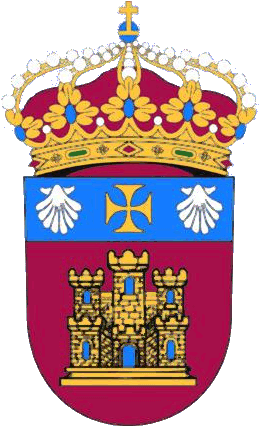
Universidad de Burgos

Escuela Politécnica Superior

Ingeniería Informática



Sistemas Informáticos

Plugin de Refactorización 3.0

Memoria

**Alumnos:**

Míryam Gómez San Martín

Íñigo Mediavilla Saiz

**Tutor:**

Raúl Marticorena Sánchez

**Burgos, Mayo de 2011**

# Índice de contenido

[Índice de contenido 2](#_Toc286652238)

[Índice de ilustraciones 4](#_Toc286652239)

[Índice de tablas 4](#_Toc286652240)

[1. Agradecimientos 5](#_Toc286652241)

[2. Lista de cambios 6](#_Toc286652242)

[2.1. Cambios 6](#_Toc286652243)

[3. Introducción 7](#_Toc286652244)

[4. Objetivos del proyecto 8](#_Toc286652245)

[4.1. Objetivos Técnicos 8](#_Toc286652246)

[5. Conceptos teóricos 9](#_Toc286652247)

[6. Comparativa entre alternativas 10](#_Toc286652248)

[6.1. Comparar con la versión anterior del proyecto 10](#_Toc286652249)

[6.2. Desaparición de alternativas respecto a versiones anteriores del plugin 10](#_Toc286652250)

[6.3. Jdeodorant 11](#_Toc286652251)

[6.3.1. Ventajas 11](#_Toc286652252)

[6.3.2. Desventajas 11](#_Toc286652253)

[6.3.3. Conclusión 12](#_Toc286652254)

[6.4. Refactory 12](#_Toc286652255)

[6.4.1. Ventajas 13](#_Toc286652256)

[6.4.2. Desventajas 13](#_Toc286652257)

[6.5. RefactoringNG 14](#_Toc286652258)

[6.5.1. Introducción 14](#_Toc286652259)

[6.6. Comparar con la api de refactorizaciones de Eclipse 20](#_Toc286652260)

[7. Técnicas y herramientas 21](#_Toc286652261)

[7.1. Programación orientada a objetos 21](#_Toc286652262)

[7.2. Patrones de diseño 21](#_Toc286652263)

[7.3. Integración continua 23](#_Toc286652264)

[7.3.1. Teoría 23](#_Toc286652265)

[7.3.2. Prácticas recomendadas 24](#_Toc286652266)

[7.3.3. Apache Maven 26](#_Toc286652267)

[7.4. Control de versiones 29](#_Toc286652268)

[7.4.1. Ventajas del control de versiones 29](#_Toc286652269)

[7.4.2. Control de versiones con Git 30](#_Toc286652270)

[7.4.3. Características de Git 31](#_Toc286652271)

[7.4.4. El modelo de objetos de Git 32](#_Toc286652272)

[7.4.5. Elección de Git como sistema de control de versiones 39](#_Toc286652273)

[7.5. Gestión de tareas 43](#_Toc286652274)

[7.5.1. Ventajas de la gestión de tareas 43](#_Toc286652275)

[7.5.2. Fogbugz y Foglyn 44](#_Toc286652276)

[7.6. Balsamiq Mockup 45](#_Toc286652277)

[8. Aspectos relevantes del desarrollo 47](#_Toc286652278)

[9. Conclusiones y líneas futuras 48](#_Toc286652279)

[10. Bibliografía 49](#_Toc286652280)

[10.1. Referencias 49](#_Toc286652281)

## Índice de ilustraciones

[Ilustración 1. Refactory aplicando una refactorización a un meta-modelo 13](#_Toc284896181)

[Ilustración 2. Representación de un objeto blob 27](#_Toc284896182)

[Ilustración 3. Representación de un objeto árbol 28](#_Toc284896183)

[Ilustración 4. Representación de un objeto Commit 29](#_Toc284896184)

[Ilustración 5. Modelo completo de objetos de Git 31](#_Toc284896185)

[Ilustración 6. Representación de un objeto Tag 32](#_Toc284896186)

[Ilustración 7. Prototipo creado con Balsamiq Mockup 39](#_Toc284896187)

## Índice de tablas

[Tabla 1. Clasificación de los patrones de diseño 17](#_Toc283642152)

# Agradecimientos

# Lista de cambios

## 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 pero de forma incompleta. | Míryam Gómez San Martín  Íñigo Mediavilla Saiz |
| 2 | 22/02/11 | Añadidos RefactoringNg y API de Eclipse 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 |

# Introducción

# Objetivos del proyecto

## Objetivos Técnicos

Otros proyectos de plugin de refactorización han caído en desuso por la supremacía de las refactorizaciones proporcionados por los IDEs (por ejemplo ContraCT y RefactorIt). Una mirada al [ref:“eclipse Marketplace”] muestra que los plugins de refactorización que tienen éxito son los que proporcionan valores añadidos algunos de los cuales se estudian en la sección de comparativa entre alternativas.

El proyecto dispone de ese factor diferenciador de valor añadido en las siguientes características:

* Planes de refactorizaciones
* Refactorizaciones adicionales de interés. Junit3 a Junit4, Genéricas...
* Posibilidad de definir refactorizaciones para otros lenguajes
* Historial de refactorizaciones realizadas, posibilidad de deshacer

Pero en ocasiones esto supone una complejidad para el usuario que encuentra dificultad para comprender el modelo de “javamoon” con las entradas, pre/poscondiciones y acciones a la hora de crear una refactorización. O que no encuentra la refactorización que necesita dado que la opción de mostrar refactorizaciones en base al contexto no le es suficiente. Era necesaria una respuesta para hacer al usuario partícipe de estas interesantes posibilidades que el plugin ofrece y debe hacerlo de forma natural para el usuario.

Desde el principio del proyecto se pensó que la solución a estos problemas debía ser ofrecer al usuario una ayuda en forma de catálogo con toda la información sobre las refactorizaciones, pero clasificada en base a criterios lógicos. El usuario agradecería también la posibilidad en el estudio del catálogo de personalizar la forma en que las refactorizaciones le aparecen organizadas. Para la personalización el catálogo creado se permite al usuario escoger su clasificación preferida, incluso definir su propia clasificación y asignar las refactorizaciones que pertenecen a él. El catálogo también permite hacer búsquedas de refactorizaciones en base a criterios (entre los disponibles filtros por categorías, nombre o palabras clave) que se aplican como un filtro en el catálogo que se puede componer de varias condiciones en base a distintos criterios.

# Conceptos teóricos

# Comparativa entre alternativas

## Comparar con la versión anterior del proyecto

## Desaparición de alternativas respecto a versiones anteriores del plugin

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 (RefactorIt) y ConTRaCT (ConTraCT) son ambas previas a 2008.

No es necesario un estudio concienzudo de la situación para comprender que la principal razón de este decaimiento 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 e importancia de las refactorizaciones en los procesos de desarrollo y 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. Como añadido 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 por cada uno de ellos, para luego comparar en que aspectos confluye o diverge con el del plugin de refactorización valorando ventajas e 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.

## Jdeodorant

JDeodorant (JDeodorant) es un plugin de Eclipse que identifica problemas de diseño en el software (bad smells) y los resuelve aplicando las refactorizaciones apropiadas.

Actualmente la herramienta identifica cuatro tipos de problemas de diseño: envidia de características (Feature Envy), Comprobación de tipos (Type Checking), Método excesivamente largo (Long Method) y Clase omnipotente (God Class).

* Los problemas de “Envidia de características” son resueltos utilizando la refactorización “Mover Método”.
* Los problemas de 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 de “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.

### 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.

### 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, que es el que 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.

### 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, que función desempeñan y a que 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.

## Refactory

Refactory (Reimann) 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 (DSLs).

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 meta-modelos 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 meta-modelos como se quiera.

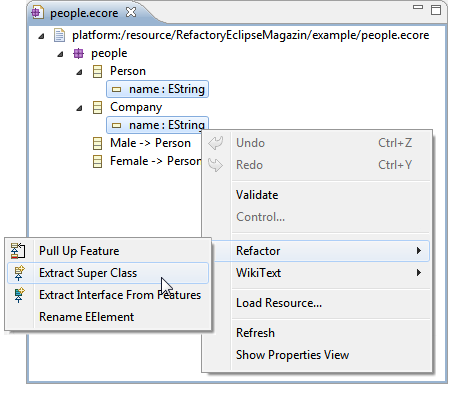


Ilustración 1. Refactory aplicando una refactorización a un meta-modelo

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.

### Ventajas

La principal ventaja que este plugin ofrece es que su enfoque es 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 (DSLs). 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.

### Desventajas

Respecto al plugin de refactorización la principal pega 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, precondiciones, poscondiciones y acciones. Con Refactory es necesario conocer el modelo de refactorizaciones basado en roles definido en (Role-based Generic Model Refactoring, 2010) .

## RefactoringNG

### Introducción

RefactoringNG [ref] 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 regla 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:

#### Árbol, atributos y contenido

Los árboles toman el nombre de los ASTs 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:

Literal [kind: NULL\_LITERAL]

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:

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 substracción. 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 substracción.

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 ejempo, 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]

}

#### 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> { }

}

}

}

#### 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"]

}

}

#### 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.

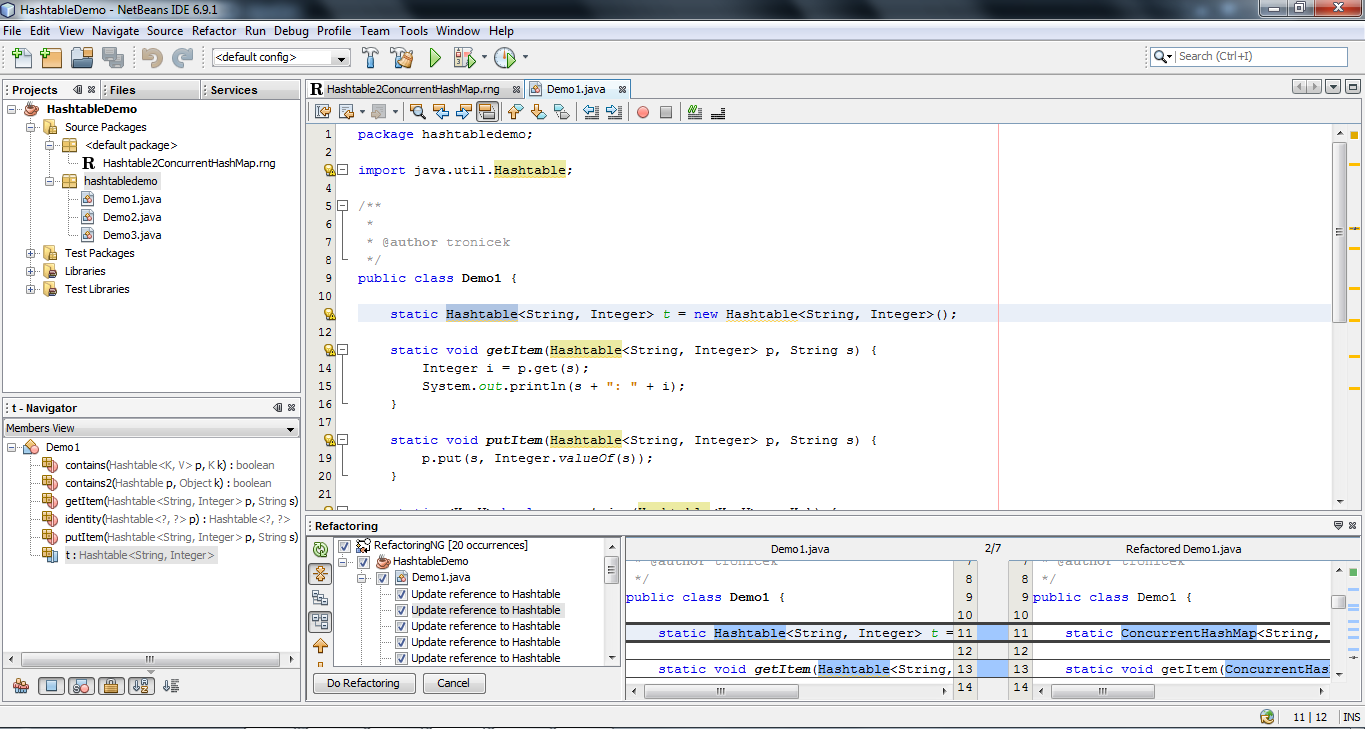


Ilustración 2. Previsualización de resultados con RefactoringNG

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 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 afectan 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. Es muy difícil crear una refactorización de este tipo, sin provocar errores de compilación en los fuentes al ser aplicada.

En definitiva, se puede considerar este plugin con una versión avanzada de una búsqueda y reemplazo basada en expresiones regulares que se soporta sobre los árboles AST. Al igual que esta es potente y flexible pues permite definir transformaciones muy precisas, pero carece de la inteligencia necesaria para definir precondiciones y poscondiciones 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.

## Comparar con la 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 cuales son las modificaciones más importantes a llevar a cabo para implementar una refactorización propia.

### Pasos del proceso de refactorización

* 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" (la clase que define el Wizard).

La información sobre lo que había seleccionado cuando el usuario disparo la ejecución de la refactorización es recuperada. Con la 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 invoca el método "checkInitialConditions()" en la instancia de la refactorización. Este método que ha debido de ser definido por la clase de la refactorización personalizada debe comprobar que se cumplen las condiciones básicas para llevar a cabo la 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 continua 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: Tras determinar que no obstáculos fundamentales que impidan llevar a cabo la refactorización, se muestra el asistente que pide al usuario que introduzca lainformación 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 esta fuera del ámbito de este minitutorial. Para una introducción a la creación de cambios ver (ref Kai-Uwe Mätzel: Safely Manipulating the Contents of Files - How to Get it Right, Presentation at the EclipseCon 2005: http://www.eclipsecon.org/2005/presentations/EclipseCon2005\_5.1Maetzel.pdf )
* 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 pendientes de ser ejecutados en detalle (y con la posibilidad de deseleccionarlos si fuera necesario). En este punto, todavía ninguno de los cambios han sido efectuados, estos se harán efectivos cuando sean confirmados definitivamente por el usuario.

### 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 extension con la personalización de la clase Refactoring. La extensión personalizada debe definir el método "checkInitialConditions()" en el que se comprueba si se cumplen las condiciones básicas para llevar a cabo la refactorización y notificar en caso contrario. Se define el método "checkFinalConditions()" en el que se comprueba que ciertas condiciones se cumplirán tras llevar a cabo el proceso. 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 páginas del Wizard, de la navegación del usuario por el mismo y 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 Wizard una referencia a la refactorización y añadir las páginas adicionales del Wizard (que derivarán de "UserInputWizardPage") si la refactorización las exige.

### 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 el 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.

# 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.

## 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 del siguiente libro para ampliar conocimientos (Meyer, 1998).

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.

## 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, 1979).

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 (contexto).
* Solución: describe los elementos que conforman el diseño, sus relaciones, 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. 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 | *Factory Method* | *Adapter(Class)* | *Interpreter*  *Template Method* |
| Objeto | *Abtract Factory*  *Builder*  *Prototype*  *Singleton* | *Adapter(Object)*  *Bridge*  *Composite*  *Decorator*  *Facade*  *Flyweight*  *Proxy* | *Chain of*  *Responsibility*  *Command*  *Iterator*  *Mediator*  *Memento*  *Observer*  *State*  *Strategy*  *Visitor* |

Tabla 1. Clasificación de los patrones de diseño

## 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 la integración continua se basa 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 para finalizar con una explicación de la herramienta utilizada en el proyecto para la automatización del proceso: Apache Maven.

### Teoría

En la mayoría de los proyectos actuales el equipo comparte el código 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 acaba siendo tan diferente de las líneas de desarrollo de los desarrolladores que se entra en lo que se conoce como “integration hell” (2009), 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.

### 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 ser construible 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 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 (Santos, 2008).

#### Automatizar el proceso de construcción

El sistema completo debería 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 caso, 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 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 hacer commits de todos los cambios al menos una vez al día (una vez por cada característica añadida) y además ejecutar un build nocturno.

#### Cada commit a la rama principal debe disparar un build

El sistema debe construirse cada vez que se actualiza la rama principal para verificar que los cambios se han integrado correctamente. Una práctica habitual es utilizar integración continua automatizada, aunque esto se puede hacer 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 build de forma automática.

#### El build debe ser rápido

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

#### Ejecutar los tests en una copia del entorno de producción

Tener un entorno de prueba puede llevar a fallos en los sistemas testados 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. Las pruebas tempranas reducen 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.

#### Todo el mundo debe poder ver los resultados del último ensamblado del producto

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).

### Apache 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 automatización del proceso de construcción de software. Es principalmente utilizada para la programación 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 componentes y el orden de construcción. Maven trae una serie de objetivos (targets) predefinidos que permiten ejecutar tareas habituales como la compilación del código o su empaquetado.

Maven es capaz de descargar dinámicamente bibliotecas Java y extensiones a su propio núcleo (plug-ins) desde uno o varios repositorios. La herramienta proporciona soporte para la descarga de paquetes desde el repositorio central sin 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 un build con éxito.

#### Project Object Model (POM)

El POM proporciona toda la configuración necesaria para un proyecto. La configuración general incluye el nombre del 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 encargado 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 POMs pueden heredar configuración de otros ficheros POM, de hecho todos los POM heredan del Super POM (Apache Software Foundation, 2011).

#### Plugins

La mayor parte de la funcionalidad de 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 están pre configuradas 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 las 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.

#### Ciclos de vida de un build

El ciclo de vida de maven es una lista de fases que pueden ser utilizadas para establecer un orden de ejecución de los objetivos. Uno de los ciclos de vida estándar de maven es el ciclo de vida por defecto que incluye las siguientes fases en el orden apuntado (Apache Software Foundation, 2010):

* 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. Cuando el comando:

mvn test

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” asociado con la frase 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 el comando “mvn install”.

#### 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 (2009) es utilizado por defecto en la búsqueda de librerías, pero se pueden 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 librerías 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.

## Control de versiones

El control de versiones aparecía como uno de los principios básicos necesarios para la adopción del proceso de integración continua. A continuación se van a comentar 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.

### Ventajas del control de versiones

* 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” estos 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 que permite 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 commits marcan finalización de tareas y las tareas apuntan a los commits 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.
* Los últimos servicios de repositorio web (como Github o Gitorious por ejemplo) facilitan la dinamización del desarrollo gracias a 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.

### Control de versiones con Git

#### Introducción y breve historia de Git

El desarrollo de Git (2010) 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. Así que, inmediatamente tras el desarrollo de la versión 2.6.12-rc2, Torvalds decidió escribió 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) contestó: “Soy un bastardo egocéntrico que nombra todos los proyectos en base a sí mismo. Primero Linux y ahora git.”

### Características de Git

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

* Desarrollo distribuido. Como 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 merge 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 e 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 historiales 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 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 desarrollar scripts que proporcionen nuevas funcionalidades.

### El modelo de objetos de Git

#### The 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 la misma manera en cada repositorio, el mismo contenido guardado en dos repositorios siempre será guardado bajo el mismo nombre.
* Git puede detector 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 es:

* Un “blob” es utilizado para almacenar datos. Es generalmente un fichero.
* Un árbol es básicamente como un directorio. Referencia a un conjunto de otros árboles o blobs (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 meta-información sobre ese momento: como la fecha, el autor de los cambios desde el último commit, un puntero al commit previo, etc.
* Una etiqueta es una manera de especificar un commit como especial de alguna manera. Es habitualmente utilizado para 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 mayoría de los sistemas SCM con los que los desarrolladores están familiarizados. Subversion, CVS, Perforce y Mercurial por ejemplo todos 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.

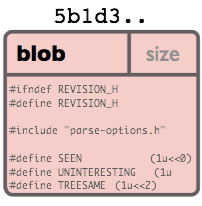


Ilustración 3. Representación de un objeto blob

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 árbol

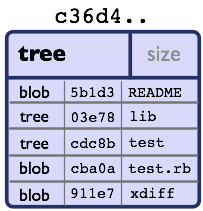


Ilustración 4. Representación de un objeto árbol

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 289b046a443c0647624607d471289b2c7dcd470b 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 forma 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



Ilustración 5. 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 2be7fcb4764f2dbcee52635b91fedb1b3dcf7ab4

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 padre; 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: el nombre de la persona responsable del cambio, junto con su fecha.
* Responsable del commit: el nombre de la persona que hizo efectivo el 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 encargo 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 contiene 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 padre. En particular, Git no registra 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 padre es la última revisión 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

E hiciéramos un commit a un repositorio de Git este se representaría de la siguiente manera:

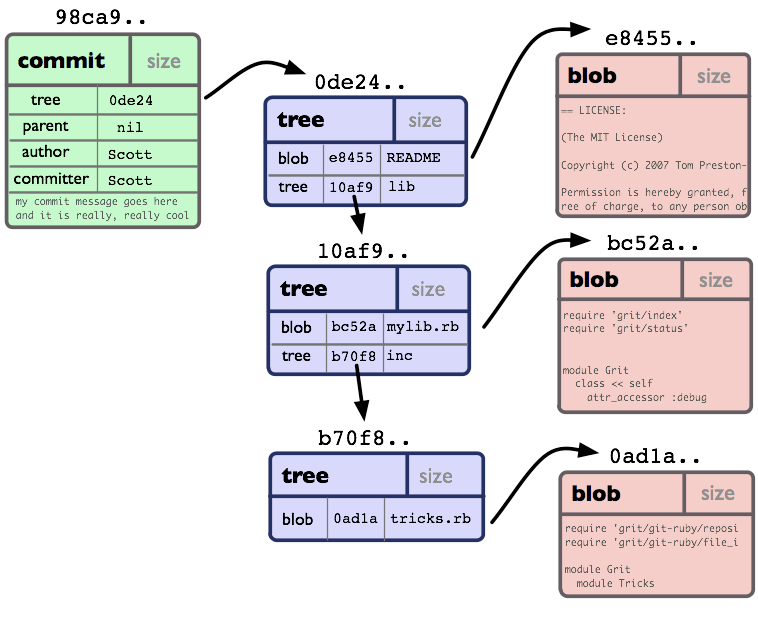


Ilustración 6. Modelo completo de objetos de Git

Se puede comprobar que se ha creado un árbol de objetos para cada directorio (incluido el raíz) y un blob para cada fichero. Así que 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)



Ilustración 7. 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 (“tagger”) que creó el tag y un mensaje que puede contener una firma, como puede se puede ver ejecutando el commando 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)

iD8DBQBF0lGqwMbZpPMRm5oRAuRiAJ9ohBLd7s2kqjkKlq1qqC57SbnmzQCdG4ui

nLE/L9aUXdWeTFPron96DLA=

=2E+0

-----END PGP SIGNATURE-----

### Elección de Git como sistema de control de versiones

Cuando se planteó el dilema 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 cumplir.

* 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 (Códice Software, 2010) 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 hosting de nuestro repositorio online de forma gratuita. Esto se veía facilitado por el hecho de que nuestro proyecto iba a ser liberado con licencia LGPL, existiendo diversas opciones de repositorios gratuitos para proyectos de código libre.
* Debía ser rápido puesto 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).
* Amplia comunidad de usuarios. Solo el servicio de hosting de repositorios Git 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 (GitLangStats).
* Amplio abanico de documentación en internet con una gran variedad de libros de introducción y perfeccionamiento en su uso disponibles en (ProGitBook) y (GitCommunity), una vasta documentación con ejemplos para cada comando disponible en la herramienta, un número enorme de tutoriales y blogs haciendo referencia al sistema y en torno a 6000 dudas etiquetadas como git en stackoverflow, frente a las 7.000 de un sistema tan implantado como Subversion o las sólo 1900 de su competidor más directo mercurial (estadísticas tras consulta a 31/1/2011).
* 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 al de 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 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), 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) o (Gitorious) 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 y dispone que pueden abrumar a un usuario novato. Es muy recomendable iniciar 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 todas esos 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 vasta 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.

## Gestión de tareas

En el proyecto se ha utilizado una metodología basada en las 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 utilizado para la implementación de este enfoque.

### Ventajas de la gestión de tareas

* El proceso se basa en priorizar las tareas e implementar las tareas con mayor prioridad, es decir las que han ser consideradas como capaces de proporcionar mayor valor de forma a 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 tracking 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 releases próximas.
  + Permite hacer referencias a commits en los que ciertas tareas son solucionadas o en el sentido contrario marcar en los commits que bugs han sido resueltos o que nuevas características han sido implementadas.
  + Permite utilizar el patrón de desarrollo de “branch per task” (una rama por tarea) (Códice Software, 2010) que se basa en que un desarrollador cuando decide solucionar un bug 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 sólo cuando la tarea ha sido completada y se ha comprobado que el build funciona y los test no fallan. De este modo se consigue que la rama principal del repositorio siempre esté libre de fallos de build, que no haya problemas de integración porque las tareas son pequeños cambios bien definidos y que siempre se pueda conocer a la perfección que tareas han sido implementadas en que commit del control de versiones.

### Fogbugz y Foglyn

#### FogBugz

Fogbugz (Fog Creek Software, 2000) 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 (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 de un usuario a partir del trabajo realizado en las tareas, por día.

##### 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, usuarios, proyectos, así como los parámetros de los mismos.

#### Foglyn

Foglyn es un plug-in 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)

## 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 , 2011). 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 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 (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 prototipo como una imagen para poder enviarlo como correo electrónico o parar imprimirlo directamente.

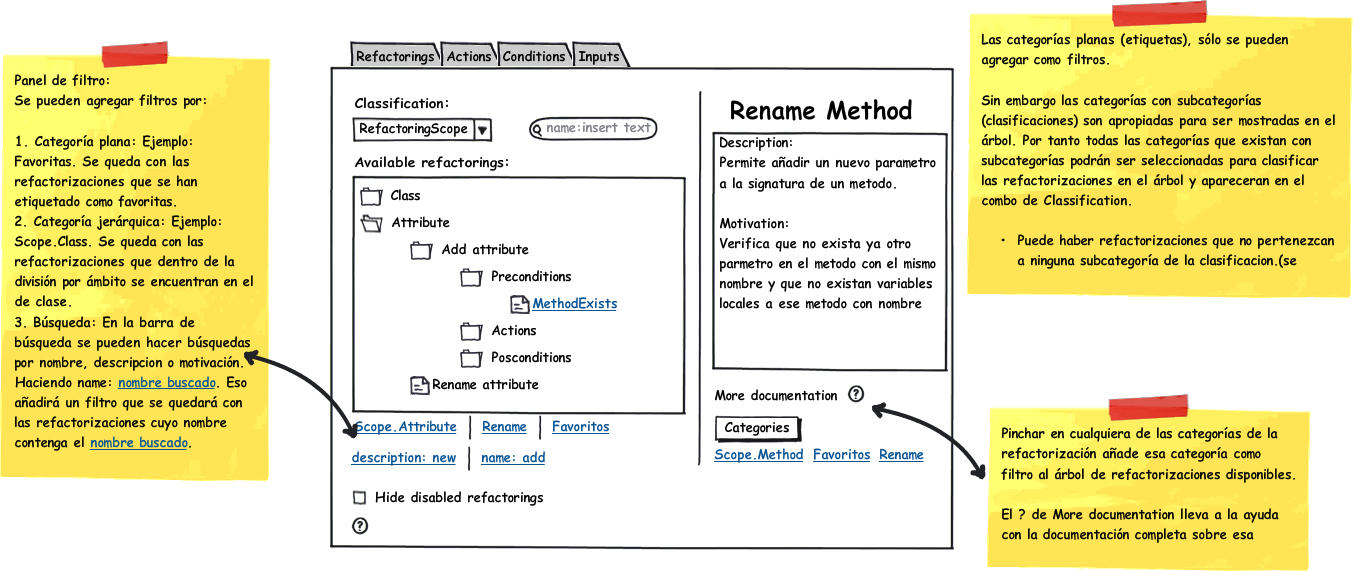


Ilustración 8. Prototipo creado con Balsamiq Mockup

# Aspectos relevantes del desarrollo

# Conclusiones y líneas futuras

# Bibliografía

## Referencias

**Alexander, C. 1979.** *The Timeless Way of Building.* s.l. : Oxford University Press, Oxford. , 1979.

**Apache Software Foundation. 2011.** Super POM. [En línea] 2011. http://maven.apache.org/guides/introduction/introduction-to-the-pom.html#Super\_POM.

**—. 2010.** Super\_POM. [En línea] 2010. http://maven.apache.org/guides/introduction/introduction-to-the-pom.html#Super\_POM.

**Balsamiq Studios, LLC . 2011.** Balsamiq Mockup. [En línea] 2011. http://balsamiq.com/products/mockups.

**Códice Software. 2010.** Task Driven Development. [En línea] 2010. http://www.plasticscm.com/features/task-driven-development.aspx.

ConTraCT Plugin for Eclipse. [En línea] http://roots.iai.uni-bonn.de/research/contract/eclipsePlugin.

EGit. [En línea] http://www.eclipse.org/egit/.

**Fog Creek Software. 2000.** [En línea] Noviembre de 2000. http://www.fogcreek.com/FogBugz/.

**2010.** Git - SCM. [En línea] 2010. http://git-scm.com/.

Git Benchmarks. [En línea] https://git.wiki.kernel.org/index.php/GitBenchmarks.

Git Community Book. [En línea] http://book.git-scm.com/.

GitHub. [En línea] http://github.com.

GitLangStats. [En línea] https://github.com/ezgraphs/R-Programs/blob/master/git\_lang\_stats.txt.

Gitorious. [En línea] http://gitorious.org/.

**2009.** Integration Hell. [En línea] 2009. http://c2.com/cgi/wiki?IntegrationHell.

JDeodorant. [En línea] http://www.jdeodorant.com/.

**2009.** Maven 2 Central Repo. [En línea] 2009. http://repo1.maven.org/maven2/.

**Meyer, B. 1998.** *Construcción de Software Orientado a Objetos, 2ª Ed.* s.l. : Prentice Hall, 1998.

ProGitBook. [En línea] http://progit.org/book/.

RefactorIt. [En línea] http://www.aqris.com/display/A/Refactorit.

**Reimann, Jan.** Refactory. [En línea] http://emftext.org/index.php/Refactoring.

**Ries, Eric. 2009.** Continuous deployment. [En línea] 30 de Marzo de 2009. http://radar.oreilly.com/2009/03/continuous-deployment-5-eas.html.

*Role-based Generic Model Refactoring.* **Jan Reimann, Mirko Seifert & Uwe Aßmann. 2010.** s.l. : ACM/IEEE 13th International Conference on Model Driven Engineering Languages Systems, 2010.

**Santos, Pablo. 2008.** Integracion Continua vs Controlada. [En línea] 2008. http://www.drdobbs.com/architecture-and-design/205917960.