

**PARTE I
DESCRIPCIÓN DEL PROYECTO**

ÍNDICE DE CONTENIDO

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

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

ÍNDICE DE ILUSTRACIONES

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

ÍNDICE DE TABLAS

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

LISTA DE CAMBIOS

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

1. INTRODUCCIÓN

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

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

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

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

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

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

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

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

2. OBJETIVOS DEL PROYECTO

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

2.1. Requisitos del software

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

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

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

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

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

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

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

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

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

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

2.2. *Objetivos técnicos*

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

2.2.1. Mejora del proceso de desarrollo

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

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

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

2.2.2. Mejora de la calidad del código

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

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

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

3. CONCEPTOS TEÓRICOS

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

3.1. *Modelo conceptual MOON*

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

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

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

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

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

3.2. Refactorización de código

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

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

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

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

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

3.2.1. Pasos al aplicar una refactorización

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

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

3.2.2. Construcción dinámica de refactorizaciones

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

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

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

seguir los siguientes pasos para la construcción de refactorizaciones:

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

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

3.3. Programación orientada a objetos

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

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

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

3.4. Patrones de diseño

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

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

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

Los elementos que constituyen un patrón de diseño son:

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

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

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

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

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

Tabla 1: Clasificación de patrones de diseño

3.5. Metodología de desarrollo

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

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

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

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

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

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

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

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

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

3.6. Integración continua

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

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

3.6.1. Teoría

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

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

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

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

3.6.2. Prácticas recomendadas

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

Mantener un repositorio de código

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

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

Automatizar el proceso de construcción

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

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

Subir cambios a la rama principal todos los días

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

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

Cada commit a la rama principal debe disparar la construcción

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

La construcción debe ser rápida

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

Ejecutar las pruebas en una copia del entorno de producción

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

Facilitar el acceso a los últimos entregables

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

Resultados del último proceso accesibles

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

Automatizar el despliegue

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

4. TÉCNICAS Y HERRAMIENTAS

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

4.1. Automatización de la construcción - Maven

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

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

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

4.1.1. POM: Project Object Model

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

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

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

4.1.2. Plugins

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

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

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

```
mvn compiler:compile
```

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

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

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

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

4.1.3. Ciclos de vida de la fase de construcción

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

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

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

```
mvn test
```

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

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

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

4.1.4. Dependencias

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

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

4.2. Control de versiones - Git

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

4.2.1. Ventajas del control de versiones

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

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

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

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

4.2.2. Introducción y breve historia de Git

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

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

Torvalds definió como criterios para el próximo sistema:

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

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

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

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

4.2.3. Características de Git

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

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

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

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

4.2.4. *El modelo de objetos de Git*

A continuación se detalla el modelo de objetos de Git:

El SHA

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

tiene un aspecto como el siguiente:

```
6ff87c4664981e4397625791c8ea3bbb5f2279a3
```

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

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

Los objetos

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

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

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

Diferente de otros SCM

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

Objeto blob

Un blob generalmente contiene el contenido de un fichero.

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

Ilustración 1: Representación de un objeto blob

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

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

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

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

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

Objeto tree

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

Ilustración 2: Representación de un objeto árbol

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

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

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

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

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

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

Objeto commit

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

Ilustración 3: Representación de un objeto commit

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

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

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

Como se ha podido ver un commit se define por:

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

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

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

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

El modelo de objetos

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

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

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

```

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

2 directories, 3 files

```

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

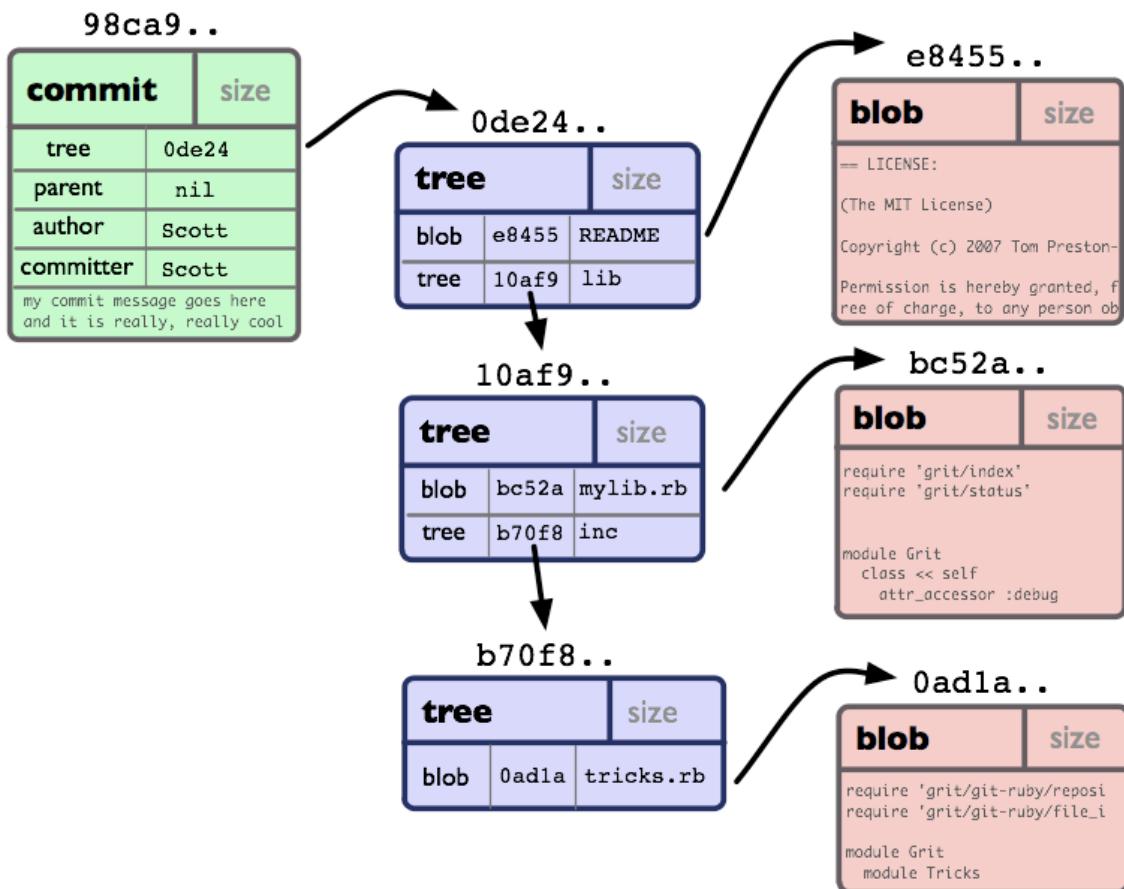


Ilustración 4: Modelo completo de objetos de Git

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

Etiquetas (tags)

49e11..

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

Ilustración 5: Representación de un objeto tag

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

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

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

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

4.2.5. Elección de Git como sistema de control de versiones

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

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

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

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

Ventajas

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

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

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

Desventajas

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

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

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

Conclusión

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

4.3. Gestión de tareas - Fogbugz

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

4.3.1. Ventajas de la gestión de tareas

Las ventajas de la gestión de tareas son las siguientes:

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

ágiles.

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

4.3.2. FogBugz

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

Características:

Gestión de proyectos:

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

Gestión del tiempo:

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

Gestión general:

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

los gráficos de datos actuales o históricos.

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

4.3.3. Foglyn

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

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

Características:

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

4.4. Prototipado de interfaces - Balsamiq Mockup

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

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

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

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

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

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

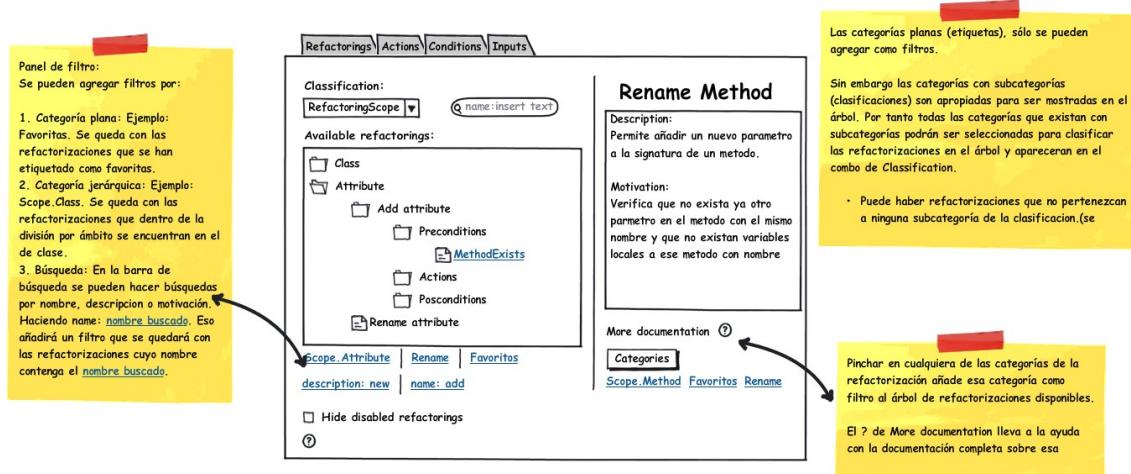


Ilustración 6: Prototipo creado con Balsamiq Mockup

4.5. Pruebas de interfaz - SWTBot

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

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

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

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

4.6. Integración continua - Hudson

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

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

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

Las características que Hudson ofrece se pueden resumir:

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

The screenshot shows the Hudson web interface. At the top, there's a blue header bar with the Hudson logo on the left and a search bar on the right. Below the header, there's a sidebar on the left containing links like 'Crear nueva Tarea', 'Administrar Hudson', 'Actividad', 'Historia de ejecuciones', 'Dependencia entre proyectos', and 'Comprobar firma de ficheros'. The main area is titled 'Todo' and contains a table with two rows of project information. The columns are labeled 'S' (Status), 'W' (Workdir), 'Tarea' (Job), 'Último éxito' (Last success), 'Último fallo' (Last failure), and 'Última duración' (Last duration). The first row shows a blue circle icon for status, a sun and cloud icon for workdir, the job name 'dynamicrefactoring.plugin', a success time of '48 Min (#106)', a failure time of '8 dias 13 Hor (#103)', and a duration of '8 Min 59 Seg'. The second row shows a grey circle icon for status, a sun and cloud icon for workdir, the job name 'dynamicrefactoring.plugin(local)', a success time of '3 Mes 2 dias (#10)', a failure time of '1 Mes 0 dias (#29)', and a duration of '9 Min 0 Seg'. At the bottom of the main area, there are icons for 'S' (Small), 'M' (Medium), and 'L' (Large) and links to subscribe to RSS feeds for 'todos los trabajos', 'sólo los fallidos', and 'los más recientes'.

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

4.7. Control de calidad del código - Sonar

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

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

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

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

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

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

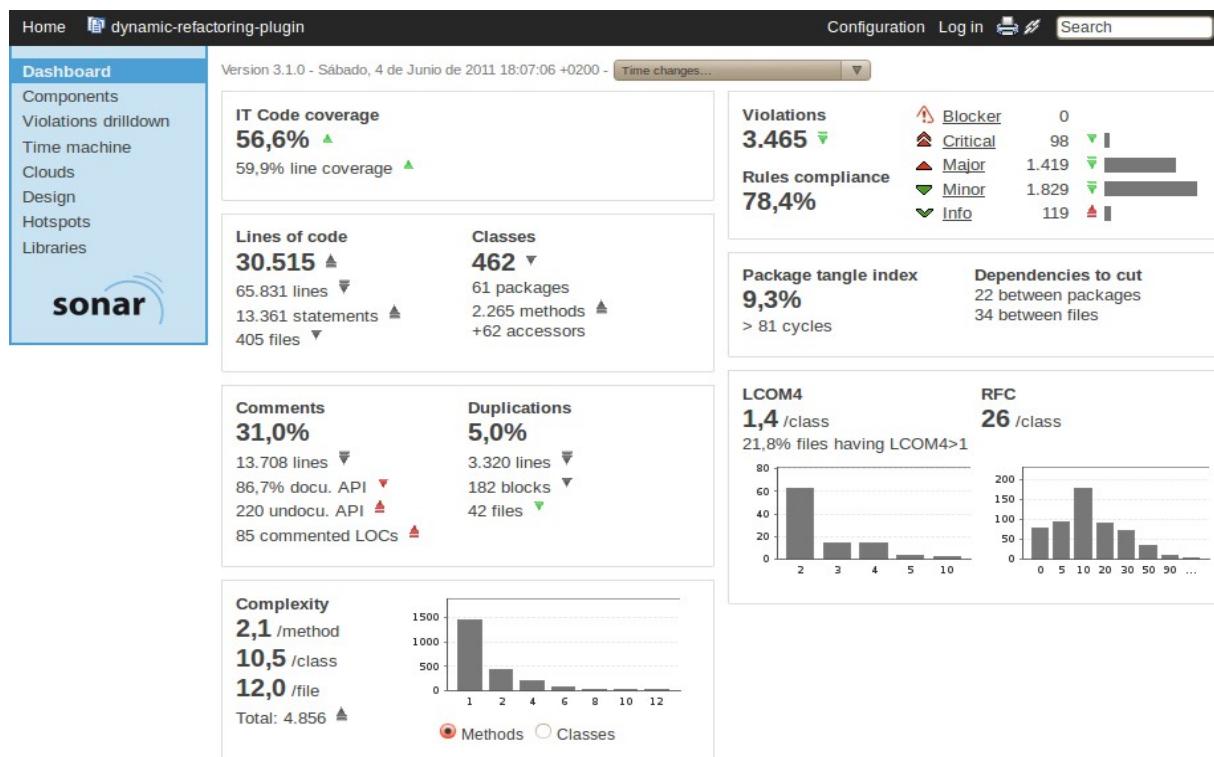


Ilustración 8: Pantalla principal de Sonar

5. ASPECTOS RELEVANTES DEL PROYECTO

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

5.1. Ciclo de vida

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

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

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

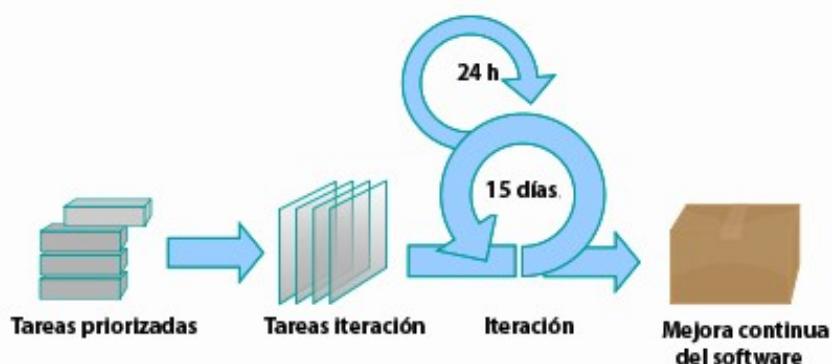


Ilustración 9: Ciclo de vida utilizado en el proyecto

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

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

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

5.2. Proceso de integración continua

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

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

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

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

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

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

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

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

5.2.1. Tareas incorporadas al proceso de construcción

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

Los pasos necesarios para introducir esta mejora fueron los siguientes:

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

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

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

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

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

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

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

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

5.2.2. Configuración avanzada

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

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

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

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

Configuración inicial

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

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

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

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

Configuración definitiva

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

5.3. Repositorio único de refactorizaciones y clasificaciones

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

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

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

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

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

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

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

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

5.3.1. Solución adoptada

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

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

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

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

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

El resto de clases es totalmente independiente del funcionamiento interno del

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

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

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

5.4. Versionado de refactorizaciones

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

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

Para dar solución a este problema se plantearon tres aproximaciones:

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

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

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

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

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

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

opción de realizar una copia modificable.

5.5. Carga de clases

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

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

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

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

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

5.6. Buscador del asistente de refactorizaciones

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

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

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

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

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

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

5.7. Bug encontrado en Eclipse Forms

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

6. TRABAJOS RELACIONADOS

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

6.1. Mejoras incorporadas

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

editables.

6.2. Desaparición de alternativas

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

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

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

6.3. JDeodorant

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

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

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

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

6.3.1. Ventajas

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

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

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

6.3.2. Desventajas

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

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

6.3.3. Conclusión

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

6.4. Refactory

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

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

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

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

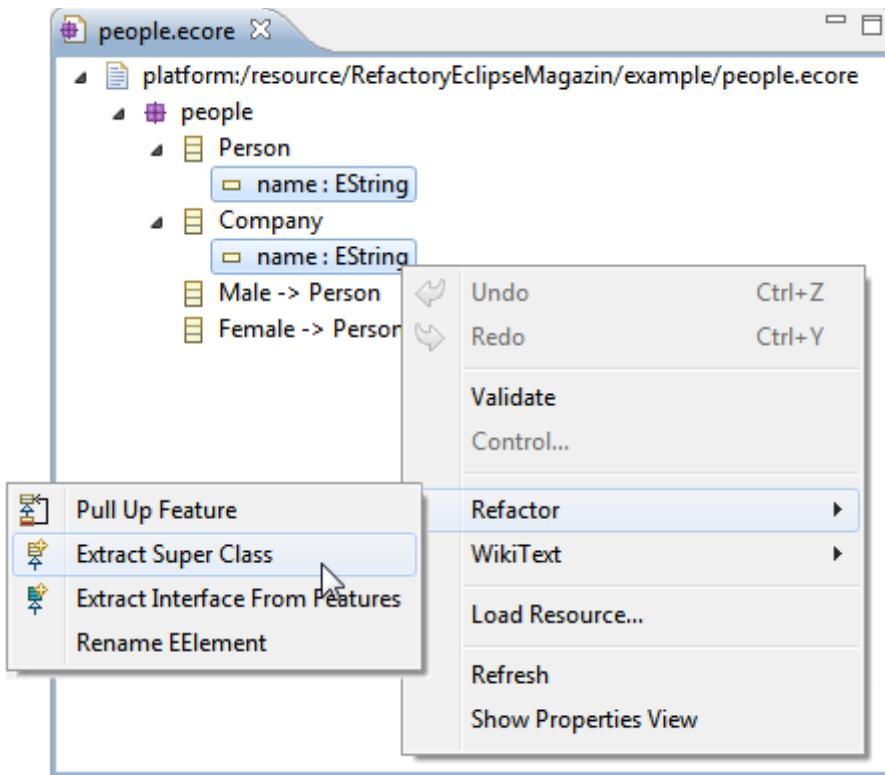


Ilustración 10: Refactory aplicando una refactorización a un metamodelo

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

6.4.1. Ventajas

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

6.4.2. Desventajas

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

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

6.5. RefactoringNG

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

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

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

```

    Literal [kind: INT_LITERAL, value: 0]
}

```

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

6.5.1. Árbol, atributos y contenido

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

Identifier

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

```
Literal [kind:Identifier]
```

indica cualquier identificador y :

```
Literal [kind: INT_LITERAL]
```

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

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

```

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

```

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

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

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

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

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

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

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

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

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

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

indica bien una suma o una sustracción.

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

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

```

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

```

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

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

```

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

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

} ->

Binary [kind: b#kind] {

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

}

```

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

```

Variable [id: v] {

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

} ->

Variable [name: v#name] {

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

}

```

6.5.2. Listas

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

```
List<Expression>
```

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

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

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

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

El atributo `size` especifica el número de elementos.

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

```
}
```

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

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

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

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

6.5.3. *NoneOf*

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

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

```

    } ,

    Literal [kind: DOUBLE_LITERAL, value: 3.14]

} ->

Assignment {

    Expression [ref: i],

    MemberSelect [identifier: "PI"] {

        Identifier [name: "Math"]

    }

}

```

6.5.4. Conclusiones

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

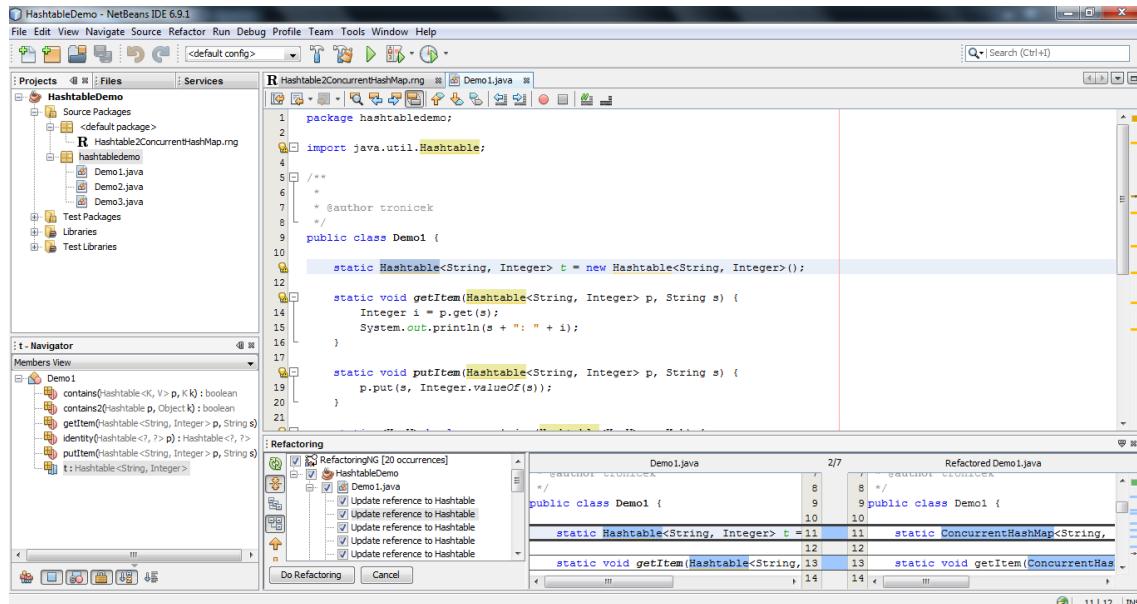


Ilustración 11: Previsualización de resultados con RefactoringNG

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

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

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

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

6.6. API de refactorizaciones de Eclipse

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

6.6.1. Pasos del proceso de refactorización

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

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

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

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

6.6.2. Modificaciones necesarias más importantes

Por tanto los cambios necesarios más importantes para definir una refactorización

son:

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

6.6.3. Conclusiones

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

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

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

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

7. CONCLUSIONES Y LÍNEAS DE TRABAJO FUTURO

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

7.1. Conclusiones

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

7.1.1. Requisitos funcionales

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

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

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

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

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

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

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

7.1.2. *Objetivos técnicos*

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

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

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

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

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

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

7.2. Líneas de trabajo futuro

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

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

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

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

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