

**Universidad de Burgos**  
**ESCUELA POLITÉCNICA SUPERIOR**  
**INGENIERÍA INFORMÁTICA**



Anexo 3. Especificación del Diseño

**Dynamic Refactoring Plugin 3.0**

Alumnos:  
Míryam Gómez San Martín  
Íñigo Mediavilla Saiz

Tutor:  
Raúl Marticorena Sánchez

Burgos, Junio de 2011

# **ANEXO III**

## **ESPECIFICACIÓN DEL DISEÑO**



# ÍNDICE DE CONTENIDO

<b><u>Lista de cambios.....</u></b>	<b><u>9</u></b>
<b><u>1. INTRODUCCIÓN.....</u></b>	<b><u>11</u></b>
<b><u>2. DISEÑO DE DATOS.....</u></b>	<b><u>12</u></b>
2.1. DTD de refactorizaciones.....	12
2.2. DTD de clasificaciones.....	15
<b><u>3. DISEÑO ARQUITECTÓNICO.....</u></b>	<b><u>18</u></b>
3.1. Arquitectura lógica.....	19
3.2. Diagramas de clases.....	20
3.2.1. dynamicrefactoring.domain.....	20
3.2.2. dynamicrefactoring.domain.metadata.....	22
3.2.3. dynamicrefactoring.domain.metadata.classifications.xml.imp.....	25
3.2.4. dynamicrefactoring.domain.metadata.condition.....	26
3.2.5. dynamicrefactoring.interfaz.view.....	27
3.2.6. dynamicrefactoring.interfaz.editor.classifeditor.....	28
3.2.7. dynamicrefactoring.interfaz.wizard.classificationscombo.....	29
3.2.8. dynamicrefactoring.interfaz.wizard.search.internal.....	30
3.2.9. dynamicrefactoring.interfaz.wizard.search.javadoc.....	31
<b><u>4. DISEÑO DE LA INTERFAZ.....</u></b>	<b><u>33</u></b>
4.1. Asistente para creación y edición de refactorizaciones.....	33
4.2. Interfaz para la exportación de refactorizaciones.....	43
4.3. Vista Available Refactorings.....	45
4.4. Vista Refactoring Catalog Browser.....	47
4.5. Editor de Clasificaciones.....	48
<b><u>5. DISEÑO PROCEDIMENTAL.....</u></b>	<b><u>50</u></b>
5.1. Diagramas de secuencia.....	50
5.1.1. RF 1: Visualizar refactorizaciones según clasificación.....	51
5.1.2. RF 2: Refrescar visualización de refactorizaciones .....	53
5.1.3. RF 3: Añadir filtro de refactorizaciones .....	55
5.1.4. RF 4: Seleccionar opción aplicar filtro .....	57
5.1.5. RF 5: Eliminar filtro de refactorizaciones .....	59
5.1.6. RF 6: Eliminar todos los filtros de refactorizaciones .....	61
5.1.7. RF 7: Seleccionar opción ver refactorizaciones filtradas .....	63
5.1.8. RF 8: Visualizar detalle refactorización .....	65
5.1.9. RF 9: Añadir clasificación .....	67
5.1.10. RF 10: Editar clasificación .....	69
5.1.11. RF 11: Eliminar clasificación .....	73
5.1.12. RF 12: Añadir categoría a una clasificación .....	75
5.1.13. RF 13: Renombrar categoría de una clasificación .....	77
5.1.14. RF 14: Eliminar categoría de una clasificación .....	79
5.1.15. RF 15: Mostrar resumen elemento seleccionado .....	81

5.1.16. RF 16: Realizar búsqueda de elementos .....	83
<b>6. REPRESENTACIÓN CTTE DE DYNAMIC REFACTORING.....</b>	<b>85</b>
<b>7. PATRONES DE DISEÑO.....</b>	<b>86</b>
7.1. Patrón Objeto Constructor - Builder.....	86
7.2. Patrón Fachada.....	88
7.3. Patrón Singleton.....	89
7.4. Patrón Comando .....	91
<b>8. REFERENCIA CRUZADA CON LOS REQUISITOS.....</b>	<b>94</b>
<b>9. DISEÑO DE PRUEBAS.....</b>	<b>96</b>
9.1. RF 1: Visualizar refactorizaciones según clasificación .....	96
9.2. RF 2: Refrescar visualización de refactorizaciones .....	97
9.3. RF 3: Añadir filtro de refactorizaciones .....	98
9.4. RF 4: Seleccionar opción aplicar filtro .....	99
9.5. RF 5: Eliminar filtro de refactorizaciones .....	99
9.6. RF 6: Eliminar todos los filtros de refactorizaciones .....	100
9.7. RF 7: Seleccionar opción ver refactorizaciones filtradas .....	100
9.8. RF 8: Visualizar detalle refactorización .....	101
9.9. RF 9: Añadir clasificación .....	101
9.10. RF 10: Editar clasificación .....	102
9.11. RF 11: Eliminar clasificación .....	103
9.12. RF 12: Añadir categoría a una clasificación .....	103
9.13. RF 13: Renombrar categoría de una clasificación .....	104
9.14. RF 14: Eliminar categoría de una clasificación.....	104
9.15. RF 15: Mostrar resumen elemento seleccionado .....	105
9.16. RF 16: Realizar búsqueda de elementos .....	105
<b>10. SELECCIÓN DE MÉTRICAS.....</b>	<b>106</b>
<b>11. ENTORNO TECNOLÓGICO DE LA APLICACIÓN.....</b>	<b>110</b>
<b>12. PLAN DE DESARROLLO E IMPLANTACIÓN.....</b>	<b>110</b>
12.1. Diagrama de componentes.....	111
12.2. Diagrama de despliegue.....	117

## ÍNDICE DE ILUSTRACIONES

Ilustración 1: DTD de clasificaciones.....	16
Ilustración 2: Diagrama global de paquetes.....	19
Ilustración 3: D. de clases de dynamicrefactoring.domain.....	20
Ilustración 4: Detalle clase Enumeración Scope.....	22
Ilustración 5: D. de clases de dynamicrefactoring.domain.metadata.....	24
Ilustración 6: D. de clases de domain.metada.classifications.xml.imp.....	25
Ilustración 7: D. de clases de dynamicrefactoring.domain.metadata.condition.....	26
Ilustración 8: D. de clases relevantes de dynamicrefactoring.interfaz.view.....	27
Ilustración 9: D. de clases de dynamicrefactoring.interfaz.editor.classifieditor.....	28
Ilustración 10: D. de clases dynamicrefactoring.interfaz.wizard.classificationscombo.....	29
Ilustración 11: D. de clases de dynamicrefactoring.interfaz.wizard.search.internal.....	30
Ilustración 12: D. de clases de dynamicrefactoring.interfaz.wizard.search.javadoc.....	31
Ilustración 13: Asistente creación/edición refactorizaciones.....	34
Ilustración 14: Selección refactorización antes de modificaciones.....	35
Ilustración 15: Selección refactorización después de modificaciones.....	35
Ilustración 16: Primera página del asistente antes de modificaciones.....	36
Ilustración 17: Primera página del asistente después de modificaciones.....	37
Ilustración 18: Segunda página del asistente antes de modificaciones.....	38
Ilustración 19: Segunda página del asistente después de modificaciones.....	39
Ilustración 20: Páginas 3,4 y 5 antes de modificaciones.....	40
Ilustración 21: Páginas 3, 4 y 5 después de modificaciones.....	41
Ilustración 22: Séptima página del asistente antes de modificaciones.....	42
Ilustración 23: Séptima página del asistente después de modificaciones.....	43
Ilustración 24: Interfaz exportación refactorizaciones antes de modificaciones.....	44
Ilustración 25: Interfaz exportación refactorizaciones después de modificaciones.....	45
Ilustración 26: Vista Available Refactorings antes de modificaciones.....	46
Ilustración 27: Vista Available Refactorings después de modificaciones.....	47
Ilustración 28: Vista Refactoring Catalog Browser.....	48
Ilustración 29: Editor de clasificaciones.....	49
Ilustración 30: D. de secuencia de RF 1: Visualizar ref. según clasificación.....	51
Ilustración 31: D. de secuencia de RF 2: Refrescar visualización de refactorizaciones.....	53
Ilustración 32: D. de secuencia de RF 3: Añadir filtro de refactorizaciones.....	55
Ilustración 33: D. de secuencia de RF 4: Seleccionar opción aplicar filtro.....	57
Ilustración 34: D. de secuencia de RF 5: Eliminar filtro de refactorizaciones.....	59
Ilustración 35: D. de secuencia de RF 6: Eliminar todos los filtros de refactorizaciones.....	61
Ilustración 36: D. de secuencia de RF 7: Seleccionar opción ver ref. filtradas.....	63
Ilustración 37: D. de secuencia de RF 8: Visualizar detalle refactorización.....	65
Ilustración 38: D. de secuencia de RF 9: Añadir clasificación.....	67
Ilustración 39: D. de secuencia de RF 10: Detalle - Renombrar una clasificación.....	69
Ilustración 40: D. de secuencia de RF 10: Detalle - Modificar desc. de clasificación.....	71
Ilustración 41: D. de secuencia de RF 11: Eliminar clasificación.....	73
Ilustración 42: D. de secuencia de RF 12: Añadir categoría a una clasificación.....	75
Ilustración 43: D. de secuencia de RF 13: Renombrar categoría.....	77
Ilustración 44: D. de secuencia de RF 14: Eliminar categoría de una clasificación.....	79
Ilustración 45: D. de secuencia de RF 15: Mostrar resumen elemento seleccionado.....	81
Ilustración 46: D. de secuencia de RF 16: Realizar búsqueda de elementos.....	83
Ilustración 47: D. de clases del PD Builder aplicado a InputParameter.....	88
Ilustración 48: D. de clases del PD Facade aplicado a SearchingFacade.....	89
Ilustración 49: D. de clases del PD Singleton aplicado a EclipseBasedJavadocReader.....	91
Ilustración 50: D. de clases del PD Command aplicado a RefreshViewAction.....	93
Ilustración 51: Diagrama de componentes.....	115

Ilustración 52: Diagrama de despliegue.....	119
---	-----

## ÍNDICE DE TABLAS

Tabla 1: Requisitos funcionales.....	94
Tabla 2: Referencia cruzada de requisitos con módulos de diseño.....	95
Tabla 3: Pruebas de RF1: Visualizar refactorizaciones según clasificación.....	96
Tabla 4: Pruebas de RF 2: Refrescar visualización de refactorizaciones.....	97
Tabla 5: Pruebas de RF 3: Añadir filtro de refactorizaciones.....	98
Tabla 6: Pruebas de RF 4: Seleccionar opción aplicar filtro.....	99
Tabla 7: Pruebas de RF 5: Eliminar filtro de refactorizaciones.....	99
Tabla 8: Pruebas de RF 6: Eliminar todos los filtros de refactorizaciones.....	100
Tabla 9: Pruebas de RF 7: Seleccionar opción ver refactorizaciones filtradas.....	100
Tabla 10: Pruebas de RF 8: Visualizar detalle refactorización.....	101
Tabla 11: Pruebas de RF 9: Añadir clasificación.....	101
Tabla 12: Pruebas de RF 10: Editar clasificación.....	102
Tabla 13: Pruebas de RF 11: Eliminar clasificación.....	103
Tabla 14: Pruebas de RF 12: Añadir categoría a una clasificación.....	103
Tabla 15: Pruebas de RF 13: Renombrar categoría de una clasificación.....	104
Tabla 16: Pruebas de RF 14: Eliminar categoría de una clasificación.....	104
Tabla 17: Pruebas de RF 15: Mostrar resumen elemento seleccionado.....	105
Tabla 18: Pruebas de RF 16: Realizar búsqueda de elementos.....	105

**LISTA DE CAMBIOS**

<b>Número</b>	<b>Fecha</b>	<b>Descripción</b>	<b>Autor/es</b>
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 clasification (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 1: 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 NMTOKENS #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 NMTOKENS (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.

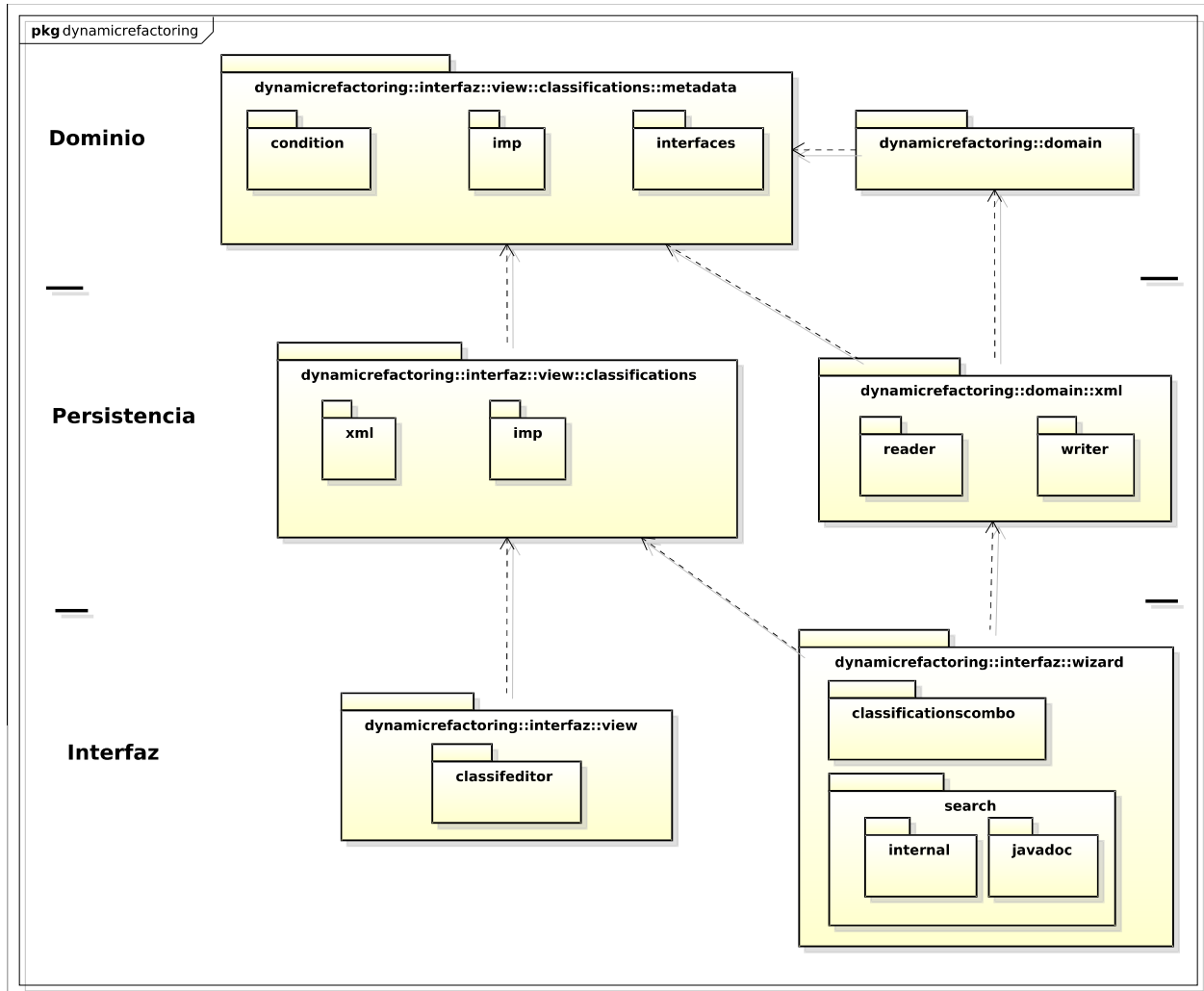
### ***3. DISEÑO ARQUITECTÓNICO***

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 2: 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*

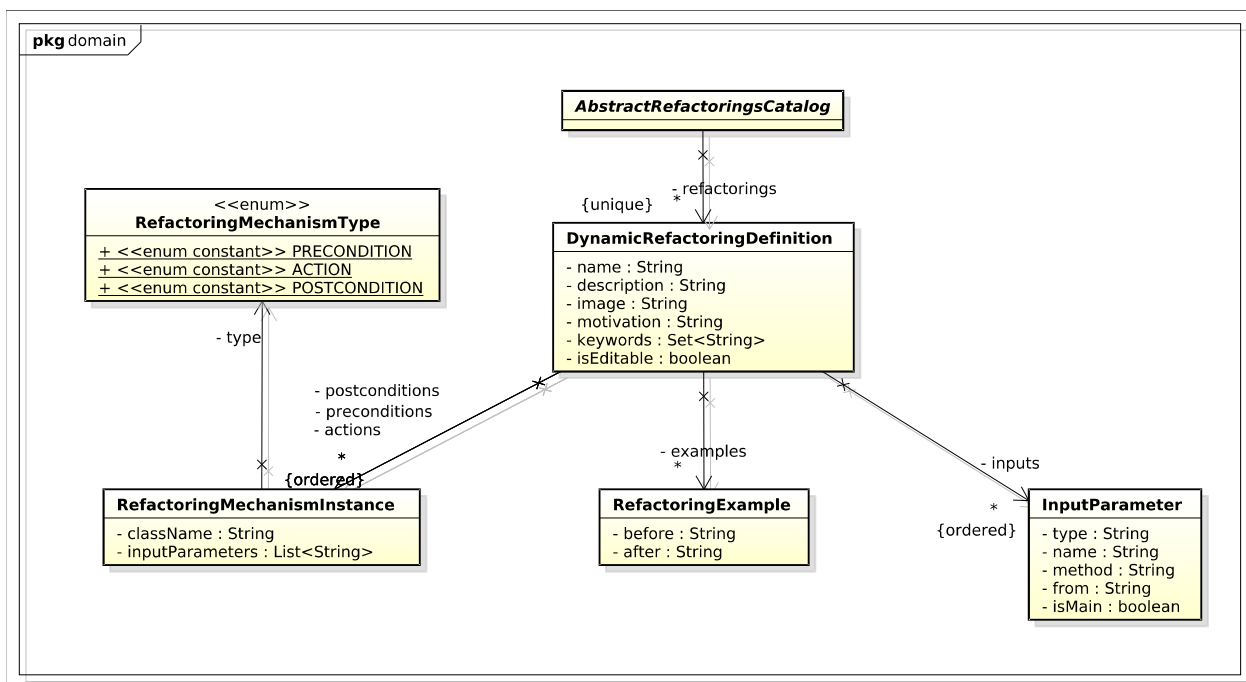


Ilustración 3: D. de clases de *dynamicrefactoring.domain*

En el centro del diagram 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

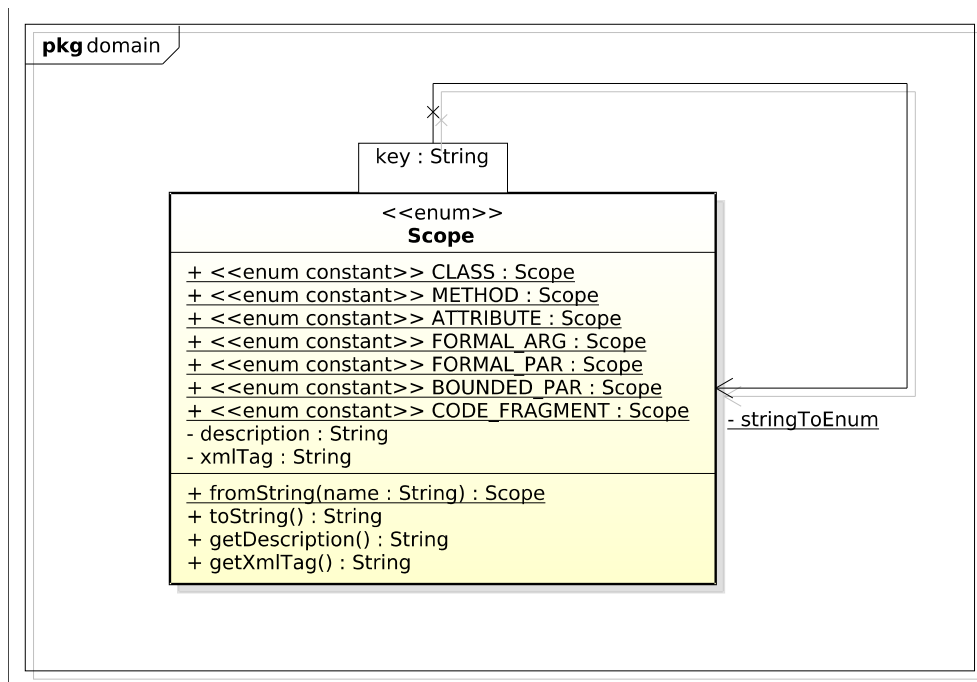
refactorización es editable (pertenece al usuario) o no lo es (pertenece al plugin). Este es un cambio que ha sido necesario para hacer frente al problema del versionado de las refactorizaciones que requería que se crearan dos tipos de refactorizaciones las de usuario y las del plugin.

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 postcondition. 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 4: 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.

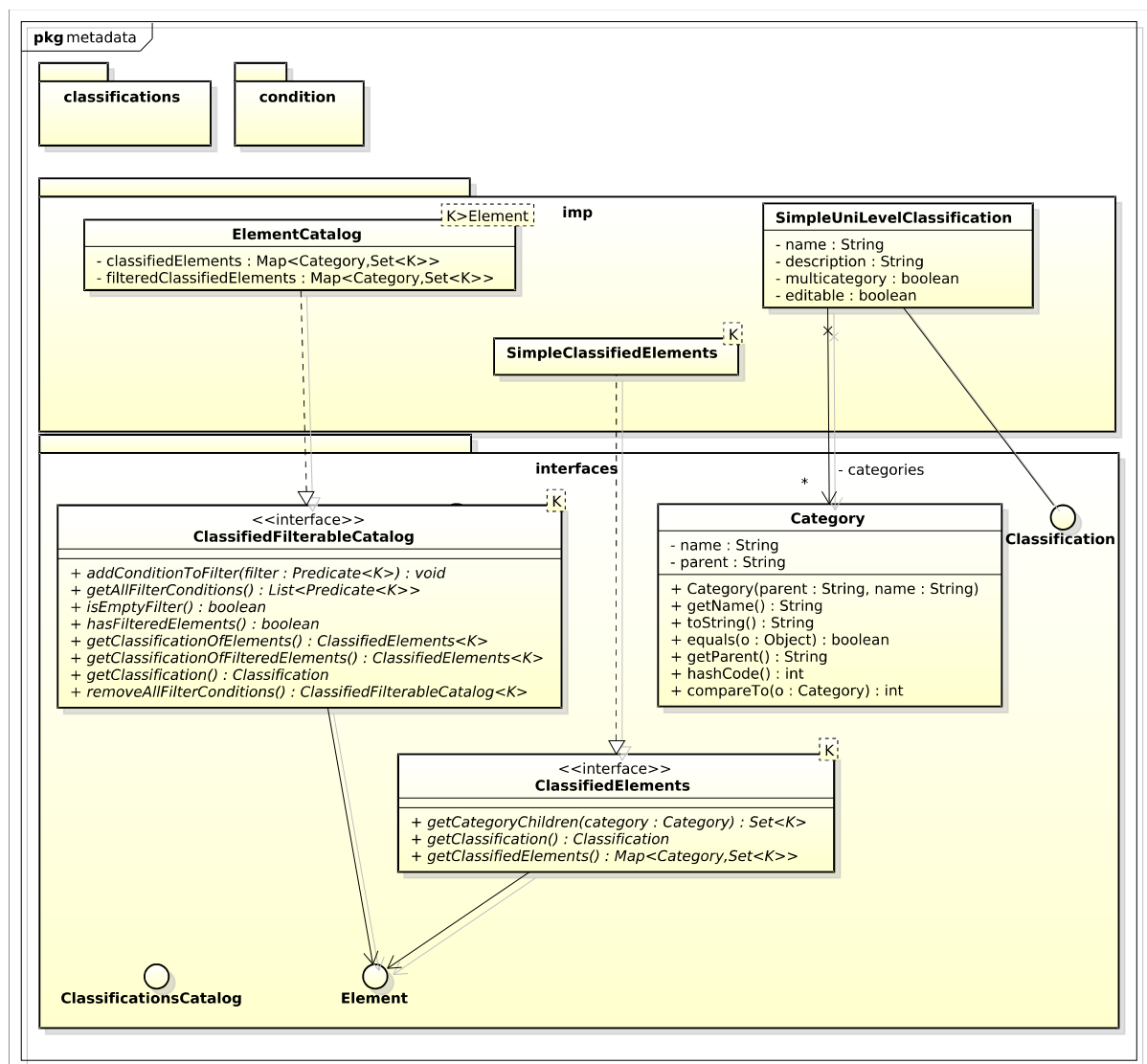


Ilustración 5: D. de clases de `dynamicrefactoring.domain.metadata`

Sobre las interfaces en la figura Ilustración 4 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*

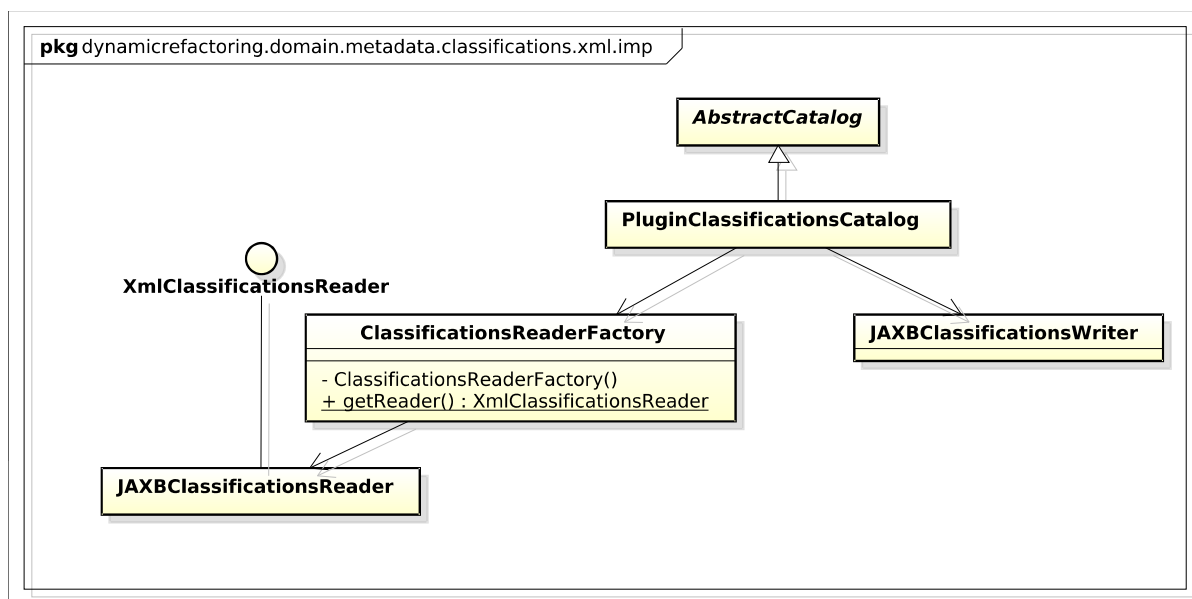


Ilustración 6: D. de clases de *domain.metada.classifications.xml.imp*

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 por tanto 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 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*

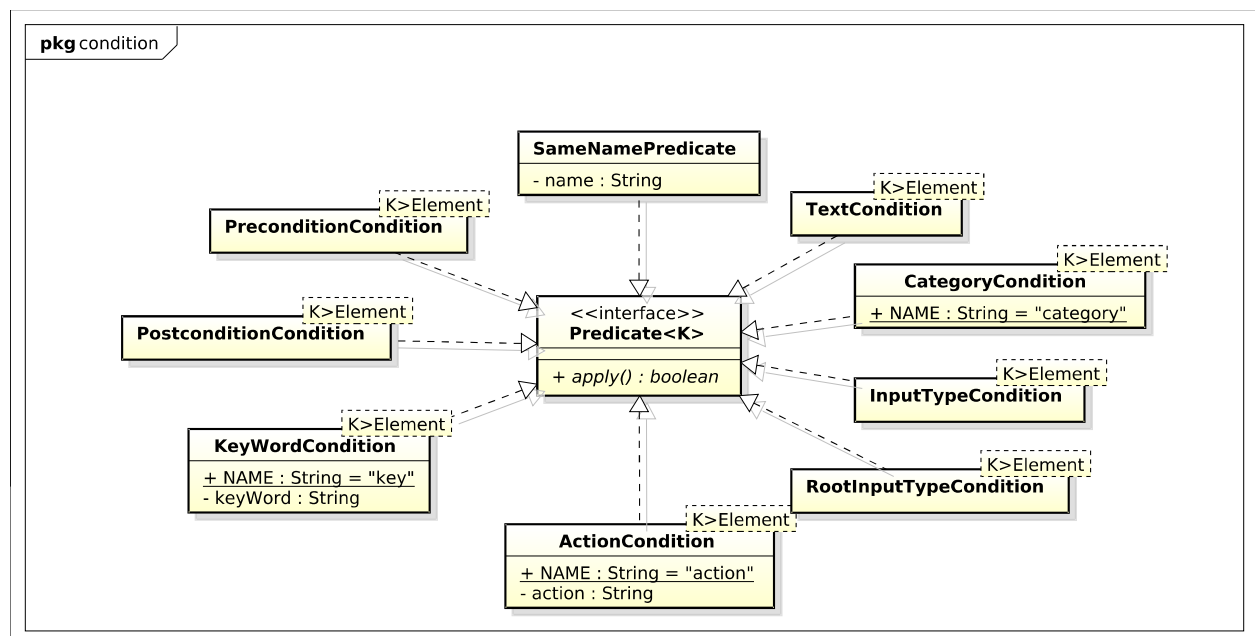


Ilustración 7: D. de clases de *dynamicrefactoring.domain.metadata.condition*

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*

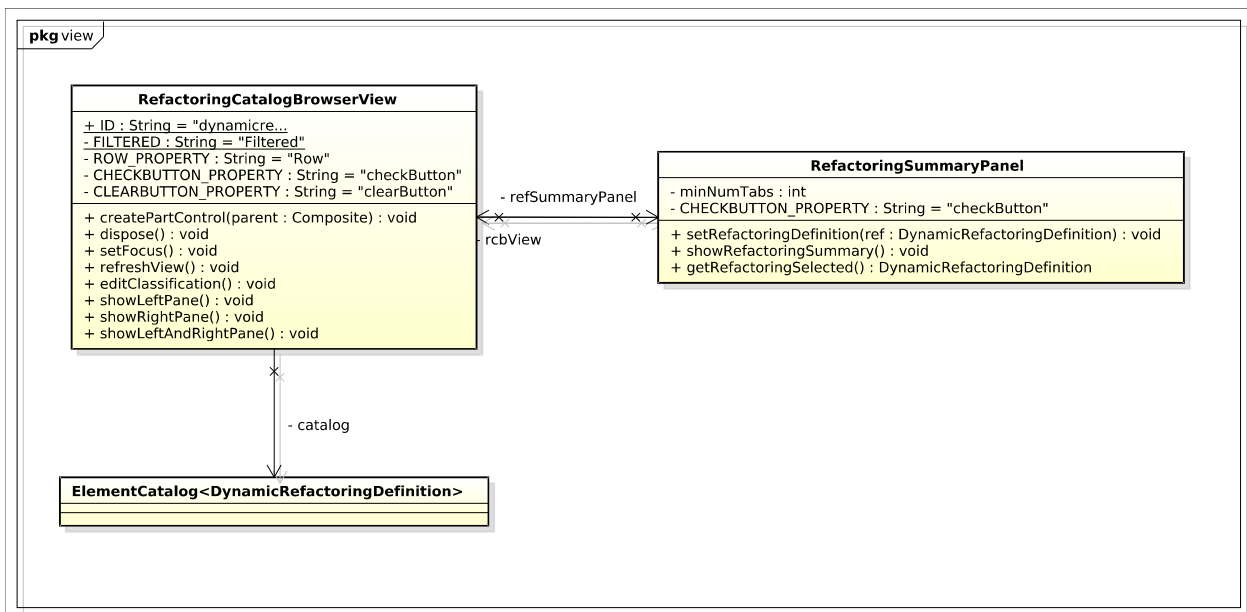


Ilustración 8: D. de clases relevantes de *dynamicrefactoring.interfaz.view*

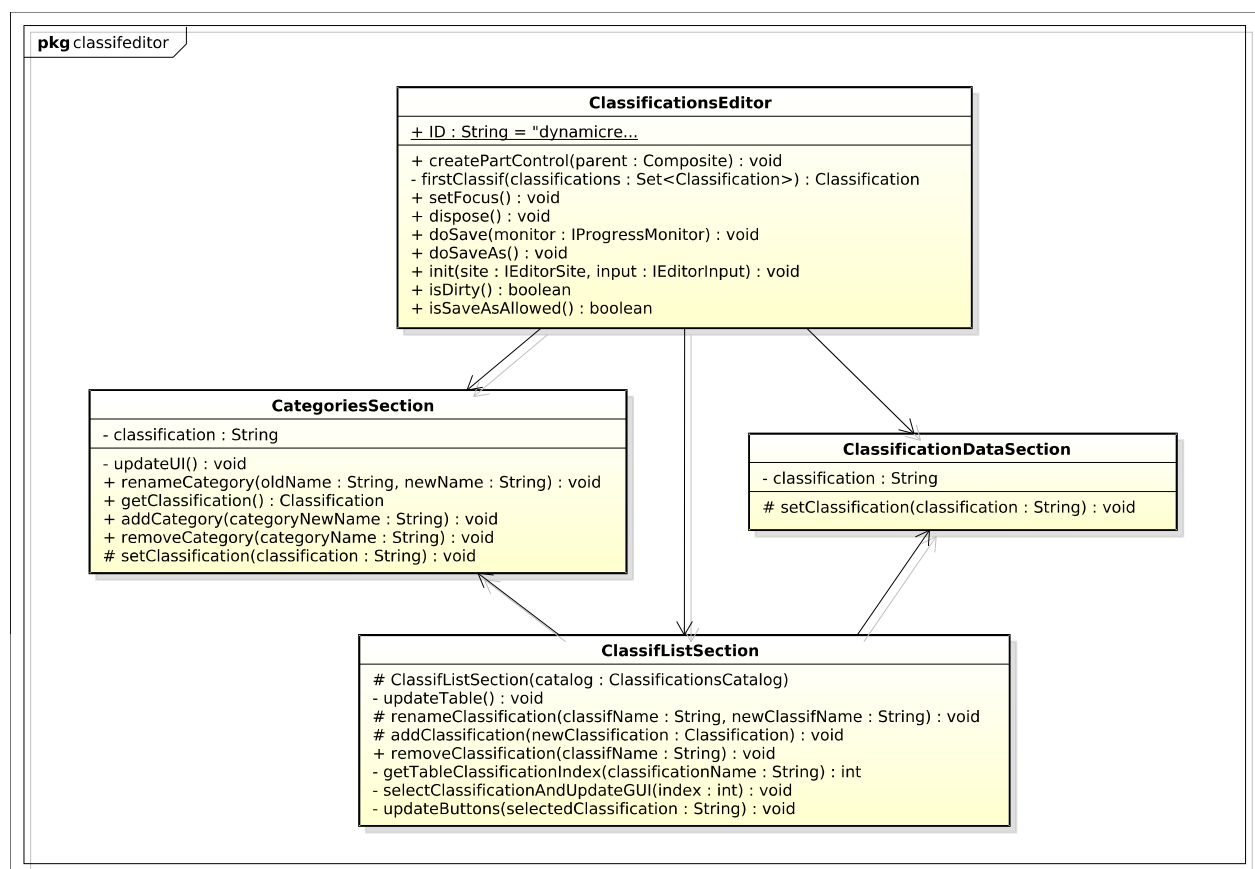
En el diagrama reflejado en Ilustración 14 aparecen las clases más relevantes de las que se han agregado en esta última versión del plugin para el paquete `dynamicrefactoring.interfaz.view`. La clase `RefactoringCatalogBrowserView` representa la vista para el catálogo con todas las refactorizaciones. 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.classifeditor*



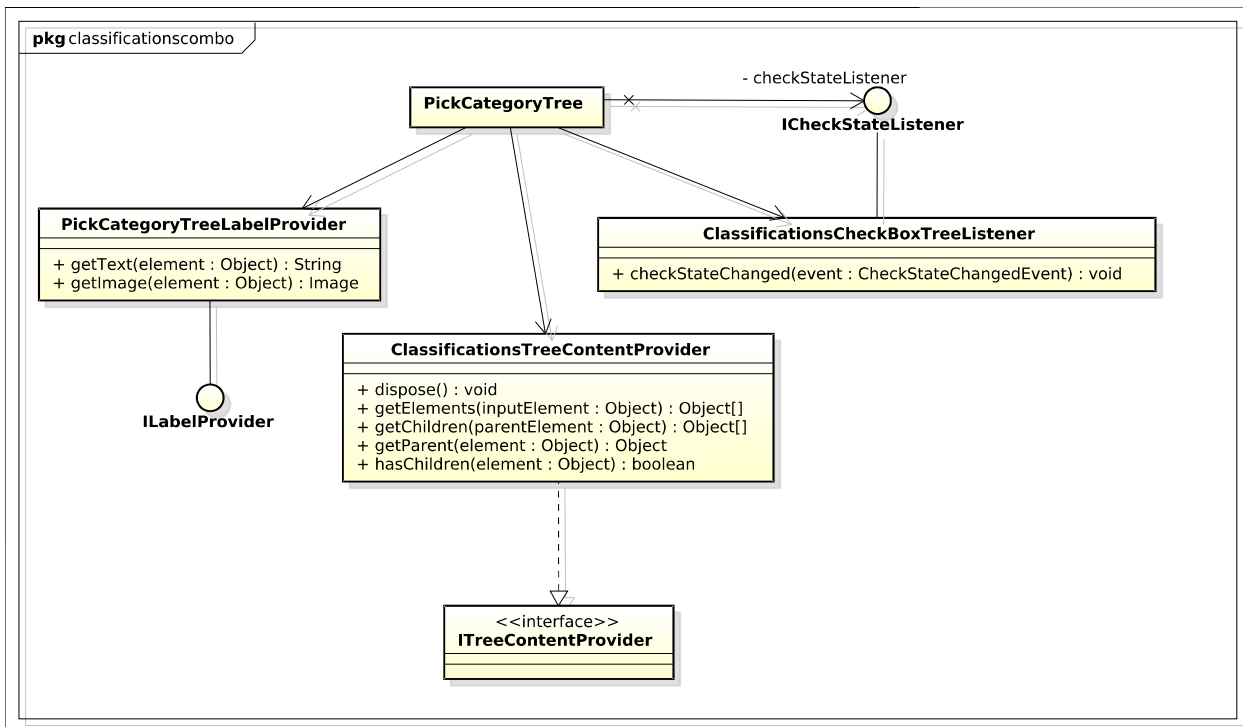
*Ilustración 9: D. de clases de dynamicrefactoring.interfaz.editor.classifeditor*

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 10: 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.

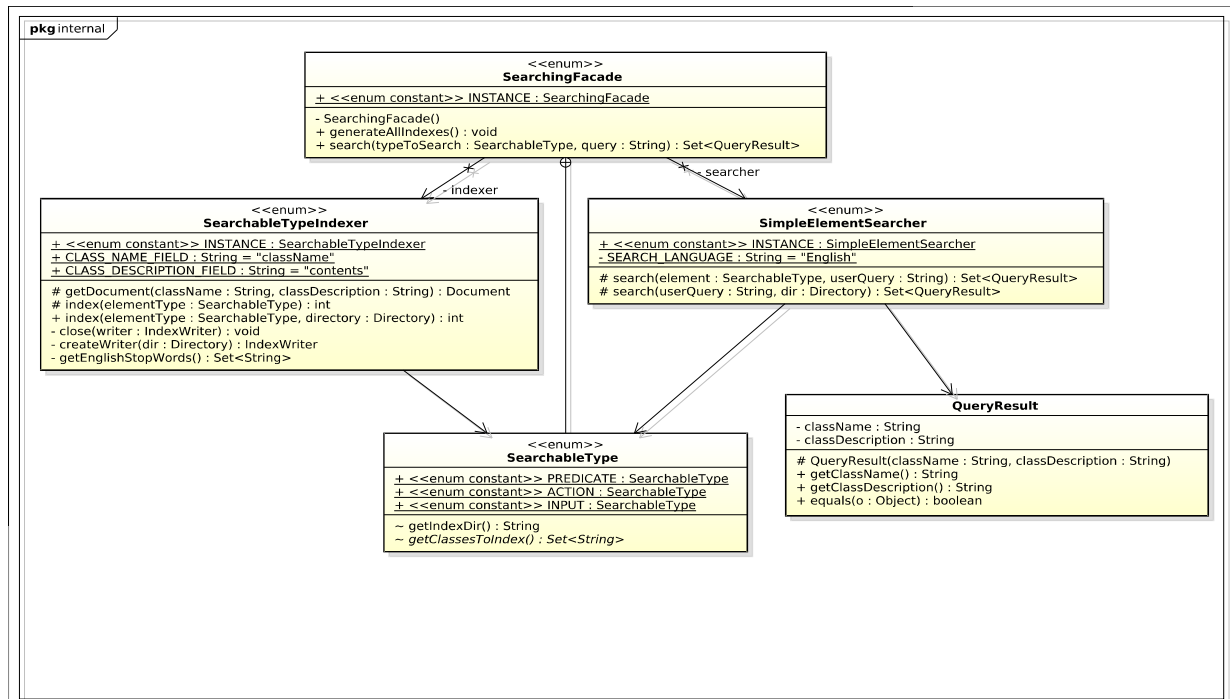
`PickCategoryTree` es la clase principal de este paquete dado que se ocupa de crear el árbol, que es el objeto principal en la elección de categorías, junto con otros objetos que se encargan de gestionar el comportamiento del árbol. Estos objetos son instancias del resto de clases que aparecen en el 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 11: 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*

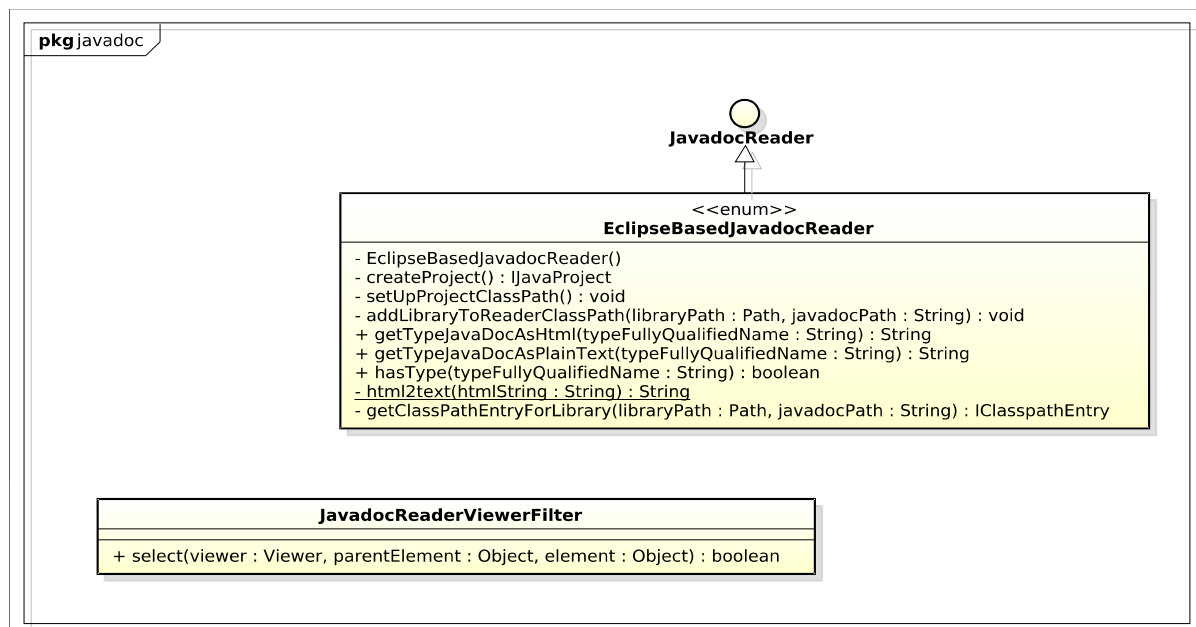


Ilustración 12: D. de clases de *dynamicrefactoring.interfaz.wizard.search.javadoc*

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.

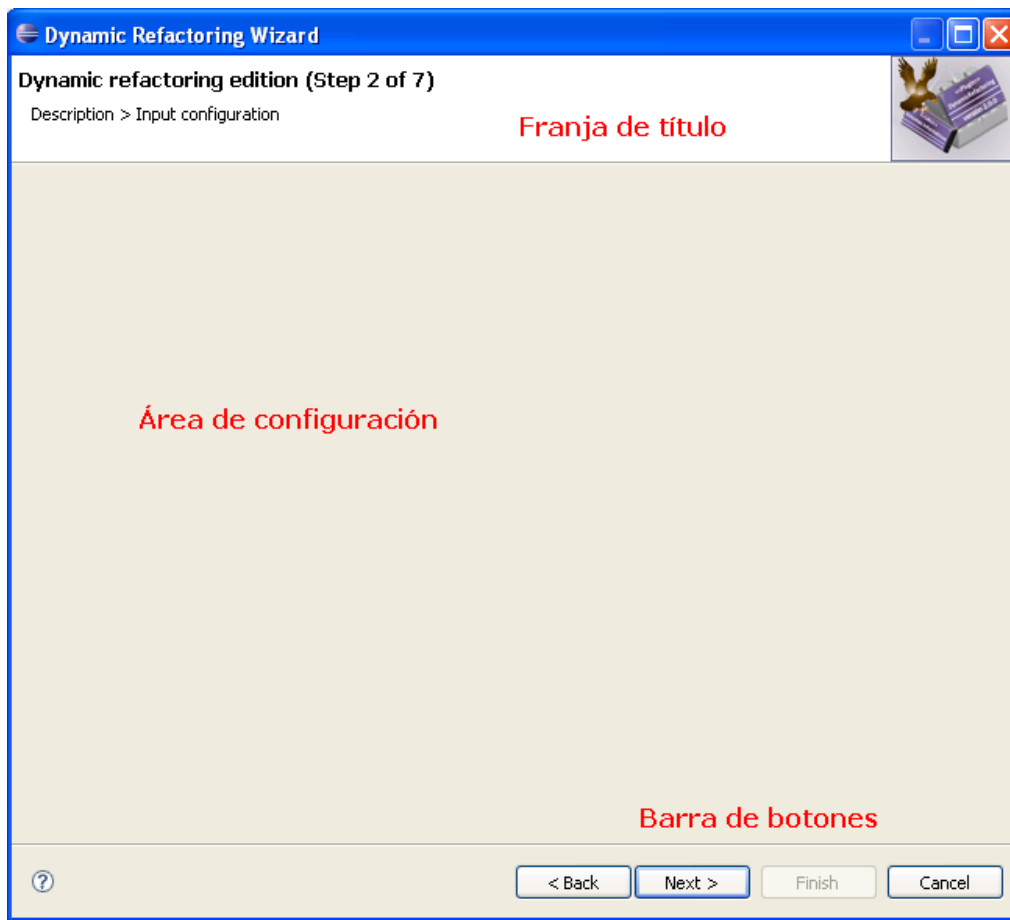
Las interfaces de usuario se tienen que crear lo suficientemente intuitivas para que el usuario pueda saber cómo completar la tarea en cada paso y permitir un aprendizaje rápido sobre la utilización de la herramienta.

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 13: 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 refactorización 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 refactorización 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 refactorización a partir de una ya existente, es decir, crear una copia que poder customizar.

A continuación se muestran de forma visual los campos sufridos:

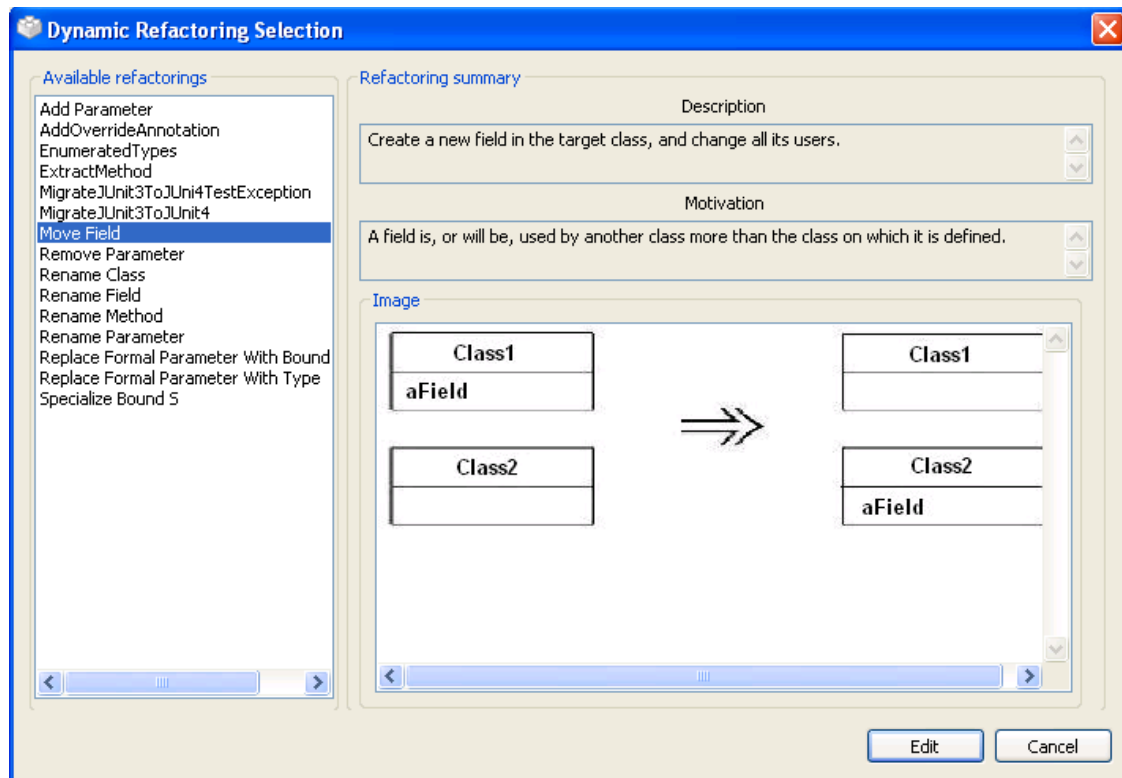


Ilustración 14: Selección refactorización antes de modificaciones

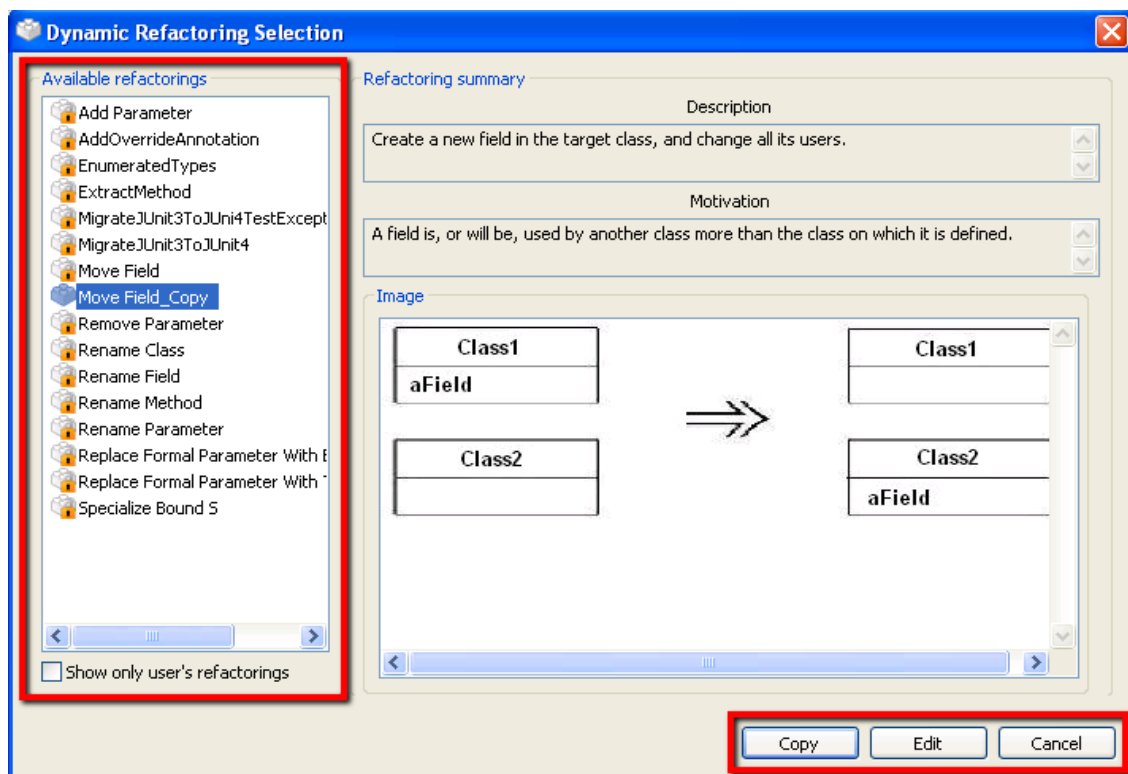
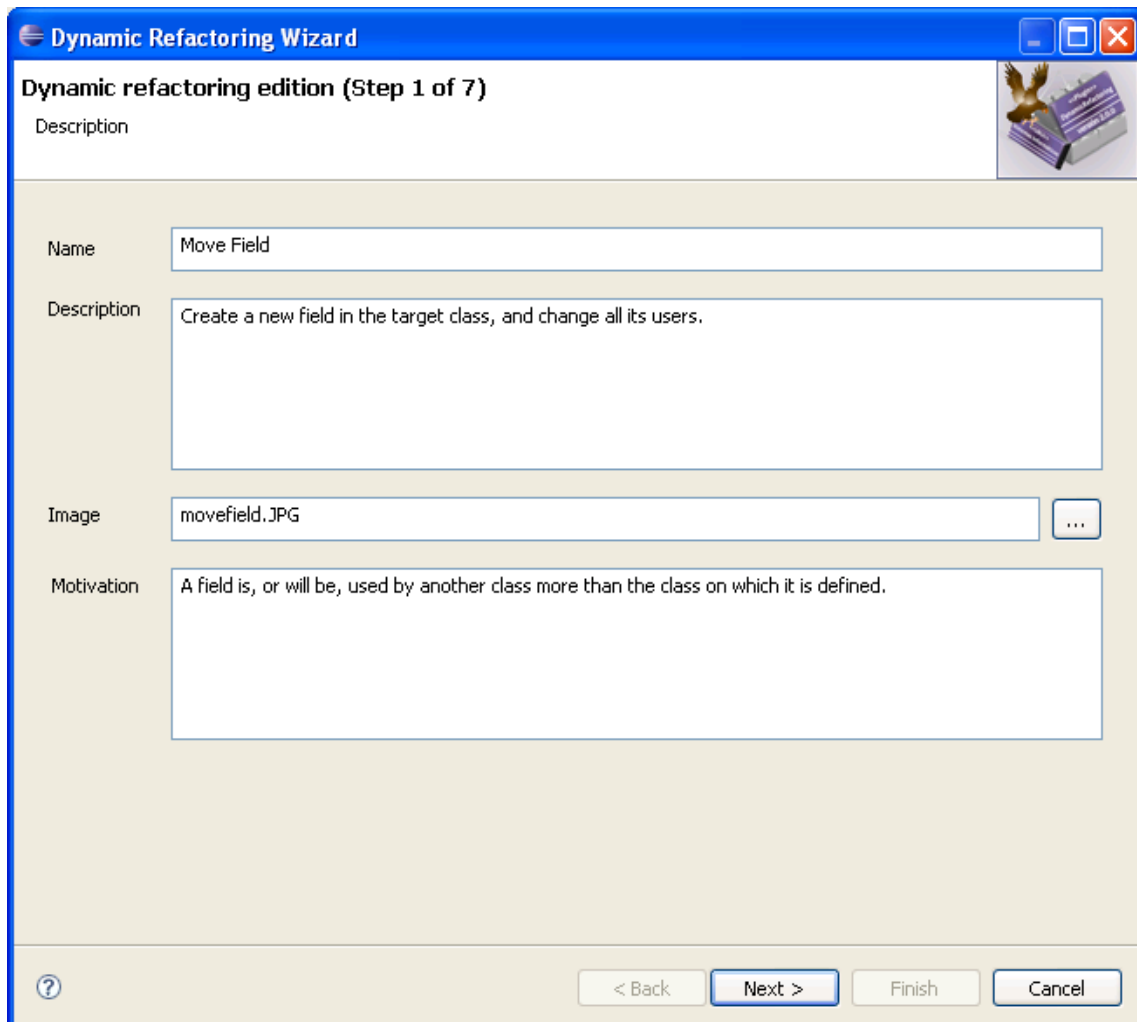


Ilustración 15: Selección refactorización después de modificaciones

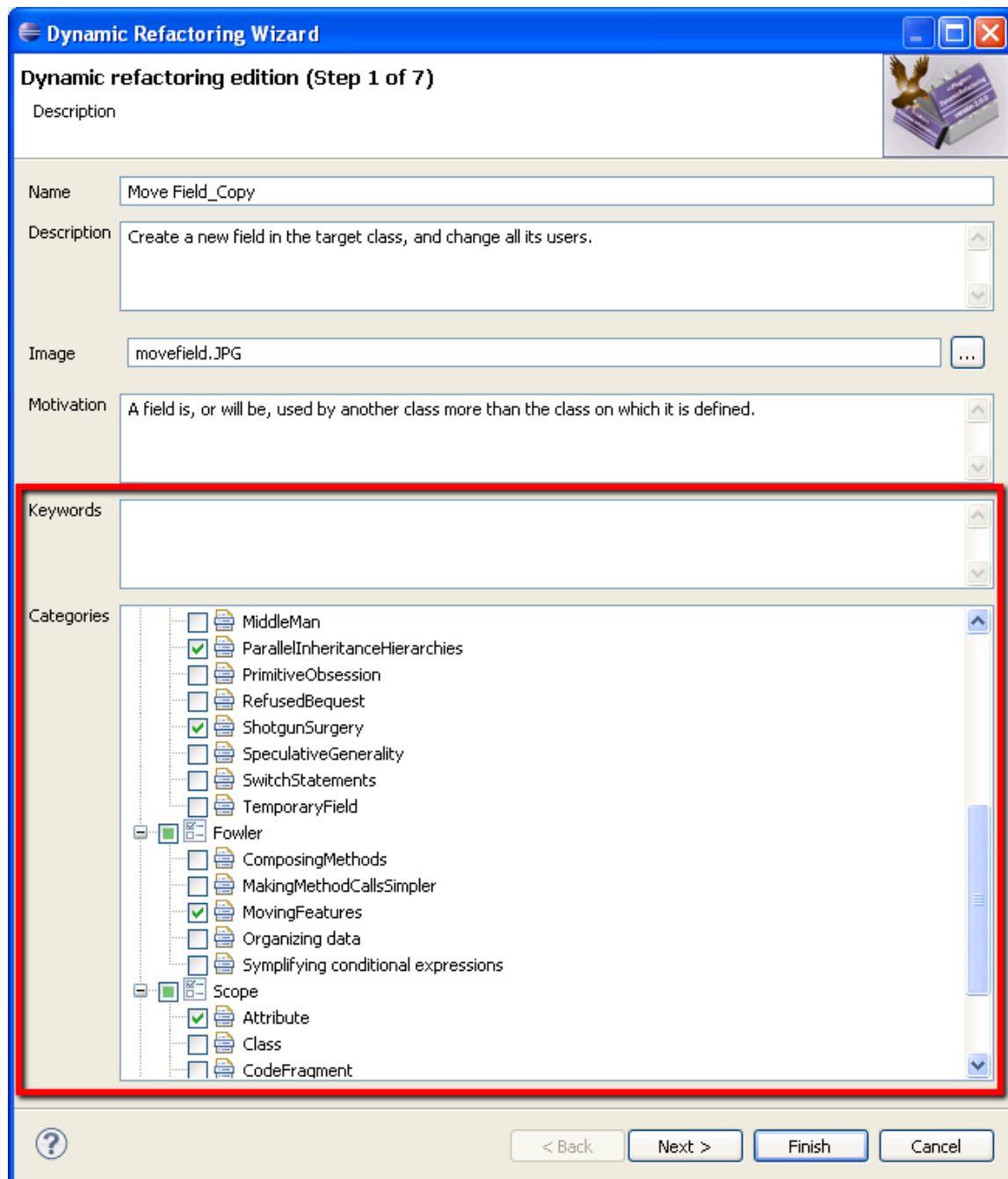


Una vez ya nos encontramos en el asistente de creación y edición de refactorizaciones y a lo largo de cada una de las paginas que lo componen se van a apreciar los cambios que a continuación se detallan.

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 pertenecer la refactorización para cada una de las clasificaciones disponibles.



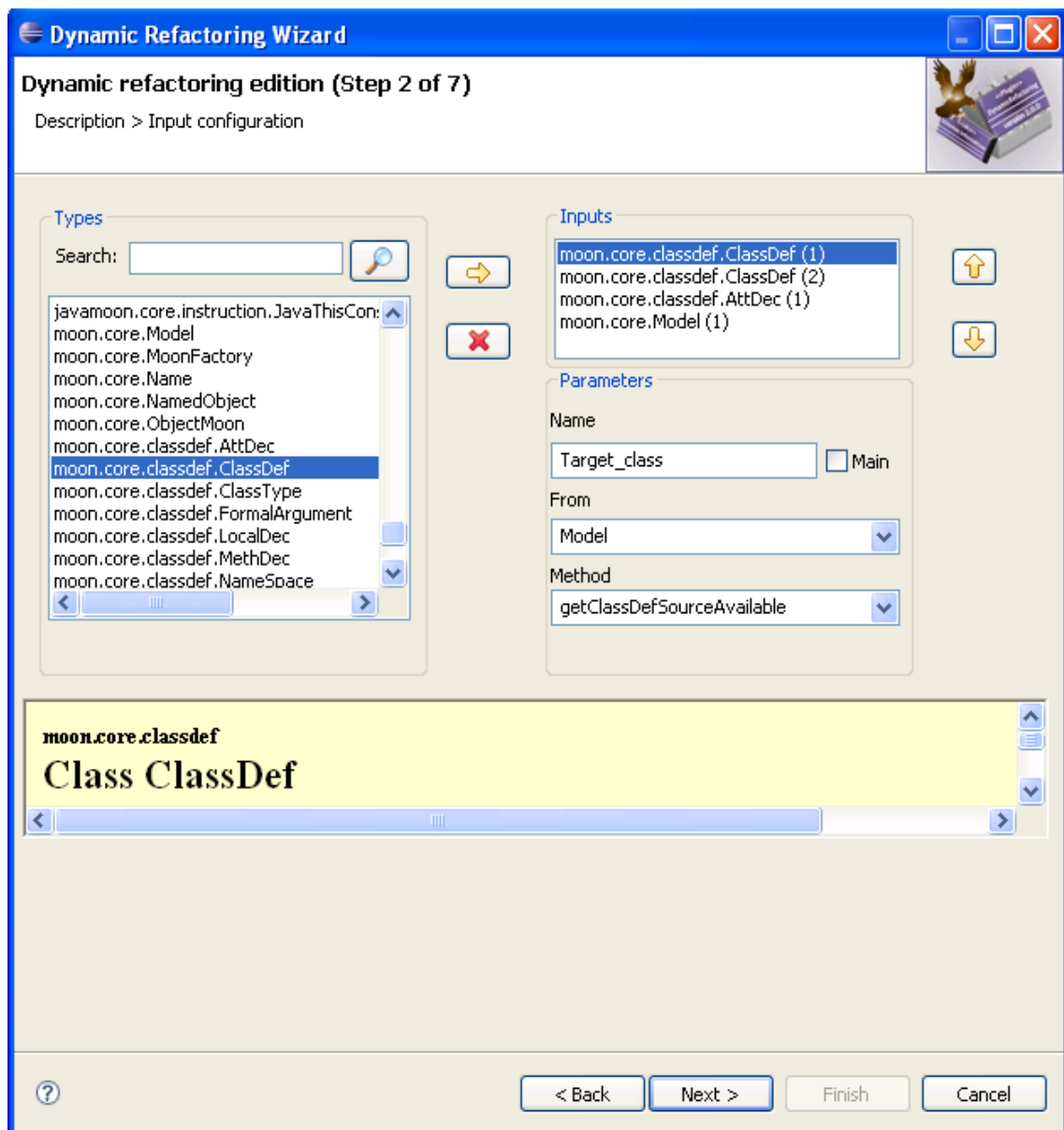
*Ilustración 16: Primera página del asistente antes de modificaciones*



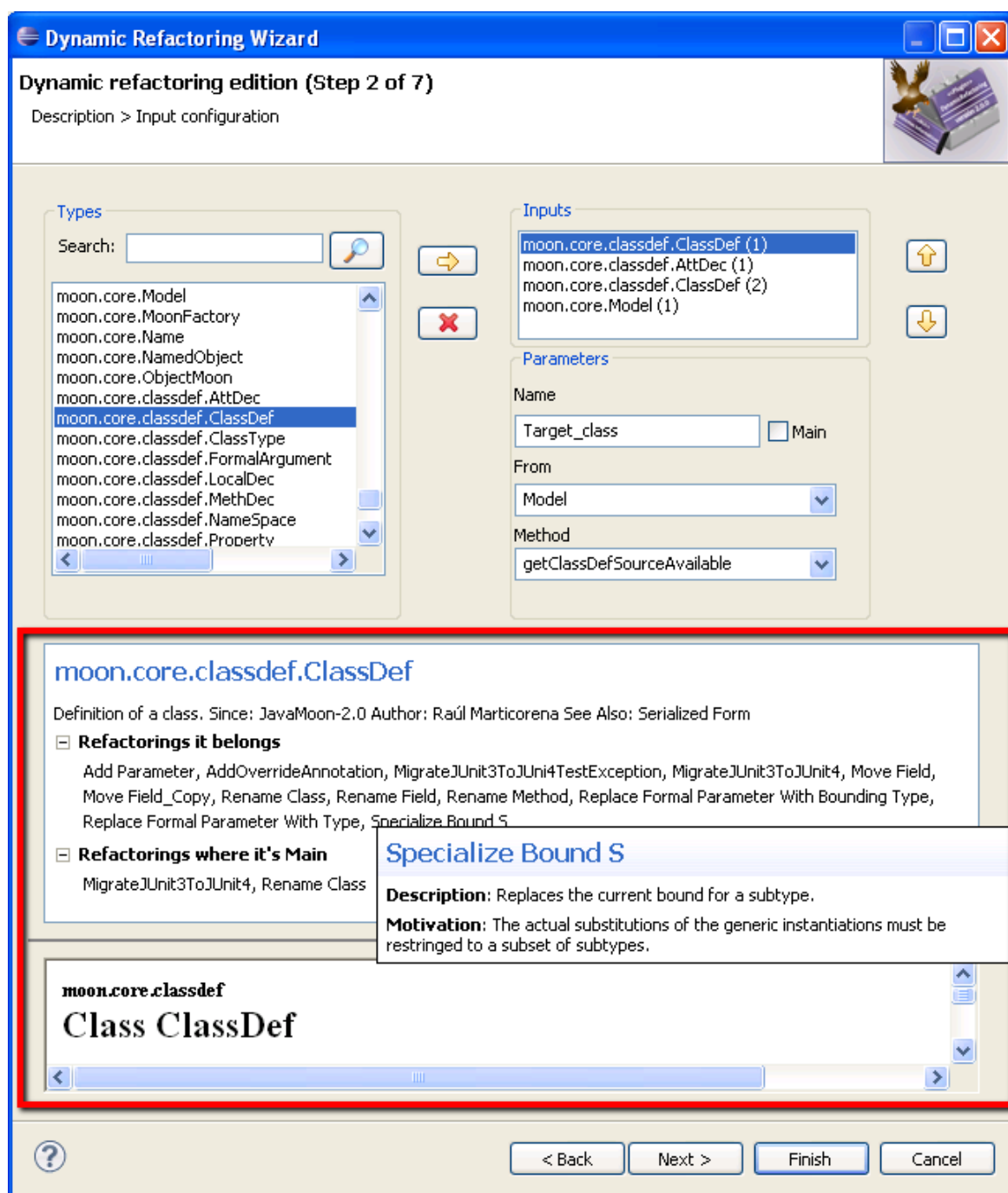
*Ilustración 17: Primera página del asistente después de modificaciones*

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 18: Segunda página del asistente antes de modificaciones*



*Ilustración 19: 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 refactorizaciones que le están 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.

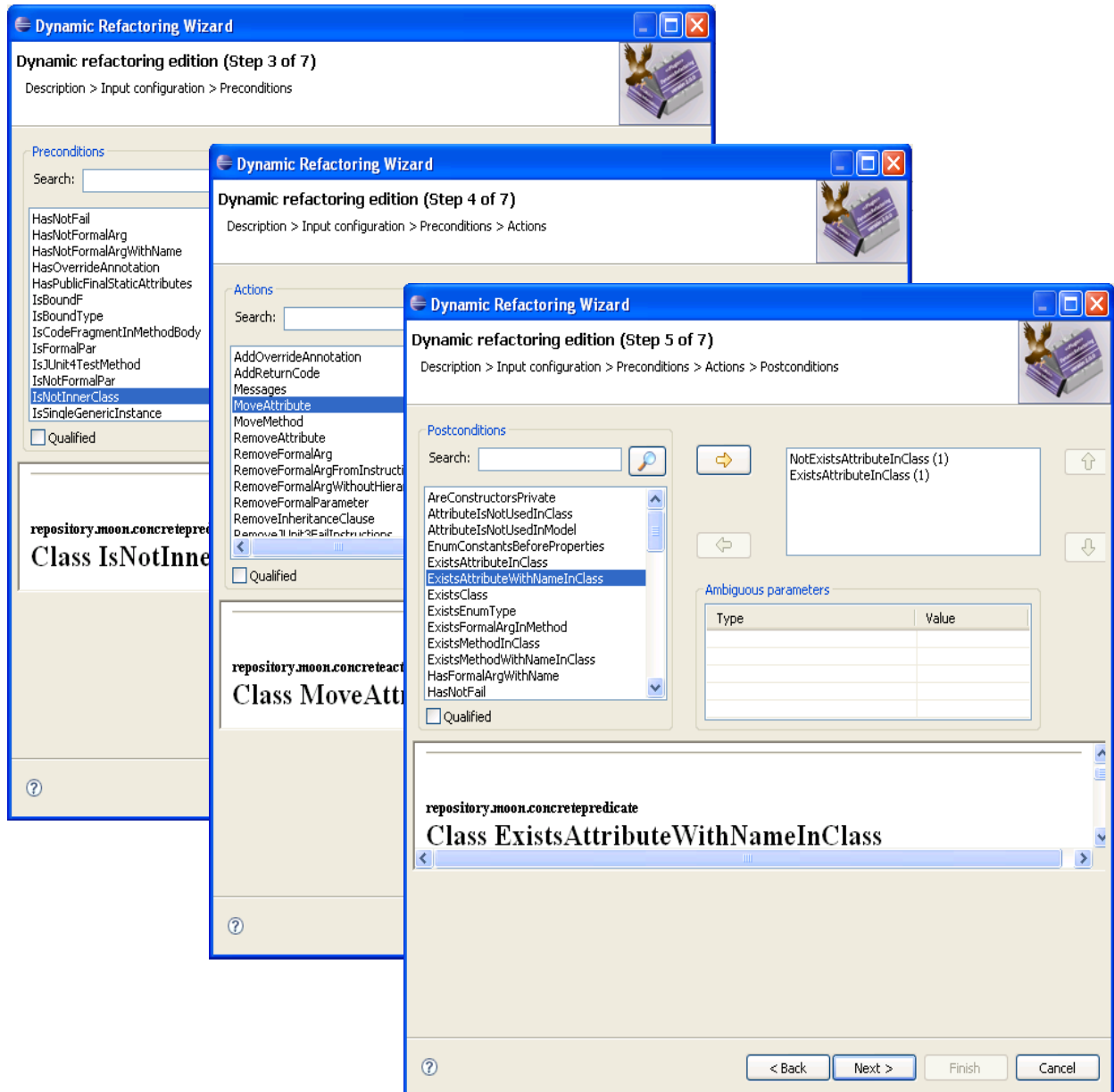


Ilustración 20: Páginas 3,4 y 5 antes de modificaciones

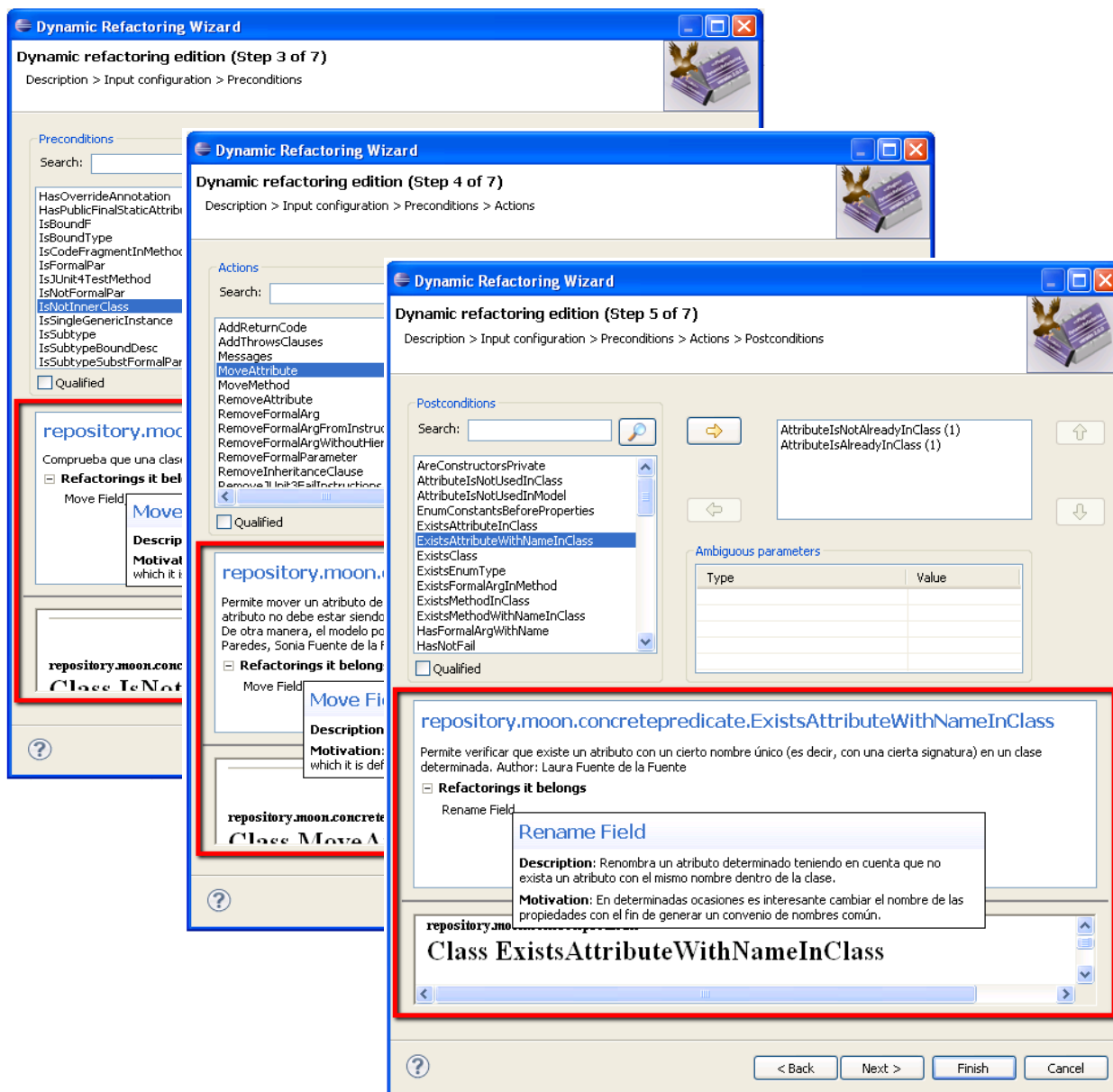
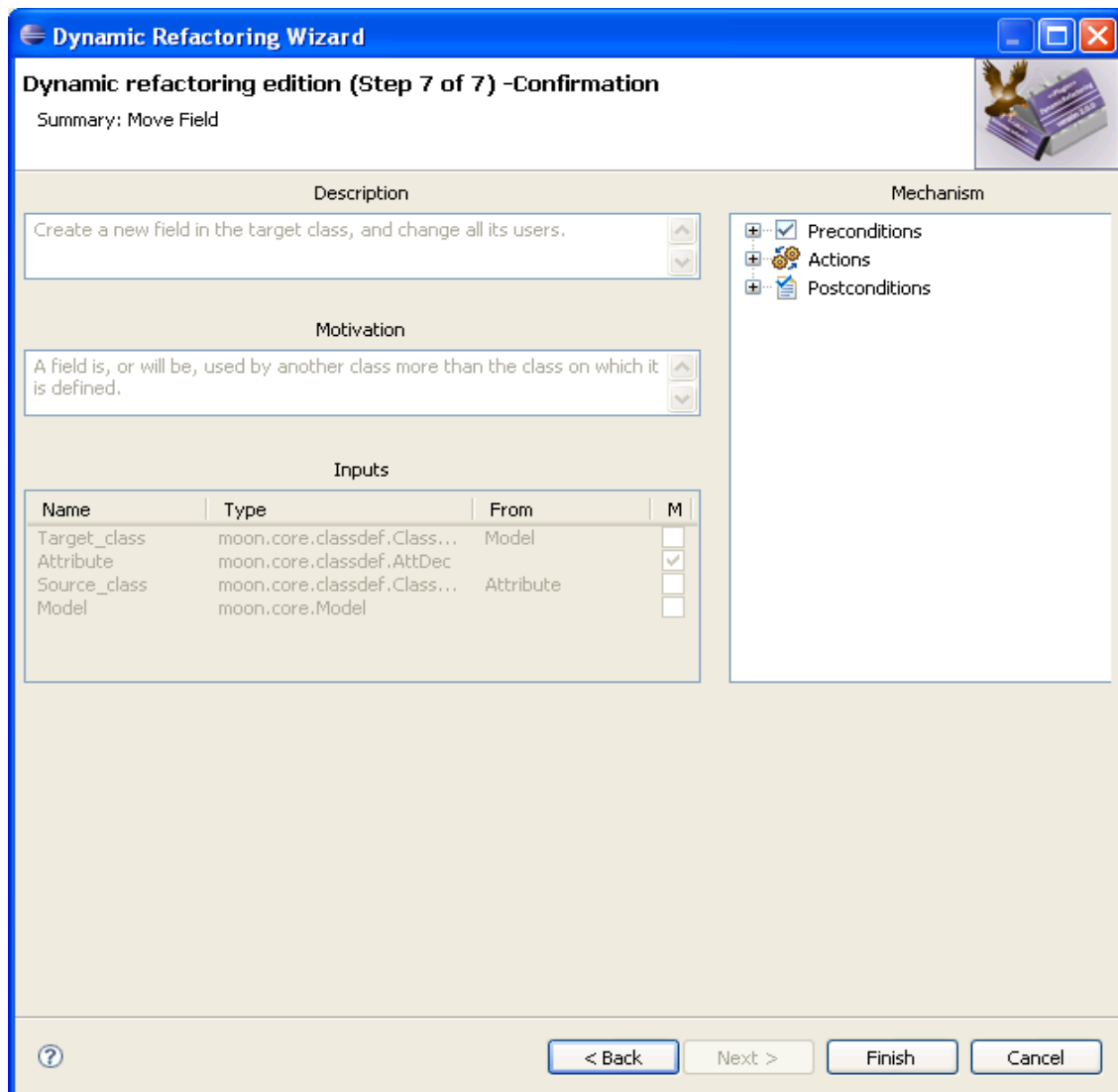


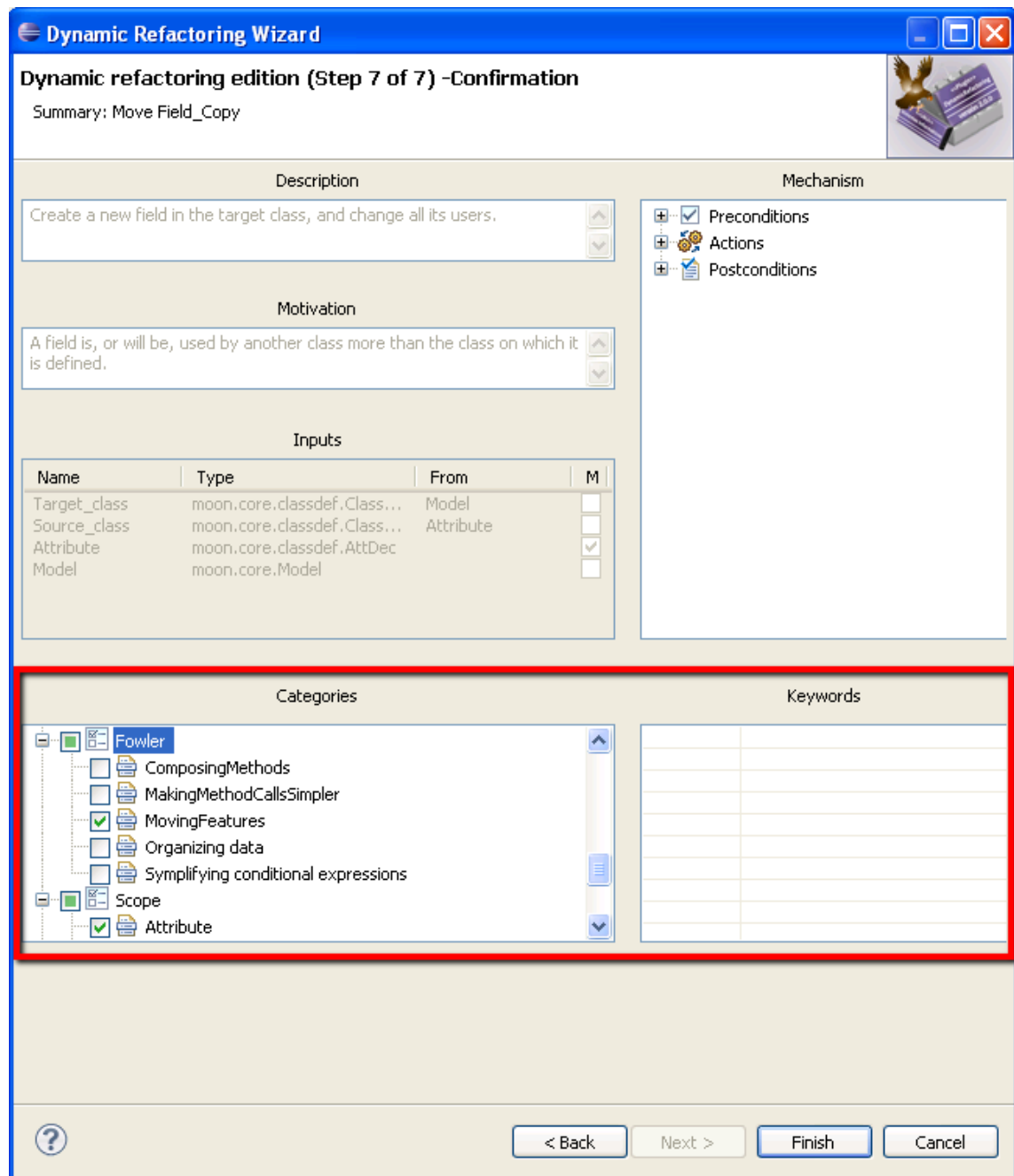
Ilustración 21: Páginas 3, 4 y 5 después de modificaciones

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 22: Séptima página del asistente antes de modificaciones*



*Ilustración 23: Séptima página del asistente después de modificaciones*

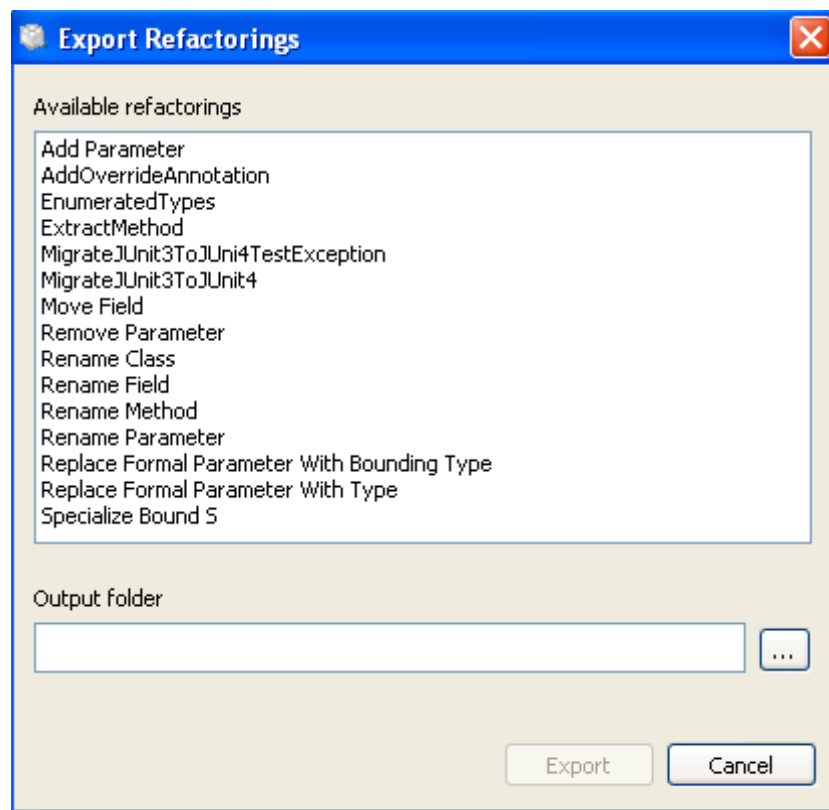
## 4.2. Interfaz para la exportación de refactorizaciones

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.

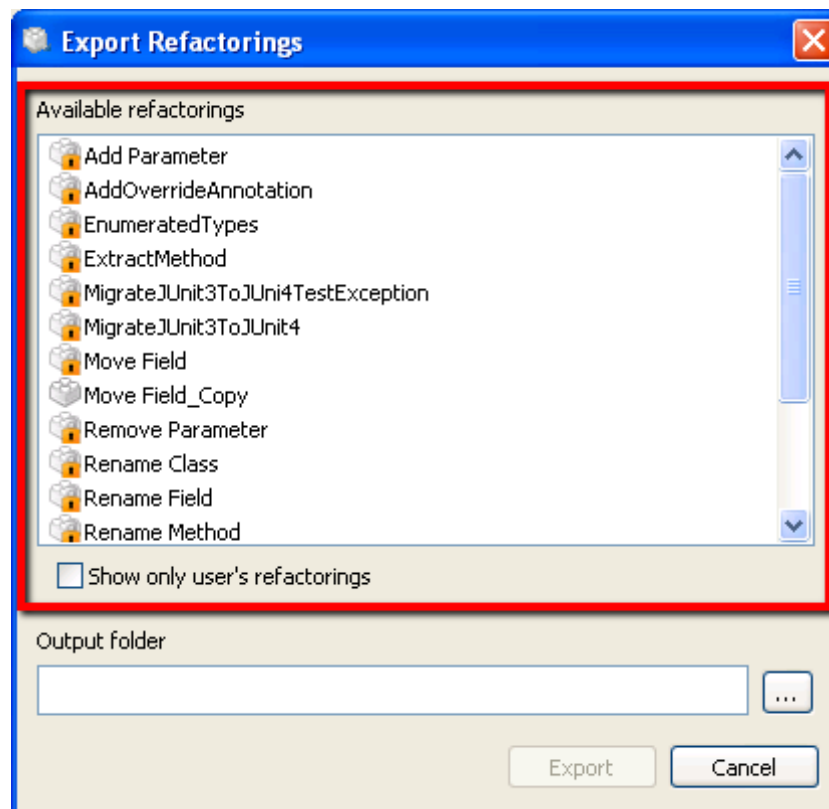


La interfaz que hace posible la exportación de refactorizaciones muestra la relación de refactorizaciones que se encuentran disponibles. De estas, el usuario seleccionará las que desea exportar e indicará la ruta en la que lo quiere realizar.

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 24: Interfaz exportación refactorizaciones antes de modificaciones*



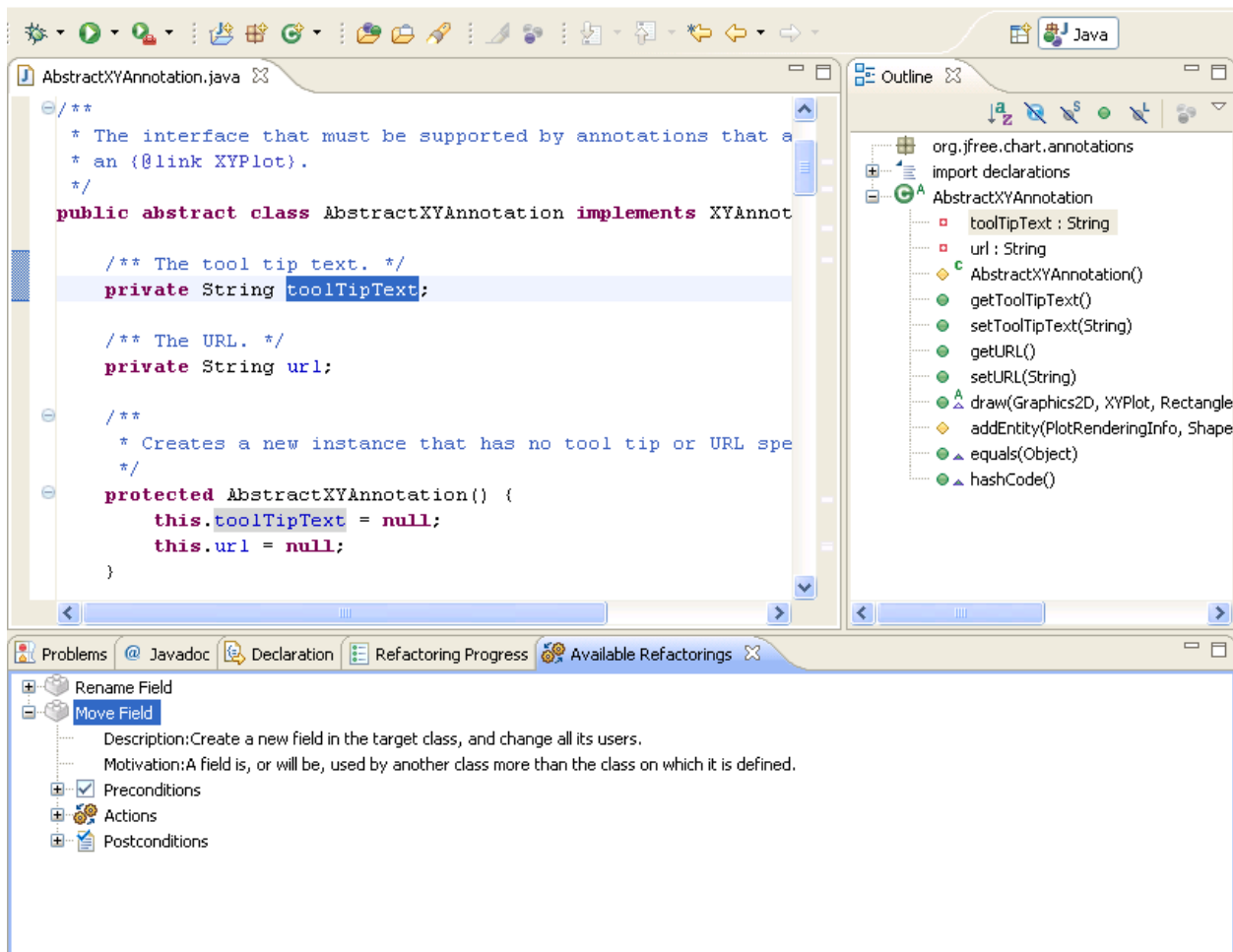
*Ilustración 25: Interfaz exportación refactorizaciones después de modificaciones*

### 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 icono 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 26: Vista Available Refactorings antes de modificaciones*

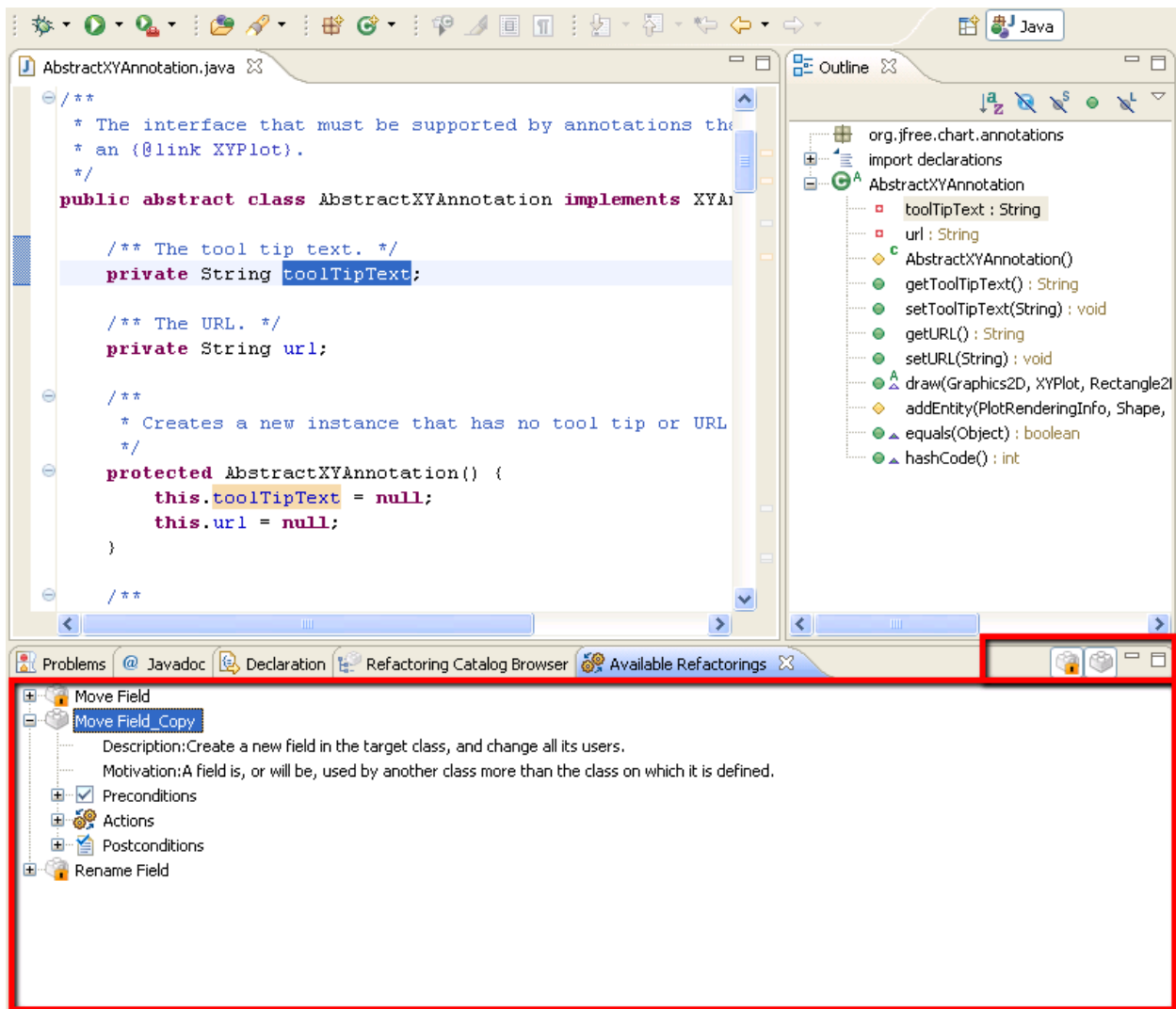


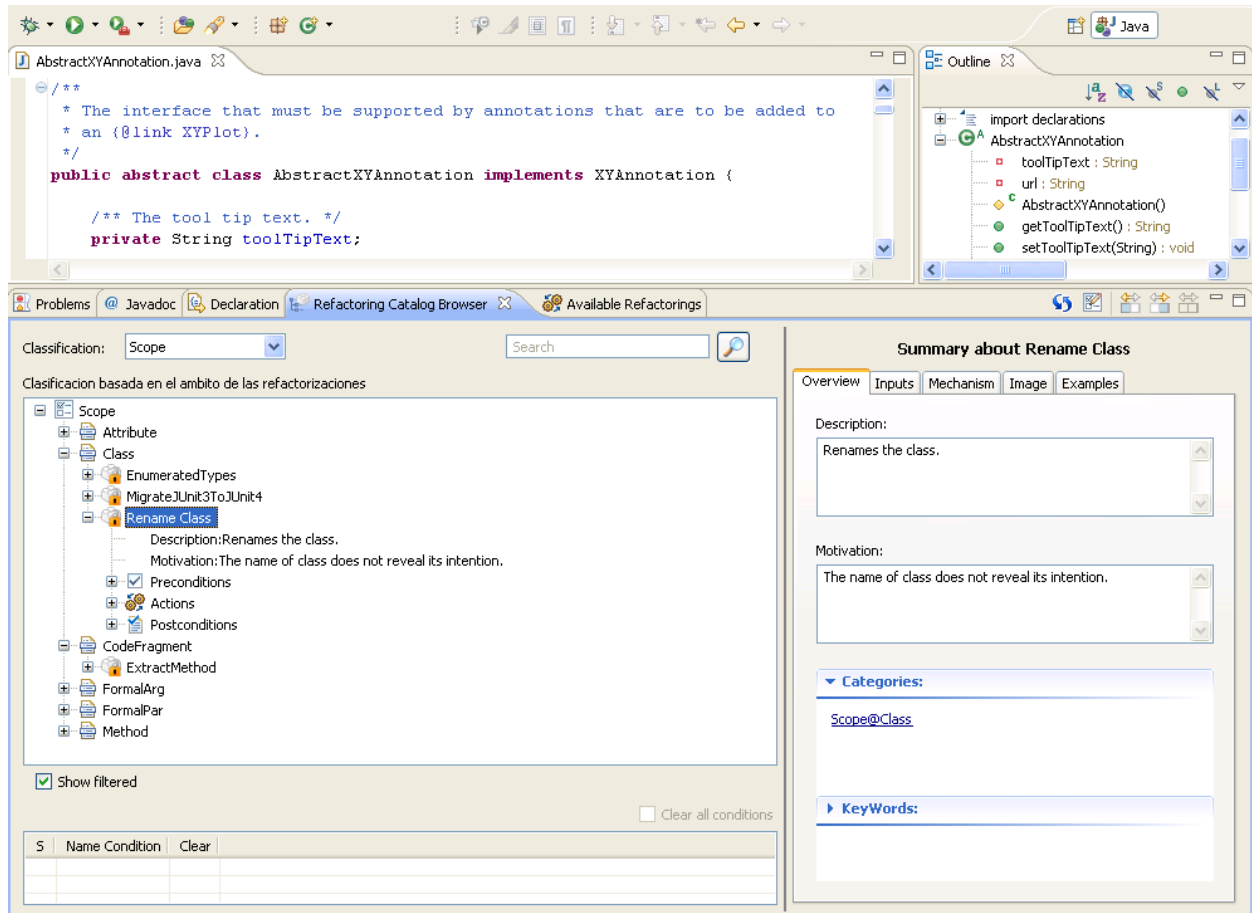
Ilustración 27: Vista Available Refactorings después de modificaciones

#### 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 adquiriera el conocimiento de estas de una forma mucho mas amigable.

La ilustración que a continuación se presenta muestra la nueva vista:



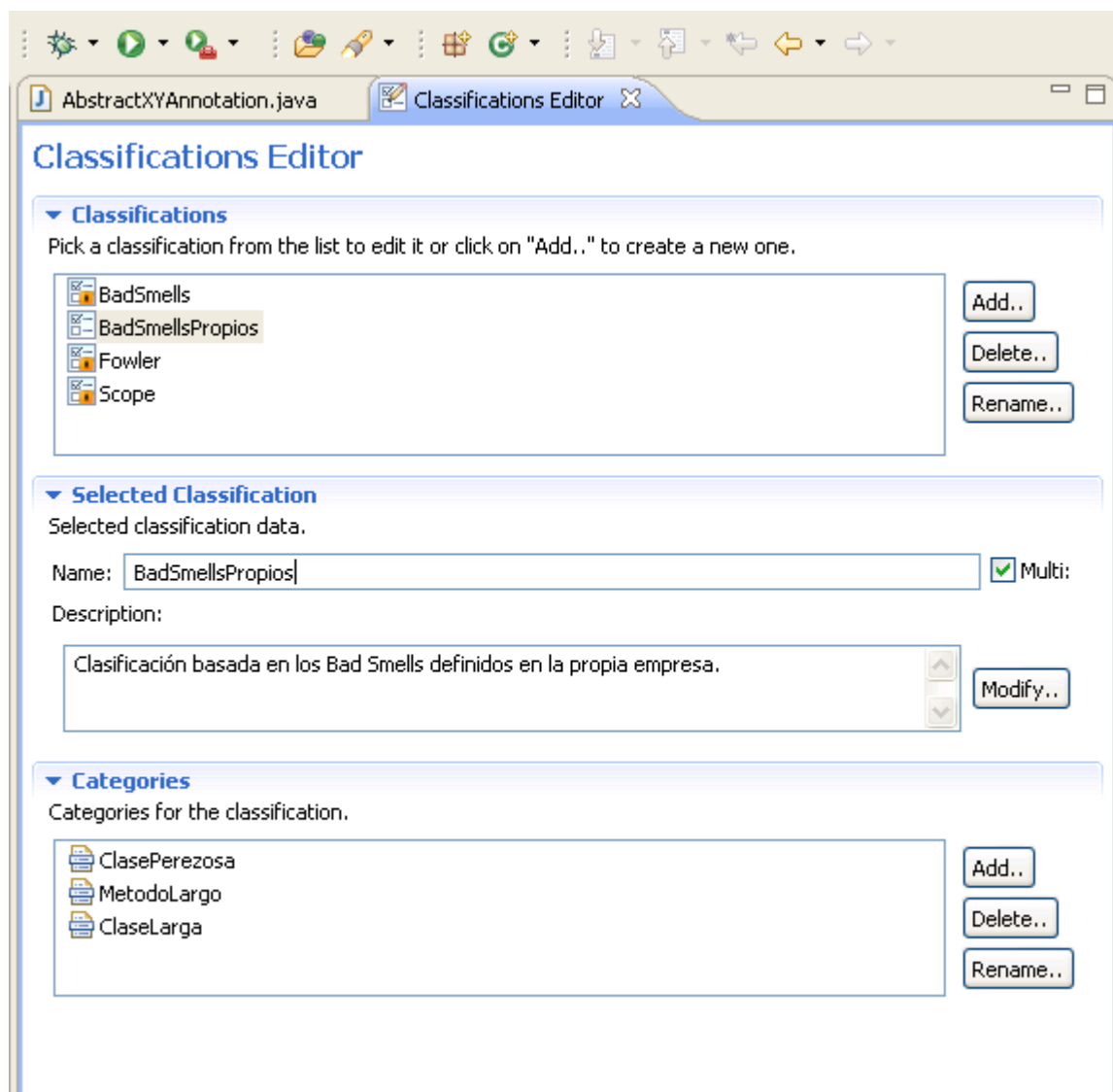
*Ilustración 28: Vista Refactoring Catalog Browser*

## 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 29: Editor de clasificaciones*

## **5. DISEÑO PROCEDIMENTAL**

Este apartado esta 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

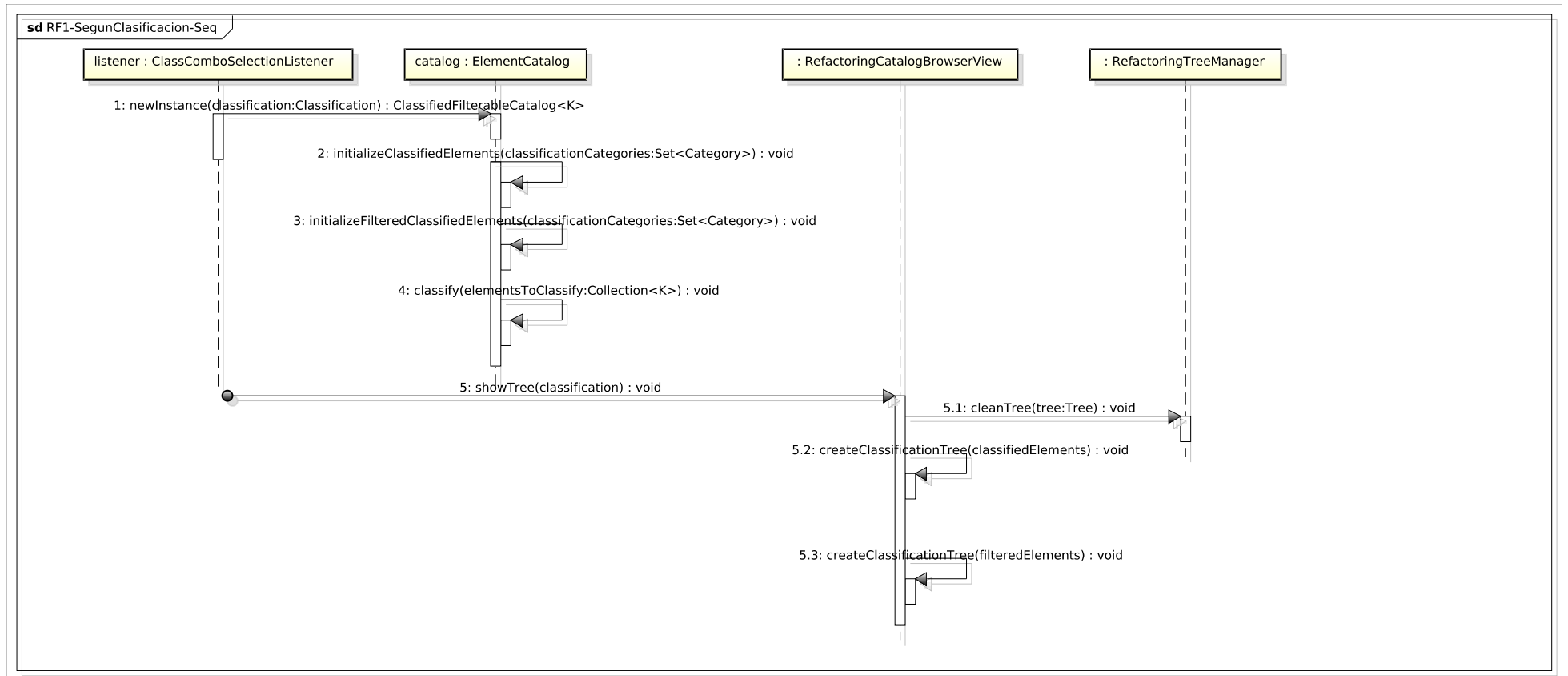


Ilustración 30: D. de secuencia de RF 1: Visualizar ref. según clasificación

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

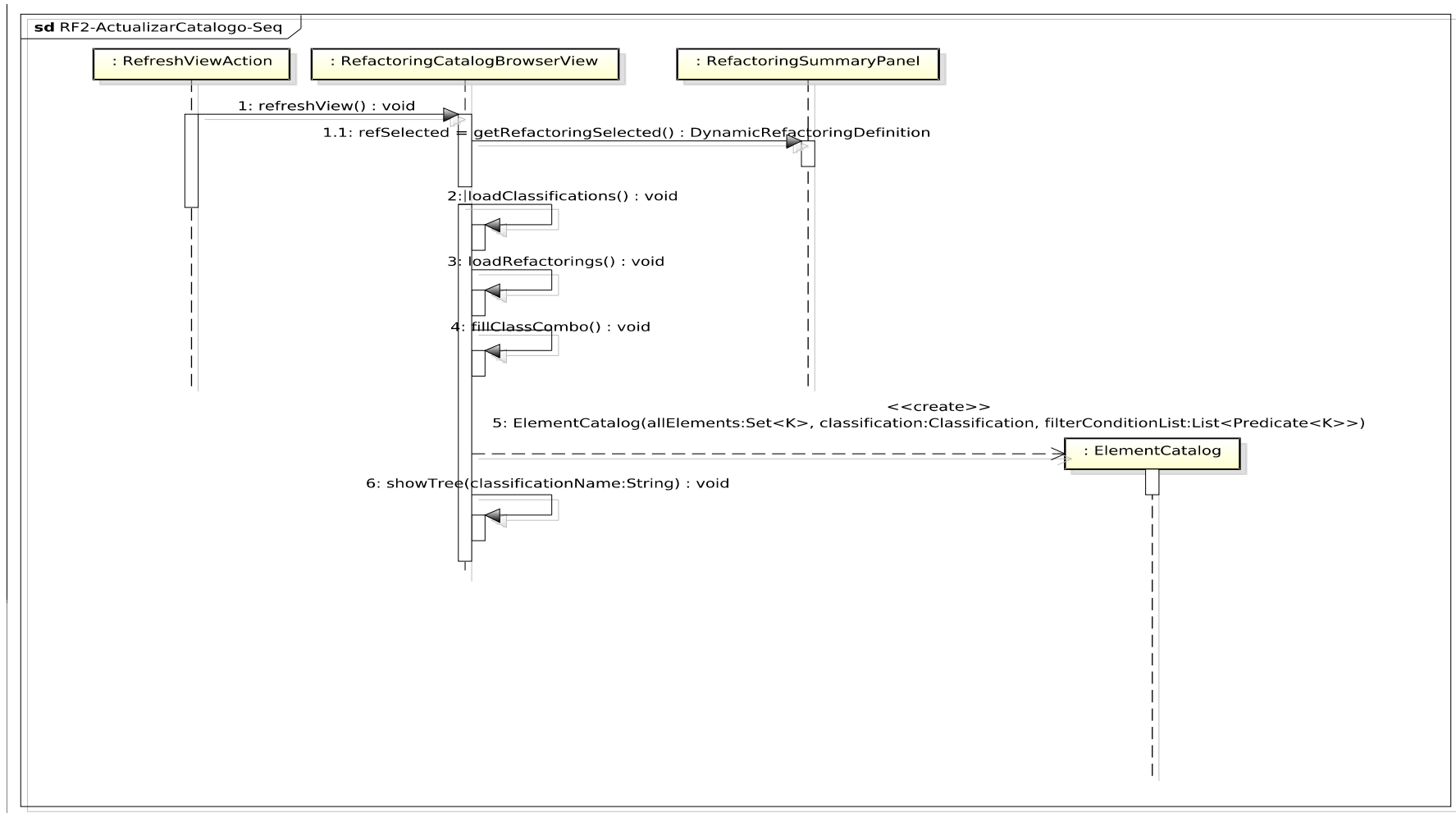


Ilustración 31: D. de secuencia de RF 2: Refrescar visualización de refactorizaciones

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

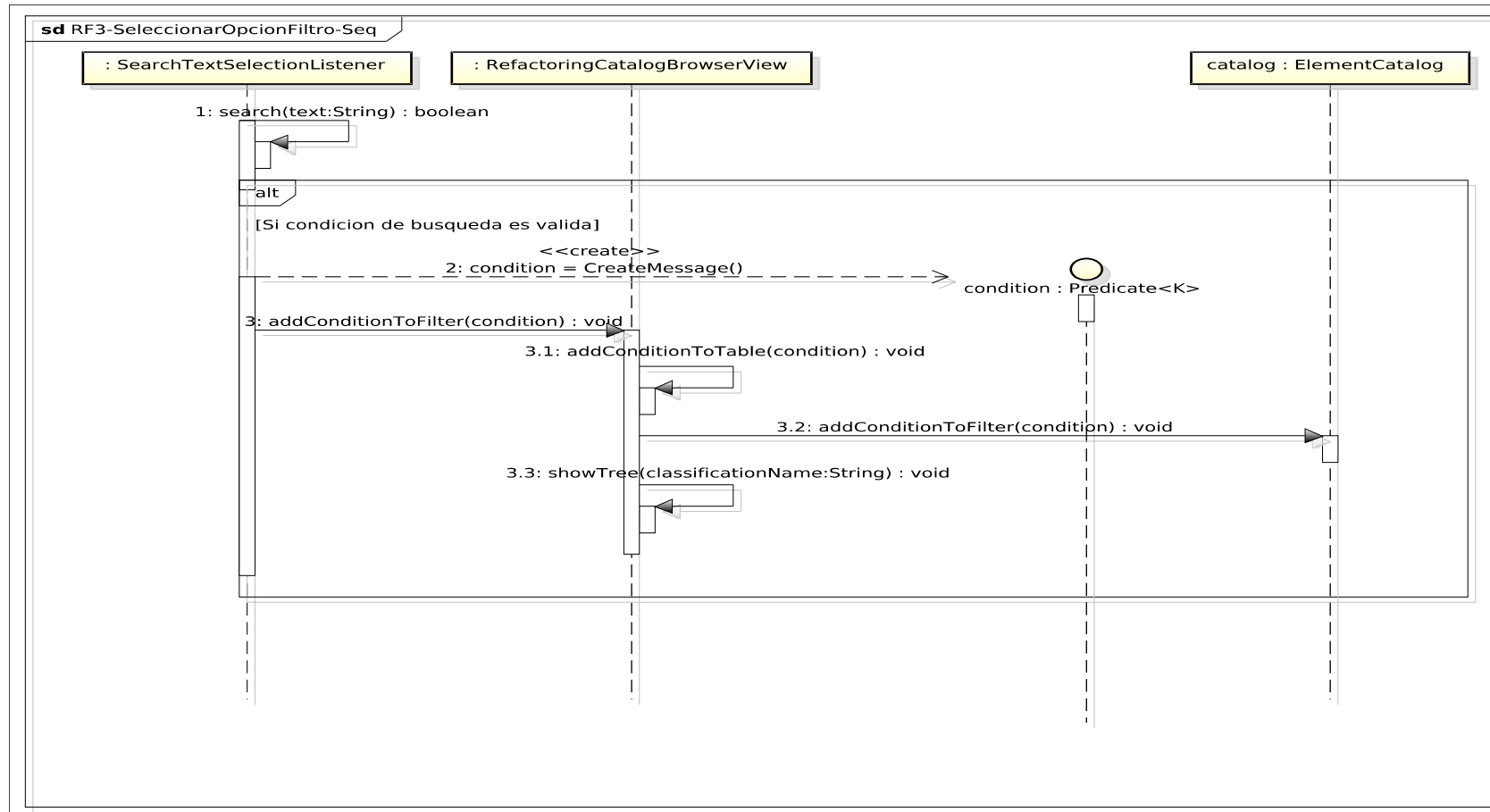


Ilustración 32: D. de secuencia de RF 3: Añadir filtro de refactorizaciones

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

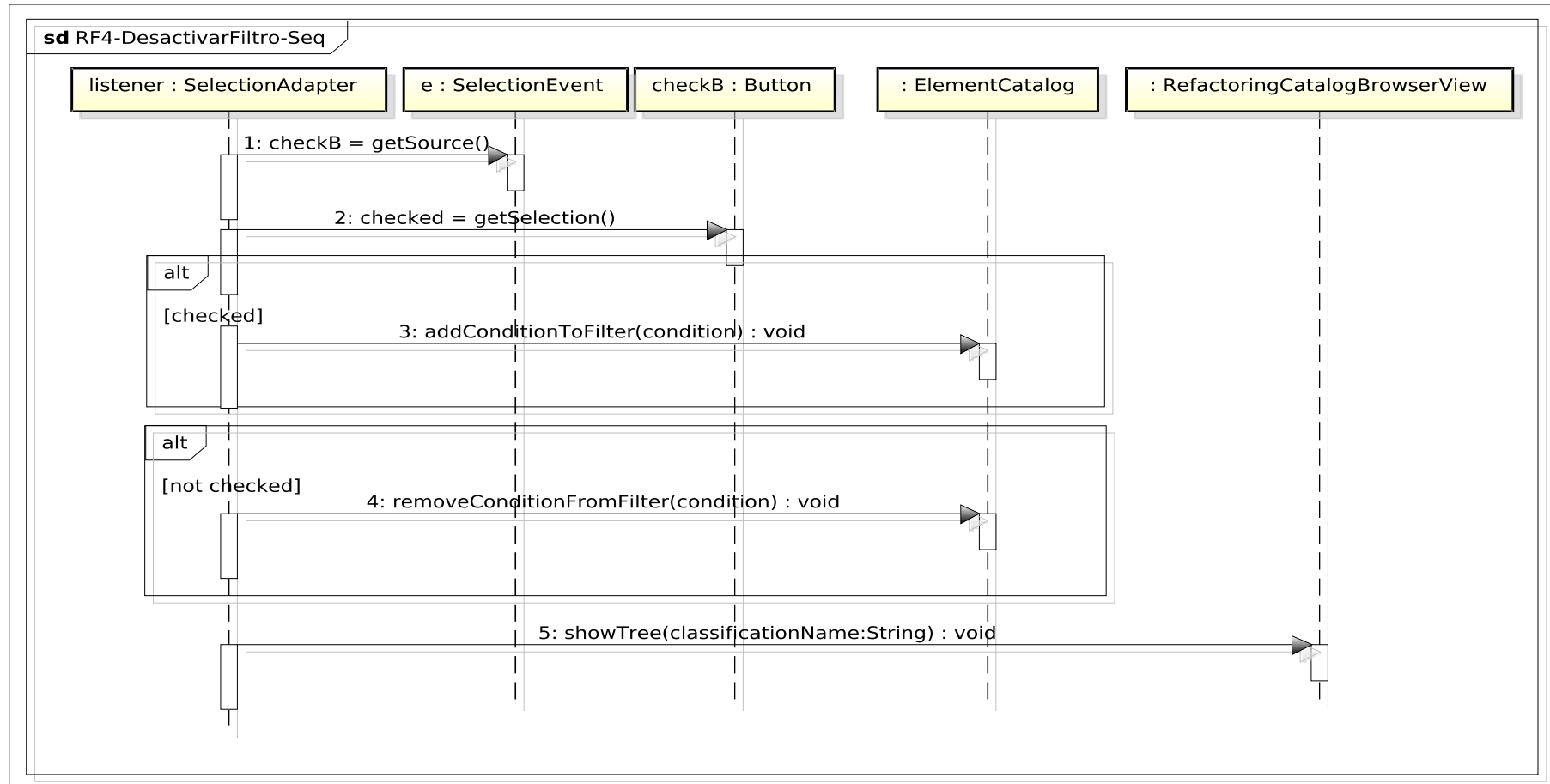


Ilustración 33: D. de secuencia de RF 4: Seleccionar opción aplicar filtro

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

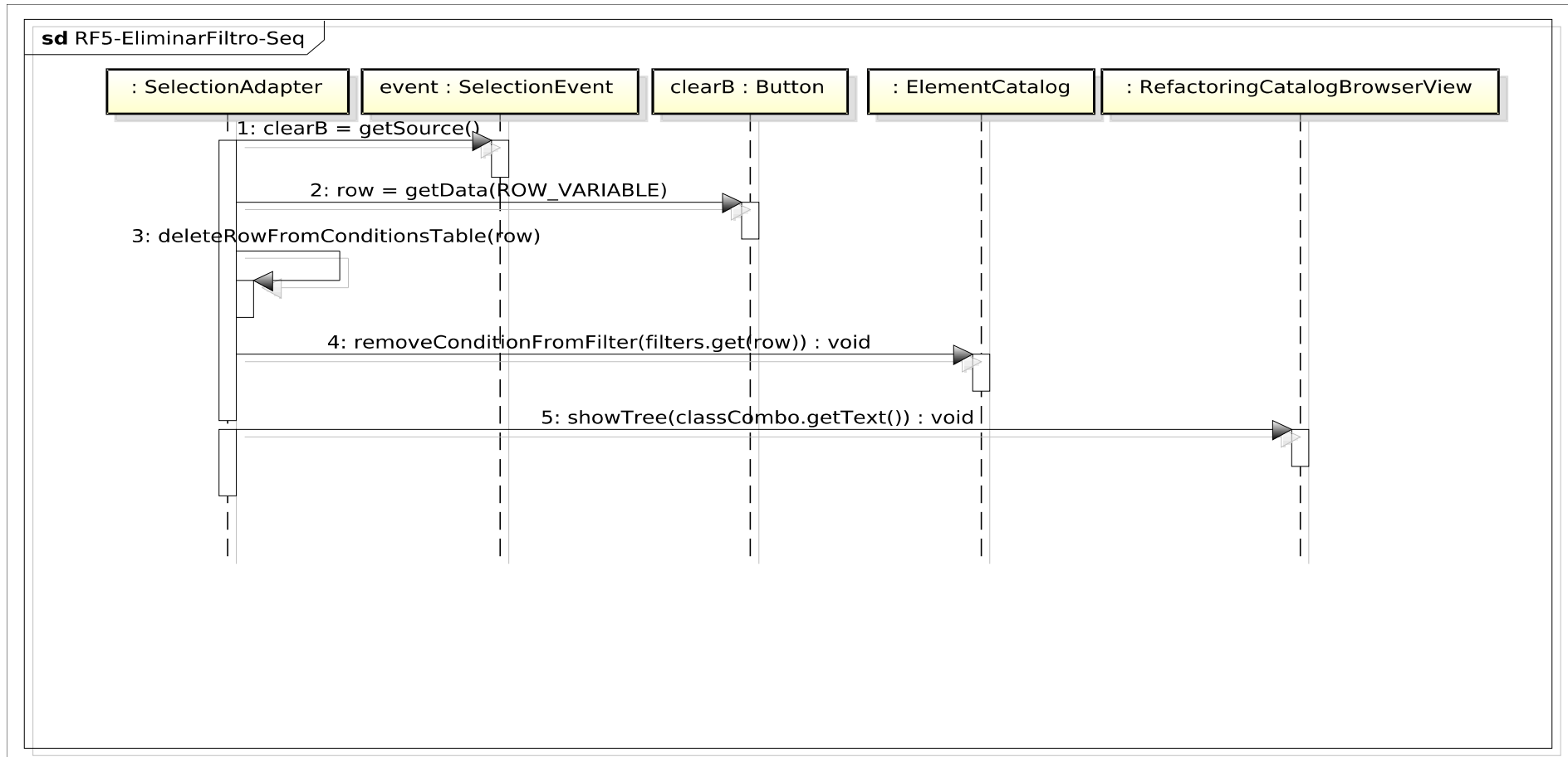


Ilustración 34: D. de secuencia de RF 5: Eliminar filtro de refactorizaciones

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

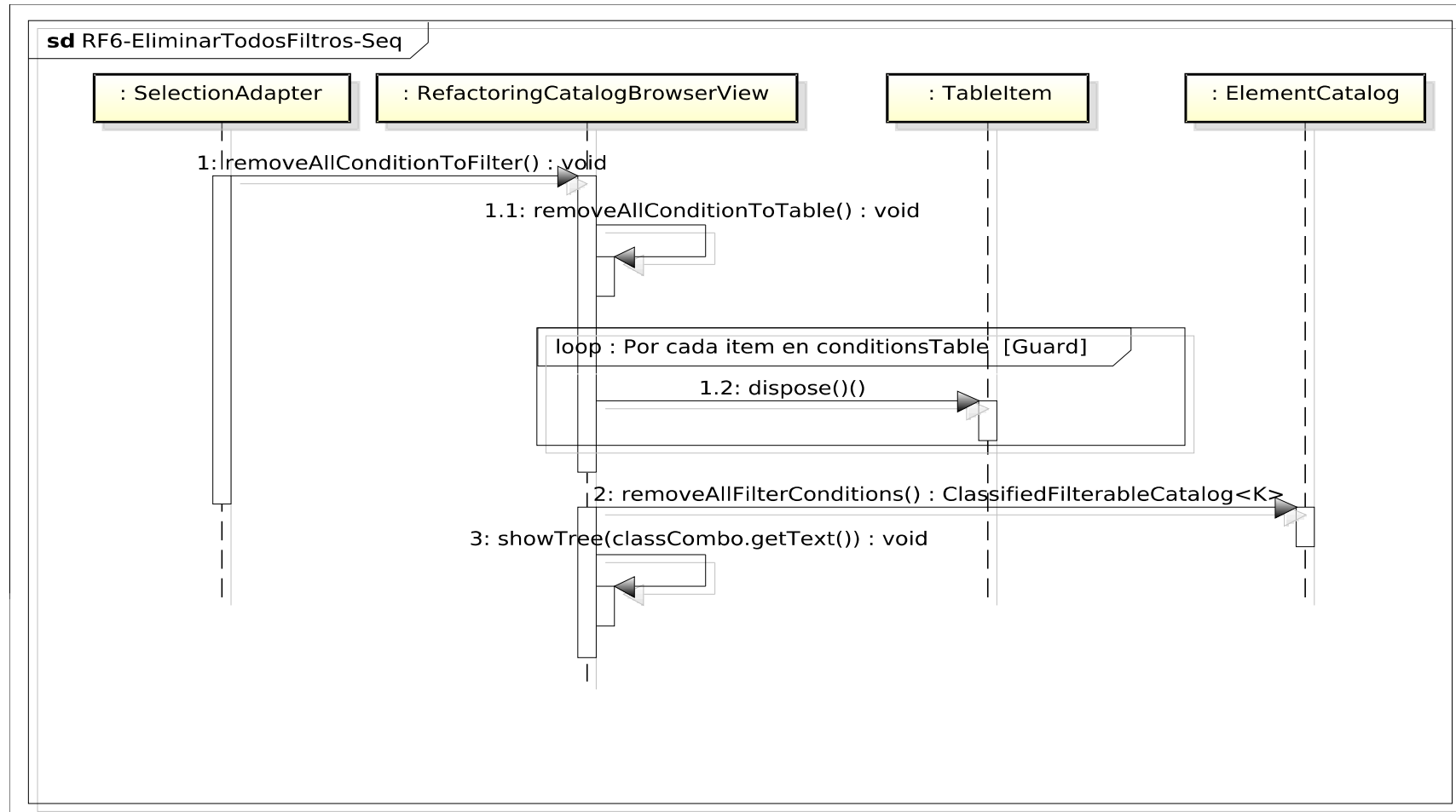


Ilustración 35: D. de secuencia de RF 6: Eliminar todos los filtros de refactorizaciones

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

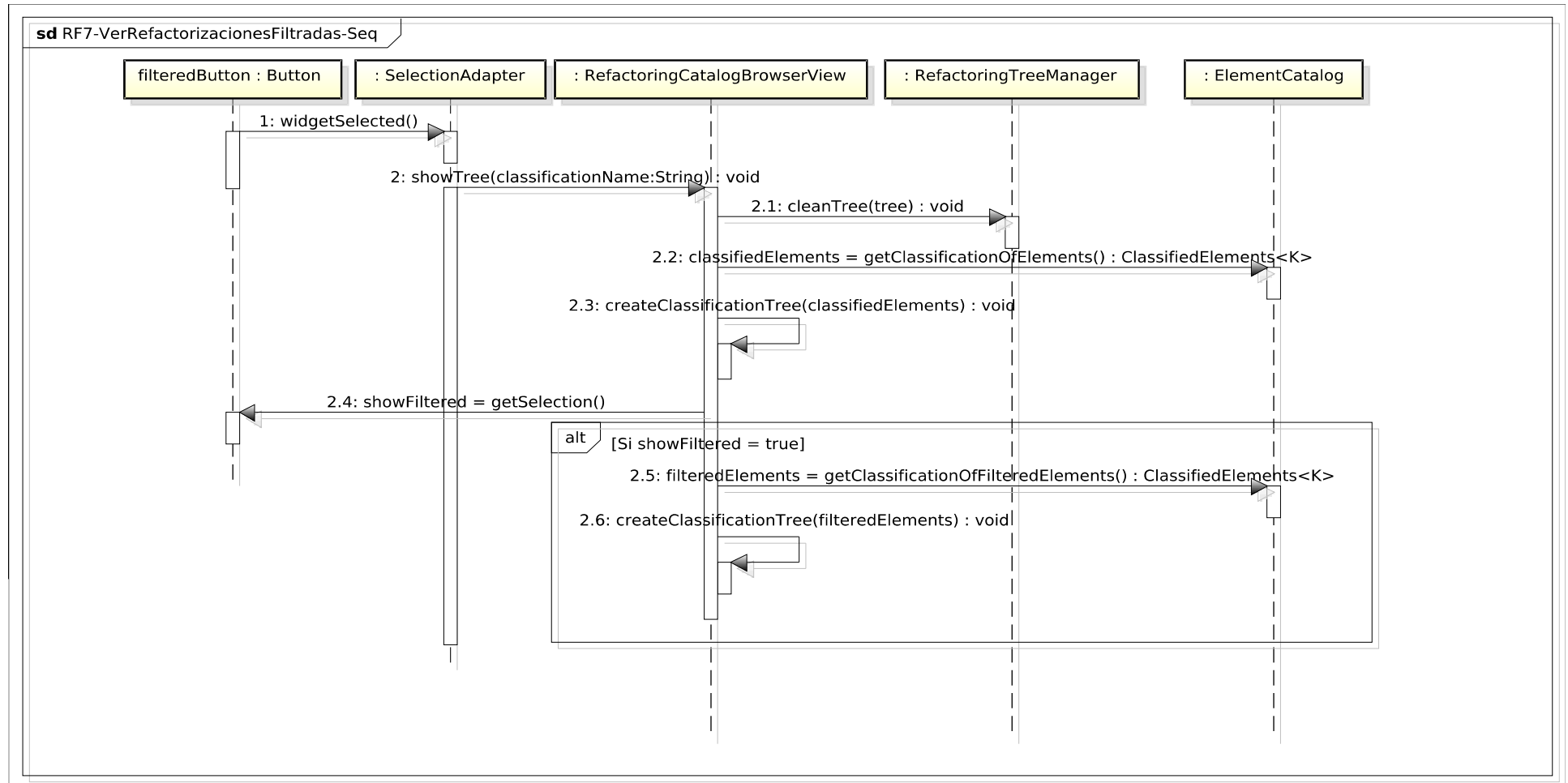


Ilustración 36: D. de secuencia de RF 7: Seleccionar opción ver ref. filtradas

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

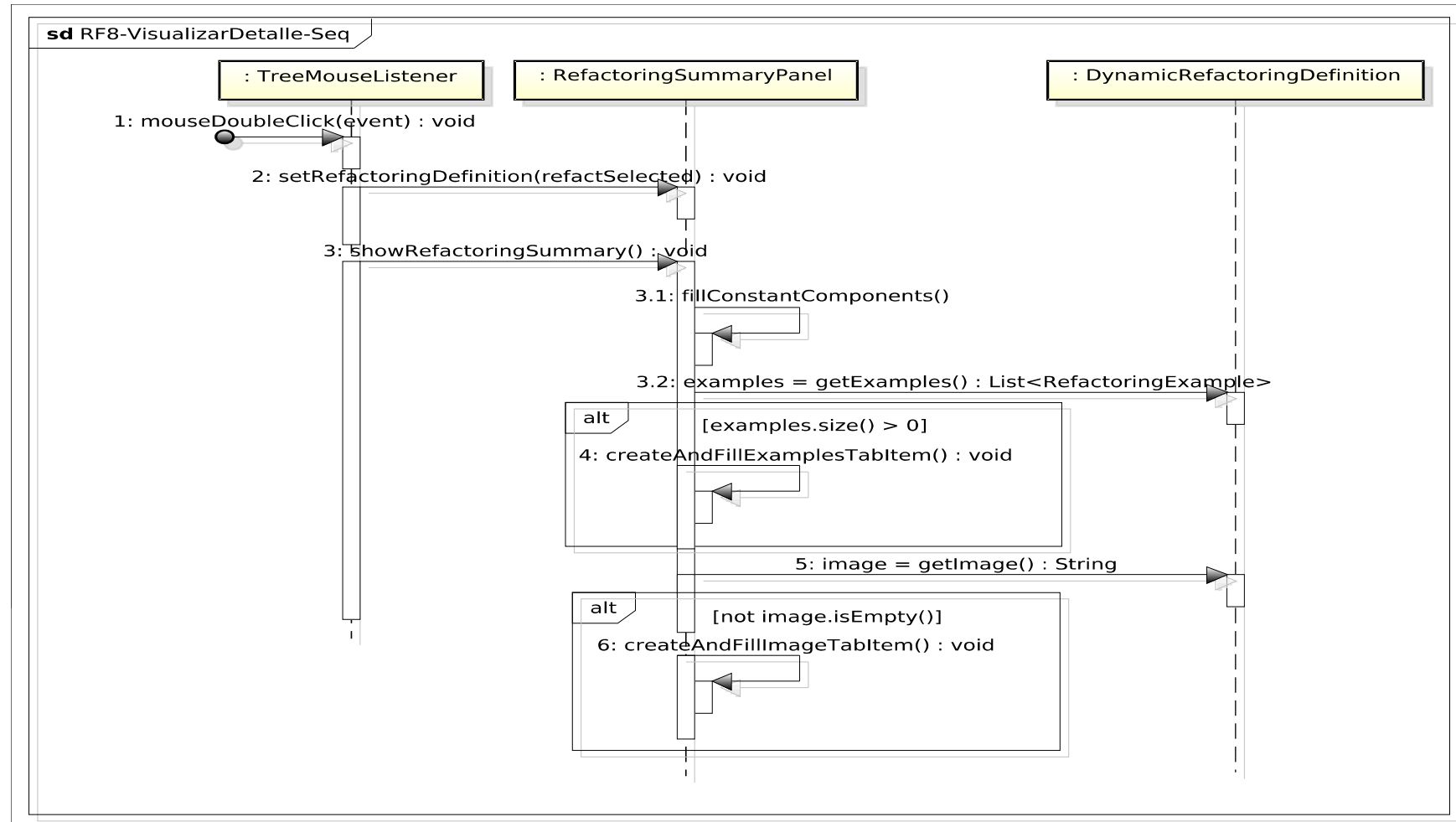


Ilustración 37: D. de secuencia de RF 8: Visualizar detalle refactorización

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

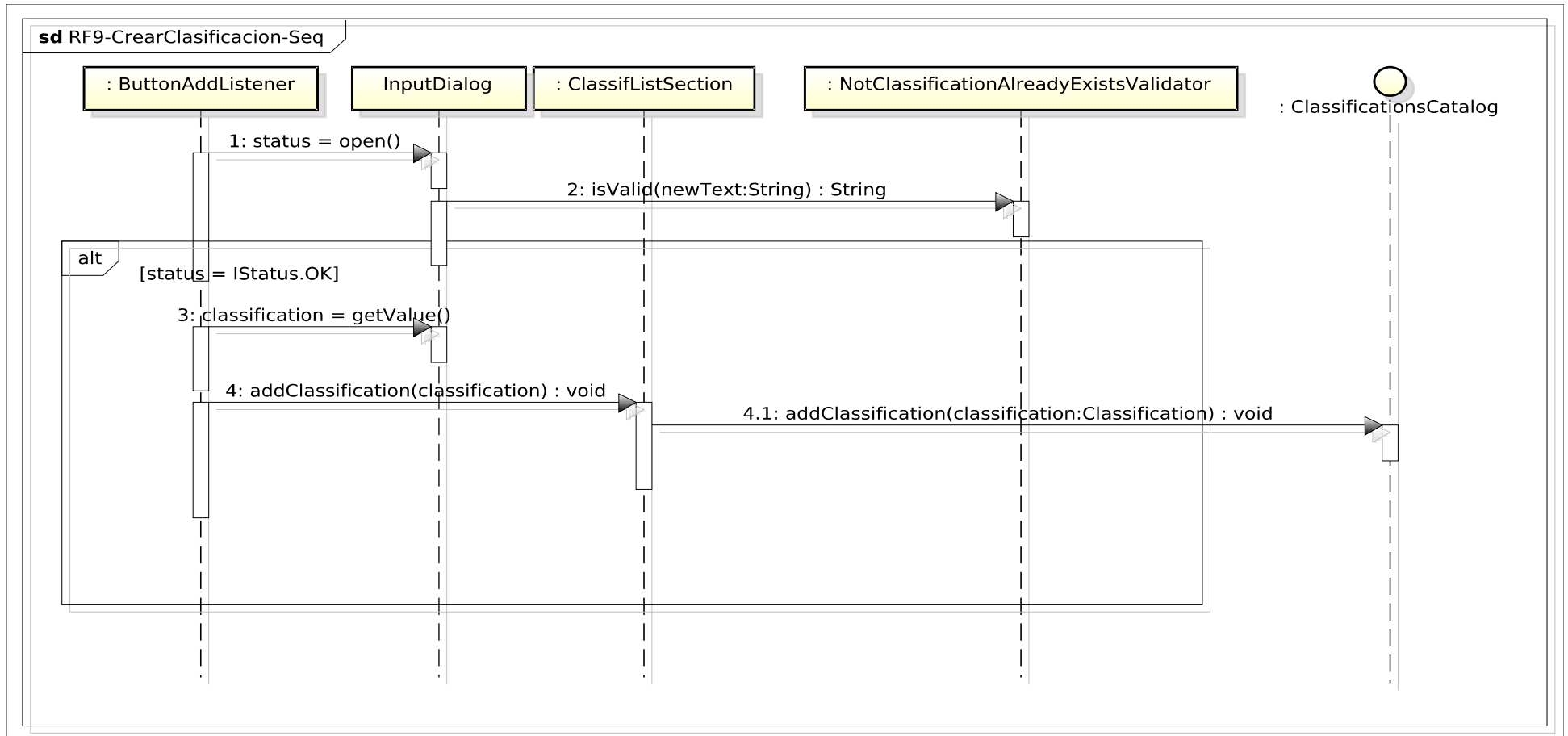


Ilustración 38: D. de secuencia de RF 9: Añadir clasificación

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

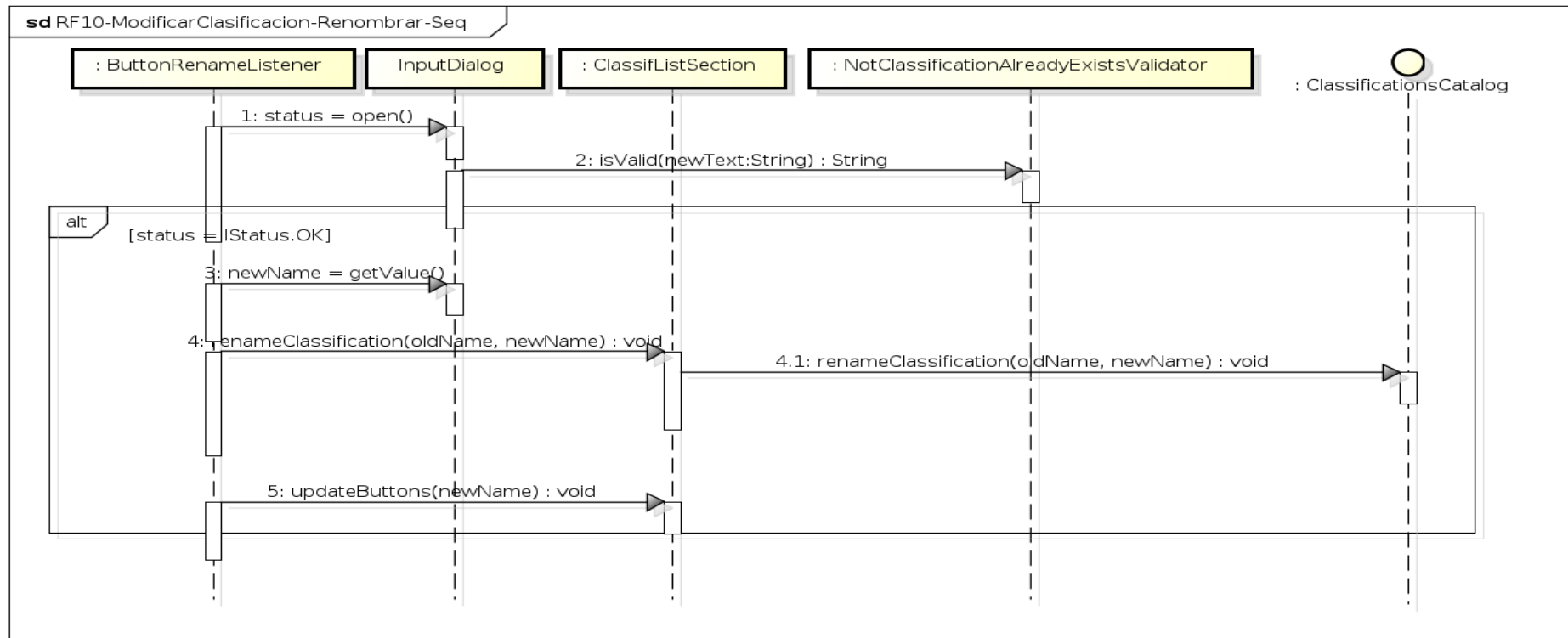
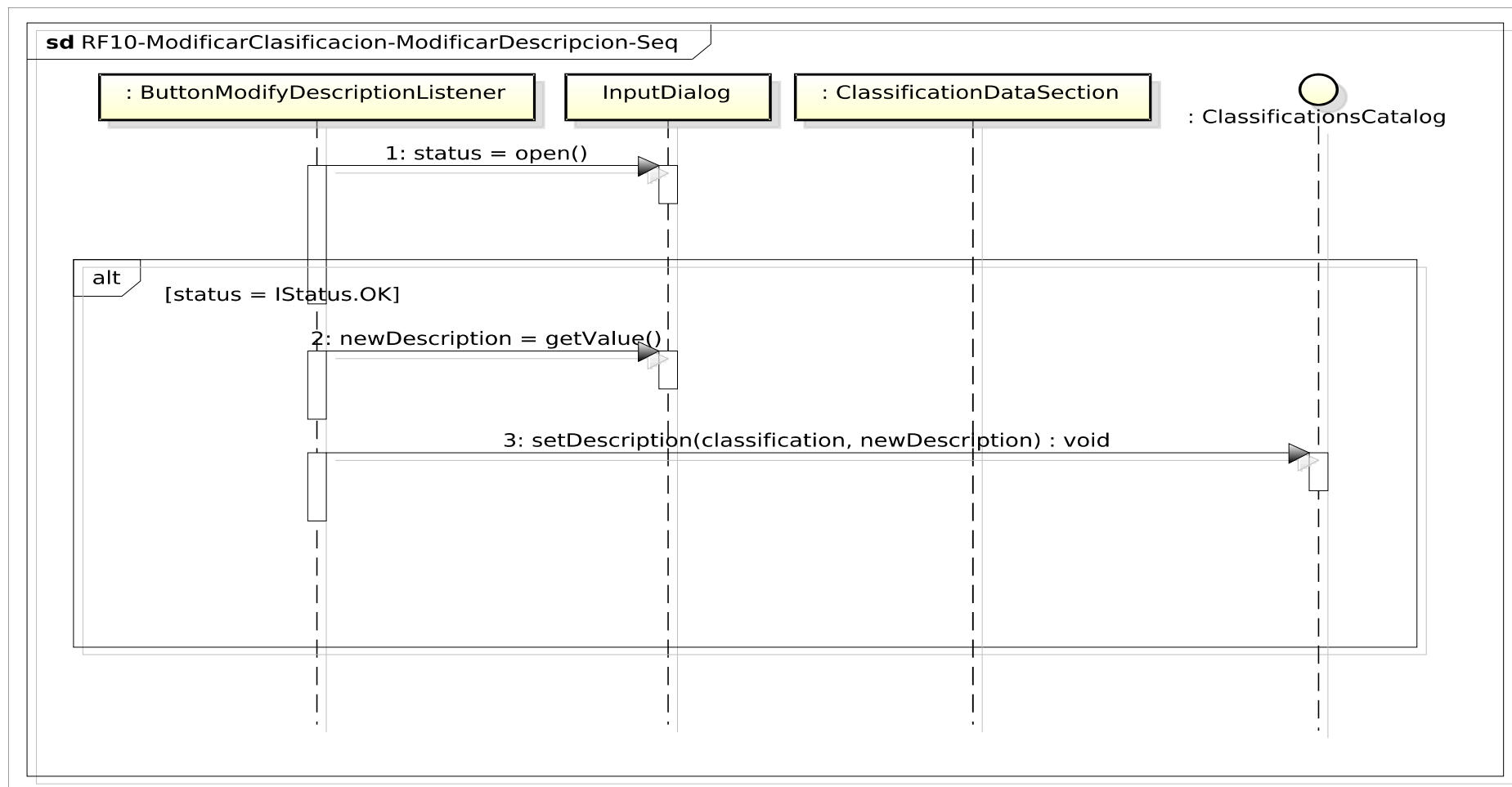


Ilustración 39: D. de secuencia de RF 10: Detalle - Renombrar una clasificación

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 30 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 40: 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

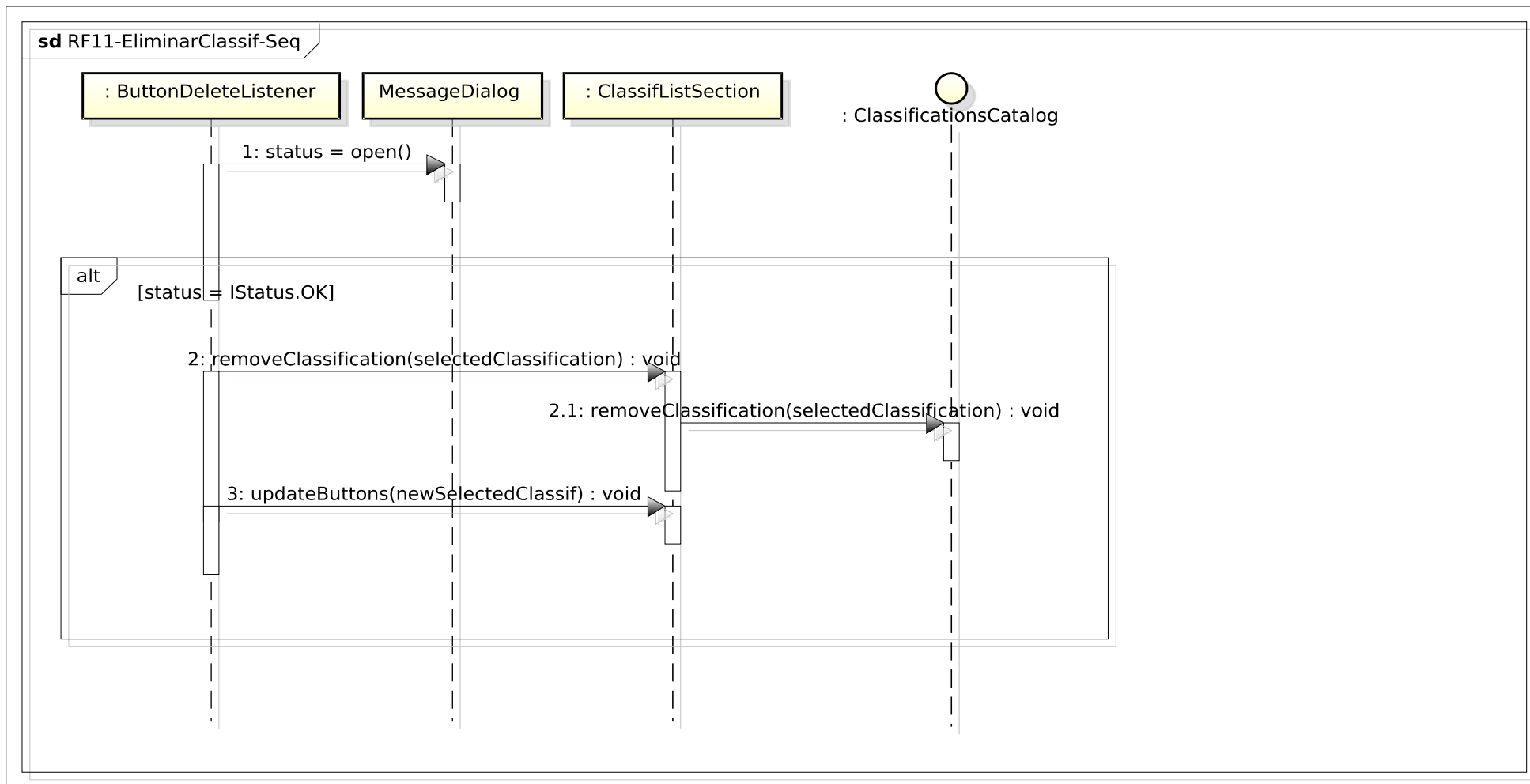


Ilustración 41: D. de secuencia de RF 11: Eliminar clasificación

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

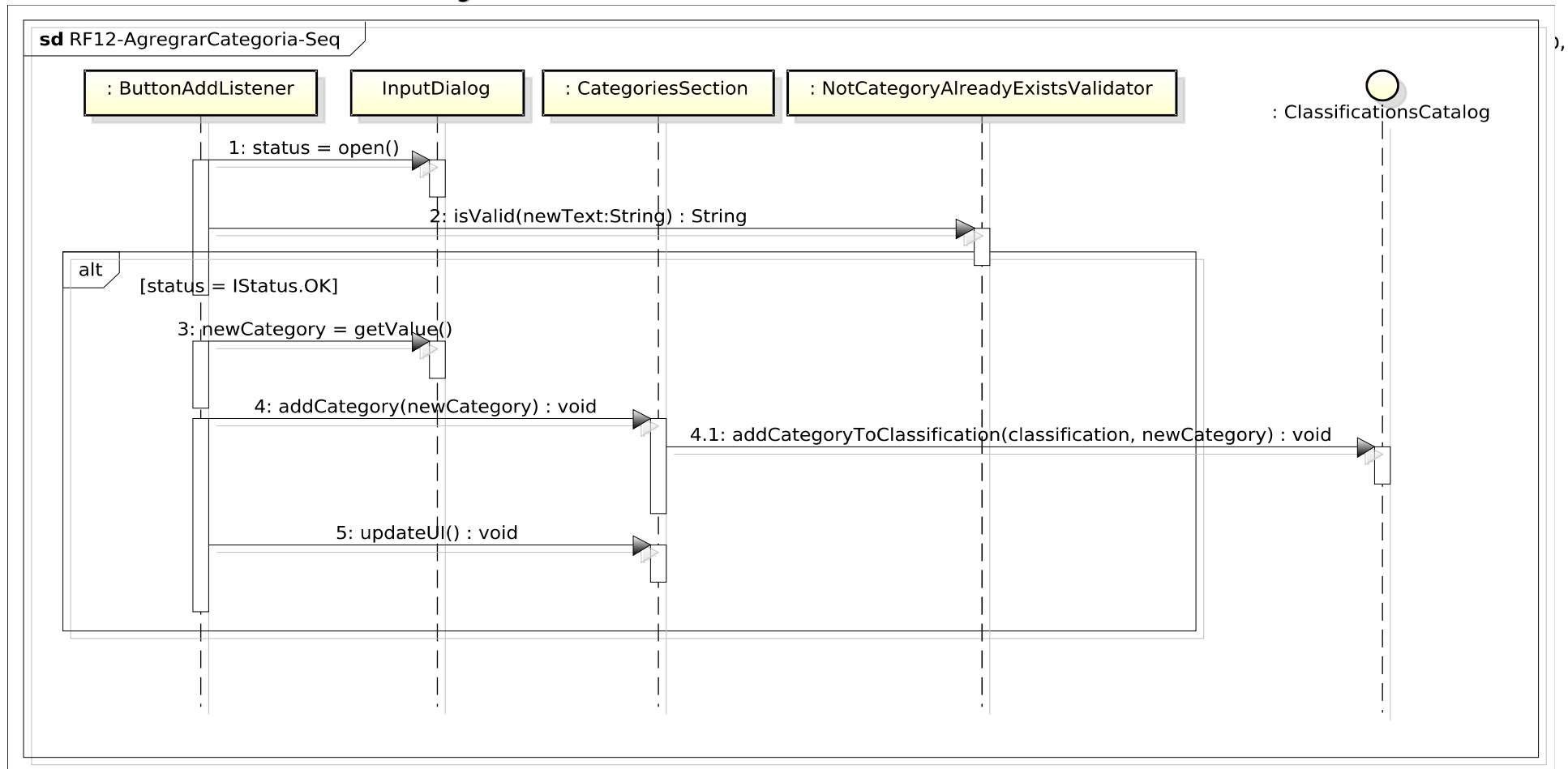


Ilustración 42: D. de secuencia de RF 12: Añadir categoría a una clasificación

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

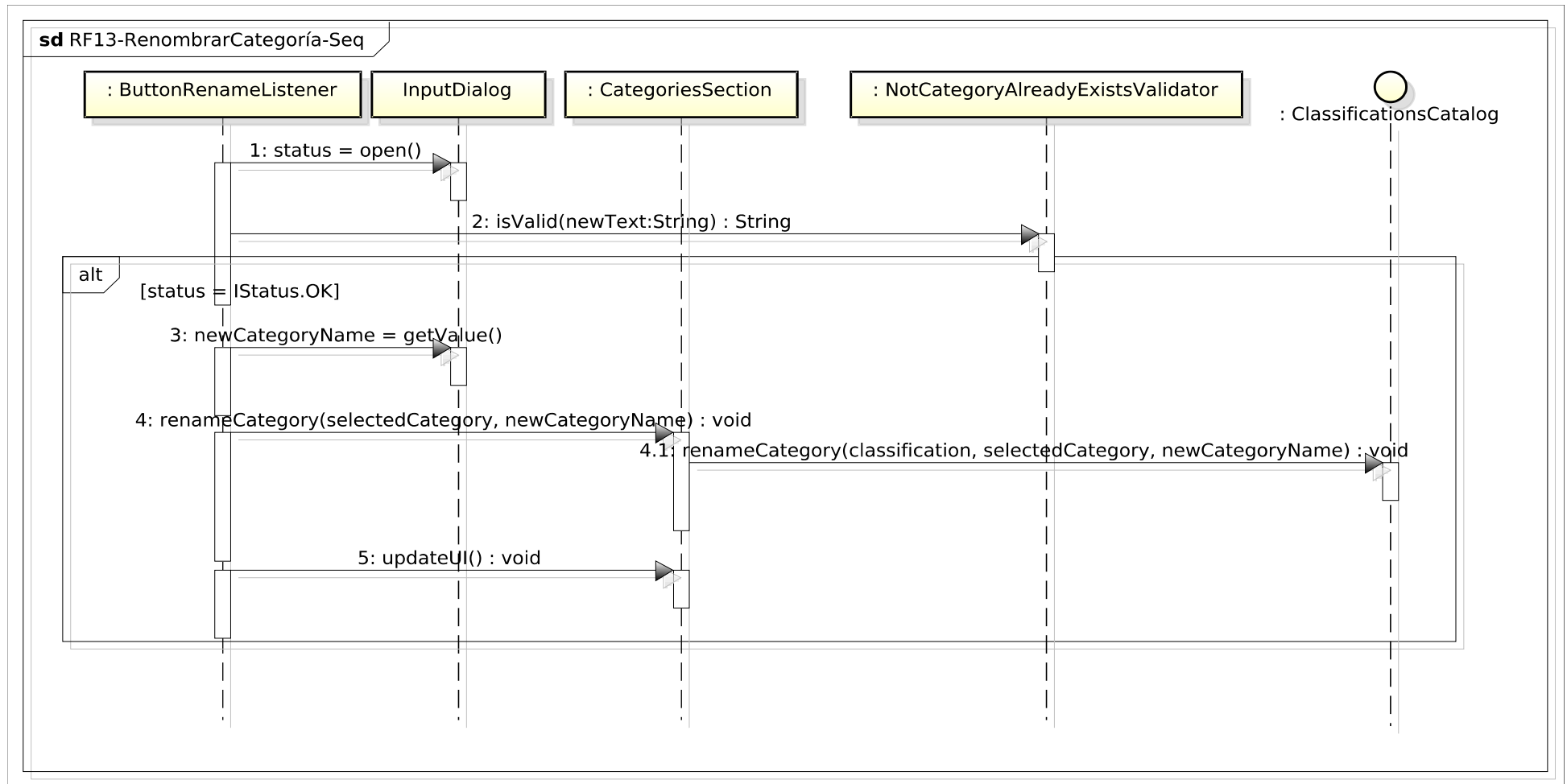


Ilustración 43: D. de secuencia de RF 13: Renombrar categoría

Este diagrama guarda paralelismo con el diagrama de Ilustración 30, 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

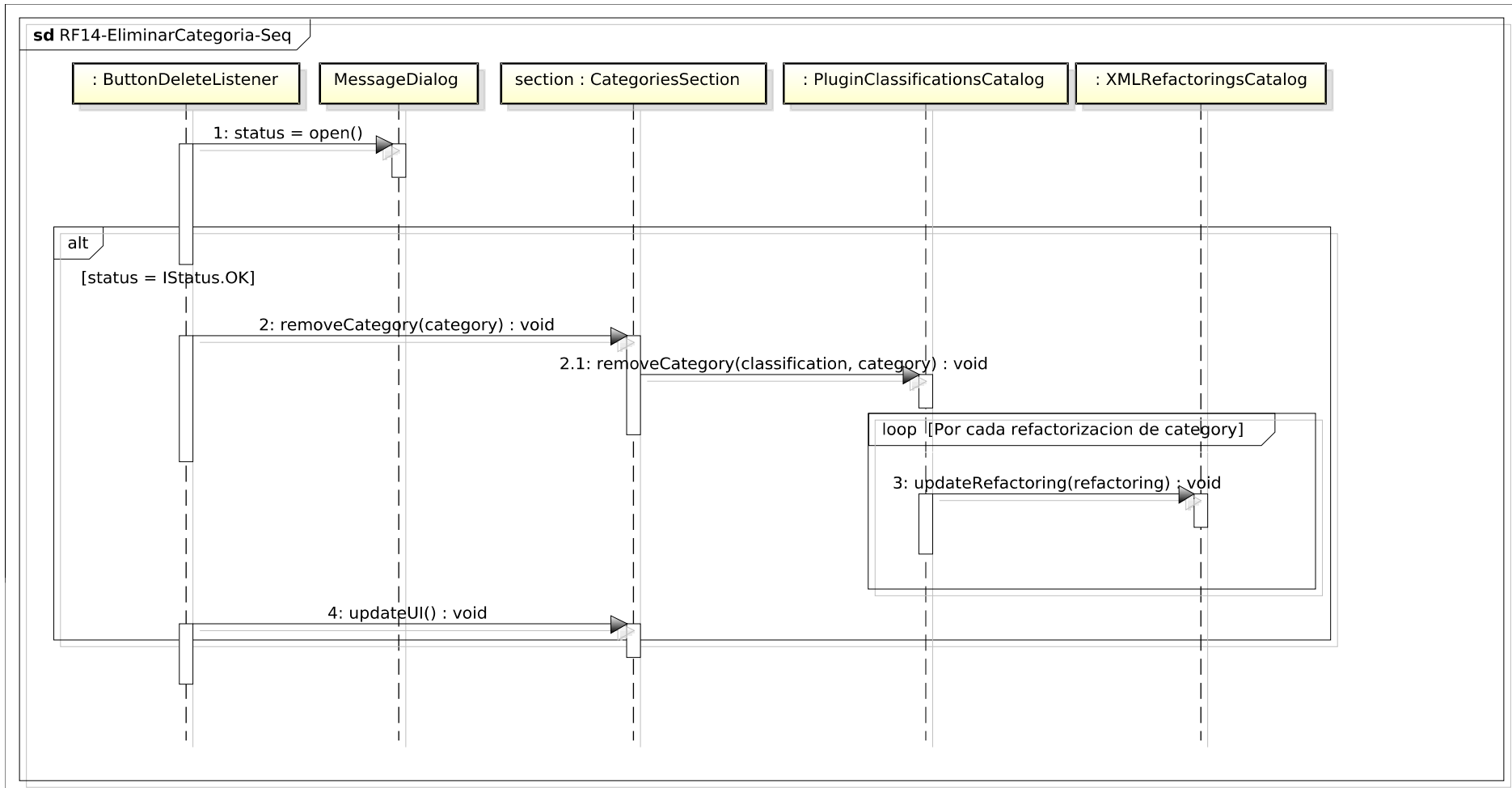
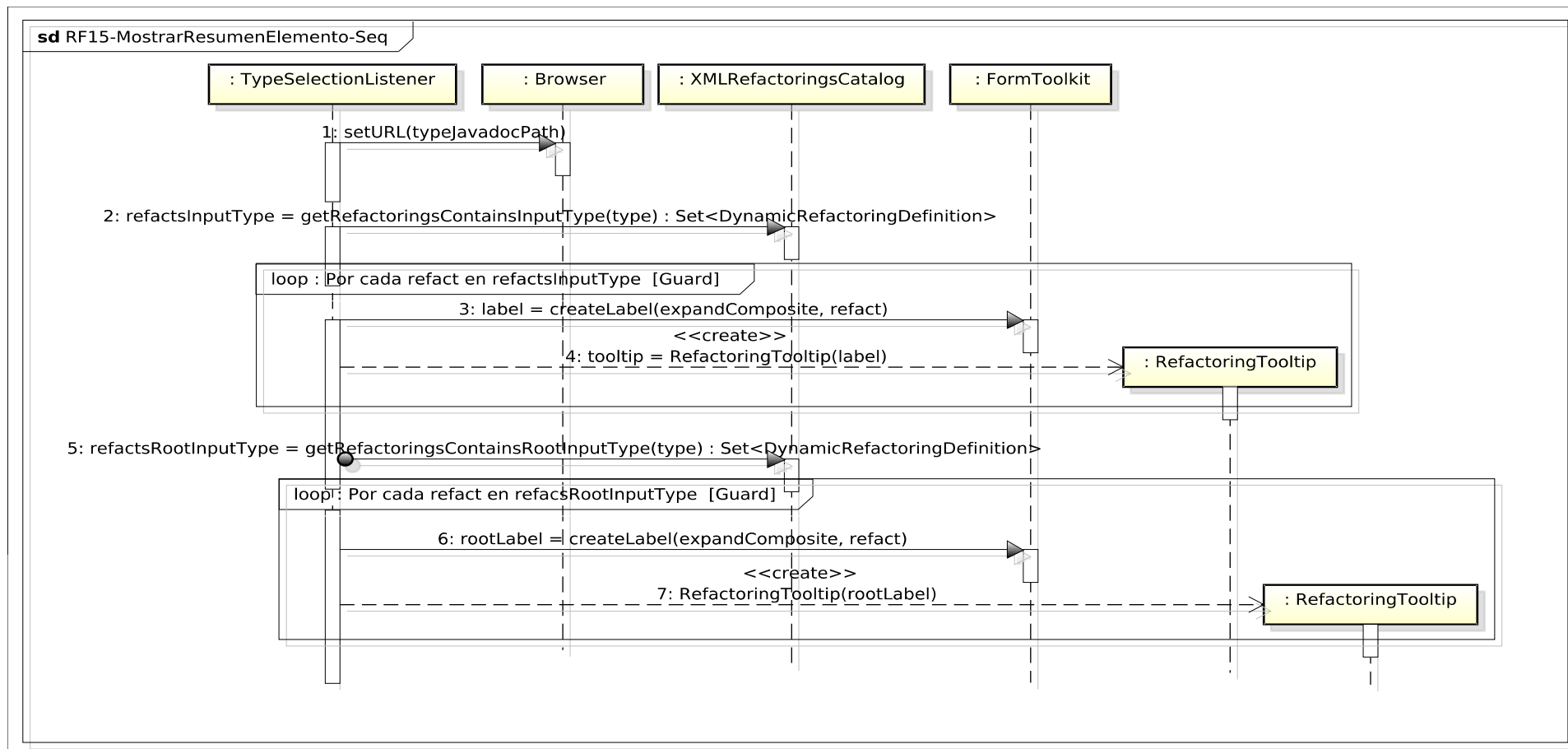


Ilustración 44: D. de secuencia de RF 14: Eliminar categoría de una clasificación

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 45: 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

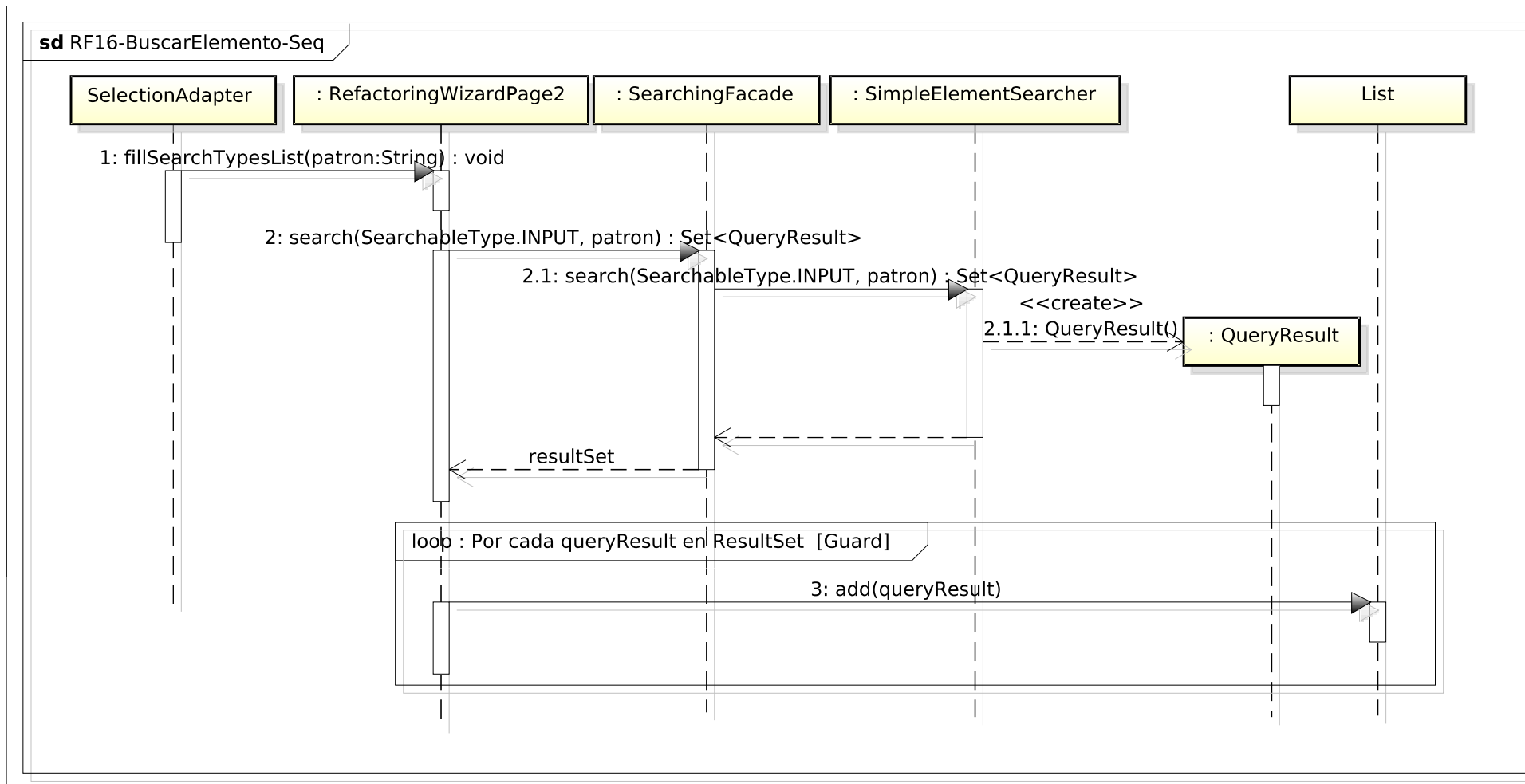


Ilustración 46: D. de secuencia de RF 16: Realizar búsqueda de elementos

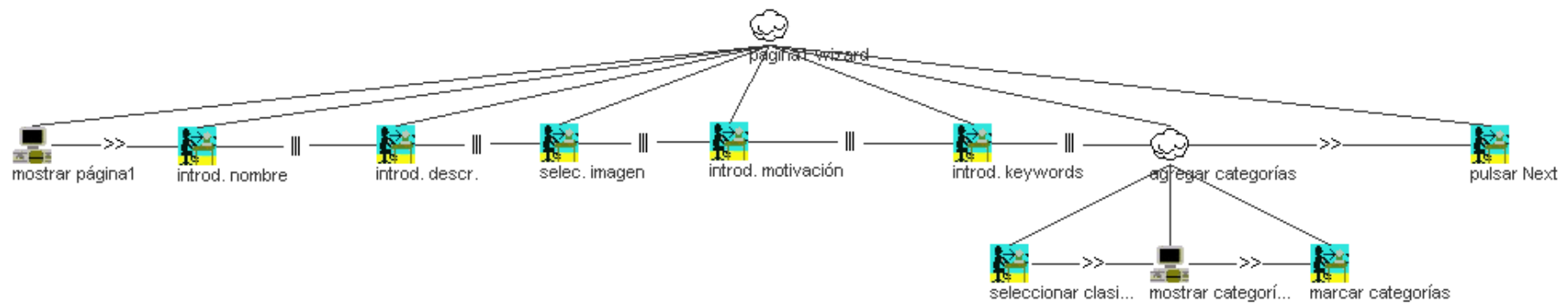
Para realizar la búsqueda de elementos la página del asistente llama a la fachada de búsqueda con el patrón a buscar y el tipo sobre el que se realizará la búsqueda (en este caso de tipo entrada). Esta delega al buscador quien se encarga de crear los resultados de la búsqueda y devolvérselos. Finalmente 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 47: Ctte de la primer página del asistente*

## 6.2. Visualizar catálogo de Refactorizaciones

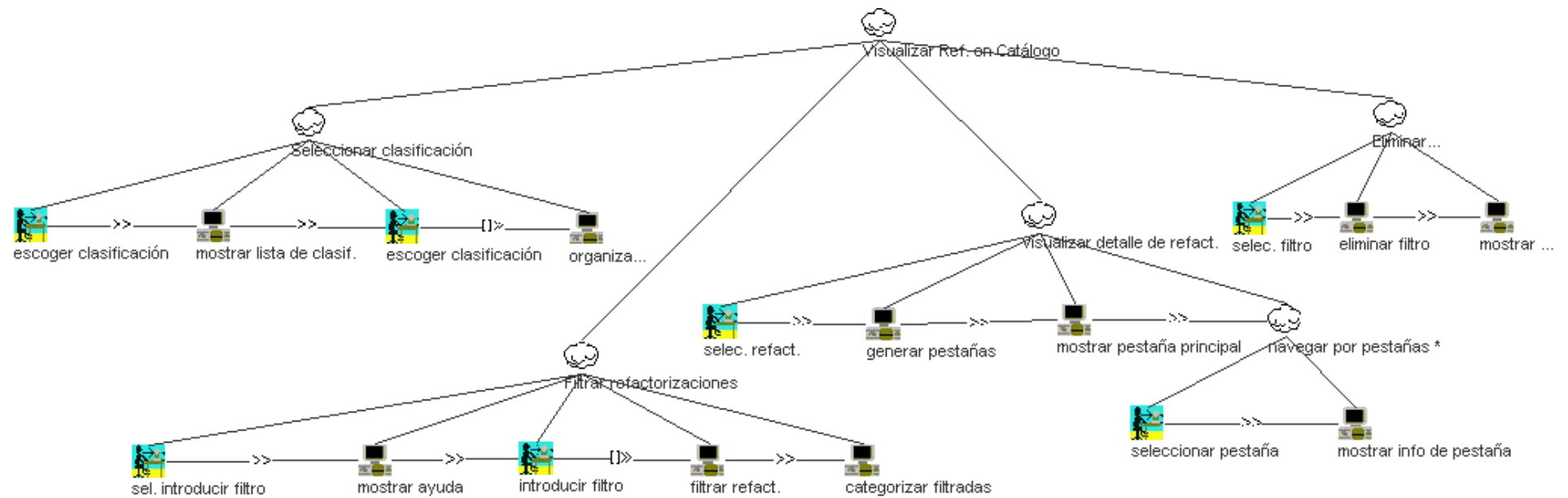


Ilustración 48: CTTE de visualizar catálogo de refactorizaciones

### 6.3. Editar clasificaciones

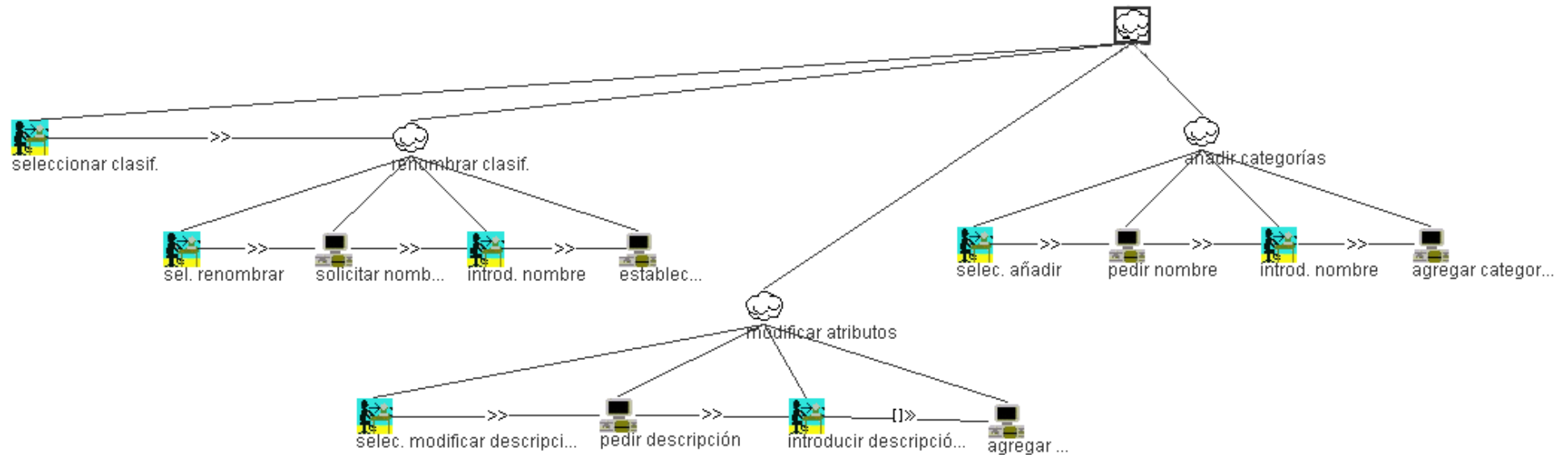


Ilustración 49: CTTE de edición de clasificaciones

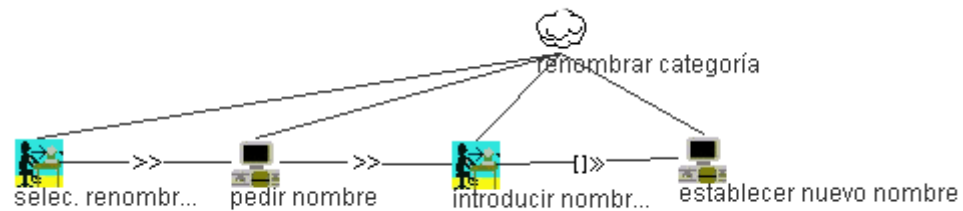


Ilustración 50: CTTE de renombrar una categoría

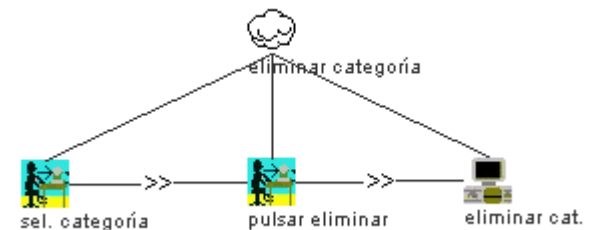


Ilustración 51: CTTE de eliminar categoría

#### 6.4. Añadir una nueva clasificación

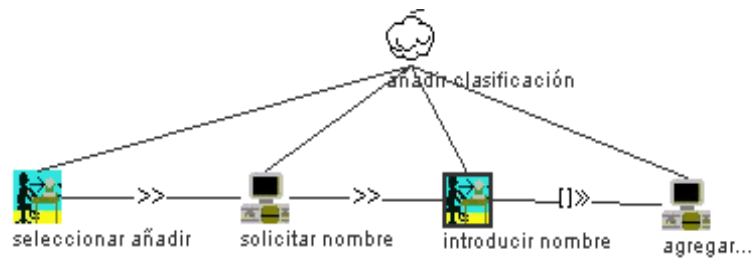


Ilustración 52: CTTE de agregar una nueva clasificación

#### 6.5. Eliminar una clasificación



Ilustración 53: 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 esta 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 reemplazó 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 atributos opcionales 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 54: D. de clases del PD Builder aplicado a InputParameter*

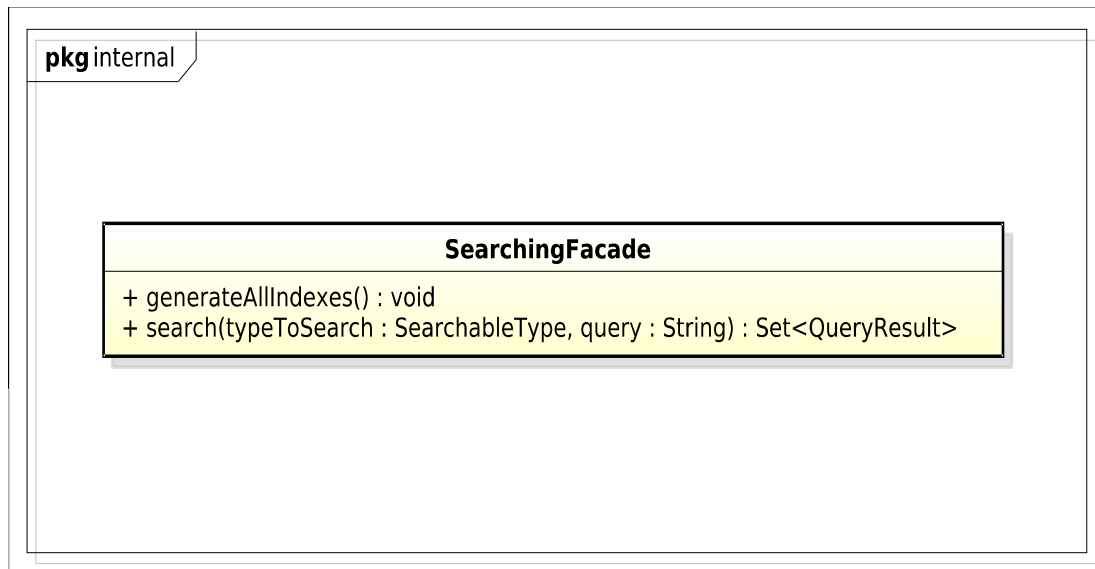
## 7.2. *Patrón Fachada*

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 puede 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 55: 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.

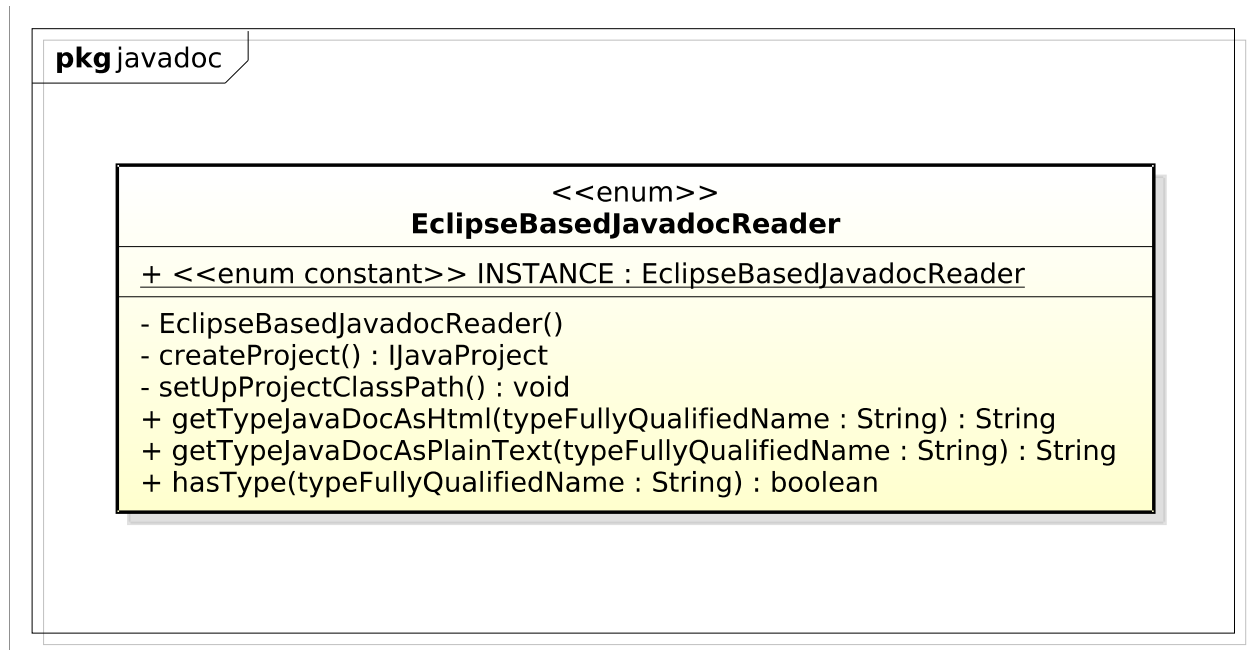


Otra implementación del patrón *Singleton* en Java es sugerida en [Bloch 2008] como alternativa a la implementación basada en un método fábrica. La implementación basada en un método otorga la flexibilidad de que da la posibilidad de cambiar de opinión y sustituir el comportamiento como *Singleton* por 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 56: 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.

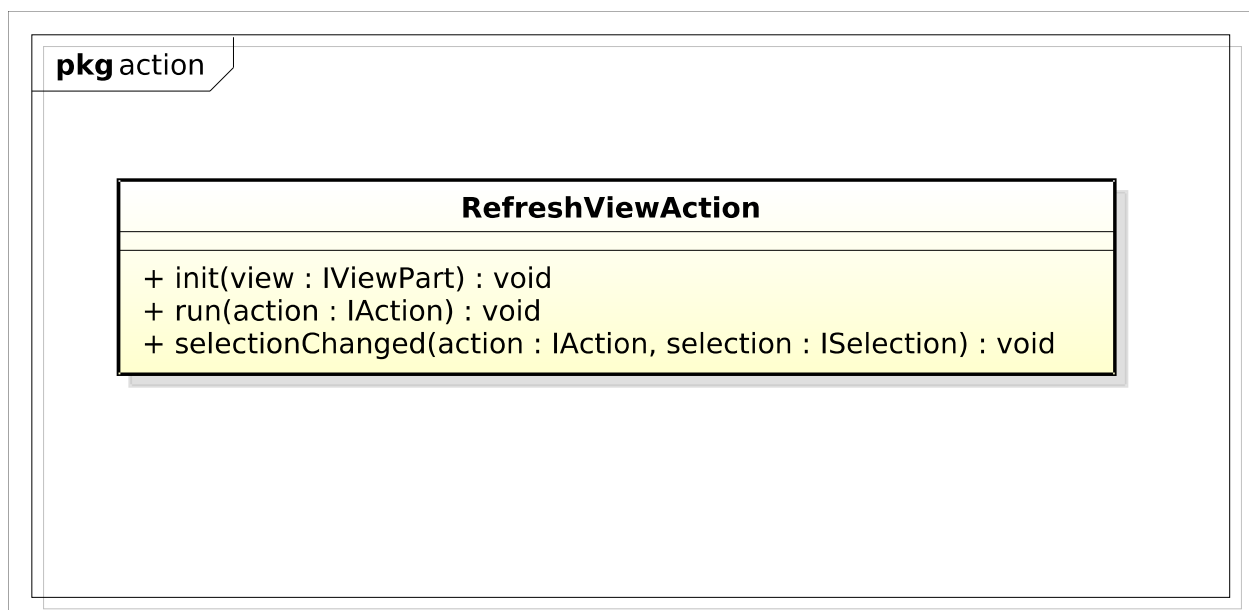
- Estructurar un sistema en base a transacciones.

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 que encapsula la acción de actualizar la vista del catálogo de refactorizaciones. Esta clase implementa la interfaz IviewActionDelegate que cumple el rol de la interfaz comando.



*Ilustración 57: 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 1: 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.interfaces	dynamicrefactoring.interfaces.view	dynamicrefactoring.interfaces.editor.classifieditor	dynamicrefactoring.interfaces.wizard	dynamicrefactoring.interfaces.wizard.search.internal	dynamicrefactoring.interfaces.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 2: 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
<b>RF 1: Visualizar refactorizaciones según clasificación</b>	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 3: Pruebas de RF1: Visualizar refactorizaciones según clasificación

## 9.2. RF 2: Refrescar visualización de refactorizaciones

Requisito	Aspectos a probar	Resultado esperado
<b>RF 2: Refrescar visualización de refactorizaciones</b>	Si el usuario añade una refactorización la vista debe reflejarlo al pulsar el botón de refrescar.	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 debiera seleccionarse otra y actualizar la vista con la nueva selección.

Tabla 4: Pruebas de RF 2: Refrescar visualización de refactorizaciones

### 9.3. RF 3: Añadir filtro de refactorizaciones

Requisito	Aspectos a probar	Resultado esperado
<b>RF 3: Añadir filtro de refactorizaciones</b>	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 5: Pruebas de RF 3: Añadir filtro de refactorizaciones



#### 9.4. RF 4: Seleccionar opción aplicar filtro

Requisito	Aspectos a probar	Resultado esperado
<b>RF 4: Seleccionar opción aplicar filtro</b>	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 un filtro desactivado se activa, todas las refactorizaciones que no cumplen con dicho filtro volverán a ocultarse.

Tabla 6: Pruebas de RF 4: Seleccionar opción aplicar filtro

#### 9.5. RF 5: Eliminar filtro de refactorizaciones

Requisito	Aspectos a probar	Resultado esperado
<b>RF 5: Eliminar filtro de refactorizaciones</b>	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 7: Pruebas de RF 5: Eliminar filtro de refactorizaciones

### 9.6. RF 6: Eliminar todos los filtros de refactorizaciones

Requisito	Aspectos a probar	Resultado esperado
<b>RF 6: Eliminar todos los filtros de refactorizaciones</b>	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 8: 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
<b>RF 7: Seleccionar opción ver refactorizaciones filtradas</b>	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 9: Pruebas de RF 7: Seleccionar opción ver refactorizaciones filtradas

### 9.8. RF 8: Visualizar detalle refactorización

Requisito	Aspectos a probar	Resultado esperado
<b>RF 8: Visualizar detalle refactorización</b>	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 10: Pruebas de RF 8: Visualizar detalle refactorización

### 9.9. RF 9: Añadir clasificación

Requisito	Aspectos a probar	Resultado esperado
<b>RF 9: Añadir clasificación</b>	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 11: Pruebas de RF 9: Añadir clasificación

### 9.10. RF 10: Editar clasificación

Requisito	Aspectos a probar	Resultado esperado
<b>RF 10: Editar clasificación</b>	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 12: Pruebas de RF 10: Editar clasificación

### 9.11. RF 11: Eliminar clasificación

Requisito	Aspectos a probar	Resultado esperado
<b>RF 11: Eliminar clasificación</b>	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 13: Pruebas de RF 11: Eliminar clasificación

### 9.12. RF 12: Añadir categoría a una clasificación

Requisito	Aspectos a probar	Resultado esperado
<b>RF 12: Añadir categoría a una clasificación</b>	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 14: Pruebas de RF 12: Añadir categoría a una clasificación

### 9.13. RF 13: Renombrar categoría de una clasificación

Requisito	Aspectos a probar	Resultado esperado
<b>RF 13: Renombrar categoría de una clasificación</b>	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 15: Pruebas de RF 13: Renombrar categoría de una clasificación*

#### 9.14. RF 14: Eliminar categoría de una clasificación

Requisito	Aspectos a probar	Resultado esperado
<b>RF 14: Eliminar categoría de una clasificación</b>	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 categorías 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 16: Pruebas de RF 14: Eliminar categoría de una clasificación*

#### 9.15. RF 15: Mostrar resumen elemento seleccionado

Requisito	Aspectos a probar	Resultado esperado
<b>RF 15: Mostrar resumen elemento seleccionado</b>	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 17: Pruebas de RF 15: Mostrar resumen elemento seleccionado*

### 9.16. RF 16: Realizar búsqueda de elementos

Requisito	Aspectos a probar	Resultado esperado
<b>RF 16: Realizar búsqueda de elementos</b>	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 18: 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 asignatura<sup>1</sup>, 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 kiviati 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 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:

---

1 <http://pisuerga.inf.ubu.es/lsi/Asignaturas/SI>



#### 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.
- **javamoon-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.
- **jgrapht-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

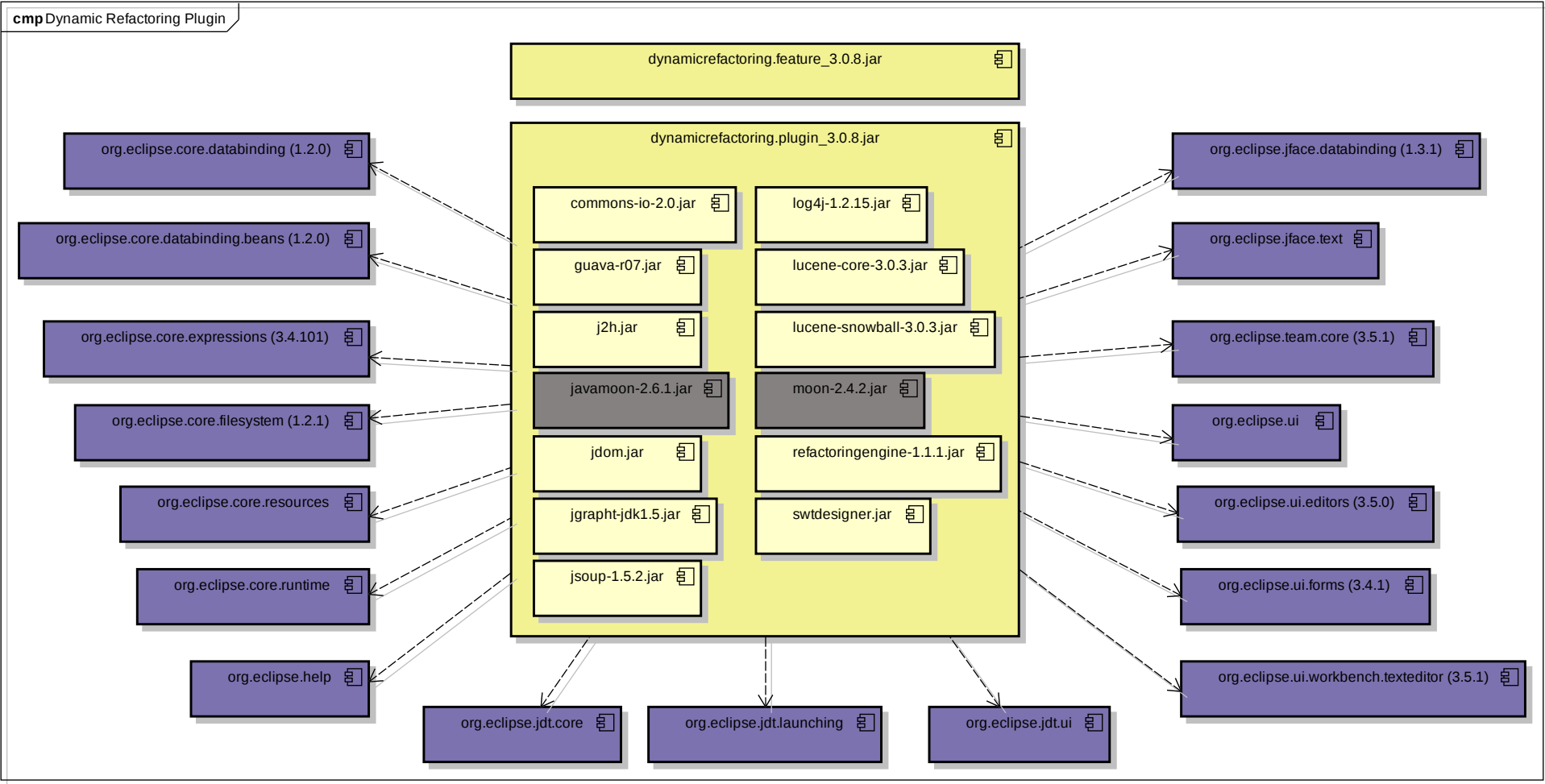


Ilustración 58: Diagrama de componentes

Leyenda:

Dynamic Refactoring Plugin

Plugins Eclipse

Bibliotecas auxiliares

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	

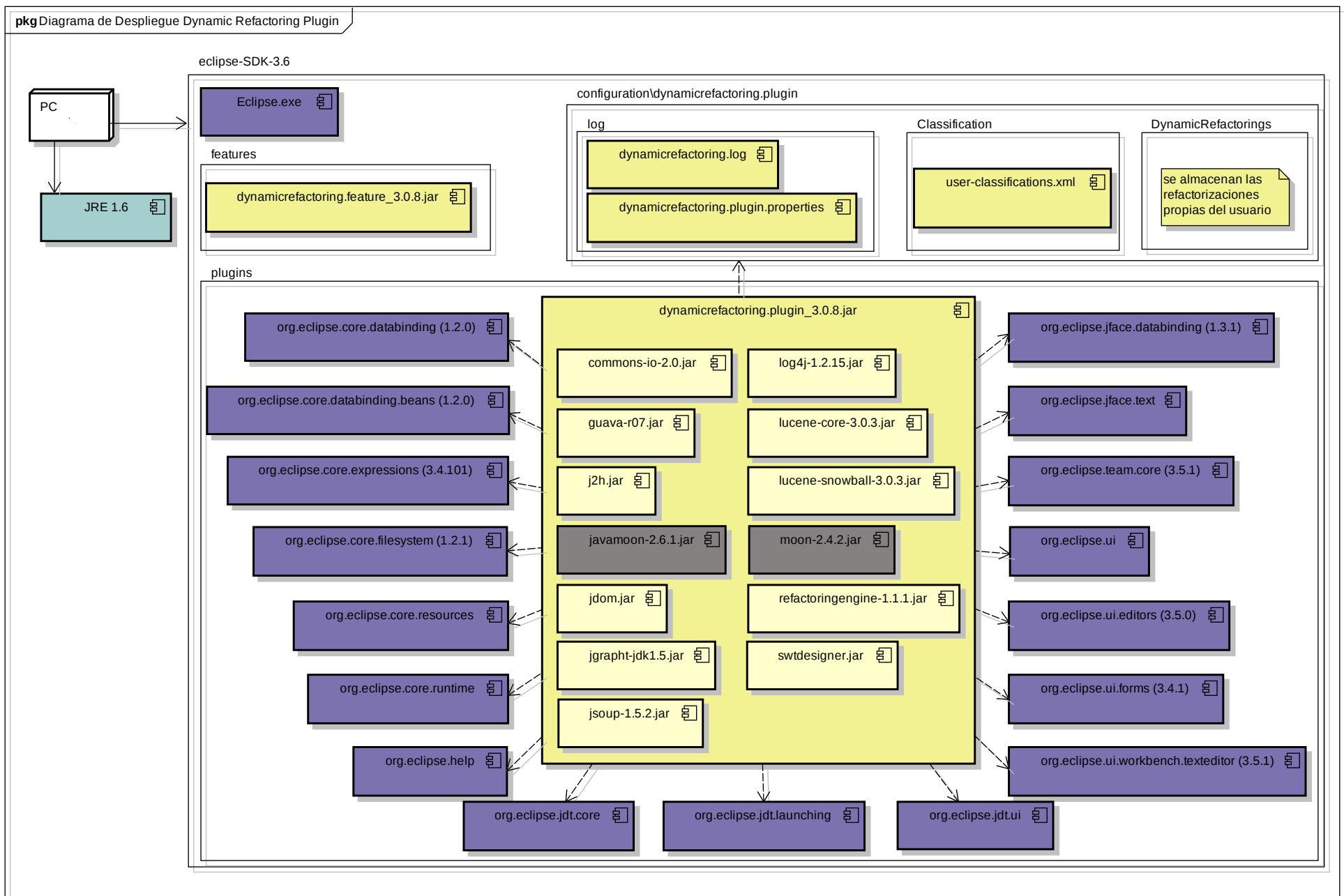


Ilustración 59: Diagrama de despliegue