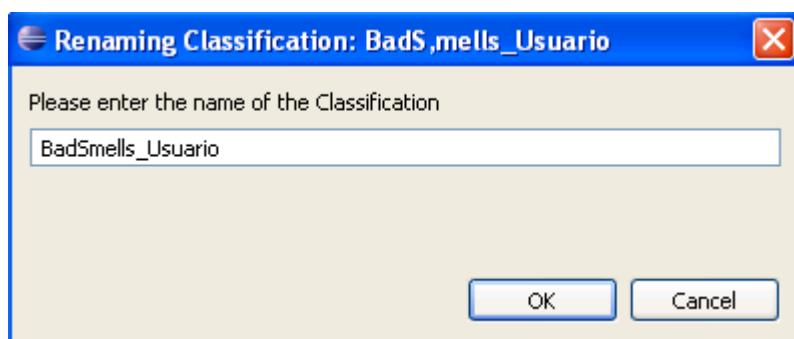


Ilustración 255: *Classifications Editor - Rename Classification*

En caso de introducir un nombre que ya está siendo utilizado por otra clasificación la aplicación mostrará el siguiente aviso, no permitiendo su creación mientras no se introduzca un nombre válido.

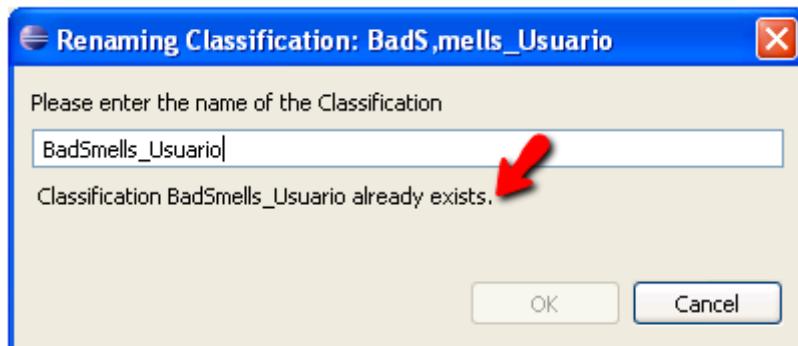


Ilustración 256: *Classifications Editor - Rename Classification Error*

5.2. Selected Classification

La sección *Selected Classification* se sitúa en la zona central del editor *Classifications Editor*, esta destinada a la visualización y edición de la información relativa a la clasificación que se encuentre seleccionada en la sección anterior, *Classifications*. Cabe destacar que la edición de la misma solo estará habilitada en caso de tratarse de una clasificación propia del usuario.

A screenshot of the "Selected Classification" section within the "Classifications Editor". The section has a title bar with a dropdown arrow and the text "Selected Classification". Below it is a sub-section titled "Selected classification data." with the text "Selected classification data." followed by a table. The table has two rows: "Name:" with the value "BadSmells_Usuario" and a checked checkbox "Multi:"; and "Description:" with the value "Clasificación basada en los Bad Smells definidos por el grupo de desarrolladores." and a "Modify.." button. There are also up and down arrows to change the order of the descriptions.

Ilustración 257: *Classifications Editor - Sección Selected Classification*

En esta sección, entre la información relativa a la clasificación que se encuentre

seleccionada se puede distinguir los siguientes campos:

Name

Permite visualizar el nombre de la clasificación de la cual se muestra su información relativa, y de la que se da la posibilidad de editar si así corresponde.

Name:

Ilustración 258: Classifications Editor - Name

Description

Permite visualizar la descripción que ha sido dada a la clasificación que se encuentra seleccionada, con el objetivo de tener un conocimiento mayor de la misma.

Description:

Clasificación basada en los Bad Smells definidos por el grupo de desarrolladores.



Ilustración 259: Classifications Editor - Description

Para permitir establecer o editar la descripción de la clasificación seleccionada se dispone del botón *Modify...* , el cual una vez pulsado dará paso al siguiente dialogo.

En este dialogo estableceremos la descripción en el cuadro de texto disponible para tal fin y acto seguido pulsaremos el botón *OK* para guardar los cambios. En caso contrario, pulsaremos *Cancel* o *Close (X o ESC)*.

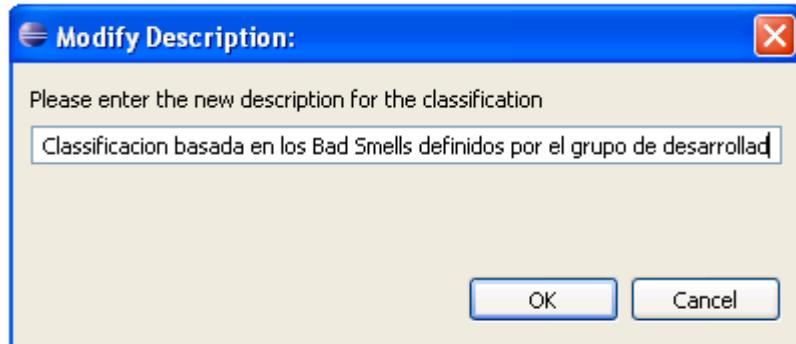


Ilustración 260: *Classifications Editor - Modify Description*

Multicategory Multi:

Permite establecer la propiedad multicategoría en la clasificación que se encuentra seleccionada. Si queremos permitir que las refactorizaciones que sean clasificadas con esta clasificación puedan pertenecer a varias categorías de la misma activaremos esta propiedad, en caso contrario la dejaremos sin seleccionar.

5.3. Categories

La sección *Categories* se sitúa en la zona inferior del editor *Classifications Editor* y esta destinada a la visualización y gestión de las categorías pertenecientes a la clasificación que se encuentre seleccionada en la sección *Classifications*. Cabe destacar que la gestión de categorías únicamente estará habilitada en caso de tratarse de una clasificación propia del usuario.



Ilustración 261: *Classifications Editor - Sección Categories*

En la parte izquierda de esta sección se podrá visualizar el listado de categorías que

se encuentran disponibles para la clasificación que se encuentra seleccionada.

En la parte derecha de esta sección se puede observar un conjunto de botones que van a permitir realizar las diferentes acciones para la gestión de categorías. Para que estos puedan estar habilitados tendremos que seleccionar previamente un clasificación que sea editable, es decir, propia del usuario.



Ilustración 262: Classifications Editor - Botones sección Categories



Su función es la de añadir una nueva categoría a la clasificación de usuario que se encuentra seleccionada, con la que poder clasificar a las refactorizaciones. Por lo tanto, si se desea realizar esta acción el primer paso será pulsar este botón que dará paso al siguiente dialogo, el cual solicitará el nombre que se le quiere dar a la nueva categoría. Una vez introducido, bastará pulsar el botón *OK* para su creación.

En caso de querer abandonar sin salvar pulsaremos *Cancel* o *Close (X o ESC)*.

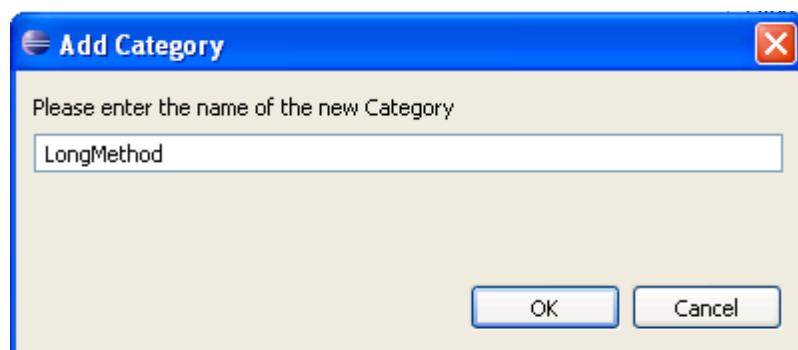


Ilustración 263: Classifications Editor - Add Category

En caso de introducir un nombre que ya está siendo utilizado por otra categoría de la clasificación la aplicación mostrará el siguiente aviso, no permitiendo su creación mientras no se introduzca un nombre válido.

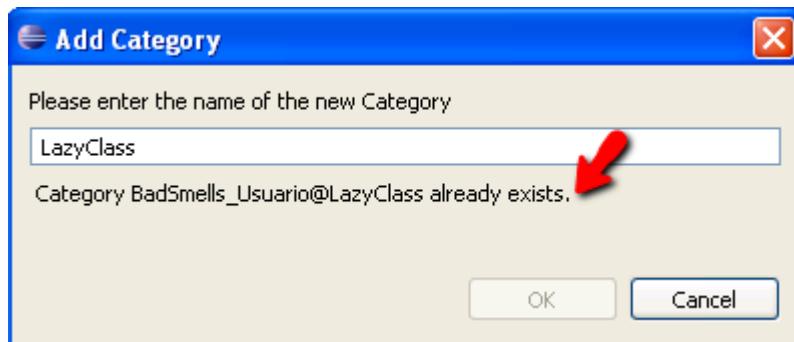


Ilustración 264: Classifications Editor - Add Category Error

Delete

Su función será la de eliminar la categoría seleccionada en la lista de categorías disponibles para la clasificación de usuario elegida. Se pide confirmación previa antes de proceder a su borrado, informando que al eliminar la categoría también realizará lo propio en la definición de aquellas refactorizaciones que estuviesen clasificadas con la categoría a eliminar.

En caso de estar seguro de querer eliminarla pulsaremos el botón *Proceed*, en caso contrario pulsaremos *Cancel* o *Close* (*X* o *ESC*).

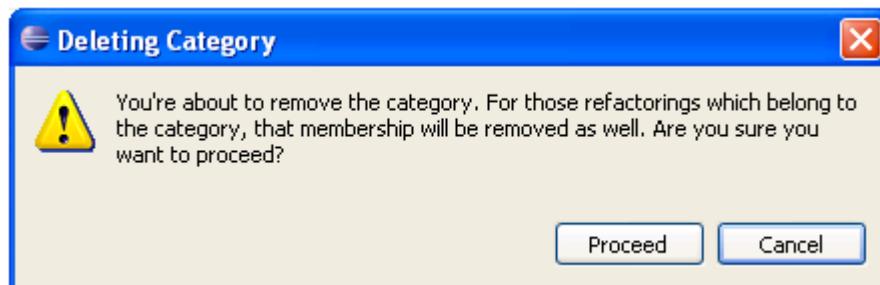


Ilustración 265: Classifications Editor - Delete Category

Rename

Su función será la de renombrar la categoría seleccionada en la lista de categorías disponibles para la clasificación de usuario elegida. Se introducirá el nuevo nombre que se quiera otorgar a la categoría en el cuadro de texto destinado para tal fin y

pulsaremos *OK*.

En caso de querer abandonar sin renombrar pulsaremos *Cancel* o *Close (X o ESC)*.

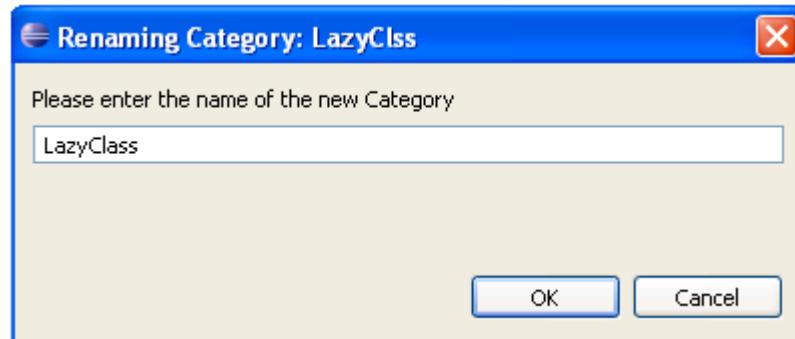


Ilustración 266: *Classifications Editor - Rename Category*

En caso de introducir un nombre que ya esté siendo utilizado por otra categoría de la clasificación la aplicación mostrará el siguiente aviso, no permitiendo su creación mientras no se introduzca un nombre válido.

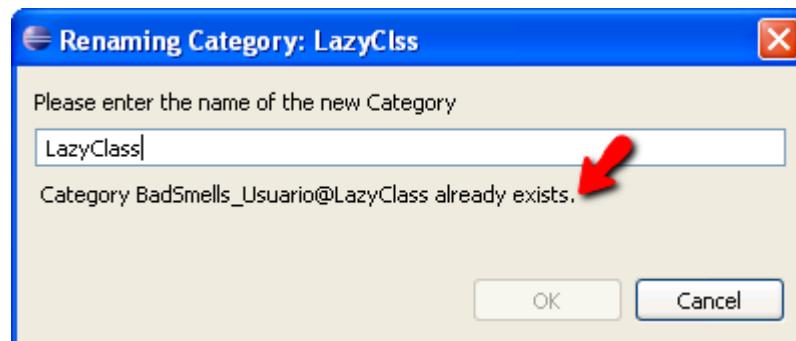


Ilustración 267: *Classifications Editor - Rename Category Error*

6. CONSULTAR AYUDA

Siempre que lo considere necesario el usuario puede acceder a las páginas de ayuda del plugin de refactorizaciones del mismo modo en que lo haría para consultar la ayuda asociada a cualquier otra funcionalidad del propio Eclipse, ya que la ayuda del plugin se encuentra integrada en este.

Para ello, nos dirigiremos al menú *Help > Help Contents* que abrirá la ventana con la ayuda de Eclipse.

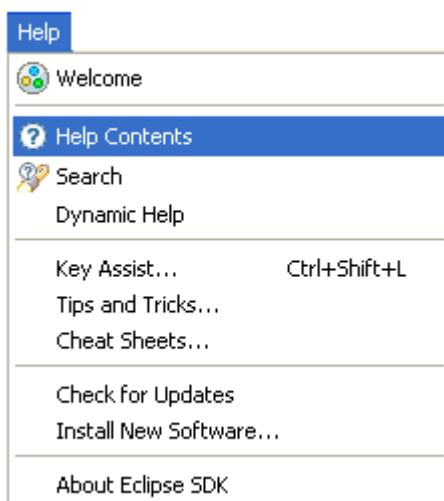


Ilustración 268: *Help > Help Contents*

Una vez abierta la ventana correspondiente a la ayuda de Eclipse, en su parte izquierda se puede observar una tabla de contenidos. Nos dirigiremos a ella con el objetivo de seleccionar la entrada correspondiente al plugin de refactorizaciones, es decir, la que aparece con el nombre *Dynamic Refactoring Plugin*, para acceder al árbol de contenidos de ayuda específico de la aplicación.

Una vez que se ha accedido al árbol de contenidos, ya se puede navegar por él seleccionando aquellos elementos que se consideren de nuestro interés. Además, también se puede hacer uso del resto de funcionalidades que nos ofrece como es la búsqueda de contenidos.

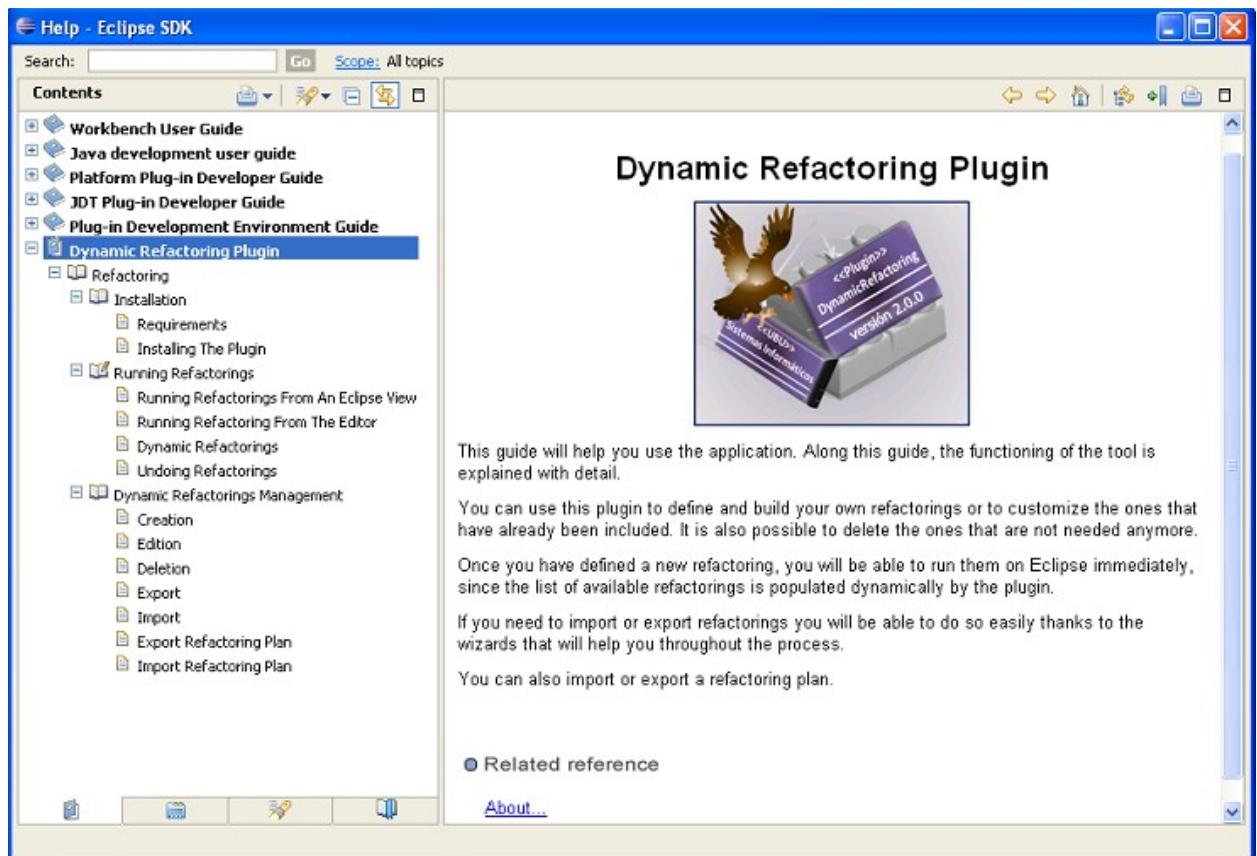


Ilustración 269: Help - Dynamic Refactoring Plugin

7. CONFIGURACIÓN ADICIONAL

Este apartado ha sido realizado con vista a usuarios más avanzados, en él se recordará qué es el catálogo de clasificaciones, dónde se utiliza y cómo se construye, es decir, la estructura del mismo con el fin de hacer más sencilla la tarea de modificación o creación del mismo en caso de que se desee hacer de forma manual.

7.1. Catálogo de clasificaciones

La vista del catálogo de refactorizaciones permite ver las refactorizaciones agrupadas por las categorías a las que pertenecen dentro de una clasificación. Por defecto el plugin dispone de varias clasificaciones predefinidas, como son las clasificaciones *Scope*, *Fowler* y *Bad Smell* pero el usuario puede crear sus propias clasificaciones. Para ello dispone de dos vías o bien realizarlo con el editor de clasificaciones, *Classifications Editor*, disponible para tal efecto o bien hacerlo de forma manual editando el propio fichero XML donde estas clasificaciones de usuario se encuentran almacenadas. En el siguiente apartado de detallará esta segunda vía.

7.2. El fichero XML de clasificaciones

Como hemos comentado anteriormente el catálogo de clasificaciones puede ser ampliado o modificado mediante clasificaciones de usuario. Para ello se editará el fichero `user-classifications.xml` que se encuentra en el siguiente directorio de la instalación de Eclipse: `configuration/dynamicrefactoring.plugin/Classification`. La estructura del fichero XML viene definida por su fichero DTD (*Document Type Definition*) correspondiente que posteriormente se mostrará, ahora detallaremos cada una de sus partes.

En concreto la estructura del fichero consta de un elemento raíz `classifications` que contiene un conjunto de elementos `classification` cada uno de los cuales posee un elemento `categories`. El elemento `categories` tiene a su vez un elemento `category` para cada una de las categorías que conforman la clasificación.

Además, la etiqueta `classification` consta de tres atributos que a continuación van a ser descritos, estos son: `name`, `description` y `multicategory`.

- El atributo `name` contiene el nombre de la clasificación. Este atributo además

de dar nombre a la clasificación también sirve para identificarla de forma única en la clasificación de refactorizaciones.

- El atributo `description` es un texto que debe contener la motivación u origen de la clasificación, el cual es mostrado en algunos apartados de la interfaz del plugin.
- El atributo `multicategory` indica si la refactorización puede o no pertenecer a varias categorías en la clasificación. Es un atributo booleano.

Por ejemplo una refactorización aparecerá en varias categorías de la clasificación *Bad Smell* si dicha refactorización puede servir como medida correctora de varios de los indicadores de defectos en el código. Por contra una refactorización no puede pertenecer a varias categorías según la clasificación de *Fowler* o *Scope*.

El elemento `category` no tiene atributos y su contenido representa el nombre de las categorías de las que consta la clasificación. Este nombre coincidirá con las categorías que se asignan a una refactorización en su definición.

A continuación se muestra el fichero DTD que define su estructura:

```
<!-- ===== Defined Types ===== -->
<!-- A "Boolean" is the string representation of a boolean variable. -->
<!ENTITY % Boolean "(true|false)">

<!-- ===== Top Level Elements ===== -->

<!ELEMENT classifications ( classification+ ) >
<!ATTLIST classifications version CDATA #IMPLIED>

<!ELEMENT classification ( categories+ ) >
<!ATTLIST classification name NMTOKENS #REQUIRED >
<!ATTLIST classification description CDATA #REQUIRED >
<!ATTLIST classification multicategory %Boolean; "false" >

<!ELEMENT categories ( category+ ) >

<!ELEMENT category ( #PCDATA ) >
```

Ilustración 270: DTD de clasificaciones

Por lo tanto para agregar una nueva clasificación a las ya existentes se puede añadir un nuevo elemento `classification` con los atributos adecuados y su correspondiente elemento `categories` al fichero. Por último, dentro del elemento `categories` se creará una etiqueta `category` para cada una de las categorías que se deseen definir para la clasificación en cuestión.

7.3. Exportación de refactorizaciones

Una de las funcionalidades que ofrece el plugin es la de exportar las refactorizaciones dinámicas existentes con el objetivo de que estas se puedan instalar fácilmente en otra instalación de Eclipse que cuente con el plugin de refactorizaciones.

En caso de que se hayan definido clasificaciones propias del usuario y estas se hayan utilizado en la definición de las refactorizaciones que son objeto de exportación se deberá tener muy presente que tendremos que llevarnos consigo el fichero de clasificaciones de usuario, `user-classifications.xml`, el cual se encuentra en el directorio de instalación de Eclipse siguiente: `configuration/dynamicrefactoring.plugin/Classification`, ya que de lo contrario estaremos perdiendo la información de estas.

En el sistema que se realice la importación de dichas refactorizaciones para hacer lo propio con las clasificaciones de usuario nos dirigiremos al mismo directorio de Eclipse pero de la otra instalación: `configuration/dynamicrefactoring.plugin/Classification` y en caso de no existir ninguno simplemente pegaremos el fichero XML. Sin embargo, si ya existiese uno tendríamos que realizar su edición para incluir las nuevas clasificaciones al final del mismo. Una vez realizado este paso manual, ya se podría disfrutar de las nuevas refactorizaciones clasificadas con las clasificaciones de usuario que se acaban de incluir.

BIBLIOGRAFÍA

- Agile Manifesto. *Manifiesto por el Desarrollo Ágil de Software*. Available at: <http://agilemanifesto.org/iso/es/> [Accessed June 5, 2011].
- Alexander, Christopher. 1980. *The timeless way of building*. Second printing. New York: Oxford University Press.
- Amazon Elastic Compute Cloud (Amazon EC2). Available at: <http://aws.amazon.com/es/ec2/> [Accessed March 11, 2011].
- Apache Lucene. *Welcome to Apache Lucene!* Available at: <http://lucene.apache.org/> [Accessed April 11, 2011].
- Apache Maven. “Maven - Introduction to the POM.” Available at: http://maven.apache.org/guides/introduction/introduction-to-the-pom.html#Super_POM [Accessed March 10, 2011].
- Aqrис. “Refactorit - Aqrис - Aqrис wiki.” Available at: <http://www.aqrис.com/display/A/Refactorit> [Accessed March 10, 2011].
- Balsamiq Studios, LLC. “Balsamiq Mockups.” *Balsamiq Mockups | Balsamiq*. Available at: <http://balsamiq.com/products/mockups>. [Accessed March 10, 2011].
- Beck, Kent. 2000. *Extreme programming eXplained : embrace change*. Reading MA: Addison-Wesley.
- Bloch, Joshua. 2008. *Effective Java*. 2nd ed. Upper Saddle River NJ: Addison-Wesley.
- Checkstyle. Available at: <http://checkstyle.sourceforge.net/> [Accessed March 11, 2011].
- Códice Software. “Task driven development.” Available at: <http://www.plasticscm.com/features/task-driven-development.aspx> [Accessed March 10, 2011].
- Dr Dobb’s Journal. “Creating and Destroying Java Objects.” Available at: <http://drdobbs.com/architecture-and-design/208403883?pgno=2> [Accessed June 6, 2011].
- Download - Hudson.war. Available at: <http://hudson-ci.org/downloads/war/> [Accessed May 30, 2011].
- Eclipse Downloads. Available at: <http://www.eclipse.org/downloads/> [Accessed April 5, 2011].
- Eclipse Extension Points and Extensions - Tutorial. Available at: <http://www.vogella.de/articles/EclipseExtensionPoint/article.html> [Accessed May 17, 2011].
- Eclipse Market Place. *Eclipse Plugins, Bundles and Products - Eclipse Marketplace*. Available

at: <http://marketplace.eclipse.org/> [Accessed March 10, 2011].

Eclipse Updates 3.5. Available at: <http://download.eclipse.org/eclipse/updates/3.5/> [Accessed March 11, 2011].

EGit. Available at: <http://www.eclipse.org/egit/> [Accessed March 10, 2011].

ezgraphs. “git_lang_stats.” *git_lang_stats.txt at master from ezgraphs/R-Programs - GitHub*. Available at: https://github.com/ezgraphs/R-Programs/blob/master/git_lang_stats.txt [Accessed March 10, 2011].

FAQ What is a plug-in fragment? - Eclipsepedia. Available at:

http://wiki.eclipse.org/FAQ_What_is_a_plug-in_fragment%3F [Accessed May 17, 2011].

FindBugs™. *FindBugs™ - Find Bugs in Java Programs*. Available at:

<http://findbugs.sourceforge.net/> [Accessed March 11, 2011].

Fowler, Martin. 2000. “Continuous Integration (original version).” Available at:

<http://martinfowler.com/articles/originalContinuousIntegration.html> [Accessed April 11, 2011].

Fuente de la Fuente, Laura. “Plugin de refactorización para Eclipse 2.0.”

Fuente, Sonia, and Enrique Herrero. “Plugin de refactorización para Eclipse.”

Gamma, Erich. 1995. *Design patterns: elements of reusable object-oriented software*. Reading Mass.: Addison-Wesley.

Gerrit. Available at: <http://code.google.com/p/gerrit/> [Accessed March 11, 2011].

GitBenchmarks - Git SCM Wiki. Available at:

<https://git.wiki.kernel.org/index.php/GitBenchmarks> [Accessed March 10, 2011].

Git Community Book. Available at: <http://book.git-scm.com/> [Accessed March 10, 2011].

Git - Fast Version Control System. Available at: <http://git-scm.com/> [Accessed March 10, 2011].

Github. “Help.GitHub - Set Up Git (Linux).” Available at: <http://help.github.com/linux-set-up-git/> [Accessed March 28, 2011a].

Github. “Help.GitHub - Set Up Git (Windows).” Available at: <http://help.github.com/win-set-up-git/> [Accessed March 28, 2011b].

GitHub. *Secure source code hosting and collaborative development - GitHub*. Available at: <https://github.com/> [Accessed March 10, 2011].

Gitorious. Available at: <http://gitorious.org/> [Accessed March 10, 2011].

Google Code. Available at: <http://code.google.com/intl/es-ES/> [Accessed April 24, 2011].

guava-libraries. *guava-libraries - Guava: Google Core Libraries for Java 1.5+ - Google Project*

Hosting. Available at: <http://code.google.com/p/guava-libraries/> [Accessed May 14, 2011].

Help - Eclipse SDK - OSGI bundle manifest. Available at:
[http://help.eclipse.org/helios/index.jsp?
topic=/org.eclipse.platform.doc.isv/reference/misc/bundle_manifest.html](http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/misc/bundle_manifest.html) [Accessed May 17, 2011].

How to run SWTBot tests with Tycho - Tycho - Confluence. Available at:
<https://docs.sonatype.org/display/TYCHO/How+to+run+SWTBot+tests+with+Tycho> [Accessed May 31, 2011].

Hudson Continuous Integration. Available at: <http://hudson-ci.org/> [Accessed April 24, 2011].

Install Sonar - Sonar - Codehaus. Available at:
<http://docs.codehaus.org/display/SONAR/Install+Sonar> [Accessed May 30, 2011].

Integration Hell. Available at: <http://c2.com/cgi/wiki?IntegrationHell> [Accessed March 10, 2011].

Intro Content File XML Format. Available at: [http://help.eclipse.org/helios/index.jsp?
topic=/org.eclipse.platform.doc.isv/reference/extension-points/introContentFileSpec.html](http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/extension-points/introContentFileSpec.html) [Accessed May 28, 2011].

JaCoCo. *EclEmma - JaCoCo Java Code Coverage Library*. Available at:
<http://www.eclemma.org/jacoco/index.html> [Accessed March 11, 2011].

JDeodorant. Available at: <http://www.jdeodorant.com/> [Accessed March 10, 2011].

Kniesel, Günter. “ConTraCT Plugin for Eclipse.” Available at: <http://roots.iai.uni-bonn.de/research/contract/eclipsePlugin> [Accessed March 10, 2011].

López, Carlos, Juan José Rodríguez, and Raúl Marticorena. “Gestión de Trabajos Fin de Carrera.”

Mätzel, Kai-Uwe. 2005. “Safely Manipulating the Contents of Files - How to Get it Right.” Available at:
http://www.eclipsecon.org/2005/presentations/EclipseCon2005_5.1Maetzel.pdf.

Maven Central Repo. *Index of /maven2/*. Available at: <http://repo1.maven.org/maven2/> [Accessed March 10, 2011].

Maven Download. *Maven - Download Maven*. Available at:
<http://maven.apache.org/download.html> [Accessed March 11, 2011].

Maven - Introduction to the Build Lifecycle. Available at:
<http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html> [Accessed March 11, 2011].

Meyer, Bertrand. 1999. *Construcción de software orientado a objetos*. 2nd ed. Madrid [etc.]:

Prentice Hall.

msysgit - Git for Windows - Google Project Hosting. Available at:
<http://code.google.com/p/msysgit/> [Accessed May 30, 2011].

Oracle Technology Network for Java Developers. Available at:
<http://www.oracle.com/technetwork/java/index.html> [Accessed April 4, 2011].

OSGi with Eclipse Equinox - Tutorial. Available at:
<http://www.vogella.de/articles/OSGi/article.html> [Accessed May 17, 2011].

Plastic SCM blog: Branch per task workflow explained. Available at:
<http://codicesoftware.blogspot.com/2010/08/branch-per-task-workflow-explained.html>
[Accessed March 11, 2011].

PMD. Available at: <http://pmd.sourceforge.net/> [Accessed March 11, 2011].

Pro Git - Book. *ProGit - Table of Contents*. Available at: <http://progit.org/book/> [Accessed March 10, 2011].

Query Parser Syntax. *Apache Lucene - Query Parser Syntax*. Available at:
http://lucene.apache.org/java/2_4_0/queryparsersyntax.html [Accessed May 31, 2011].

RefactoringNG — Project Kenai. Available at: <http://kenai.com/projects/refactoringng> [Accessed March 10, 2011].

Reimann, Jan, Mirko Seifert, and Uwe Aßmann. 2010. “Role-based Generic Model Refactoring.”

Reimman, Jan. “Refactory.” Available at: <http://emftext.org/index.php/Refactoring> [Accessed March 10, 2011].

Ries, Eric. 2009. “Continuous deployment in 5 easy steps - O'Reilly Radar.” Available at:
<http://radar.oreilly.com/2009/03/continuous-deployment-5-eas.html> [Accessed March 10, 2011].

Schwaber, Ken. 2004. *Agile project management with Scrum*. Redmond Wash.: Microsoft Press.

Seguridad Social. Available at: http://www.seg-social.es/Internet_1/index.htm [Accessed April 24, 2011].

Sonar. Available at: <http://www.sonarsource.org/> [Accessed March 11, 2011].

Sonar» Download. Available at: <http://www.sonarsource.org/downloads/> [Accessed May 30, 2011].

Spolsky, Joel. “FogBugz.” *FogBugz - Bug & Issue Tracking, Project Management, Help Desk Software | FogBugz from Fog Creek Software*. Available at:
<http://www.fogcreek.com/fogbugz/> [Accessed March 10, 2011].

SWTBot/CI Server - Eclipsepedia. Available at: http://wiki.eclipse.org/SWTBot/CI_Server

[Accessed May 31, 2011].

SWTBot - User Guide. *SWTBot/UsersGuide - Eclipsepedia*. Available at:
<http://wiki.eclipse.org/SWTBot/UsersGuide> [Accessed May 12, 2011].

The GNU General Public License v3.0 - GNU Project - Free Software Foundation (FSF).
Available at: <http://www.gnu.org/licenses/gpl.html> [Accessed May 27, 2011].

Tycho. Available at: <http://www.eclipse.org/tycho/> [Accessed March 11, 2011].

GLOSARIO

En este apartado se recogen algunos de los términos que pudieran necesitar alguna aclaración precisa acerca de su significado y que, por no formar parte del propio trabajo desarrollado en el proyecto o no estar directamente relacionados con sus objetivos, no se hayan explicado en profundidad en el resto de documentos.

API: del inglés *Application Program Interface*. Corresponde con la interfaz que un sistema operativo, una biblioteca o un servicio proporciona para poder dar soporte a las peticiones realizadas por otros programas o otros componentes. Se implementa a través de un conjunto de funciones y procedimientos que proporcionan una capa de abstracción para acceder a la funcionalidad subyacente.

Doclet: es un programa escrito en Java que utiliza el *API* específica de Java para la especificación del contenido y el formato de la salida generada por la herramienta JavaDoc de generación de documentación. El *doclet* que la herramienta utiliza por defecto genera documentación *HTML* en un formato más o menos estándar para la documentación técnica de *APIs*. Sin embargo, existen muchos otros disponibles para obtener otros tipos de salida.

DTD: del inglés *Document Type Definition*. Es uno de los lenguajes de definición de esquemas de *SGML* y *XML*. Se utiliza para definir un esquema mediante el uso de un conjunto de declaraciones que especifiquen las marcas que van a conformar con dicho esquema. Permite así declarar el conjunto de restricciones que debe cumplir un documento cuyo esquema se defina mediante este lenguaje. Los documentos *XML* se describen mediante un subconjunto del lenguaje *DTD* que impone un cierto número de restricciones a la estructura del documento.

GUI: del inglés *Graphical User Interface*. Se trata de un tipo de interfaz de usuario que permite al usuario interactuar con una máquina o aplicación. Su particularidad reside en el hecho de estar basada en una representación visual del sistema, sus procesos y las actuaciones del usuario. Para ello, utiliza iconos, etiquetas y todo tipo de indicadores visuales o elementos gráficos que permiten manipulación directa por parte del usuario.

IDE: del inglés *Integrated Development Environment*. Es un aplicación software que proporciona diferentes tipos de funcionalidad orientadas a la mejora y a la facilitación

del proceso de programación y desarrollo. Normalmente incluye, como mínimo, un editor de código fuente, un compilador o intérprete, herramientas de automatización de procesos de construcción y, habitualmente, además un depurador. Dependiendo del paradigma o el lenguaje al que esté orientado, puede incluir también de forma típica funcionalidad específica del entorno, como puede ser un explorador de objetos, diagramas *UML*, etc.

Locale: conjunto de parámetros que definen el idioma del usuario, el país, y otro conjunto variable de preferencias que el usuario pueda haber querido modificar en relación con la interfaz de usuario de un sistema y, posiblemente, también con el idioma que utiliza. La *locale* utilizada puede afectar al idioma en que se muestra la interfaz, el formato de fecha y hora, la zona horaria, o los formatos utilizados para representar números (separador de miles y decimales, por ejemplo).

OSGI: del inglés *Open Services Gateway initiative framework*. Es un sistema de módulos y una plataforma de servicios para Java. Facilita la modularización de aplicaciones y la separación de responsabilidades por paquetes llamados *bundles*. Los paquetes pueden definir servicios y las implementaciones de dichos servicios pueden ser sustituidas en tiempo de ejecución de forma transparente para los clientes de dichos servicios. Es la plataforma para la modularización que utiliza Eclipse.

Parser: programa o componente, a menudo parte de un compilador o un intérprete, y que recibe una entrada en forma de una secuencia de instrucciones de código, comandos interactivos en línea, etiquetas de marcado, o algún otro tipo de contenido de interfaz definida, y la descompone en partes menores (unidades gramaticales) que puedan ser utilizadas en el procesamiento por parte de algún otro componente.

SDK: del inglés *Software Development Kit*. Se trata de un conjunto de herramientas de desarrollo que permiten crear aplicaciones para un cierto paquete software, para un *framework* determinado, para una plataforma, etc. A menudo se puede reducir simplemente a una *API* con la que acceder a un set de bibliotecas, pero puede incluir también complejas estructuras de acceso a hardware y otros sistemas.

UML: del inglés *Unified/Universal Modeling Language*. Se trata de un lenguaje estándar de especificación visual para el modelado de objetos. El lenguaje *UML* es una herramienta genérica que sirve de cierta notación gráfica para crear modelos abstractos de un sistema, que se denominan *modelos UML*. Abarca diagramas de diversos tipos, entre los que se encuentran los diagramas de clases, de paquetes, de

actividad y de casos de uso.

XML: del inglés *Extensible Markup Language*. Es una especificación de propósito general para la creación de lenguajes de marcado adecuados a dominios específicos. Permite definir elementos propios de cada entorno, de ahí que lleve el calificativo de *extensible*. Su objetivo principal es facilitar la transmisión de datos estructurados entre diferentes sistemas de información.