Universidad del País Vasco Euskal Herriko Unibertsitatea

▪ **P**royecto **F**in de **G**rado ▪

Titulación: **Grado en Ingeniería Informática**

Especialidad: **Ingeniería del Software**

Gestión de la Evolución de Familias de Producto Software en Escenarios Complejos.

Alumno: Asier Gil Bahón

Director: Oscar Javier Saiz Cuesta

Codirector: Oscar Díaz García

Proyecto Fin de Grado, enero de 2014

Resumen

En los últimos años, la dinámica de mercado de productos y servicios ha cambiado sustancialmente: los clientes demandan cada vez productos y servicios más personalizados a sus necesidades específicas.

Esta dinámica también ha llegado a la industria de producción de software. Para dar respuesta a esta demanda, se utilizan técnicas para, a partir de un producto común, generar variantes del mismo y cubrir las diferentes exigencias de los clientes. Estas técnicas se agrupan bajo la disciplina de líneas de producto software.

Sin embargo, incluso con este paradigma, el número de artefactos a gestionar puede ser elevado. Ante esta situación se plantea un proyecto de investigación acerca de metodologías, técnicas y tecnologías de gestión para ofrecer soluciones eficientes.

Este proyecto se apoya en diversos escenarios de proyectos software de una empresa concreta para estudiar las dificultades que revelan y a partir de ellas definir una estrategia. Asimismo, se presenta una herramienta que implementa esa estrategia y facilita su uso a los usuarios.

Palabras clave: sistemas de control de versiones, Git, branch (rama), eGit.

ÍNDICE

1. INTRODUCCIÓN 11
1.1 Situación actual y problemática 11
1.2 Antecedentes y motivación 12
1.3 Objetivo y descripción general del proyecto 13
1.4 Planificación y gestión 14
1.4.1 Diagrama EDT 14
1.4.2 Diagrama de GANTT 15
2 ESTADO DEL ARTE 18
2.1 Control de versiones 18
2.1.1 Introducción 18
2.1.2 Tipos de VCS: Centralizados y Distribuidos 20
2.1.3 Operaciones básicas 22
2.1.4 Flujos de trabajo 28
2.1.5 Interfaces gráficas de usuario (GUIs) 31
2.1.6 Servicio de alojamiento de repositorios en la web 32
2.2 Herramientas de gestión 33
2.2.1 Sistemas de seguimiento de incidencias / errores 34
2.2.2 Comparación de ficheros/ directorios 35
2.2.3 Revisión de código 38
2.3 Plugins de Eclipse 39
2.3.1 jGit 39
2.3.2 eGit 40
2.3.3 Plug- in Development Enviroment 41
2.4 Otros 42
2.4.1 gitFlow 42
3 SOLUCIÓN PLANTEADA 44
3.1 Contribución innovadora 44
3.2 Escenarios de aplicación 45
3.2.1 Punto de partida para un nuevo proyecto 46
3.2.2 Fusionar ramas lejanas o no conectadas directamente 47
3.2.3 Control de versión software de múltiples proyectos 47

3.2.4 Alcance de los errores de código ocasionados en un proyecto 48
3.3 Estrategia de ramificación 49
3.3.1 Contexto 49
3.3.2 Diseño 51
3.3.3 Límites 51
3.3.4 Solución a los escenarios mediante la estrategia 54
3.4 Validación de la estrategia 57
4 IMPLEMENTACIÓN 58
4.1 Descripción de la herramienta 58
4.2 Requisitos 58
4.2.1 Requisitos funcionales 58
4.2.2 Requisitos no funcionales 60
4.3 Análisis 60
4.3.1 Casos de uso 60
4.4 Diseño 69
4.4.1 Diagrama de contexto 69
4.4.2 Diagramas de clases 70
4.4.3 Diagramas de secuencia 74
4.5 Desarrollo 81
4.5.1 Estructura 81
4.5.2 Implementación 83
4.6 Pruebas 87
4.6.1 Pruebas de funcionalidad 87
4.6.2 Pruebas de carga 90
5 CONCLUSIONES 92
5.1 Contribución 92
5.2 Conclusiones 92
5.3 Líneas futuras 93
5.4 Lecciones aprendidas 94
6 BIBLIOGRAFÍA 96
7 ANEXOS 98
7.1 Anexo A: Fundamentos básicos de Git 98
7.1.1 Conceptos 98
7.1.2 Operaciones 102

7.2 Anexo B: Manual de usuario 110
7.2.1 Inicializar repositorio 111
7.2.2 Crear nuevo proyecto 112
7.2.3 Crear nueva característica 113
7.2.4 Cerrar característica 114
7.2.5 Buscar errores en el código 115
7.2.6 Ver diferencias a tres vías 117
7.3 Anexo C: Fragmentos de código más relevantes 119
7.3.1 Menú dinámico 119
7.3.2 UI de los proyectos 120
7.3.3 UI de las características 121
7.3.4 Crear proyecto 122
7.3.5 Crear característica 123
7.3.6 Cerrar característica 123
7.3.7 Buscar errores en el código 124
7.3.8 Ver diferencias a tres vías 124

ÍNDICE DE ILUSTRACIONES

Figura 1 - Diagrama EDT 15
Figura 2 - Diagrama GANTT 17
Figura 3 - Principio básico control de versiones 18
Figura 4 - VCS centralizado 20
Figura 5 - VCS Distribuido 21
Figura 6 - Ejemplo de branching 22
Figura 7 - Merge y Rebase 25
Figura 8 - Cherry pick 26
Figura 9 - Resolver conflictos 27
Figura 10 - Flujo de trabajo centralizado 28
Figura 11 - Flujo de trabajo del gestor de integraciones 30
Figura 12 - Flujo de trabajo con dictador y tenientes 31
Figura 13 - Github: ejemplo de repositorio público 33
Figura 14 - Jira: Visualizar incidencias 35
Figura 15 - Kdiff3: Comparación de directorios 37
Figura 16 - Semanticmerge: Fusión 38
Figura 17 - Ejemplo de crucible 39
Figura 18 - eGit: Comparador de diferencias 40
Figura 19 - eGit: Historial de revisiones 40
Figura 20 - Modelo de ramificación de Vincent Driessen 43
Figura 21 - Metodología convencional 44
Figura 22 - Nueva metodología 45
Figura 23 - de partida para un nuevo desarrollo A 46
Figura 24 - Punto de partida para un nuevo desarrollo B 46
Figura 25 - Fusionar ramas lejanas 47
Figura 26 - Control de versión software 48
Figura 27 - Alcance de los bugs en un proyecto 49
Figura 28 - Estrategia de ramificación 50
Figura 30 - Mensaje de confirmación en eGit 53
Figura 29 - Ejemplo de árbol con etiquetas 53
Figura 31 - Capas software del desarrollo 69
Figura 32 - Diagrama de clases: Patrón 70
Figura 33 - Diagrama de clases: Inicializar proFlow 71
Figura 34 - Diagrama de clases: Crear proyecto 72
Figura 35 - Diagrama de clases: Crear característica 72
Figura 36 - Diagrama de clases: Cerrar característica 73
Figura 37 - Diagrama de clases: Buscar errores en el código 73
Figura 38 - Diagrama de clases: Ver diferencias a tres vías 74
Figura 39 - Diagrama de secuencia: Inicializar proFlow 75

Figura 40 - Diagrama de secuencia: Crear proyecto 76
Figura 41 - Diagrama de secuencia: Crear característica 77
Figura 42 - Diagrama de secuencia: Cerrar característica 78
Figura 43 - Diagrama de secuencia: Buscar errores en el código 79
Figura 44 - Diagrama de secuencia: Ver diferencias a tres vías 80
Figura 45 - eGit arquitectura 81
Figura 46 - Estructura del menú de la herramienta 82
Figura 47 - Menú de la herramienta en ejecución 83
Figura 48 - eGit: Tipos de referencias 85
Figura 49 - Modelado de datos por diferencias 98
Figura 50 - Modelado de datos por instantáneas 99
Figura 51 - Ejemplo de revisiones 99
Figura 52 - Apuntadores en Git 100
Figura 53 - HEAD en Git 101
Figura 54 - Avance de los apuntadores en Git 102
Figura 55 - Menús desplegables 110
Figura 56 - Opciones del historial de revisiones 111
Figura 57 - Asistente para inicializar proFlow 112
Figura 58 - Checkout de ejemplo 112
Figura 59 - Asistente para crear proyectos 113
Figura 60 - Nuevo proyecto creado 113
Figura 61 - Nueva característica creada 114
Figura 62 - Opciones de fusionado 114
Figura 63 - Característica fusionada 115
Figura 64 - Especificar rango de búsqueda 116
Figura 65 - Error de código encontrado 116
Figura 66 - Especificar revisiones a comparar 117
Figura 67 - Diferencias a tres vías 118

ÍNDICE DE TABLAS

Tabla 1 - Tipos de VCS 21
Tabla 2 - GUIs por VCS 32
Tabla 3 - GUIs por sistemas operativos 32
Tabla 4 - Servicios de alojamientos de repositorios en la web 33
Tabla 5 - Requisitos no funcionales 60
Tabla 6 - Caso de uso: Seleccionar repositorio 61
Tabla 7 - Caso de uso: Establecer el estándar de codificación 61
Tabla 8 - Caso de uso: Actualizar el directorio de trabajo 62
Tabla 9 - Caso de uso: Visualizar el árbol histórico de revisiones 62
Tabla 10 - Caso de uso: Crear nuevo proyecto 63
Tabla 11 - Caso de uso: Fusionar proyecto 63
Tabla 12 - Caso de uso: Crear nueva característica 64
Tabla 13 - Casos de uso: Fusionar característica 64
Tabla 14 - Caso de uso: Eliminar rama 65
Tabla 15 - Caso de uso: Etiquetar revisiones 65
Tabla 16 - Caso de uso: Eliminar etiquetas 66
Tabla 17 - Caso de uso: replicar los cambios de una o varias revisiones 66
Tabla 18 - Caso de uso: Confirmar los cambios realizados 67
Tabla 19 - Caso de uso: Buscar errores en el código 68
Tabla 20 - Caso de uso: Invertir los cambios realizados en una revisión 68
Tabla 21 - Caso de uso: Búsqueda en el repositorio 68
Tabla 22 - Caso de prueba 1: Inicializar proFlow 87
Tabla 23 - Caso de prueba 2: Crear nuevo proyecto 88
Tabla 24 - Caso de prueba 3: Crear nueva característica 88
Tabla 25 - Caso de prueba 4: Cerrar característica 89
Tabla 26 - Caso de prueba 5: Buscar errores en el código 89
Tabla 27 - Caso de prueba 6: Ver diferencias a tres vías 90

ACRÓNIMOS Y SIGLAS

VCS	Version control system
GUI	Graphical user interface
IDE	Integrated Development Environment
RCP	Rich Client Platform

1. INTRODUCCIÓN

Este documento describe el desarrollo del Proyecto Fin de Grado (PFG) realizado por Asier Gil Bahón durante los meses de junio de 2013 a enero de 2014 en la empresa IK4-Ikerlan, bajo la supervisión de Oscar Javier Saiz Cuesta en la empresa y siendo director del proyecto en la universidad Oscar Díaz García. El proyecto se denomina "Gestión de la evolución de familias de producto software en escenarios complejos", y está basado en una problemática común detectada en varias empresas y validado contra una empresa concreta: ULMA Packaging S.Coop**.** En este apartado se expone la problemática a abordar en el proyecto, los objetivos y la planificación de las tareas. Además, se explica la distribución estructural de este documento.

1.1 Situación actual y problemática

En los últimos años, la dinámica de mercado de productos y servicios ha cambiado sustancialmente. Los clientes demandan cada vez productos y servicios más personalizados a sus necesidades específicas.

Como respuesta a esta demanda de una forma eficiente, las empresas han avanzado desde el paradigma de la producción en masa (mass- production)[1], a la personalización en masa (mass- customization)[2].

Esta misma evolución se ha dado también en la industria de producción de software. Hasta hace muy poco las empresas desarrollaban programas genéricos que sacaban al mercado. Ahora, crean productos software personalizados a las necesidades de cada cliente/sector/mercado. Esto ha introducido nuevos problemas: incremento del número de programas vivos, gestión de código similar o repetido, gestión de las modificaciones,…

La cantidad de productos personalizados que las empresas tienen que gestionar es en muchos casos de una magnitud considerable. Por ello, muchas optan por desarrollar variantes de un producto estándar. Es decir, se centran en desarrollar un producto genérico del que generar diferentes variantes o productos personalizados, consiguiendo así no replicar bloques iguales. Estas soluciones se estudian en la disciplina de *Línea de Producto Software*.

[1] http://en.wikipedia.org/wiki/Mass_production
[2] http://en.wikipedia.org/wiki/Mass_customization

Sin embargo, incluso con este paradigma, el número de productos a gestionar puede ser elevado. Por ello se ve necesario adoptar nuevas metodologías y herramientas para gestionar esta complejidad de forma eficiente.

Un ejemplo de herramientas son los sistemas de control de versiones [1]. Estos sistemas permiten almacenar cronológicamente las modificaciones que se realizan en diferentes momentos creando así un historial de revisiones, en el que es posible analizar las diferencias de un punto concreto a otro en el tiempo, replicar cambios, recuperar estados anteriores, generar proyectos nuevos a partir de cierta revisión, etc.

Estos sistemas permiten:

- Estar al corriente de las modificaciones que se realizan a lo largo del ciclo de vida de los proyectos.
- Facilitar al usuario información para la toma de decisiones, ya que permiten determinar qué cambios se han realizado en momentos determinados en el tiempo.
- Almacenar los diferentes estados (conjunto de cambios) de un proyecto para moverse con libertad entre ellos. Este método es útil para, por ejemplo especificar a partir de qué punto del desarrollo de un proyecto conviene empezar a desarrollar una variante que se adapte a las necesidades un nuevo cliente.

En la actualidad los VCS se utilizan para gestionar proyectos individuales, es decir, se centran únicamente en un único proyecto y en las características de este. En este proyecto, se pretende adaptar los VCS a un escenario en el que deban coexistir multitud de proyectos.

1.2 Antecedentes y motivación

En la actualidad, muchas empresas no utilizan un método sistemático que permita gestionar las familias de productos software y que facilite el desarrollo de productos software personalizados a partir de un proyecto genérico.

Hasta ahora, han venido utilizando sistemas de control de versiones para llevar un registro de la evolución de un producto software. Sin embargo se ve la necesidad de diseñar una estrategia que aporte algo más.

Cuando surge la necesidad de comenzar un nuevo proyecto que implemente modificaciones sobre un proyecto específico, como si una variante del mismo se tratase, en muchos casos no se sigue ningún tipo de pautas que indiquen cuál debe ser el origen que mejor le conviene. Es decir, si un proyecto puede comenzar de diferentes puntos de partida, por ejemplo A, B y C, cada uno con sus diferencias, no se sigue ninguna metodología para elegir de qué punto de los tres comenzar. Normalmente es el desarrollador quien debe realizar un análisis del historial para, en primer lugar elegir el origen que mejor se adecúa, y en segundo lugar realizar las distintas acciones para enlazar el nuevo proyecto con el anterior. En principio, la parte de analizar la mejor fuente de origen del nuevo proyecto es un procedimiento que no puede ser automatizable. Sin embargo, la parte de realizar las acciones pertinentes para enlazar ambos proyectos, sí que es automatizable en cierta medida: Es posible guiar al usuario en el proceso e incluso abreviar los pasos.

Cuando el número de variantes es muy elevado, resulta complicado decidir en qué parte del árbol cronológico incorporar un nuevo desarrollo. Por lo tanto el sistema debe proveer información al usuario para facilitar la selección del punto de partida.

La empresa IK4- Ikerlan apuesta por un proyecto de investigación para diseñar una estrategia que dé respuesta a la problemática de la gestión de familias de productos software, concluyendo con un estudio de factibilidad y posterior implementación de una herramienta que aplique la estrategia definida, aportando así una solución a la problemática descrita. La herramienta estará desarrollada a modo de extensión (plugin) para el IDE Eclipse.

1.3 Objetivo y descripción general del proyecto

El objetivo del proyecto es definir metodologías y herramientas que permitan gestionar la evolución de familias de producto software de manera eficiente, facilitando además información para la toma de decisiones.

Este objetivo se materializará por medio del desarrollo de las siguientes tareas:

- Realizar un estudio del estado del arte de gestores de configuración de software y de gestión de familias de producto. Establecer criterios de valoración y realizar la selección para los escenarios definidos. Realizar un análisis crítico.

- Especificar escenarios de aplicación basados en un caso industrial real. A partir de estos escenarios, se definirán las necesidades de herramienta de soporte y se explicitará el proceso para sistematizar la evolución de la familia.
- Diseñar y desarrollar-integrar una herramienta que de soporte para dichos escenarios basándose en el estado del arte o en nuevas soluciones, definiendo una metodología.

1.4 Planificación y gestión

Este apartado expone la planificación y gestión de las tareas durante el desarrollo del proyecto a través de los diagramas EDT (Estructura de Desglose de Trabajo) y GANTT.

1.4.1 Diagrama EDT

Para dar una visión clara de las tareas a realizar en el transcurso del proyecto, se ha utilizado un diagrama EDT diseñado y editado mediante la herramienta SmartArt que viene integrada en el paquete de Microsoft Word 2007. A continuación, se describen brevemente las distintas fases en las que está compuesto el proyecto, seguido del correspondiente diagrama EDT (ver Figura 1):

- **Fase 1. Iniciación**

  Se define la situación y problemática de las familias de productos software en la actualidad, los objetivos a llevar a cabo, la descripción del proyecto y la planificación del mismo.
- **Fase 2. Estudio**

  Se estudian metodologías y herramientas/tecnologías que existen en relación al proyecto.
- **Fase 3. Propuesta**

  Después del estudio, en este bloque se exponen los diferentes escenarios a analizar junto con la estrategia propuesta para dar una solución al problema. Además, se describen las herramientas elegidas para llevar a cabo la implementación de la estrategia, así como las herramientas adicionales utilizadas para el desarrollo del proyecto en general.
- **Fase 4. Requisitos, análisis y diseño**

  Se definen los requisitos que debe cumplir la aplicación, así como el análisis y diseño posterior para determinar qué y cómo debe hacer.

- **Fase 5. Desarrollo**

  Se desarrolla e implementa la aplicación.

- **Fase 6. Pruebas**

  Se definen y realizan las pruebas para la evaluación del sistema implementado.

- **Fase 7. Documentación**

  Se recoge cada fase explicada anteriormente para recopilarla en una memoria. Además se realiza una presentación sobre la memoria para su exposición.

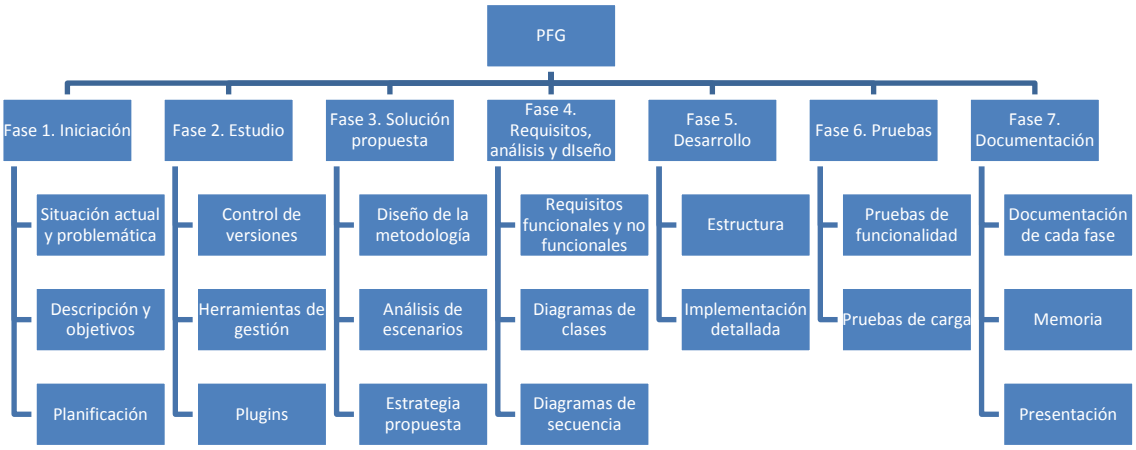


Figura 1 - Diagrama EDT

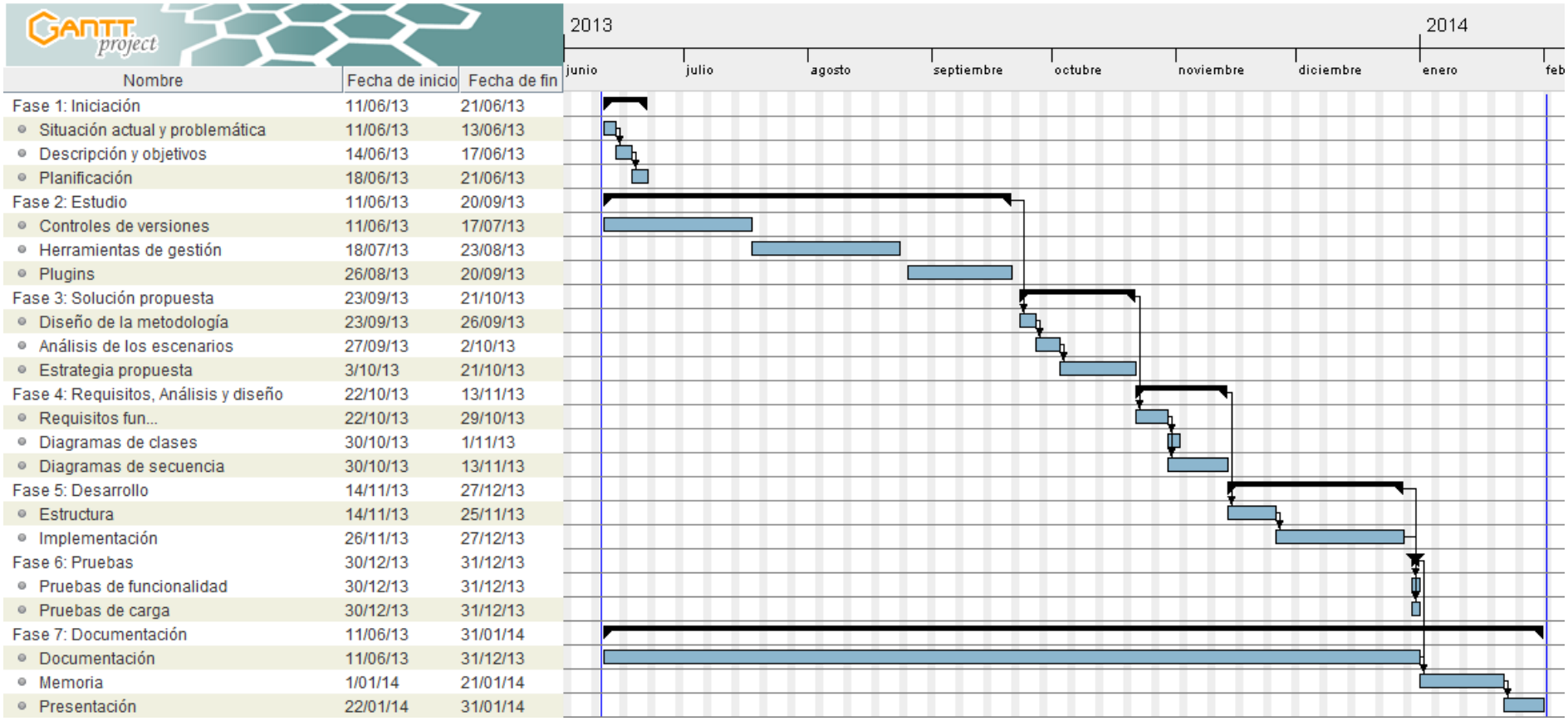
1.4.2 Diagrama de GANTT

Para realizar el diagrama de GANTT de la planificación del proyecto se ha utilizado la herramienta de software libre GanttProject. Ésta permite definir la planificación, con sus fases y subtareas de forma cronológica y visual. La descripción de las tareas a realizar se sitúa en la parte izquierda, y cada una indica su fecha de comienzo y fecha de finalización. En la parte derecha se presentan gráficamente la duración y dependencia de las tareas.

En cuanto al reparto de horas, IK4- Ikerlan establece un calendario en el que la semana laboral está definida de lunes a viernes, y días no laborables los días declarados como festivos nacionales o festivos de la empresa y fines de semana. El total a lo largo de 8 meses contando

los días festivos se han trabajado 136 días, que a una media de 8 horas diarias equivalen a 1088 horas.

En el diagrama de Gantt de la siguiente figura se pude ver como se ha distribuido el trabajo en ese periodo de tiempo.



Nombre	Fecha de inicio	Fecha de fin
Fase 1: Iniciación	11/06/13	21/06/13
Situación actual y problemática	11/06/13	13/06/13
Descripción y objetivos	14/06/13	17/06/13
Planificación	18/06/13	21/06/13
Fase 2: Estudio	11/06/13	20/09/13
Controles de versiones	11/06/13	17/07/13
Herramientas de gestión	18/07/13	23/08/13
Plugins	26/08/13	20/09/13
Fase 3: Solución propuesta	23/09/13	21/10/13
Diseño de la metodología	23/09/13	26/09/13
Análisis de los escenarios	27/09/13	2/10/13
Estrategia propuesta	3/10/13	21/10/13
Fase 4: Requisitos, Análisis y diseño	22/10/13	13/11/13
Requisitos fun...	22/10/13	29/10/13
Diagramas de clases	30/10/13	1/11/13
Diagramas de secuencia	30/10/13	13/11/13
Fase 5: Desarrollo	14/11/13	27/12/13
Estructura	14/11/13	25/11/13
Implementación	26/11/13	27/12/13
Fase 6: Pruebas	30/12/13	31/12/13
Pruebas de funcionalidad	30/12/13	31/12/13
Pruebas de carga	30/12/13	31/12/13
Fase 7: Documentación	11/06/13	31/01/14
Documentación	11/06/13	31/12/13
Memoria	1/01/14	21/01/14
Presentación	22/01/14	31/01/14

Figura 2 - Diagrama GANTT

2 ESTADO DEL ARTE

En esta sección se recogen de manera detallada las tecnologías-herramientas que han sido estudiadas para dar con una solución a la problemática del proyecto.

2.1 Control de versiones

2.1.1 Introducción

El mundo del control de versiones se centra en la gestión de los cambios de los documentos, aplicaciones, sitios web y otro tipo de colecciones. Estos cambios se identifican mediante etiquetas o códigos y marcan el estado de un elemento o un conjunto de elementos en el tiempo. Los diferentes estados en el que se encuentra el producto se denominan revisiones. Por ejemplo, partiendo de un conjunto de cambios inicial "rev1", cuando se realice un cambio, el estado resultado será "rev2", y así sucesivamente. Además, las revisiones tienen asociadas información de control: el usuario que realizó cambios, cuándo los hizo, etc. Mediante este sistema las revisiones pueden ser comparadas, restauradas, y fusionadas con otro conjunto de cambios.

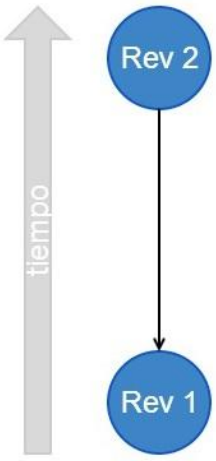


Figura 3 - Principio básico control de versiones

En la industria informática el control de versiones se concentra en la gestión del código fuente en su mayoría, pero no necesariamente debe ser así. Se puede aplicar a otros ámbitos, como la gestión de documentos, archivos de configuración, imágenes, etc.

A día de hoy, existen herramientas que dan soporte al control de versiones con el fin de facilitar su uso y llevar un control en todo momento. Estas herramientas se denominan Sistemas de Control de Versiones (Version Control System-VCS) [3]. Comúnmente funcionan como aplicaciones independientes, pero el control de versiones también está incluido en varios tipos de software, como pueden ser los procesadores de texto y hojas de cálculo, ej. Google Docs, y en varios sistemas de gestión de contenido (CMS), ej. Wikipedia.

Un sistema de este tipo presenta las siguientes características:

- Sistema de almacenamiento para gestionar distintos elementos. (ej. Archivos de texto, imágenes, documentación, etc.)
- Realizar cambios sobre los elementos almacenados. (ej. modificaciones parciales, añadir, borrar, renombrar o mover elementos.)
- Registro histórico de las acciones realizadas que permita retroceder a un estado anterior del producto, ver diferencias entre estados, etc.

Para comprender a fondo el funcionamiento de los sistemas de control de versiones a continuación se describen brevemente los términos usados en este contexto. Éstos pueden variar en función del VCS utilizado. Sin embargo la mayoría son de uso común.

- **Repositorio:** es el lugar donde se almacenan los datos actualizados y el histórico de cambios.
- **Revisión:** identifica la versión o estado del conjunto de elementos. Hay sistemas que los identifican con contadores (ej. Subversion) y otros mediante códigos. (ej. Git usa códigos SHA-1)
- **Directorio de trabajo:** es el directorio cuyo contenido de ficheros/directorios corresponden a la revisión en la que el repositorio está colocado en ese instante.
- **Etiquetas (tags):** los tags permiten identificar con un nombre preciso una confirmación. Se utilizan para marcar puntos de importancia, que estén accesibles de manera sencilla. Por ejemplo para marcar la versión del software en una confirmación determinada. (ej. v0.7, v1.0, etc.)
- **Ramas (branches):** las ramas son conjuntos de confirmaciones, que reflejan la evolución de un proyecto. Una rama puede ser bifurcada o ramificada en un instante de tiempo de manera que, desde ese momento en adelante se tienen dos copias (ramas) que evolucionan de forma independiente siguiendo su propia línea de desarrollo. Este procedimiento se puede realizar las veces que sean necesarias para crear las ramas que interesen. La ventaja de desarrollar con ramas es que, permite crear ramas de pruebas que contengan código pendiente de evaluación, crear ramas dedicadas a nuevas funcionalidades, ramas de corrección de errores, etc.
- **Despliegue (clone, check- out):** un despliegue crea una copia de trabajo local desde el repositorio.
- **Confirmación (commit):** se utiliza para almacenar el estado de los cambios realizados en la copia de trabajo en el repositorio, el resultado es una revisión.

- **HEAD:** es una etiqueta que indica en qué rama está posicionado en ese instante el directorio de trabajo.

Adicionalmente para profundizar en los fundamentos de un sistema de este tipo, en el Anexo I se han recopilado los fundamentos básicos necesarios para comenzar a utilizar Git [7] correctamente.

2.1.2 Tipos de VCS: Centralizados y Distribuidos

Actualmente existen una gran variedad de herramientas similares diferenciándose principalmente por su arquitectura. Bajo este criterio, se distinguen dos tipos: centralizados o distribuidos.

Sistemas de control de versiones centralizados

Los sistemas centralizados se basan en una comunicación cliente-servidor, de forma que existe un único repositorio al que los usuarios acceden. Los usuarios obtienen una copia del repositorio en su máquina local sobre la que trabajar y una vez hayan realizado cambios sincronizarán sus cambios con los del repositorio. Este modo obliga a los usuarios del sistema a estar conectados al repositorio cada vez que necesiten actualizarlo.

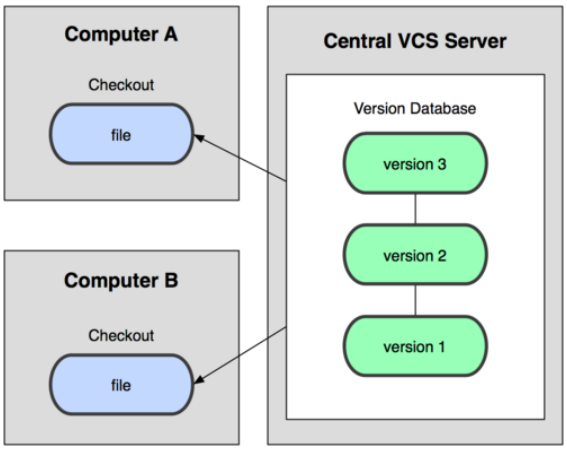


Figura 4 - VCS centralizado

Sistemas de control de versiones distribuidos

En las herramientas distribuidas en cambio, se sigue un formato de igual a igual, en el que en principio no existe un repositorio central. Cada desarrollador tiene su propio repositorio en el que desarrollar el proyecto. Esto permite a los usuarios intercambiar información entre ellos. Este enfoque permite distribuir el trabajo dando la oportunidad de que los desarrolladores se organicen por grupos para desarrollar partes concretas del código, y finalmente unificar todo.

Normalmente para facilitar la sincronización de los datos, en estos sistemas se establece un repositorio compartido al que todos acceden. Los desarrolladores siempre tendrán una copia del mismo en su máquina local, que supone trabajar sin necesidad de estar conectado a la red cada vez que se quiera realizar confirmaciones. Esto permite hacer las pruebas necesarias, incluso con otros desarrolladores, antes de actualizar la versión estable.

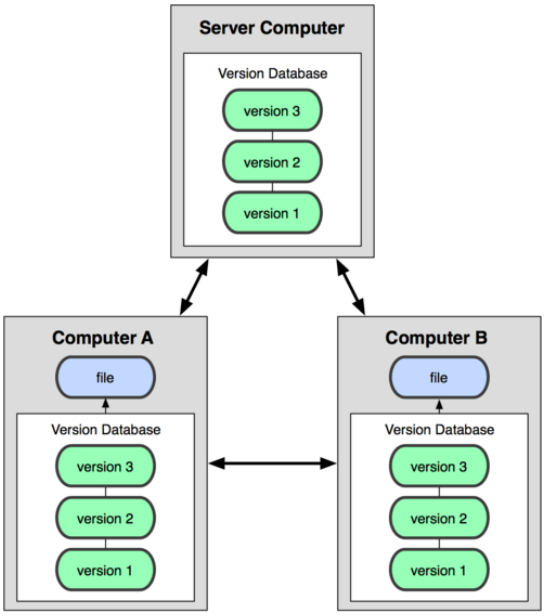


Figura 5 - VCS Distribuido

Cada tipo de VCS posee sus ventajas:

Ventajas de los distribuidos

- Se puede trabajar sin conexión (para hacer un commit no hay que estar conectado al repositorio central). Esto produce una mayor autonomía y una mayor rapidez.
- Se pueden crear ramas locales que no afecten a los demás usuarios.
- Un sistema distribuido es más ágil que uno centralizado ya que gran parte del trabajo se reparte entre cada máquina. En los centralizados, la mayor parte de trabajo la debe realizar la misma máquina, es decir el repositorio central.

Ventajas de los centralizados

- Siempre se tiene una versión actualizada del proyecto en desarrollo.
- Fácil de aprender y utilizar.

Por último, en la tabla siguiente se muestran los sistemas de control de versiones más populares:

Distribuidos	Centralizados
Git	Cvs
Mercurial	Subversion
Baazar	Team Foundation Server

Tabla 1 - Tipos de VCS

2.1.3 Operaciones básicas

Ramificación (branching)

A menudo durante el desarrollo de un proyecto, es muy común encontrarse con la necesidad de separar el trabajo en diferentes líneas de desarrollo para flexibilizar así la manera en la que los usuarios colaboran.

Ramificar consiste en realizar copias independientes por cada línea de desarrollo a partir de una base. Así, es posible realizar modificaciones asegurando que el código estable quede intacto. Por ejemplo, a partir de una rama de desarrollo principal, crear una de pruebas para probar una nueva funcionalidad. Sea cual sea el resultado la rama de desarrollo principal no sufrirá ningún cambio, lo que permitirá al usuario trabajar sobre seguro en todo momento.

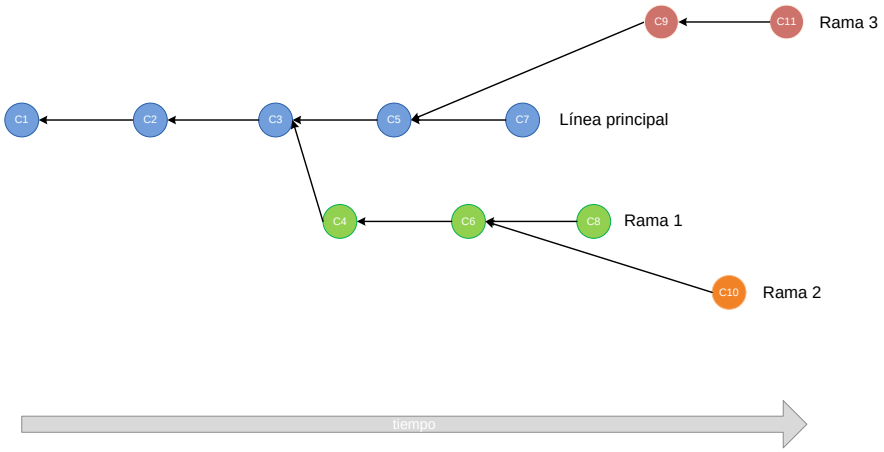


Figura 6 - Ejemplo de branching

En los proyecto es importante conocer cuándo se debe ramificar, pues no interesa llenar el repositorio de ramas cuyo cometido no está muy definido. Por ello, a continuación se exponen algunas de las recomendaciones a tener cuenta a la hora de ramificar.

Se debe ramificar cuando,

- Se siga evolucionando un software al mismo tiempo que se corrigen los errores de código.
- Surjan dos evoluciones de naturaleza distinta y por tanto no sea conveniente desarrollarlas de forma conjunta.
- Realizar un gran número de modificaciones que durante su desarrollo obligarían a dejar el repositorio en un estado inestable.

- Se quiere desarrollar una nueva funcionalidad para el proyecto base.
- Otros.

Además, a continuación se muestran algunos escenarios[3] de ejemplo posibles:

- Escenario 1: Sin ramas

My Team Project

└ **Main**

- Escenario 2: Rama de lanzamiento

My Team Project

└ **Main** → Main integration branch

│

└ **Releases**

  └ **Release 1** → Release branch

- Escenario 3: Rama de mantenimiento

My Team Project

└ **Main** → Main integration branch

│

└ **Releases** → Maintenance branch container

  └ **Release 1** → Maintenance branch

  └ **Release 2** → Maintenance branch

- Escenario 4: Rama de características

My Team Project

└ **Development** → Isolated development branch container

│ └ **Feature A** → Feature branch

│ └ **Feature B** → Feature branch

│ └ **Feature C** → Feature branch

│

└ **Main** → Main Integration branch

  └ **Source**

[3] http://msdn.microsoft.com/en-us/library/bb668955.aspx

- Escenario 5: Rama de subgrupos

My Team Project

└ **Development** → Isolated development branch container

| └ **Team 1** → Team branch

| | └ **Feature A** → Isolated branch for development

| | └ **Feature B** → Isolated branch for development

| |

| └ **Team 2** → Team branch

| └ **Feature A** → Isolated branch for development

| └ **Feature B** → Isolated branch for development

|

└ **Main** → Main Integration branch

Integrar cambios

En ocasiones surge la necesidad de integrar los cambios generados o parte de ellos de una rama de desarrollo sobre otra. Este suceso suele darse en ramas destinadas a desarrollar características específicas. Lo habitual suele ser, crear una rama a partir de un proyecto, realizar los cambios oportunos y finalmente aplicar los cambios de esa rama sobre su rama padre.

Para ello, los sistemas de control de versiones disponen de diferentes mecanismos. Por un lado existen dos mecanismo similares llamados fusión (merge) o reorganización (rebase).

La fusión (merge) aplica los cambios de una rama sobre otra creando una confirmación en la que ambas ramas se combinan. Esta técnica mantiene el orden cronológico en el que se fueron realizando las confirmaciones.

La reorganización (resabe) en cambio, replica todas las confirmaciones de una rama de trabajo determinada sobre otra. La rama que ha recibido los nuevos cambios contendrá sus propias revisiones y seguidas de estas todas las revisiones de la otra rama. Visualmente, parecerá como si las confirmaciones agregadas a la rama se hubieran realizado en esa misma rama.

A continuación se aprecia en una figura la diferencia entre ambos modos.

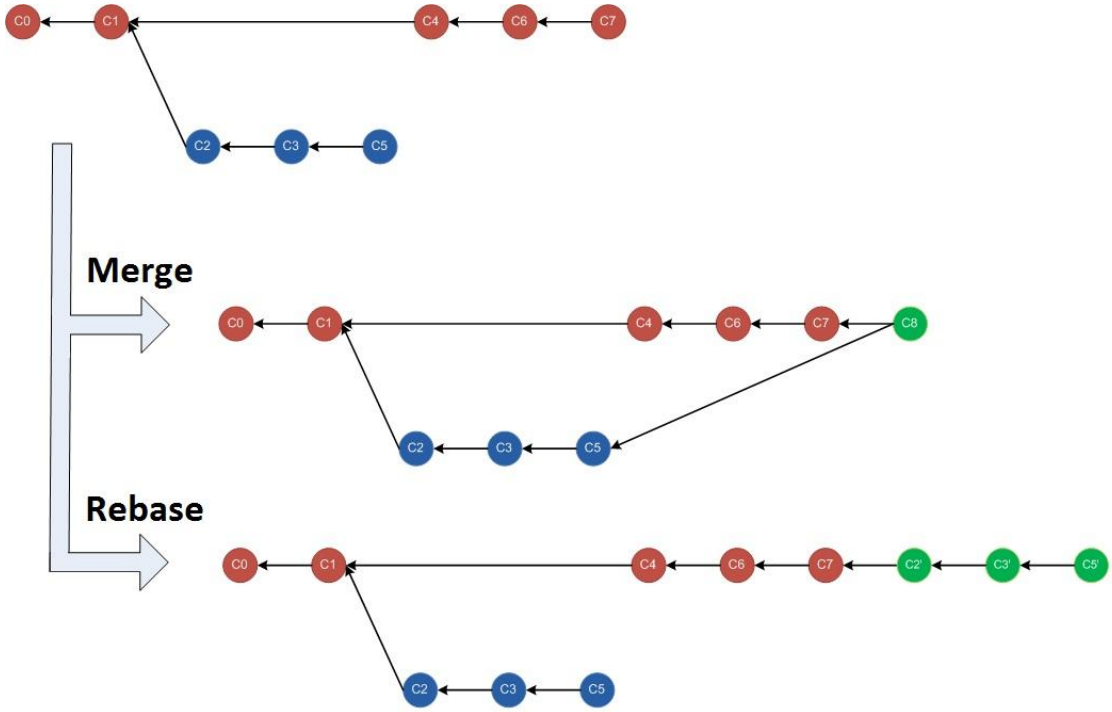


Figura 7 - Merge y Rebase

En la práctica sea cual sea el método utilizado no hay ninguna diferencia en el resultado de la integración, el contenido del directorio en el punto final será el mismo. Pero reorganizando se consigue un registro más claro. Si se examina el registro de una rama reorganizada, éste aparece siempre como un registro lineal, como si todo el trabajo se hubiera realizado en serie, aunque realmente se haya hecho en paralelo.

Los métodos descritos anteriormente son realmente útiles cuando se quiere aplicar todos los cambios de una rama sobre otra. Sin embargo muchas veces los desarrolladores tienen la necesidad de combinar cambios específicos, revisiones específicas. Para ello, existe un mecanismo llamado “cherry-pick” capaz de copiar tan sólo las revisiones que el usuario vea necesarias de una rama a otra. En realidad es muy similar al rebase, sólo que en vez de aplicar los cambios de todas las revisiones de la rama, permite elegir cuáles de ellas copiar.

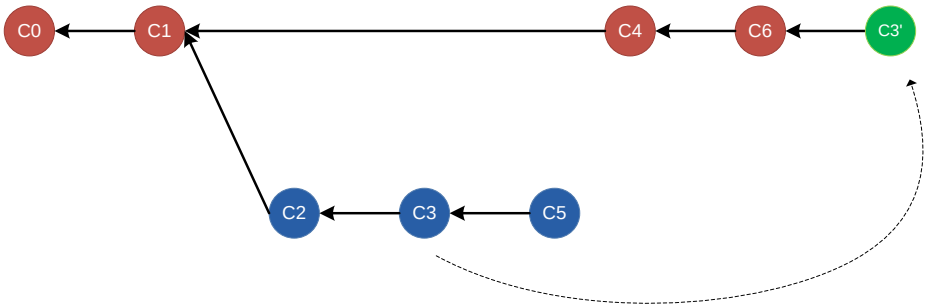


Figura 8 - Cherry pick

En cualquier caso, estos métodos para integrar cambios dependen del sistema de control de versiones. Por lo general todos son capaces de realizar fusiones, pero no todos son capaces de realizar “rebase” por ejemplo.

Revolver conflictos

En entornos de trabajo colaborativos, en los que diferentes usuarios realizan cambios simultáneamente sobre un mismo fichero o conjunto de ficheros para posteriormente fusionar los cambios realizados, entonces se pueden generar conflictos. Todos los sistemas de control de versiones detectan estos conflictos automáticamente y notifican a al menos uno de las personas involucradas de que sus cambios entran en conflicto con los de alguien más. Se diferencian dos tipos de conflictos:

1.

Conflictos que ocurren cuando dos o más personas están realizando cambios sobre un mismo fichero, pero sin que los cambios afecten a la misma porción del fichero.

A la hora de fusionar los cambios, el sistema combinará automáticamente los cambios de cada usuario.

2.

Cuando dos o más personas están realizando cambios sobre una misma porción de un fichero.

En este caso es entonces tarea de los afectados resolver el conflicto y comunicar esa resolución al sistema de control de versiones. A continuación se muestra un ejemplo de un caso real.

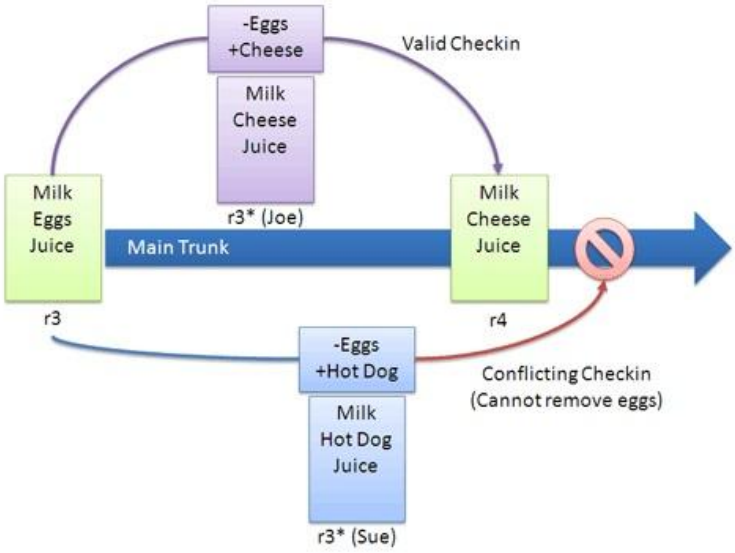


Figura 9 - Resolver conflictos

Joe quiere eliminar los huevos y reemplazarlos por queso (-eggs, +cheese), y Sue quiere reemplazar los huevos por un perrito caliente (-eggs, +hot dog).

Hay dos enfoques diferentes para solucionarlo,

1. *Aplicar de nuevo los cambios: Sincronizarse a la última versión (r4) y añadir los cambios al fichero. Es decir, manteniendo los cambios de Joe añadir los de Sue. Añadir el perrito caliente a la lista del queso.*
1. *Anular los cambios con los propios: Consultar la última versión (r4), y sobrescribirla con los cambios de Sue. Eliminará el queso reemplazándolo por el perrito caliente.*

Sin embargo, hay sistemas de control de versiones que poseen una solución alternativa al proceso anterior, son capaces de establecer bloqueos en los ficheros. De esta forma se evitan los conflictos por completo, pues sólo una persona puede realizar cambios en un momento dado. Una vez se bloquea un fichero, hasta que no se desbloquee nadie más podrá acceder, excepto el usuario que estableció el bloqueo. Cuando lo desbloquee permitirá nuevamente el acceso a todas las personas. Así se garantiza un acceso exclusivo.

2.1.4 Flujos de trabajo

Los flujos de trabajo de un VCS definen cómo se relacionan los usuarios de un grupo de trabajo para colaborar entre sí en la consecución de los objetivos del proyecto.

A diferencia de los sistemas de control de versiones centralizados, la naturaleza de los distribuidos permite más flexibilidad a la hora de colaborar en proyectos. En los sistemas centralizados, cada desarrollador contribuye a un repositorio común. En los distribuidos, cada desarrollador es tanto un repositorio como un contribuyente de un repositorio. En otras palabras, cada desarrollador puede tanto contribuir a otros repositorios, como servir de repositorio público sobre el que otros puedan basarse. Esto abre un gran abanico de posibilidades de trabajo en los proyectos.

En esta sección se muestran enfoques de ejemplo que un proyecto puede tomar de acuerdo a las necesidades del grupo de trabajo. También se pueden dar variaciones o mezclas de los mismos con el objetivo de encontrar el más adecuado a las necesidades específicas.

Flujo de trabajo centralizado

Esté modo de trabajar es el más sencillo y el más común para usuarios acostumbrados a la arquitectura de trabajo de los VCS centralizados. Consiste en utilizar un repositorio central o compartido para centralizar los datos, de modo que todos los desarrolladores sincronicen su trabajo con él. Cada desarrollador es un nodo de trabajo y operan en pie de igualdad sobre el repositorio remoto.

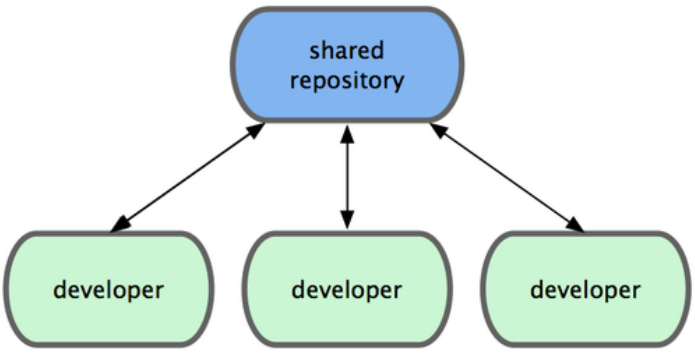


Figura 10 - Flujo de trabajo centralizado

Una desventaja de este modo de trabajo es que si dos desarrolladores copian los datos desde el repositorio remoto, y ambos hacen cambios; tan solo el primero de ellos en enviar sus cambios de vuelta lo podrá hacer limpiamente. El segundo desarrollador deberá fusionar previamente su trabajo con el del primero, antes de enviarlo, para evitar sobrescribir los cambios del primero.

Flujo de trabajo del Gestor de Integraciones

En los VCS distribuidos trabajar con varios repositorios conjuntamente es algo habitual. De este modo se permite realizar una integración por partes.

El flujo de trabajo está compuesto por el grupo de desarrolladores y por el administrador del sistema, este último será el único con acceso al repositorio central. Cada desarrollador tiene acceso a un repositorio propio sobre el que realizar modificaciones. Este repositorio es accesible sólo para lectura para los demás desarrolladores. Cuando un desarrollador quiere realizar cambios, la manera de funcionar consiste en: el desarrollador accede al repositorio común o al de uno de los desarrolladores, según sus necesidades, para copiar la parte de trabajo en su repositorio público. Después realizará los cambios pertinentes. Cuando necesite incorporar esos cambios al repositorio central, no podrá hacerlo directamente. Tendrá que comunicarle al administrador que desea incorporar los cambios. Entonces, el administrador copiará los datos del repositorio público del desarrollador que ha solicitado la incorporación y los integrará en el repositorio central.

A continuación se muestra un ejemplo de este flujo de trabajo:

1. La persona que desea contribuir, clona el repositorio principal y hace algunos cambios.

2. El contribuidor envía (push) una copia de su repositorio a su propio repositorio público.

3. Éste comunica al gestor que ha realizado cambios y que necesita integrar los cambios.

4. El gestor añade como repositorio remoto el del contribuidor y fusiona (merge) los cambios localmente.

5. Finalmente el gestor sincroniza (push) los cambios fusionados en el repositorio principal.

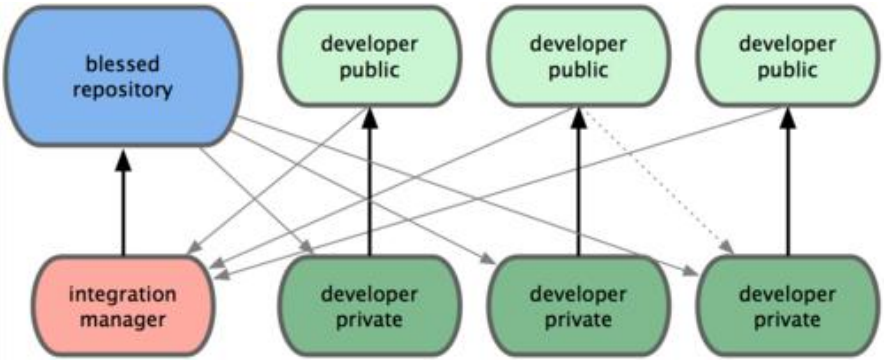


Figura 11 - Flujo de trabajo del gestor de integraciones

Flujo de trabajo con "Dictator and Lieutenants"

A partir del escenario anterior es posible construir una aplicación del mismo orientado generalmente a proyectos de mayor escala en cuanto al número de colaboradores.

Este flujo de trabajo consiste en dividir las tareas aplicando cierto grado de jerarquía entre los desarrolladores. El sistema está compuesto por desarrolladores, tenientes y un dictador. Los desarrolladores se encuentran en el nivel más bajo jerárquicamente y se encargarán principalmente del desarrollo de software. Los tenientes serán los encargados de supervisar y validar el trabajo de los desarrolladores para solicitar una incorporación del código realizado. Finalmente el dictador integrará todas las aportaciones de los tenientes publicando el trabajo en el repositorio de referencia o central al que todos los colaboradores tienen acceso. Este sistema de trabajo permite al líder del grupo (el dictador) delegar gran parte del trabajo en los tenientes, relegando su trabajo en recolectar el fruto de múltiples puntos de trabajo.

El proceso es de la siguiente manera:

1. Los desarrolladores habituales trabajan cada uno en su rama puntual y reorganizan (rebase) su trabajo sobre su rama principal. La rama principal es la del dictador.

2. Los tenientes fusionan (merge) las ramas puntuales de los desarrolladores sobre su propia rama principal.

3. El dictador fusiona las ramas principales de los tenientes en su propia rama principal.

4. El dictador envía (push) su rama principal al repositorio de referencia, para permitir que los desarrolladores reorganicen (rebase) desde ella.

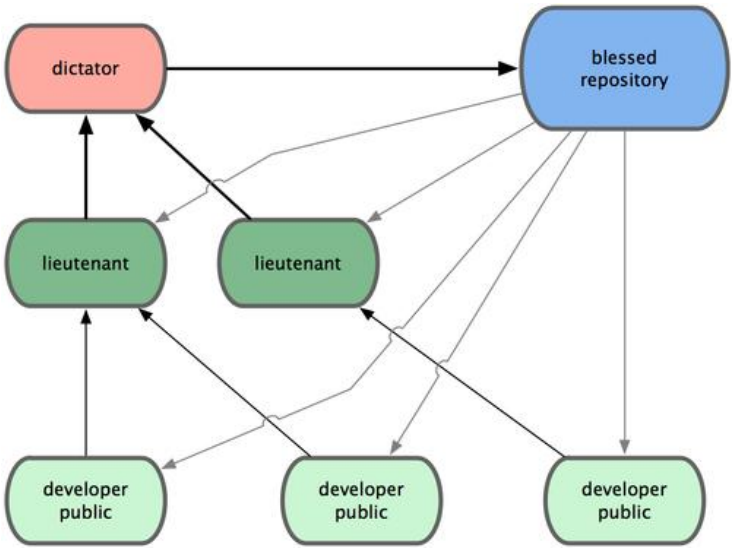


Figura 12 - Flujo de trabajo con dictador y tenientes

2.1.5 Interfaces gráficas de usuario (GUIs)

Los sistemas de control de versiones son herramientas que por sí solas no realizan ninguna acción, ya que están destinadas a actuar en función de lo que el usuario ordene. La forma de indicar las acciones a realizar es mediante la ejecución de comandos. Esto supone que el usuario ha de tener nociones mínimas sobre el uso de los comandos en el terminal, además de tener que manejarse con agilidad para no perder mucho tiempo y poder continuar con su trabajo con normalidad. En muchos casos los usuarios no tienen dicho conocimiento.

Con el objetivo de hacer la vida más fácil a los usuarios, las interfaces gráficas de usuario (GUI, por sus siglas en ingles) disminuyen parcialmente o totalmente, según el caso, la necesidad de ejecutar instrucciones en el terminal de comandos. Estas herramientas dan una visión gráfica a modo de árbol para visualizar el historial de revisiones del repositorio y poder realizar las diversas operaciones sobre él.

En la actualidad hay una gran variedad de GUIs para las diferentes plataformas existentes. En las tablas siguientes se muestran algunas de las posibilidades:

Herramienta	Sistema de Control de Versiones
Tortoise	Git - Mercurial - Subversion
EasyMercurial	Mercurial
Git Extensions	Git
SourceTree	Git - Mercurial
RabbitVC	Git
eGit(eclipse)	Git
HgEclipse (eclipse)	Mercurial
Subversive (eclipse)	Subversion

Tabla 2 - GUIs por VCS

Herramienta	Sistema Operativo
Tortoise	Windows - Linux - MacOS
EasyMercurial	Windows - Linux - MacOS
Git Extensions	Windows - Linux - MacOS
SourceTree	Windows - MacOS
RabbitVC	Linux
eGit(eclipse)	Windows - Linux - MacOS
HgEclipse (eclipse)	Windows - Linux - MacOS
Subversive (eclipse)	Windows - Linux - MacOS

Tabla 3 - GUIs por sistemas operativos

2.1.6 Servicio de alojamiento de repositorios en la web

La gestión del repositorio es una de las tareas más significativas a realizar para garantizar un correcto funcionamiento de un SCV. Lo habitual es hacerlo de forma local, en una máquina de la red de la empresa. Pero en ocasiones y en función a las necesidades del grupo de trabajo conviene albergar el repositorio en un servidor externo, en la nube (cloud). Los servicios de almacenamiento de repositorios permiten alojar en la nube el repositorio en común de un grupo de desarrolladores. Estas herramientas ofrecen un conjunto de características muy útiles para el trabajo en equipo.

Dependiendo de la naturaleza del proyecto estos repositorios pueden ser privados o públicos. Por ejemplo para grupos de desarrollo de código libre interesa utilizar repositorios públicos, para que otros desarrolladores o empresas puedan reutilizar o añadir contenido.

A continuación se muestra una tabla con los servicios de alojamiento de repositorios de las páginas más populares y un ejemplo de una de ellas. (Ver figura 13)

Servicio	SCV
Github	Git
Bitbucket	Git, Mercurial
Google code	Git, SVN, Mercurial

Tabla 4 - Servicios de alojamientos de repositorios en la web

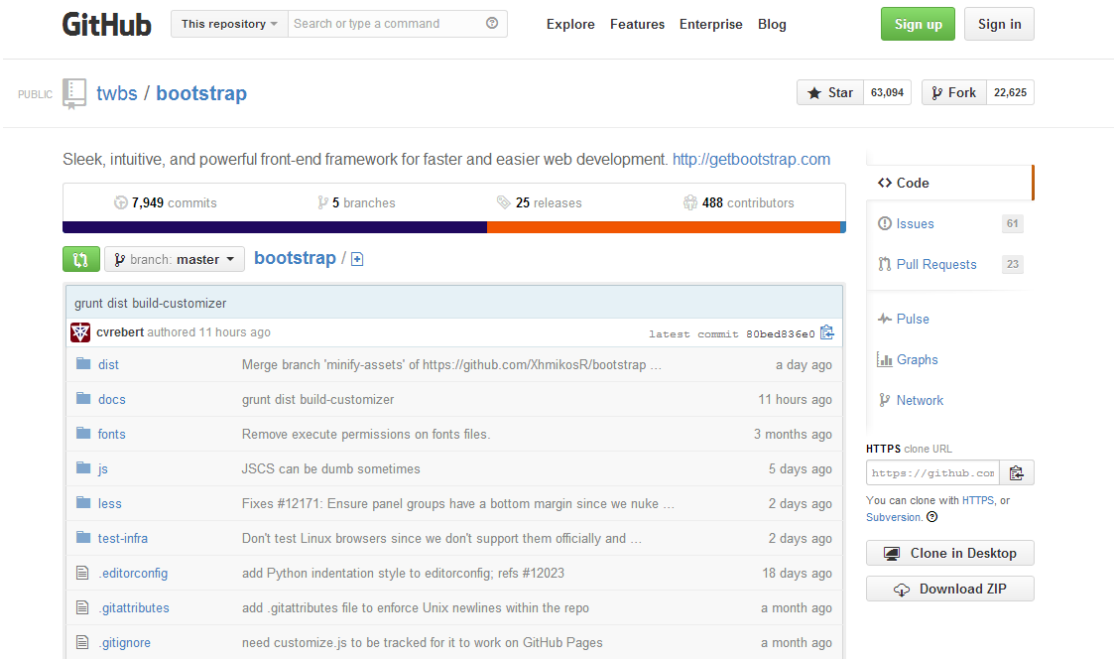


Figura 13 - Github: ejemplo de repositorio público

2.2 Herramientas de gestión

Este apartado presenta un conjunto de aplicaciones que facilitan y mejoran el desarrollo de los proyectos. En esta sección se explican, entre otras, aplicaciones que sirven para gestionar las incidencias de las herramientas, comparar diferencias en el código y validar el trabajo realizado por otros desarrolladores.

2.2.1 Sistemas de seguimiento de incidencias / errores

Los sistemas de seguimiento de incidencias [23] son herramientas de administración que gestionan listas de incidentes encontrados en los proyectos de una organización. Normalmente, se utilizan en servicios de atención al cliente para crear, actualizar y resolver todo tipo de incidencias enviadas por los usuarios o por los empleados de la organización.

Este tipo de sistemas son similares a los sistemas de seguimiento de errores, de hecho se consideran como un tipo especial de éstos. Lo normal en las empresas es beneficiarse de ambos.

Los sistemas de seguimientos de incidencias resuelven las incidencias de una lista de tickets. Un ticket es un archivo contenido en el sistema que contiene información acerca de las intervenciones de software hechas por los empleados o usuarios, que están impidiendo continuar una tarea específica. Además, estos se pueden organizar por prioridad, estado, fecha, código de identificación, etc.

A modo de ejemplo a continuación se describe brevemente una aplicación de seguimiento de incidencias.

Jira

Jira [24] es una herramienta que además de estar orientada al seguimiento de incidencias, también realiza seguimiento de errores, gestión operativa de proyectos, etc. Es una herramienta de las más potentes del mercado. Está desarrollada por la empresa Attlasian, empresa que dispone de muchas más aplicaciones para la gestión de proyectos. Algunas de las funcionalidades que tienen son:

- Árbol histórico de confirmaciones.
- Estadísticas del repositorio.
- Comparador de diferencias.
- Búsqueda de contenido.
- Revisiones y/o comentarios de las confirmaciones.
- Subir archivos.
- Wiki.

En la siguiente figura se puede ver un ejemplo del navegador de incidencias.

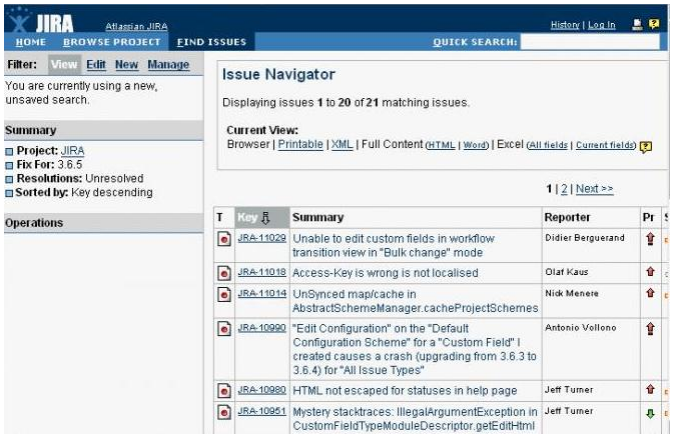


Figura 14 - Jira: Visualizar incidencias

2.2.2 Comparación de ficheros/ directorios

El trabajo colaborativo en las empresas presenta un escenario en el que los desarrolladores llegan a coincidir en el desarrollo de proyectos o mismas tareas. Hasta tal punto que probablemente coincidan también en los elementos que están desarrollando, por ejemplo generando partes de código diferentes de un mismo fichero.

En la mayoría de los casos, el trabajo realizado debe ser combinado para obtener un resultado completo, que integre las diferentes partes desarrolladas por cada desarrollador del elemento en cuestión.

Como solución a este dilema, hay aplicaciones orientadas a la comparación de ficheros y algunas incluso de directorios. Estas herramientas son capaces de mostrar visualmente línea a línea las diferencias de hasta tres ficheros. Además, permiten combinar a gusto del usuario los diferentes elementos en uno.

El problema surge cuando los ficheros muestran diferencias en una misma porción de código, que se resume en un conflicto entre los ficheros. Por ello, estás herramientas permiten resolver los conflictos generados eligiendo línea a línea que parte dar por válida. Adicionalmente, la herramienta es capaz de mostrar el estado del fichero resultado en tiempo real.

Cuando no existan conflictos entre los ficheros la herramienta permitirá fusionar los cambios en un fichero resultado.

Además algunas de estas aplicaciones son integrables con los VCS, permitiendo así realizar fusiones con facilidad.

A continuación se muestran algunas de las herramientas de comparación orientadas a comparar y revolver conflictos en las fusiones de los VCS:

Kdiff3

KDiff3 [25] es una herramienta que permite ver las diferencias entre archivos y fusionarlos una vez solventado los conflictos si los hubiese. Además también permite comparar directorios. Características:

- Compara y fusiona dos o tres archivos / directorios.
- Muestra las diferencias línea a línea y carácter a carácter.
- Proporciona una utilidad de fusión automática.
- Tiene un editor para resolver de forma cómoda los conflictos.
- Proporciona transparencia de red a través de KIO.
- Tiene opciones para resaltar u ocultar los cambios en los espacios en blanco o en los comentarios.
- Soporta Unicode, UTF-8 y otras codificaciones de archivo.
- Imprime diferencias.
- Soporta claves de control de versiones e historial de fusionado.

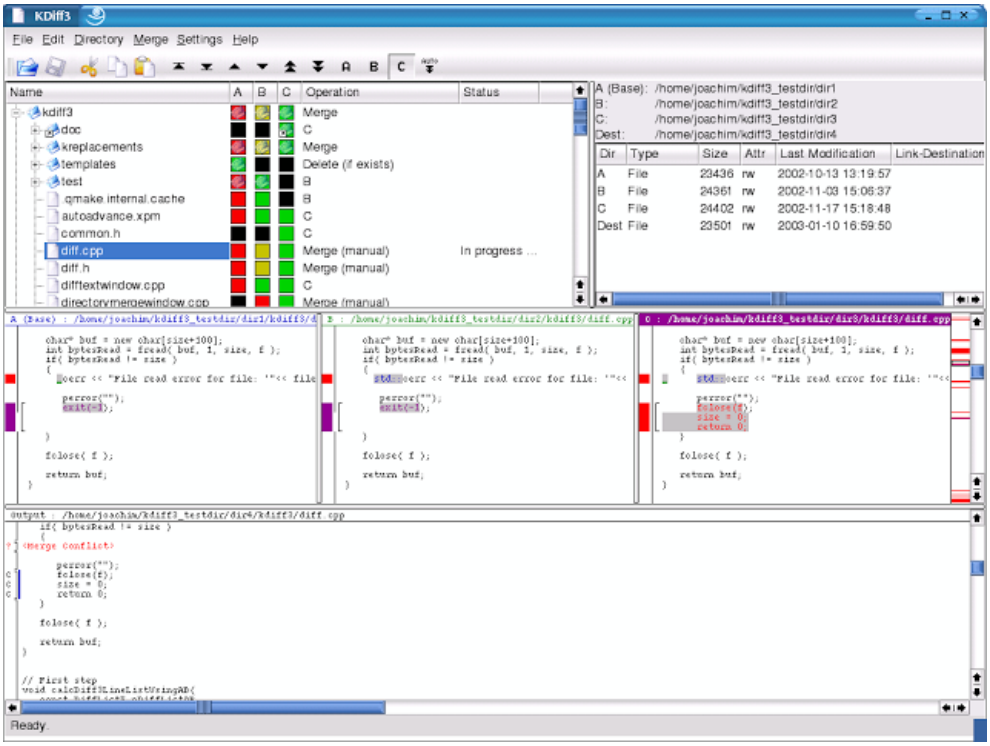


Figura 15 - Kdiff3: Comparación de directorios

Kdiff3 es una herramienta de la que interesa estudiar la capacidad de integración que tiene con otras herramientas. Existen herramientas que podrían beneficiarse de su funcionalidad para comparar ficheros o directorios. Por ejemplo:

- IDE Eclipse: permite integrar kdiff3 pero sólo como herramienta de comparación de ficheros o archivos del directorio de trabajo. No es posible realizar comparaciones entre distintos nodos del repositorio.
- Git: la integración con Git es completa, permitiendo invocar kdiff3 desde el terminal de comandos al ejecutar el comando correspondiente para ello.

SemanticMerge

SemanticMerge [26] es una aplicación en estado beta, que resuelve automáticamente conflictos entre ficheros. Se trata de la primera herramienta comercial con un método de análisis revolucionario. Permite analizar los cambios no como lo hacen la mayoría de este tipo de herramientas, a nivel de línea. Lo hace a un nivel más abstracto, analizando las diferencias entre métodos, clases, propiedades o campos de un mismo fichero. Además, permite convertir las operaciones manuales en operaciones automáticas. Actualmente soporta lenguajes como

JAVA, C# y VB.Net. Pero están trabajando para dar soporte a otros. El próximo lenguaje soportado será C++.

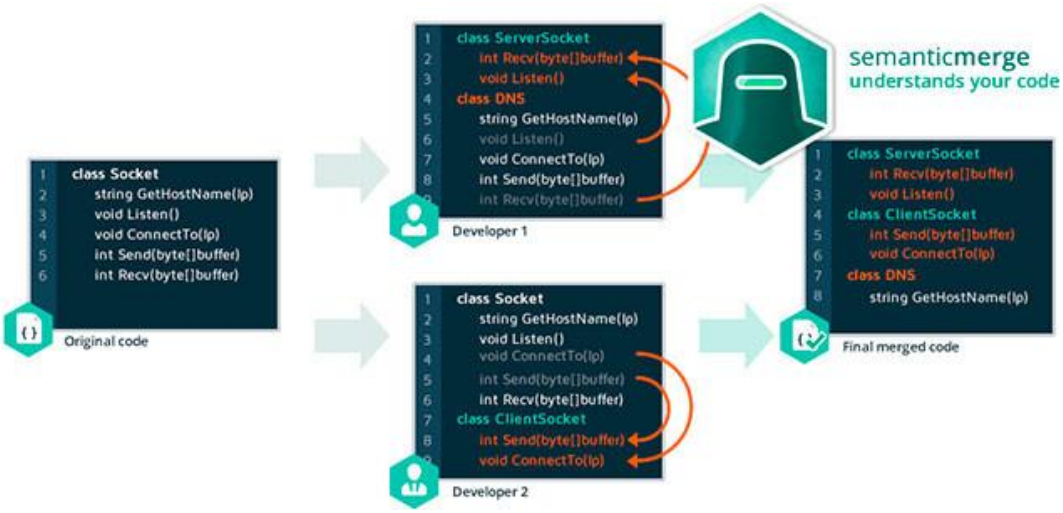


Figura 16 - Semanticmerge: Fusión

2.2.3 Revisión de código

En el desarrollo de código fuente es habitual cometer errores en el código o escribir código poco eficiente [27]. Con el objetivo de mejorar dicho código, existen herramientas para supervisar los cambios en el código, de manera que sea posible sugerir nuevas ideas o simplemente validar lo realizado. Este tipo de tareas son posibles gracias a herramientas de revisión de código. Están enfocadas a encontrar y resolver errores en la primera fase de desarrollo, mejorando tanto la calidad del software como las habilidades de los desarrolladores. A continuación se muestran algunas de ellas:

- Crucible
- Gerrit
- Barkeep
- Rhotecode

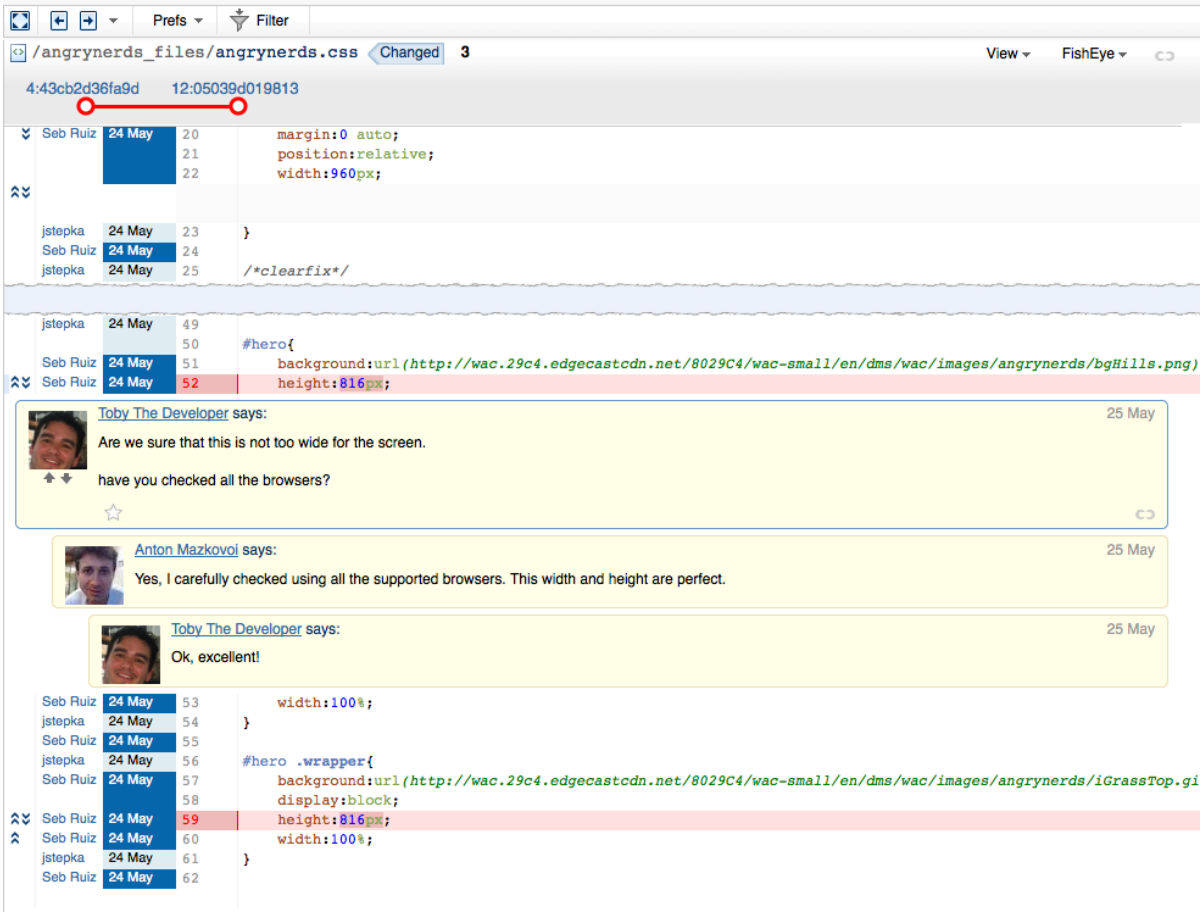


Figura 17 - Ejemplo de crucible

2.3 Plugins de Eclipse

En esta sección se muestran plugins de interés para gestionar sistemas de control de versiones del entorno de desarrollo Eclipse, IDE muy extendido y que permite extender su funcionalidad fácilmente mediante plugins.

2.3.1 jGit

JGit [13] es la librería Java que implementa el sistema de control de versiones Git. Además, permite a los desarrolladores abstraerse de conocer el funcionamiento de Git al detalle.

2.3.2 eGit

Egit [15] es un plugin de Eclipse para el desarrollo de software colaborativo que permite utilizar el sistema de control de versiones distribuido Git directamente desde Eclipse. Para ello hace uso de la librería jGit para Eclipse. Realiza la misma labor que una GUI de Git, pero para el IDE eclipse. Características:

- Comparador de ficheros

  Contiene un comparador de ficheros que muestra los cambios entre una versión y la anterior.

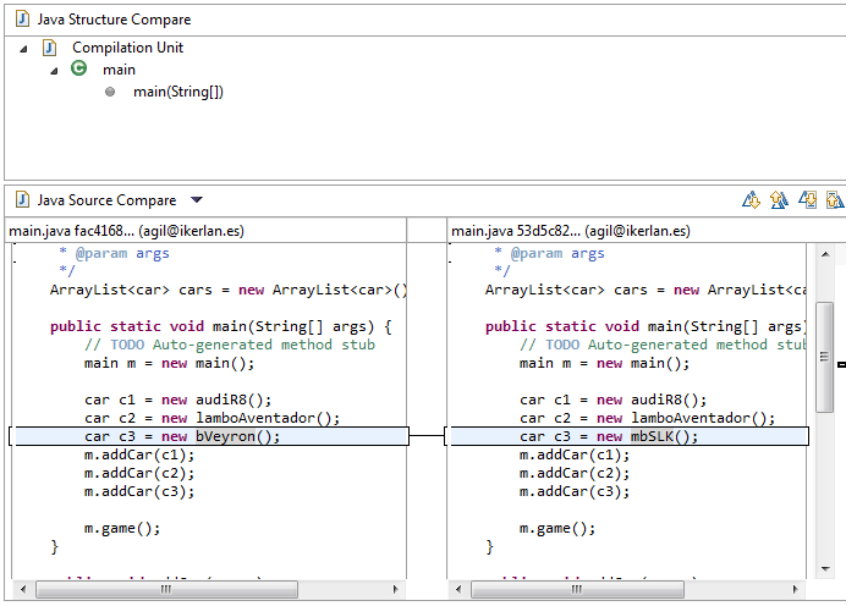


Figura 18 - eGit: Comparador de diferencias

- Historial de revisiones:

  Dispone de un árbol ordenado cronológicamente que muestra todas las revisiones del repositorio y las relaciones entre sí.

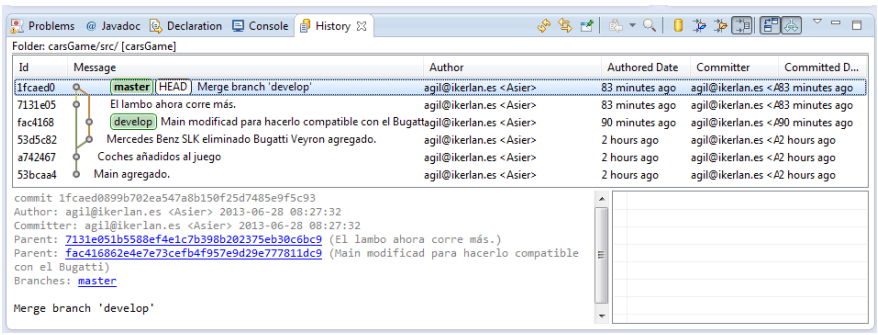


Figura 19 - eGit: Historial de revisiones

- Herramienta de resolución de conflictos

  Usa el comparador para solucionar los conflictos generados en las fusiones. Además tiene la opción de hacer la comparación a tres bandas. (Ancestro, rama actual, rama a fusionar)

2.3.3 Plug- in Development Enviroment

PDE [19] es un framework que provee herramientas para crear, desarrollar, probar, depurar, compilar y ejecutar plug-ins, fragmentos, características y productos RCP de eclipse.

El componente más importante del framework es el fichero plugin.xml. La información almacenada en este fichero indica de qué elementos de interfaz de usuario está compuesto el plugin, las dependencias entre plugin/librerías y todo tipo de aspectos de configuración.

Es importante remarcar el uso de las extensiones. Son los diferentes elementos de los que un plugin está compuesto. Además, mantienen la modularidad entre elementos para que puedan ser reutilizados fácilmente o sustituidos. Existen un amplio abanico de extensiones: menús desplegables, vistas, comandos, etc. La configuración de dichas extensiones se almacena en el fichero plugin.xml. En la figura siguiente se muestra un ejemplo de las extensiones que un plugin puede tener:

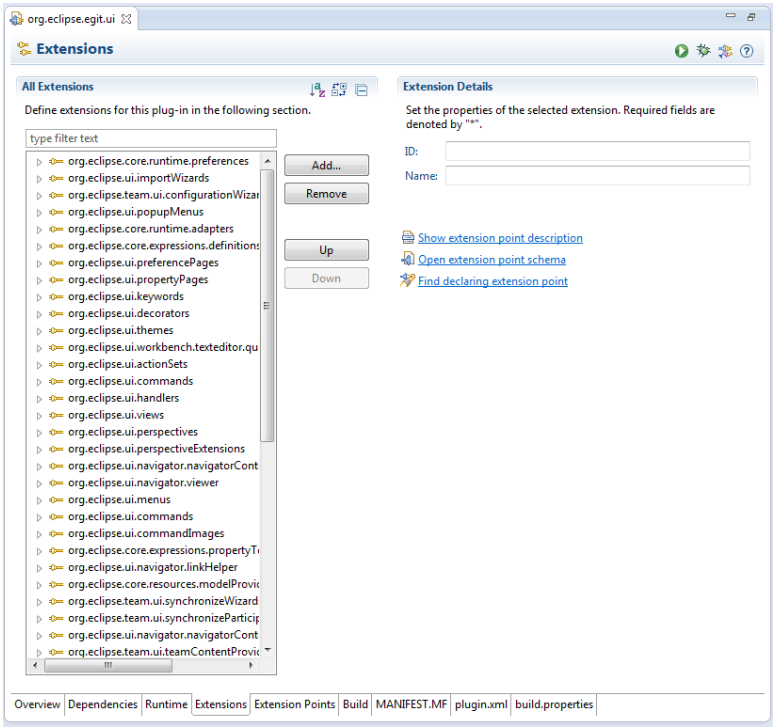


Figura 13 -- Extensiones de PDE

Finalmente, es importante mencionar que el framework también dispone de un fichero destinado a almacenar las cadenas de texto de la aplicación. El fichero se denomina plugin.properties.

2.4 Otros

Este apartado muestra presenta el estudio de otras herramientas adicionales a las anteriores.

2.4.1 gitFlow

gitFlow [21] es una extensión de Git que facilita la gestión de ramas y flujos de trabajo para Git. Git es un sistema de control de versiones en el que se pueden crear ramas rápidamente para cualquier cometido. Al igual que se puede ramificar correctamente también se puede hacer en la incorrecta. Si no se tienen unos conocimientos básicos para ramificar, el usuario puede cometer acciones inconsistentes, complicando el orden del repositorio o incluso estropeándolo. Por lo que necesariamente hay que seguir unas reglas que definan la manera de ramificar correctamente, sin lugar a fallos.

gitFlow da una solución a esta problemática proveyéndole de unas reglas a seguir. Estas reglas o metodología están adquiridas del modelo de ramificación que publicó en enero de 2010 Vincent Driessen en su blog [13].

El modelo de ramificación que él define es el siguiente:

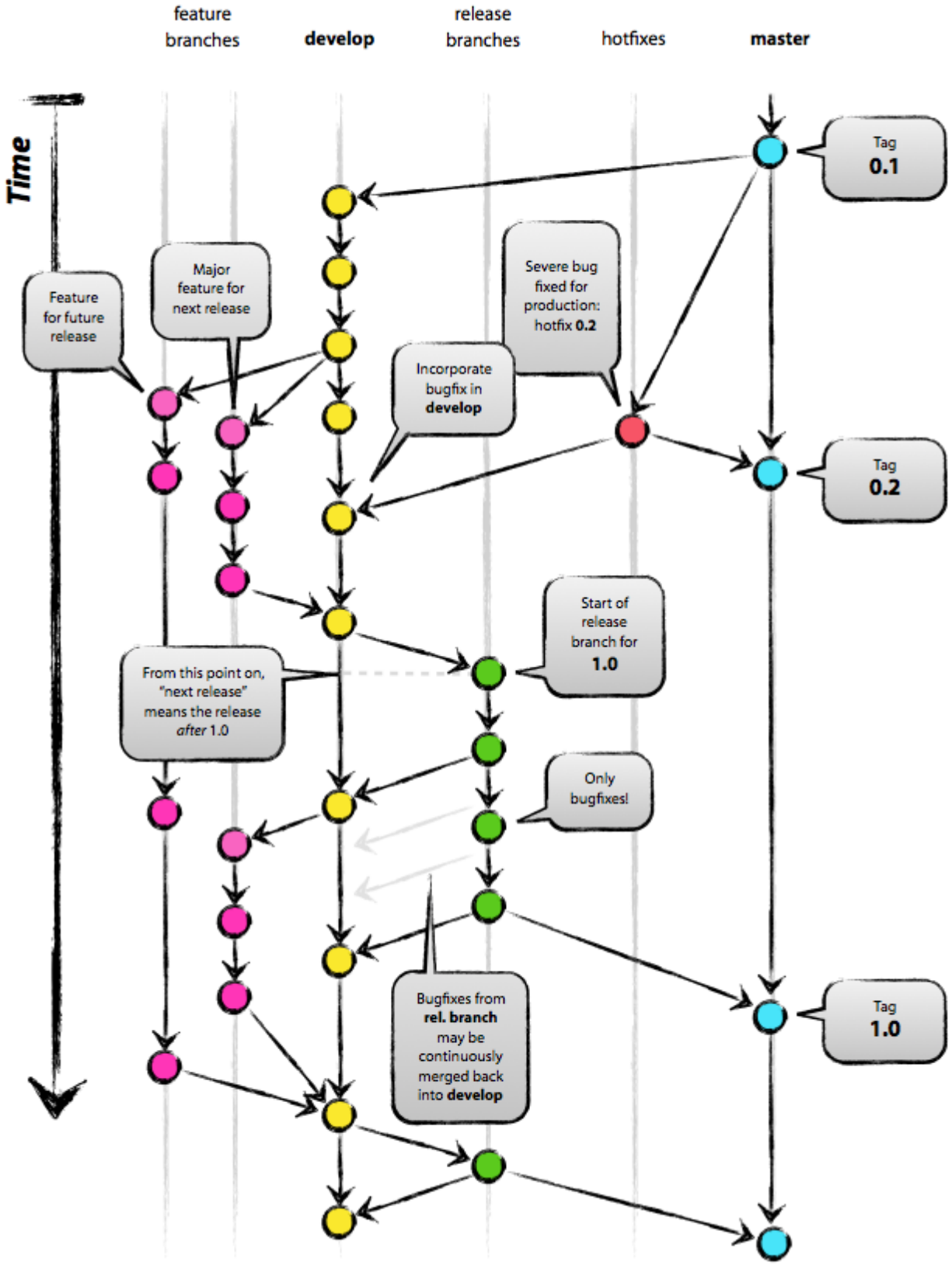


Figura 20 - Modelo de ramificación de Vincent Driessen

3 SOLUCIÓN PLANTEADA

En esta sección se describe la solución planteada a problemática descrita, en concreto la metodología a seguir, los escenarios de aplicación a investigar y la propuesta de solución. También, se encuentra en esta sección la validación del cliente a la propuesta planteada.

3.1 Contribución innovadora

En el estado del arte, existen sistemas comerciales para el control de versiones del software (e.g. CVS, SVN, GIT, etc). Lamentablemente no están diseñados para escenarios como el que nos ocupa. Los sistemas comerciales están orientados a procesos de desarrollo de software *largos y estrechos:* Procesos en los que se generan versiones (ramas) de desarrollo de los productos para desarrollar distintas funcionalidades, pero que tienen vocación de converger en el tronco común. Es decir, son productos individuales. En la figura siguiente se puede ver un ejemplo de dos proyectos diferentes:

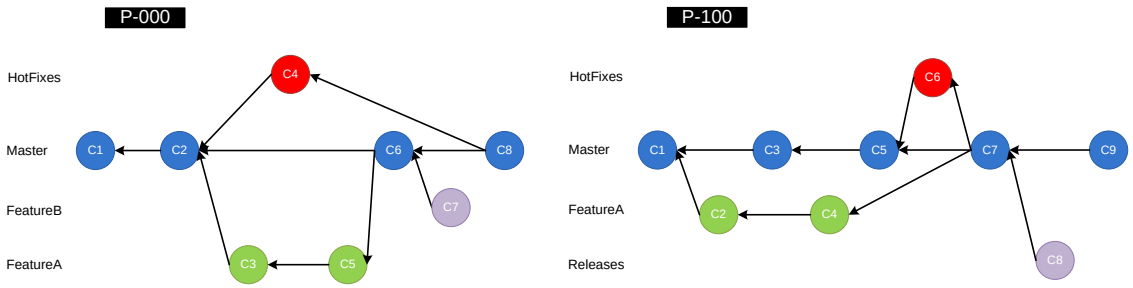


Figura 21 - Metodología convencional

En el caso que nos ocupa, por el contrario, se generan versiones (ramas) de desarrollo para proyectos específicos de clientes, que deben mantenerse abiertos para siempre (no convergerán). Es decir, son familias de productos. En la siguiente ilustración se puede observar un escenario de ejemplo para esta metodología.

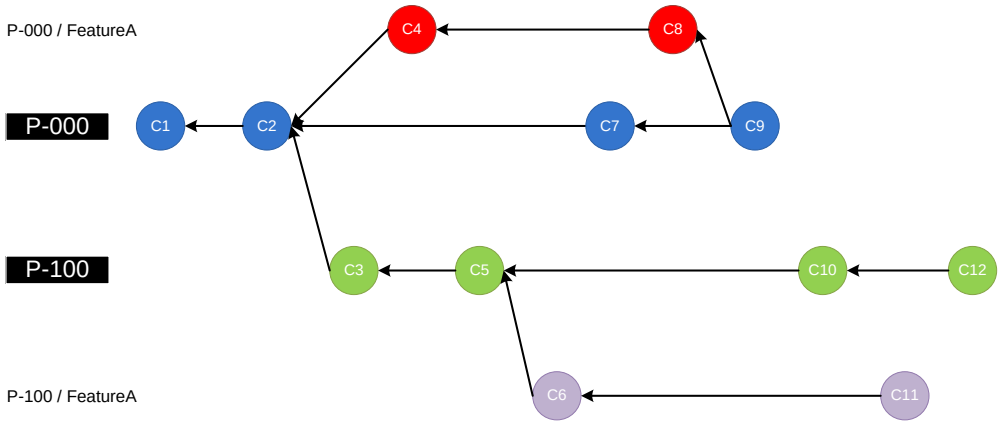


Figura 22 - Nueva metodología

Por otro lado este escenario plantea el reto de la escalabilidad: en un proceso de desarrollo de software clásico hay pocas ramas abiertas (unidades o unas pocas decenas). En el caso de productos de maquinaria industrial (como el caso de Ulma packaging), podemos estar hablando de miles de máquinas distintas vendidas al año, lo que se traduce en miles de ramas abiertas al año, y que no se cerrarán jamás.

Además de la falta de herramientas adecuadas, y quizás relacionado al hecho de que no existan, tampoco se cuenta con metodologías para racionalizar este tipo de procesos ni tampoco procesos de referencia que puedan servir como guía-modelo a la hora de abordar dicha racionalización

3.2 Escenarios de aplicación

La problemática del proyecto ya ha sido descrita de forma general. Para concretar los requerimientos del proyecto se han descrito una serie de escenarios con problemas reales que enfrentan los desarrolladores de software. El resto del proyecto consiste en resolver dichos escenarios.

Los escenarios utilizados corresponden a un caso real de la empresa ULMA Packaging S.Coop.

3.2.1 Punto de partida para un nuevo proyecto

En el desarrollo de productos software es habitual encontrarse con la necesidad de comenzar un nuevo proyecto, que es una variante de un proyecto existente al que hay que aplicarle nuevas funcionalidades. En ese proceso aparece la necesidad de elegir la fuente de origen para el nuevo proyecto. En este escenario se plantean dos opciones diferentes:

- Partir de un proyecto parecido que se hizo para un cliente. Y hacerle las modificaciones (al menos las críticas) que se han hecho desde entonces en el desarrollo principal.

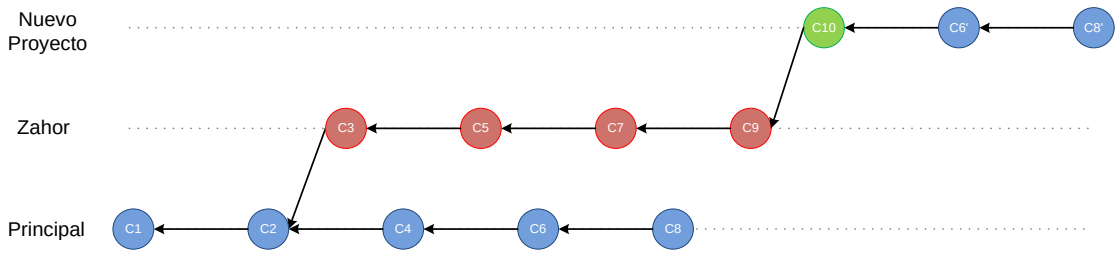


Figura 23 - de partida para un nuevo desarrollo A

Partiendo de la revisión C9 del proyecto Zahor, comenzar un nuevo proyecto y aplicarle algunos de los cambios del proyecto principal (C6' y C8').

- Partir de la rama de desarrollo principal y hacerle las modificaciones específicas que se hicieron para aquel proyecto parecido que se hizo para un cliente.

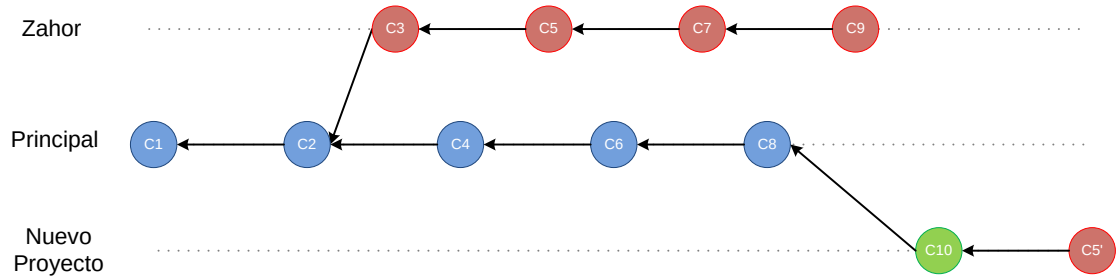


Figura 24 - Punto de partida para un nuevo desarrollo B

A partir de la revisión C8 del proyecto principal crear un nuevo de proyecto y aplicarle algunos cambios (C5') del proyecto Zahor.

3.2.2 Fusionar ramas lejanas o no conectadas directamente

Este escenario se basa fundamentalmente en la reutilización de características en un determinado proyecto. Es decir, consiste en incorporar los cambios realizados en otros proyectos sobre uno en concreto.

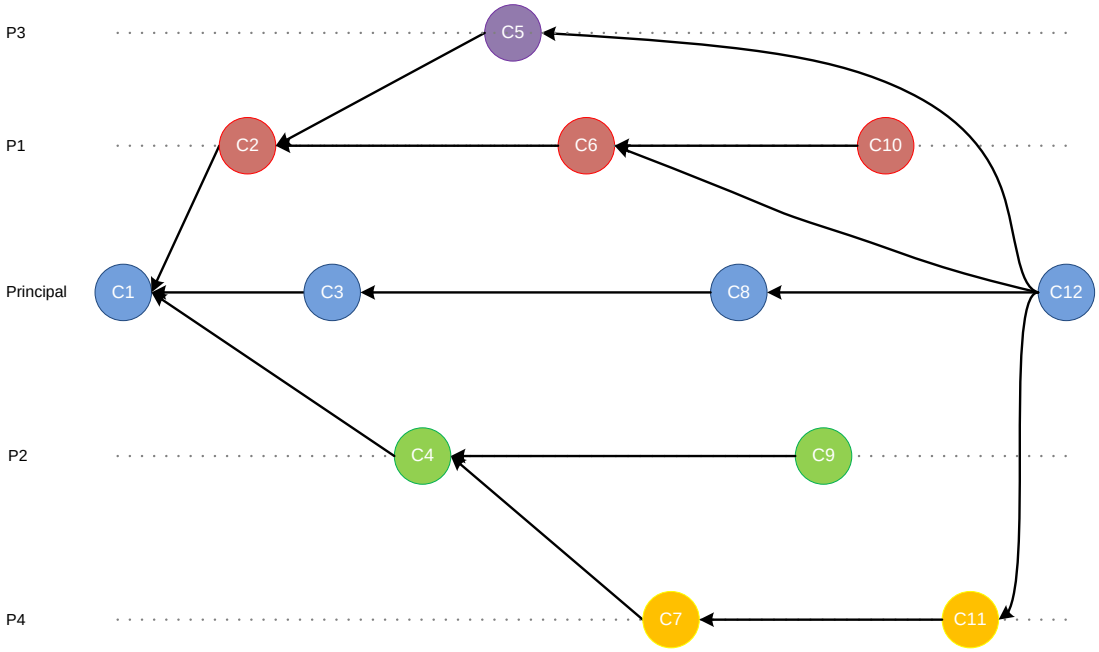


Figura 25 - Fusionar ramas lejanas

El proyecto principal incorpora a su desarrollo los cambios realizados en los proyectos P1 (hasta la revisión C6), P3 y P4.

3.2.3 Control de versión software de múltiples proyectos

En entornos donde se desarrollan distintas versiones de un mismo proyecto o variantes de un mismo proyecto, resulta complicado identificar qué versión de software corresponde a cada variante.

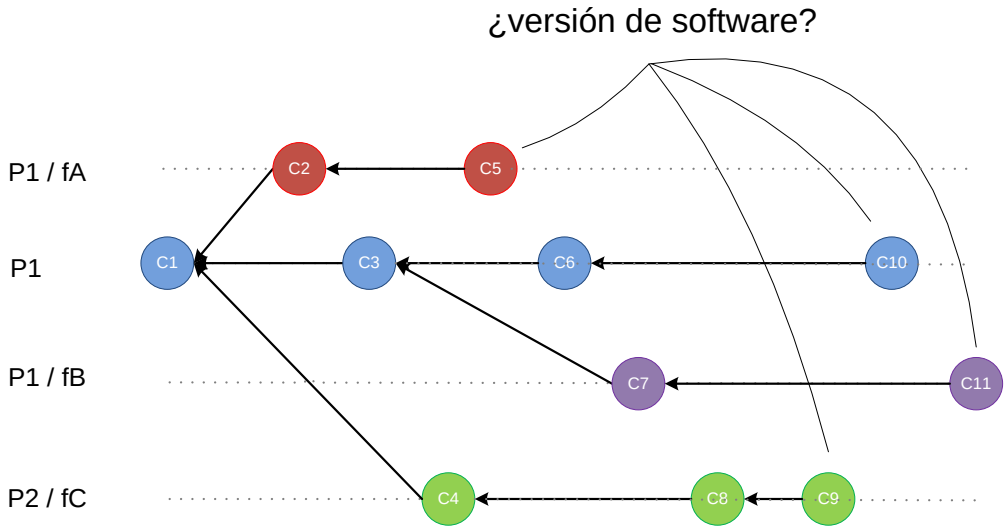


Figura 26 - Control de versión software

En la figura superior se puede observar un escenario en el que predomina el proyecto P1, del cual cuelgan variaciones del mismo, en este caso características nuevas. La figura refleja la necesidad de identificar de algún modo la versión del proyecto correspondiente a cada característica.

3.2.4 Alcance de los errores de código ocasionados en un proyecto

En desarrollos en los que intervienen varios proyectos se ha de tener mucho cuidado con la repercusión que tienen las modificaciones que se realizan a medida que avanza el tiempo. Todavía más cuando se encuentra un error de código. En éste caso no se sabe el alcance de ese error, es decir, a cuántas revisiones ha afectado. Por lo tanto, tampoco se sabe qué proyectos están influenciados.

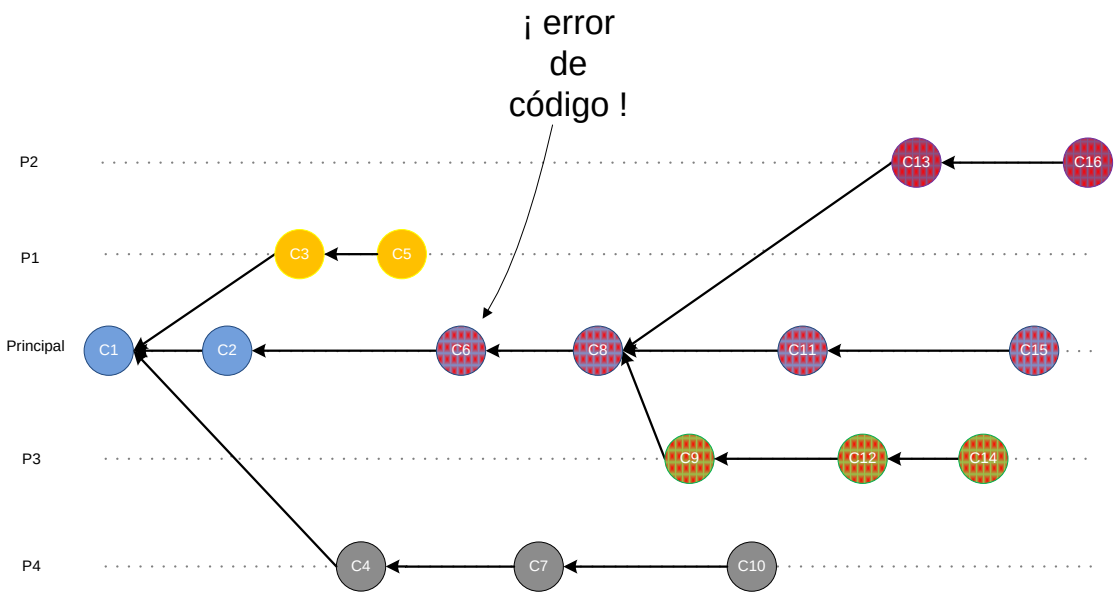


Figura 27 - Alcance de los bugs en un proyecto

En la figura superior se aprecia la prolongación de un error de código cometido en la revisión C6. El error está afectando a revisiones posteriores del mismo proyecto o diferentes proyectos.

3.3 Estrategia de ramificación

Después de analizar las características del proyecto y los escenarios definidos, se ha procedido a definir una metodología para dar una solución a la problemática. Se trata de una estrategia para facilitar las tareas a realizar por el usuario en los escenarios definidos en el apartado anterior.

3.3.1 Contexto

El sistema contendrá una rama principal de desarrollo de la que colgarán los subproyectos u otros desarrollos. De cada desarrollo podrán surgir otros desarrollos, o bien, características específicas del propio desarrollo. Así, sucesivamente.

El modelo descrito contiene diferentes tipos de ramas. Cada tipo se comportará de diferente manera. En la ilustración siguiente se puede ver un ejemplo.

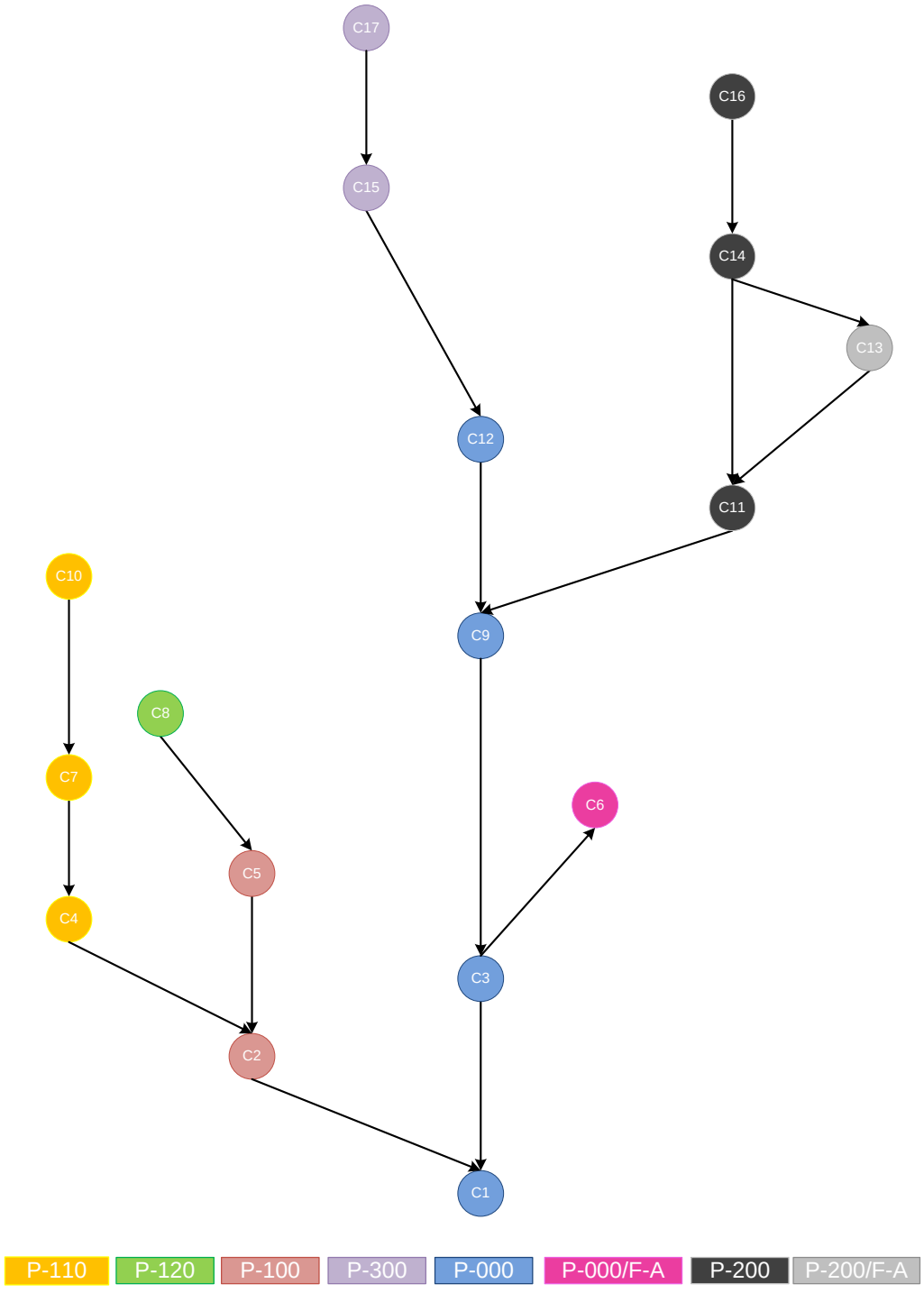


Figura 28 - Estrategia de ramificación

La figura anterior ilustra un escenario de ejemplo de aplicación de la estrategia. La rama principal corresponde al proyecto P-000. De ésta surgen los demás proyectos, en este caso P-100, P-200 y P-300. Del mismo modo también hay proyectos que surgen de otros proyectos: P-110 y P-120 de P-100.

Además, algunos de los proyectos descritos anteriormente serán la fuente de origen de ramas destinadas al desarrollo de características específicas, como por ejemplo: P-000/F-A de P-000 y P-200/F-A de P-200.

3.3.2 Diseño

La estrategia en cuestión es algo diferente a las habituales, es decir, no sigue las mismas pautas de ramificación que la mayoría de las estrategias propuestas por las organizaciones. La principal diferencia se basa concretamente en el procedimiento de generación de ramas: Lo habitual al crear una rama, es trabajar en ella para posteriormente fusionarla a la rama de origen y finalmente eliminar la rama cuando ya no se necesite. En este caso, el escenario va a tener ramas abiertas, que en muchos casos no lleguen a fusionarse nunca. Esto se debe a que los proyectos van a ser variaciones de otros proyectos, los cuales aportarán distintas funcionalidades en base a lo requerido por el cliente. El hecho de que estos desarrollos no converjan se debe a que ya no se pueden modificar (excepto para corregir bugs), porque ya están operativos.

Cuando se dé la necesidad de agregar nuevas funciones a los proyectos, se podrán dedicar ramas de características para ello.

3.3.3 Límites

Para aplicar la estrategia eficientemente necesariamente hay que especificar el alcance de ésta, definir qué acciones se pueden realizar y qué acciones no:

1. Tipos de ramas (branches)

   Existirán dos tipos de ramas: Las de desarrollo y las de características. La rama principal es una rama de desarrollo.

   <u>Operaciones</u>

   - Nueva rama de desarrollo:
     Inicialmente, sólo existirá una rama de desarrollo, que corresponderá a la principal. De ésta podrán surgir nuevos desarrollos. En este punto, se podrá

seguir creando desarrollos de la rama base o de los propios desarrollos ya creados. Por ejemplo, el P-110 a partir del P-100.

- Cerrar rama de desarrollo:
  En principio las ramas de desarrollo se mantendrán abiertas. No obstante, habrá casos en los que interese integrar un desarrollo completo o parte de él sobre otro. Para ello, se fusionará la rama en cuestión o se replicarán las confirmaciones determinadas. (Cherry-pick)

- Nueva rama de característica:
  Las ramas de características se crearán siempre a partir de un desarrollo y estarán orientadas al desarrollo de nuevas funcionalidad de los proyectos.

- Cerrar rama de característica:
  El usuario podrá fusionar la rama de característica con el proyecto padre. El proyecto adquirirá todos los cambios desarrollados en la rama de característica.

  Cerrar una rama de característica no implicará eliminarla. La etiqueta que apunta a la característica seguirá existiendo, permitiendo continuar desarrollando la rama y dando la opción de realizar cierres o fusiones posteriores. En cualquier caso, si el usuario lo desea podrá eliminar la etiqueta que apunta a la rama, para no seguir desarrollándola y darla por terminada.

Estándar de codificación

Se ve necesario establecer un estándar de codificación para guardar un orden en el repositorio y diferenciar unos desarrollos con otros. El estándar que se propone es el siguiente:

- Si se trata de una rama de desarrollo, contendrá “P-“ por delante del nombre del desarrollo para identificar la rama como un proyecto.
- Si se trata de una rama de característica (feature), contendrá primero el nombre del desarrollo padre (puede automatizarse), seguido de “F-“ por

delante del nombre de la característica y separando con un guión bajo el proyecto de la característica.

Ej. P-100_F-CSS

Los prefijos propuestos son a modo de ejemplo, el usuario puede elegir que prefijos se adaptan mejor.

2. Etiquetas (tags)

A la hora de crear un tag, se da libertad al usuario para definir qué nombre darle. Podría seguir las siguientes sugerencias:

- Anotar la versión del software. Ej. vX.X
- Junto con la anterior, indicaciones para marcar el estado del producto. Ej. Release 1.0, RC 2.0, Stable 5.0.
- Notas para remarcar algún commit en el que se introdujeron mejoras remarcables. Ej. GUI terminada.

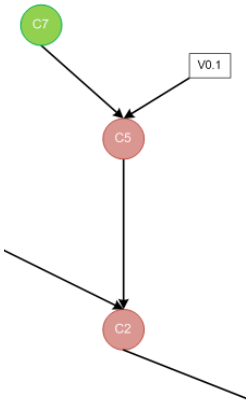


Figura 29 - Ejemplo de árbol con etiquetas

3. Confirmaciones (commits)

Los mensajes de confirmación quedan a gusto del usuario, pudiendo escribir lo que más desee.

No obstante, es recomendable utilizar mensajes explicativos, que describan de forma clara y concisa los cambios realizados. Como mejora de estos, eGit da la opción de dividir el mensaje en dos partes, resumen y detalles del mensaje. Primero se escribe el resumen, que debe ser breve (como una línea) y finalmente se escriben los detalles del mensaje. Debe haber dos saltos de línea entre ambos, un ejemplo de ello:

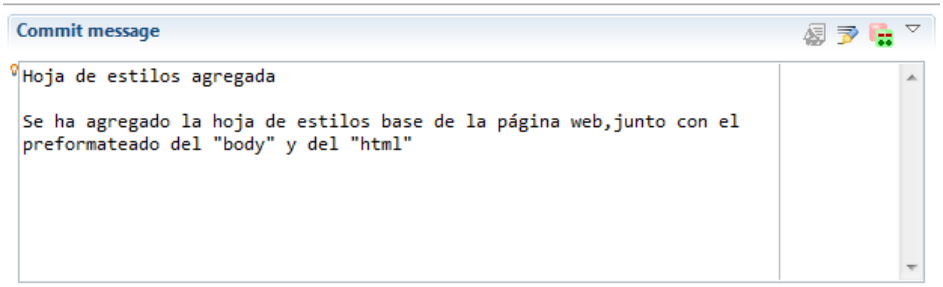


Figura 30 - Mensaje de confirmación en eGit

Para las fusiones, sería interesante definir un título adicional que identifique el punto exacto en el que se ha realizado una fusión. Por ejemplo: escribiendo como resumen del mensaje “Fusión: P-000/F-A sobre P000”. De esta manera al realizar una búsqueda por revisiones, filtrará todo mensaje y se centrará únicamente en las revisiones que comiencen de este modo.

3.3.4 Solución a los escenarios mediante la estrategia

En este apartado se describen los pasos a realizar para dar solución a los escenarios analizados en el punto anterior (3.2.3) mediante la estrategia definida.

- Escenario 1: Punto de partida para un nuevo desarrollo

Comenzar un nuevo proyecto a partir de otro para desarrollar una variante del original es un proceso que plantea dudas a la hora de definir el punto de partida. En dominios de trabajo en los que el número de proyectos en producción es elevado, definir la fuente de origen resulta complicado si no se conoce al detalle cada proyecto del dominio.

Por lo tanto, el sistema debe ser capaz de proporcionar información al usuario para facilitarle qué proyecto seleccionar como origen para uno nuevo. Mediante esta metodología es posible obtener la siguiente información:

- Las diferencias entre proyectos.
- La relación entre los proyectos del dominio.
- Información cronológica de todo el dominio.

Este proceso se solventa haciendo uso de las búsquedas, que gracias a los estándares de codificación definidos en la estrategia permitirán al usuario posicionarse en los nodos que desee para realizar un estudio. Un ejemplo del proceso se resume de la siguiente manera:

1. En primer lugar, hay que localizar y posicionarse en los proyectos a analizar. (ej. Rama A y rama B)

2. Después, mediante las herramientas de comparación se tendrá que visualizar qué punto es el más conveniente para comenzar un nuevo desarrollo, a decisión del usuario en base a las comparaciones que realice. (ej. A)
3. Una vez elegido se creará un nuevo desarrollo a partir del desarrollo escogido. (ej. Se crea C a partir de A)
4. Finalmente, si interesa integrar parte de los cambios generados en B, se replicarán las revisiones sobre el nuevo desarrollo. (ej. Replicar revisiones de B sobre C)

- Escenario 2: Fusionar ramas lejanas o no conectadas directamente

Durante el desarrollo de un proyecto, se opta por añadir funcionalidad desarrollada en proyectos diferentes, debido a que estos ya implementan la funcionalidad necesitada y replicándola se ahorra tiempo y esfuerzo.

La solución para este escenario es prácticamente trivial. Básicamente hay que buscar la rama a integrar para poder hacer la fusión. Una vez más, gracias al estándar de codificación escogido para las ramas el proceso de búsqueda será sencillo. Un ejemplo de ello:

1. Buscar la rama a integrar. (ej. Rama B)
2. Posicionarse en la rama que integrará la rama del punto anterior. (ej. Rama A)
3. Fusionar la rama localizada en el punto 1. (ej. Rama B sobre A)
4. Repetir el proceso con tantas ramas como se quiera. (ej. Ramas C, D, E,… sobre A)

- Escenario 3: Control de versión software de múltiples proyectos

En muchas ocasiones los desarrolladores de software tienen la necesidad de marcar nodos en concreto del árbol de revisiones por la importancia de los mismos, ya sea por la finalización de una nueva versión del programa, el desarrollo de cierta característica o simplemente porque interesa remarcar algo en concreto.

Para este escenario la solución es simple. Se pueden remarcar las revisiones del árbol usando las etiquetas (tags). Un ejemplo de ello:

1. Localizar el nodo de la revisión a etiquetar.

2. Etiquetar la revisión.

- Escenario 4: Alcance de los errores de código ocasionados en un desarrollo

Conocer el alcance de los errores de código de un proyecto es de gran importancia. Cometer errores en el código y dejar que se prolonguen en el tiempo, puede causar que el software deje de funcionar o que funcione incorrectamente. Más aun cuando se trabaja con VCS. Determinar dónde se produjo el error puede ser crucial para limpiar todo el desarrollo posterior y las revisiones afectadas.

Por ello la mejor manera de solucionar este problema es buscar la revisión donde se produjo el error de código y poder así revertirlo o modificarlo en las revisiones posteriores.

Este procedimiento se soluciona ejecutando una serie de comandos (git-bisect) que buscan en el repositorio hasta encontrar el origen del error. A continuación se muestra un ejemplo de uso:

1. El usuario proporciona como entrada el fragmento de código erróneo a buscar.
2. Después tiene que especificar el rango de búsqueda, es decir entre que revisiones realizar la búsqueda. Útil para focalizar la búsqueda en cierta parte del historial si se tiene seguro que estará en ese rango de revisiones. No obstante, siempre podrá realizar una búsqueda completa sobre todo el repositorio, aunque no será tan rápido para repositorios muy grandes.
3. El sistema devolverá el código identificador de la confirmación donde se introdujo por primera vez el error.

Después de localizar la fuente de origen de la revisión que contiene el error, el usuario podrá,

- Corregir el error y replicar la corrección sobre los proyectos afectados (utilizando cherry- pick por ejemplo).
- O también, se podrán deshacer los cambios realizados por la revisión en la que se introdujo el error y replicarlos en las ramas afectadas.

3.4 Validación de la estrategia

Después de estudiar la problemática del proyecto y diseñar una propuesta de solución, se decidió mostrar el resultado la empresa que estamos usando como referente (ULMA Packaging S.Coop) para contrastar lo realizado con su perspectiva.

La impresión de ULMA fue positiva, la estrategia le pareció factible y útil en líneas generales.

ULMA planteó ciertas inquietudes acerca del comportamiento de la estrategia frente a escenarios de gran carga (gestionar multitud de proyectos). Se intereso exactamente por los siguientes términos:

- Visualización: La posibilidad de visualizar el árbol histórico de revisiones a lo ancho de la pantalla.
- Rendimiento: La repuesta a las operaciones del usuario funciona con agilidad.
- Carga: La eficiencia con que el sistema gestiona el repositorio con muchos proyectos.

Los aspectos anteriores necesitaban ser probados para conocer el resultado. Por ello, se incluyo un apartado de pruebas en el que se realizaron los diferentes test sobre la herramienta que implementó la estrategia. Así, se comprobó el comportamiento de la estrategia.

4 IMPLEMENTACIÓN

Este bloque describe el desarrollo completo de la herramienta creada, abarcando las fases de requisitos, el análisis y diseño, el desarrollo y finalmente un apartado para las pruebas y validación.

4.1 Descripción de la herramienta

Con el objetivo de llevar a la práctica la estrategia propuesta en este proyecto se ha desarrollado una herramienta que la implemente. La herramienta ha sido desarrollada con el objetivo de automatizar y facilitar el uso de las diferentes funcionalidades definidas en la estrategia. De esta forma se consigue limitar las acciones de la herramienta al usuario y guiarle así en el proceso. Además, permite que el usuario utilice funciones de Git sin conocer los comandos al detalle.

Para materializar la estrategia se ha decidido extender el plugin eGit de Eclipse para añadir nuevas funcionalidades a las que ya dispone, de tal modo que queden cubiertas al completo las diferentes operaciones de la estrategia descrita anteriormente. Además, la herramienta es muy similar en cuanto al modo de funcionamiento a gitFlow. En realidad, es una especie de gitFlow, pero que ejecuta las acciones correspondientes a la estrategia que se ha definido.

La herramienta se denomina proFlow.

4.2 Requisitos

A continuación se muestran los requisitos funcionales y no funcionales que la herramienta debe soportar.

4.2.1 Requisitos funcionales

RF1 El usuario puede crear un nuevo repositorio o usar uno ya existente para comenzar a utilizar las ayudas de la herramienta.

RF2 El sistema debe ser capaz de reconocer si ya está creado el repositorio.

RF3 El usuario puede actualizar el contenido del directorio actual a la revisión/rama/etiqueta que desee.

RF4 El sistema da la opción de configurar a gusto del usuario el estándar de codificación de las ramas.

RF5 El sistema debe permitir dar una visión cronológica del árbol de revisiones del repositorio.

RF6 El usuario debe tener la opción de crear una rama de desarrollo nueva para un nuevo proyecto tomando la rama de desarrollo de otro proyecto como fuente de origen.

RF7 El sistema debe permitir fusionar el progreso de un proyecto sobre otro cuando el usuario lo crea conveniente.

RF8 El sistema debe dar la oportunidad de crear ramas de características para los proyectos.

RF9 Cuando el usuario desee debe poder fusionar cada rama de características sobre su rama padre.

RF10 El usuario debe poder eliminar cada rama de características que hayan llegado a su fin.

RF11 El sistema debe permitir identificar cualquier punto del árbol de revisiones mediante las etiquetas.

RF12 El sistema debe permitir eliminar cualquier etiqueta en el repositorio.

RF13 El usuario debe permitir integrar cambios de una determinada revisión sobre otra cualquiera. (cherry- pick)

RF14 Durante el desarrollo, el usuario debe poder almacenar los cambios que haya realizado hasta el momento. (commit)

RF15 Frente a errores en el código, siempre que el usuario indique cual es el error a encontrar, el sistema debe proporcionar la revisión donde se introdujo por primera vez el error.

RF16 Cuando se de la necesidad de modificar los cambios realizados en un cierta revisión, el sistema debe proporcionar la manera de que el usuario invierta los cambios o los modifique.

RF17 El sistema debe facilitar la búsqueda de revisiones, ramas, etiquetas, etc. en el repositorio.

4.2.2 Requisitos no funcionales

Tipo	Requisito no funcional
Usabilidad	**RNF1** La interfaz de usuario debe tener un diseño amigable e intuitivo, de modo que el usuario no demoré más de 30 segundos en realizar la acción que necesite.
Carga	**RNF2** El repositorio debe soportar una cantidad de 1000 ramas.
Disponibilidad	**RNF3** El repositorio debe ser accesible en todo momento.
Concurrencia	**RNF4** El sistema de control de versiones debe permitir que dos o más usuarios trabajen sobre algún elemento del repositorio simultáneamente.

Tabla 5 - Requisitos no funcionales

4.3 Análisis

A continuación se describen detalladamente los casos de uso que debe cumplimentar la herramienta para cubrir todas las posibilidades de la estrategia planteada. No obstante no todos han sido implementados, ya que algunos estaban resueltos por el plugin eGit.

4.3.1 Casos de uso

CU01	Seleccionar repositorio	
Descripción	Para comenzar a utilizar la herramienta, el primer paso consiste en disponer de un repositorio donde trabajar. Para ello se le da al usuario la oportunidad de crear uno nuevo. Si el repositorio ya existe se utiliza automáticamente.	
Precondición	Ninguna	
Flujo normal	**Paso**	**Acción**
	1	El sistema comprueba si existe un repositorio ligado al proyecto.
	2	Si no existe, el usuario solicita crear uno.
	3	El sistema pide la ruta donde crear el repositorio.
	4	El usuario inserta la ruta donde generar el repositorio y le hace saber al sistema que debe generar el repositorio.

	5	El sistema configura el directorio elegido para que pueda ser utilizado y le indica al usuario que el repositorio ya está creado y listo para usar.
Flujo alternativo	2	Si el repositorio ya está creado, el sistema lo reconoce automáticamente permitiendo al usuario continuar con su trabajo.
Postcondición	El repositorio está creado y listo para usar.	

Tabla 6 - Caso de uso: Seleccionar repositorio

CU02	**Establecer el estándar de codificación**	
Descripción	El usuario será el que decida que estándar de codificación utilizar, dejando personalizar su configuración o utilizando la configuración por defecto.	
Precondición	El repositorio debe haberse creado.	
Flujo normal	**Paso**	**Acción**
	1	El sistema pide al usuario al inicializar la herramienta que estándar de codificación utilizar durante todo el proceso.
	2	El usuario puede elegir entre un estándar personalizado o el de por defecto.
	3A	Si elige el personalizado, debe rellenar los campos de texto que el sistema le muestra para configurarlo a su conveniencia.
	3B	Si no, el sistema muestra el estándar de codificación que se va a utilizar por defecto.
	4	El usuario solicita al sistema que fije el estándar de codificación.
	5	El sistema guarda la configuración escogida.
Postcondición	El sistema deja configurado el estándar de codificación que se va a utilizar.	

Tabla 7 - Caso de uso: Establecer el estándar de codificación

CU03	Actualizar el directorio actual	
Descripción	El directorio de trabajo reconstruye su contenido en base a la rama, revisión o etiqueta que se le indique, reorganizando los ficheros y directorios a esa versión.	
Precondición	El repositorio debe haber sido inicializado. Debe existir más de una revisión en el repositorio. (para poder cambiar entre ellas)	
Flujo normal	**Paso**	**Acción**
	1	El usuario debe cambiar de revisión en el historial de revisiones.
	2	El sistema regenera el directorio de trabajo, adaptando el contenido a la revisión elegida por el usuario.
Postcondición	El directorio es actualizado a la revisión seleccionada.	

Tabla 8 - Caso de uso: Actualizar el directorio de trabajo

CU04	Visualizar el árbol histórico de revisiones	
Descripción	Para dar una visión gráfica del repositorio el usuario puede visualizar el histórico de revisiones a modo de árbol cronológico.	
Precondición	El repositorio debe haber sido inicializado. Debe existir un commit por lo menos.	
Flujo normal	**Paso**	**Acción**
	1	El usuario pide al sistema que muestre el historial de revisiones en forma de árbol.
	2	El sistema muestra el historial centrándose en la última parte del repositorio, pero dejando la opción de navegar a través del árbol.
Postcondición	Muestra un panel con el histórico de revisiones ordenados por fecha.	

Tabla 9 - Caso de uso: Visualizar el árbol histórico de revisiones

CU05	Crear nuevo proyecto
Descripción	En cualquier instante el usuario puede comenzar un nuevo proyecto a partir de otro para aprovechar todo el código desarrollado en ese proyecto de origen.

Precondición	El repositorio debe haber sido inicializado. Debe existir un proyecto. (por defecto: master)	
Flujo normal	**Paso**	**Acción**
	1	El usuario cambia a la rama de la que quiere generar un nuevo proyecto.
	2	Una vez posicionado debe solicitar al sistema la creación de un nuevo proyecto.
	3	El sistema solicita al usuario el nombre del nuevo proyecto.
	4	El usuario proporciona el nombre del proyecto y solicita su creación.
	5	El sistema crea una nueva rama a partir del proyecto actual y la identifica con el nombre generado (prefijo base + nombre). Además, si el usuario lo desea el directorio de trabajo puede quedar posicionado en la nueva rama.
Postcondición	El repositorio cuenta con una nueva rama de desarrollo para el proyecto.	

Tabla 10 - Caso de uso: Crear nuevo proyecto

CU06	**Fusionar proyecto**	
Descripción	Cuando un proyecto especifico está listo para su integración (no tiene por qué estar terminado), el sistema permite realizar una fusión del proyecto vigente sobre otro que interese.	
Precondición	El repositorio debe haber sido inicializado. Al menos debe haber dos ramas de proyectos.	
Flujo normal	**Paso**	**Acción**
	1	El usuario debe posicionar el directorio de trabajo en la rama del proyecto a recibir la integración.
	2	Después indica al sistema que proyecto desea fusionar con el actual.
	3	El sistema realiza la fusión.
Postcondición	El proyecto seleccionado hereda los cambios del otro, siempre y cuando no existan conflictos entre ambos.	

Tabla 11 - Caso de uso: Fusionar proyecto

CU07	Crear nueva característica	
Descripción	El sistema deja crear nuevas líneas de desarrollo sobre un proyecto a modo de rama para desarrollar funcionalidades o características nuevas.	
Precondición	El repositorio debe haber sido inicializado.	
Flujo normal	**Paso**	**Acción**
	1	El usuario se posiciona en la rama de proyecto del que abrir una nueva rama e indica al sistema que desea abrir una línea de desarrollo.
	2	El sistema le pide que introduzca un nombre para la nueva rama.
	3	El usuario le proporciona el nombre de la nueva rama de característica.
	4	El sistema crea la nueva rama y actualiza el directorio de trabajo a ese estado, si así lo desea.
Postcondición	Una rama de característica creada a partir de un proyecto.	

Tabla 12 - Caso de uso: Crear nueva característica

CU08	Fusionar rama de característica	
Descripción	Integra los cambios generados en una rama de característica sobre el proyecto padre o de origen.	
Precondición	El repositorio debe haber sido inicializado. Debe existir una rama de característica por lo menos.	
Flujo normal	**Paso**	**Acción**
	1	El usuario se sitúa en la rama de la característica desarrollada.
	2	Después, solicita al sistema la integración de rama de característica desarrollada sobre el proyecto padre.
	3	El sistema integra los cambios de la rama de característica sobre el proyecto.
Postcondición	El proyecto escogido contiene los cambios realizados en la rama de característica además de los suyos.	

Tabla 13 - Casos de uso: Fusionar característica

CU09	Eliminar rama	
Descripción	En un momento dado el usuario tiene la necesidad de borrar ramas que no le interese seguir desarrollando.	
Precondición	El repositorio debe haber sido inicializado. Debe haber ramas creadas.	
Flujo normal	**Paso**	**Acción**
	1	El usuario debe actualizar el directorio actual en cualquier punto del repositorio siempre y cuando no sea la rama a borrar.
	2	Acto seguido debe seleccionar la rama a borrar y solicitar su eliminación.
	3	El sistema elimina la rama del repositorio.
Postcondición	La rama seleccionada queda eliminada.	

Tabla 14 - Caso de uso: Eliminar rama

CU10	Etiquetar revisiones	
Descripción	Mediante etiquetas el usuario puede marcar las revisiones que vea oportunas para mantenerlas identificadas.	
Precondición	El repositorio debe haber sido inicializado. Debe contener revisiones.	
Flujo normal	**Paso**	**Acción**
	1	El usuario selecciona la revisión a etiquetar y solicita marcarla con una etiqueta.
	2	El sistema pide un nombre para la etiqueta.
	3	El usuario le facilita el nombre.
	4	La revisión queda marcada con esa etiqueta.
Postcondición	La etiqueta seleccionada queda identificada mediante la etiqueta asignada.	

Tabla 15 - Caso de uso: Etiquetar revisiones

CU11	Eliminar etiquetas	
Descripción	Eliminar la etiqueta seleccionada del repositorio.	
Precondición	El repositorio debe haber sido inicializado. Debe contener etiquetas creadas.	
Flujo normal	**Paso**	**Acción**
	1	El usuario selecciona la etiqueta a borrar del repositorio y solicita borrarla del sistema.
	2	El sistema la borra del repositorio.
Postcondición	La etiqueta elegida queda borrada del repositorio.	

Tabla 16 - Caso de uso: Eliminar etiquetas

CU12	Replicar los cambios de una o varias revisiones	
Descripción	Replicar los cambios de una o varias revisiones de una rama a otra diferente.	
Precondición	El repositorio debe haber sido inicializado. Deben existir dos ramas por lo menos y con al menos una revisión en una de ellas.	
Flujo normal	**Paso**	**Acción**
	1	El usuario actualiza el directorio actual en la rama que quiere integrar nuevos cambios.
	2	Seguidamente elige las revisiones de otra rama de las que quiere replicar sus cambios. Solicita al sistema que integre las revisiones.
	3	El sistema copia los cambios realizados en las revisiones seleccionadas sobre la rama en la que el directorio actual está posicionado.
Postcondición	La rama elegida contiene los cambios de las revisiones seleccionadas por el usuario.	

Tabla 17 - Caso de uso: replicar los cambios de una o varias revisiones

CU13	Confirmar los cambios realizados	
Descripción	Cuando se realizan modificaciones en el código el sistema puede almacenar esos cambios en el repositorio.	
Precondición	El repositorio debe haber sido inicializado. Debe generar modificaciones en el directorio de trabajo.	
Flujo normal	**Paso**	**Acción**
	1	El usuario hace saber al sistema que quiere confirmar los cambios recientes en el código.
	2	El sistema crea una revisión nueva en el repositorio, guardando el contenido del directorio actual en ese momento.
Postcondición	El repositorio contiene una revisión nueva correspondiente a los cambios realizados en el directorio.	

Tabla 18 - Caso de uso: Confirmar los cambios realizados

CU14	Buscar errores de código	
Descripción	El sistema busca en el repositorio la fuente de origen de los errores de código que los desarrolladores hayan introducido durante el desarrollo. Para ello hace uso de búsquedas binarias.	
Precondición	El repositorio debe haber sido inicializado. Debe contener una rama por lo menos.	
Flujo normal	**Paso**	**Acción**
	1	El usuario solicita realizar una búsqueda de errores de código en el repositorio.
	2	El sistema le pide al usuario el rango de revisiones a revisar en el repositorio y el código a buscar
	4	El usuario indica el rango de revisiones donde buscar y el código a buscar.
	5	El sistema realiza el proceso de búsqueda desechando toda revisión que no sea la fuente de origen donde se produjo por primera vez el error.
	6	Una vez encuentre la revisión de la cual origina el error, proporciona el código de la revisión al usuario.

Flujo alternativo	Paso	Acción
	5	Si no encuentra ningún error, significan que no se existe tal error. Con lo que advierte de ello al usuario.
Postcondición		

Tabla 19 - Caso de uso: Buscar errores en el código

CU15	Invertir los cambios realizados en una revisión	
Descripción	Deshace los cambios realizados en una revisión determinada.	
Precondición	El repositorio debe haber sido inicializado. Debe contener al menos una revisión.	
Flujo normal	**Paso**	**Acción**
	1	El usuario selecciona la revisión a invertir y pide al sistema que invierta los cambios.
	2	El sistema crea una revisión al final de la rama en la que se encuentra posicionado, invirtiendo todos los cambios realizados en la revisión escogida por el usuario.
Postcondición	Los cambios realizados en una revisión anterior quedan invertidos por una posterior.	

Tabla 20 - Caso de uso: Invertir los cambios realizados en una revisión

CU16	Búsqueda en el repositorio	
Descripción	Busca todo tipo de información en el repositorio: ramas, comentarios, revisiones, etc.	
Precondición	Debe existir un repositorio.	
Flujo normal	**Paso**	**Acción**
	1	El usuario selecciona el tipo de búsqueda que desea realizar. (Id de revisión, autor, rama, etc.) e introduce el texto.
	2	El sistema filtra el contenido del repositorio y muestra la información encontrada.
Postcondición		

Tabla 21 - Caso de uso: Búsqueda en el repositorio

4.4 Diseño

Este apartado presenta el diseño de la aplicación reflejado en diversos diagramas: diagrama de contexto, diagramas de clases y diagramas de secuencia. Únicamente se muestran los diagramas correspondientes a las nuevas funcionalidades desarrolladas en el proyecto. No se ve necesario mostrar los diagramas de las funcionalidades propias de eGit.

4.4.1 Diagrama de contexto

Antes de entrar a fondo en cómo está desarrollada la aplicación conviene conocer cómo está compuesta en sus diferentes capas software. La herramienta desarrollada utiliza diferentes componentes software o plugins del IDE eclipse. Para comenzar, la herramienta se sitúa dentro de eGit, es decir, es una extensión del plugin eGit. Debido a que necesita gran parte de las funcionalidades de eGit no tiene sentido colocarla fuera o tomarla como un plugin independiente. En la figura de a continuación se observa visualmente como está compuesta.

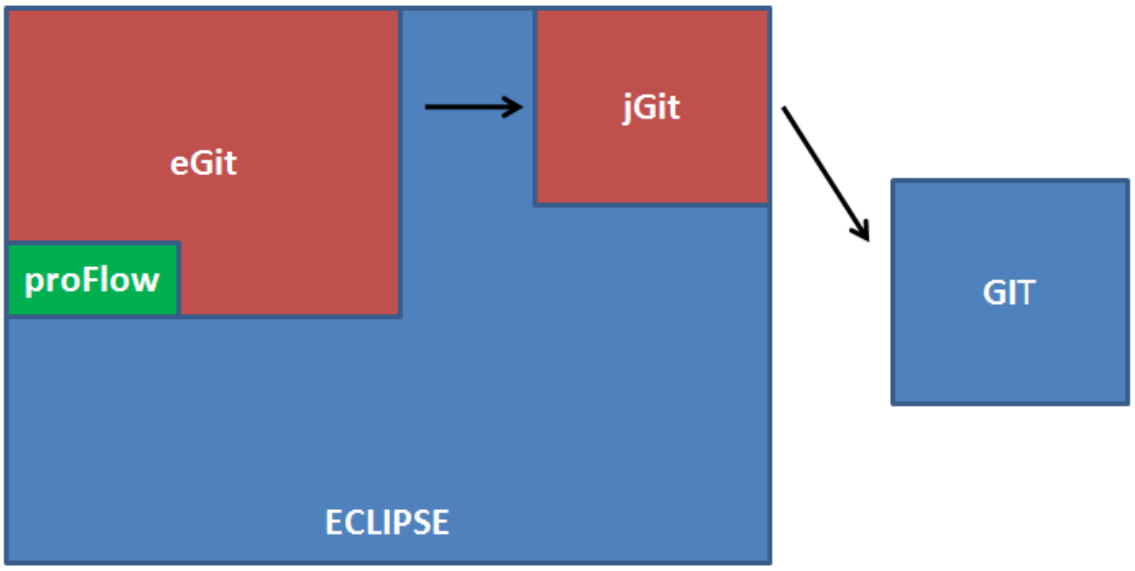


Figura 31 - Capas software del desarrollo

Atendiendo a la figura, se puede observar que la estructura es simple. Por un lado se encuentra proFlow, la herramienta desarrollada en el proyecto, que junto con eGit son utilizadas para implantar la estrategia al completo. Por otro lado está jGit, una librería de Git para Java, que hace de intermediario a modo de API entre eGit y Git. De esta forma

abstraemos al desarrollador de conocer los comandos de Git. Únicamente con entender y saber utilizar jGit es suficiente para hacer uso de gran parte de los comandos Git.

4.4.2 Diagramas de clases

Continuando con el diseño de la aplicación, en este apartado se muestra los diagramas de clases para comprender como se ha llevado a cabo la integración de la herramienta proFlow con eGit. Cada diagrama de clases corresponde a cada una de las funcionalidades desarrolladas en este proyecto. Debido a la similitud entre los diagramas y como peculiaridad en este proyecto, se ha definido una plantilla o patrón del que todas las funcionalidades heredan. Para entender los diagramas basta con fusionar la parte variable del patrón con el bloque de una funcionalidad concreta. A continuación se muestran cada uno de los diagramas de clases:

Patrón

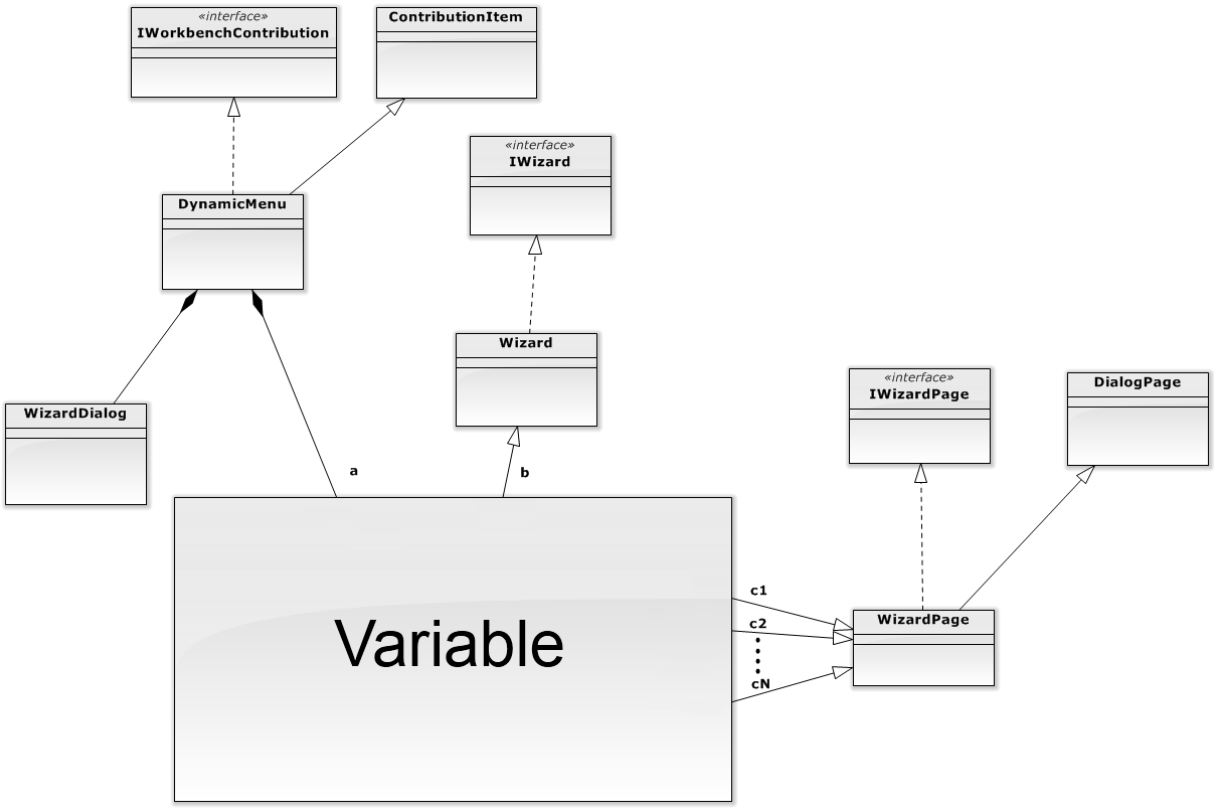


Figura 32 - Diagrama de clases: Patrón

Como se puede observar en la figura anterior, todas las funcionalidades están compuestas por varias clases en común. A grandes rasgos, cada funcionalidad contiene un Wizard el cual está compuesto a su vez por varias páginas WizardPage. Un Wizard es un asistente que provee páginas para la creación y configuración de elementos java, y los WizardPages son las páginas que constituyen los Wizards.

Además, siguiendo la figura, cada funcionalidad debe formar parte del menú de opciones, DynamicMenu.

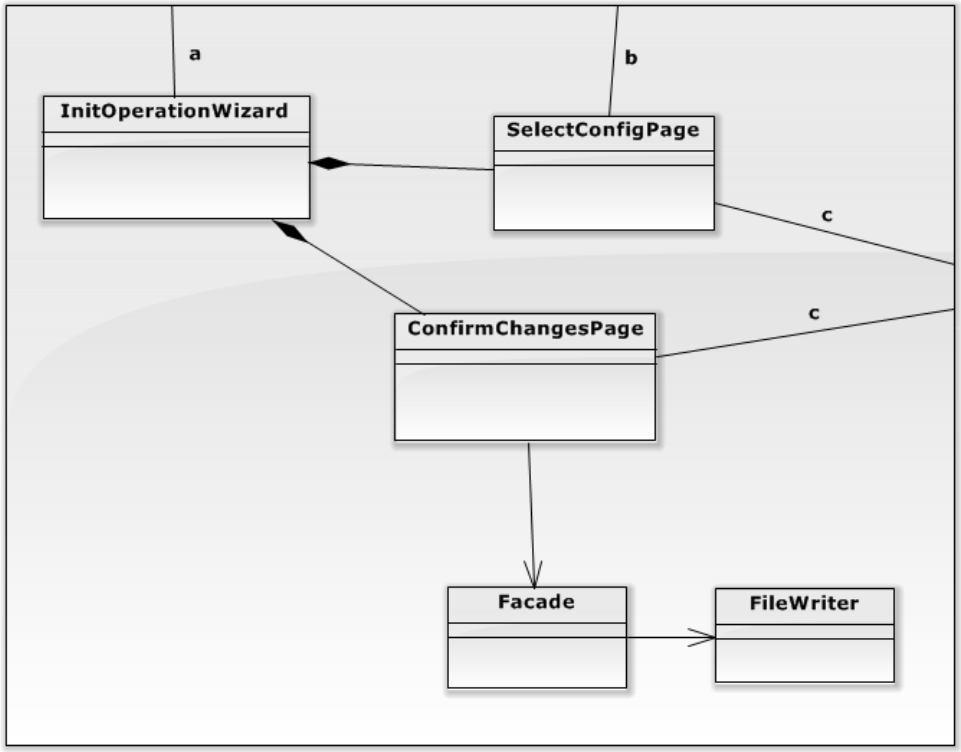


Figura 33 - Diagrama de clases: Inicializar proFlow

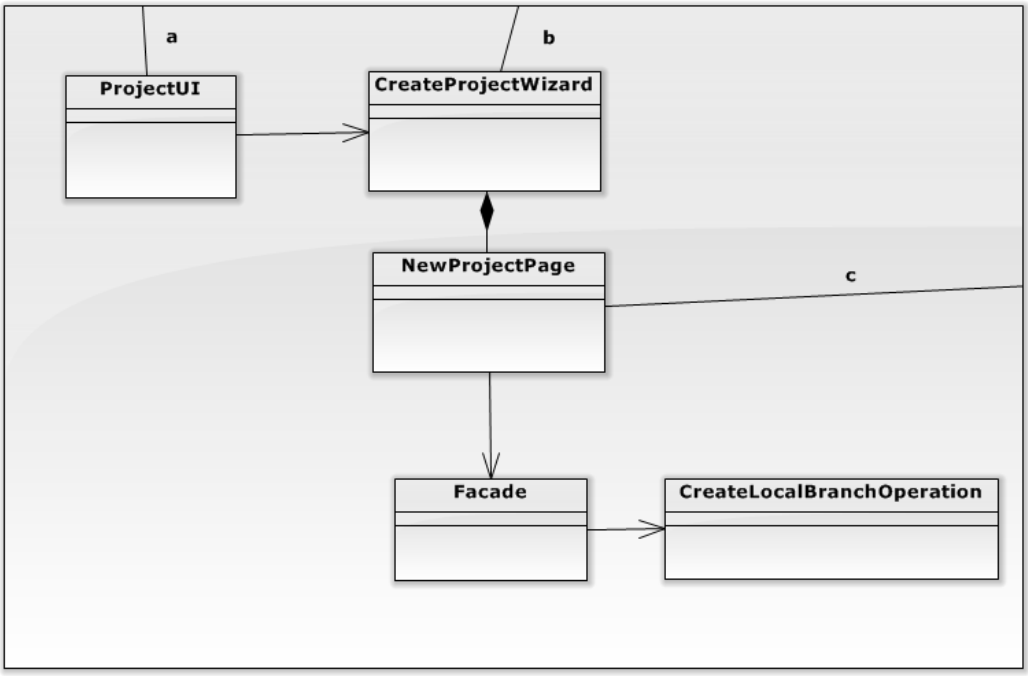


Figura 34 - Diagrama de clases: Crear proyecto

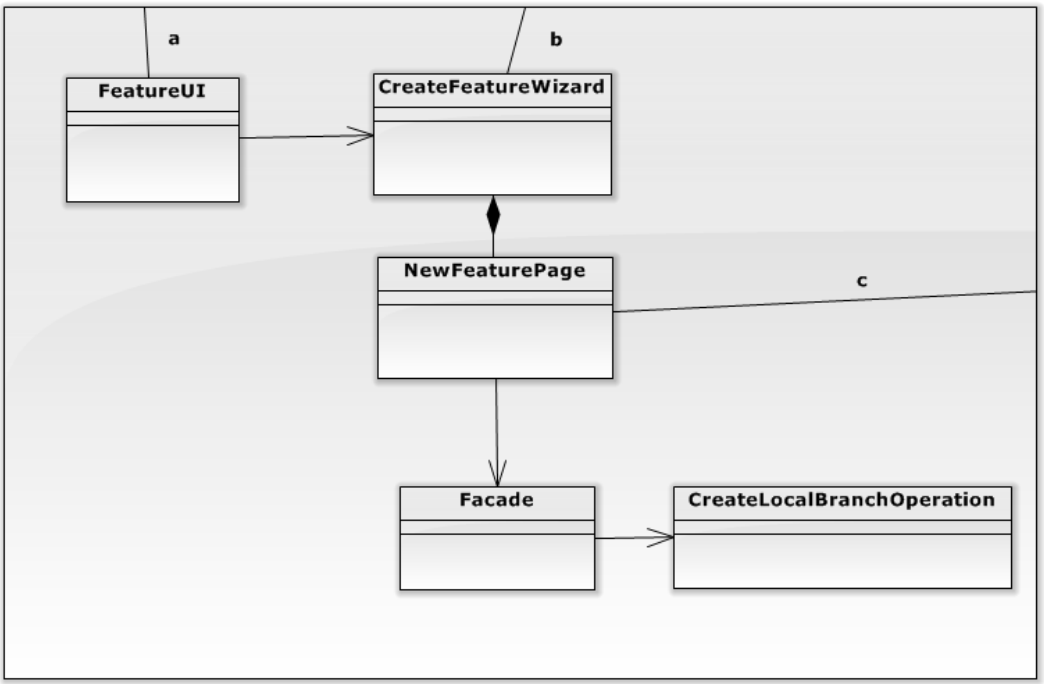


Figura 35 - Diagrama de clases: Crear característica

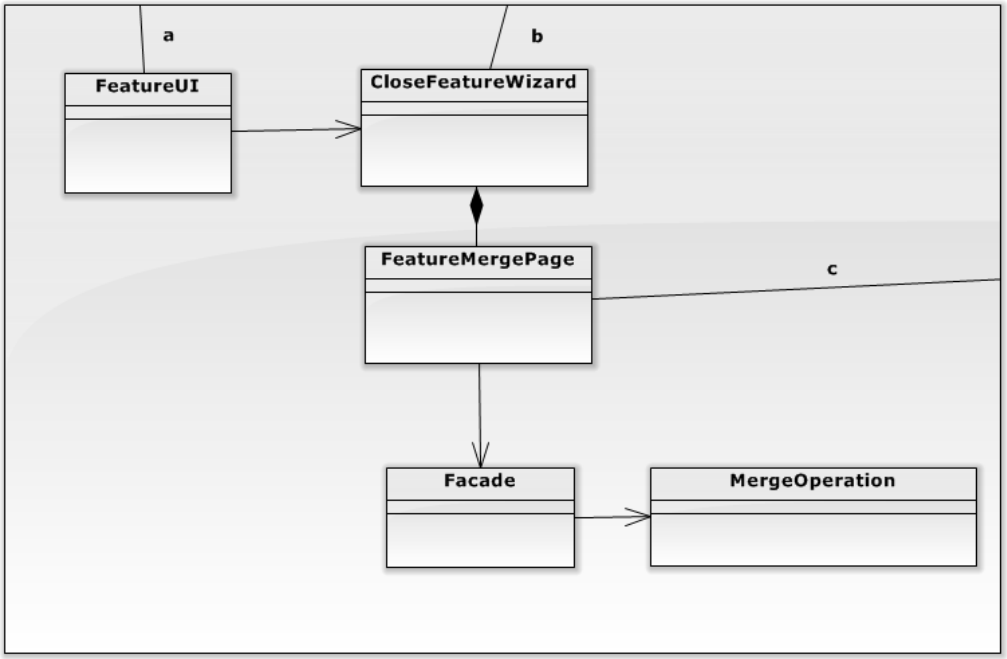


Figura 36 - Diagrama de clases: Cerrar característica

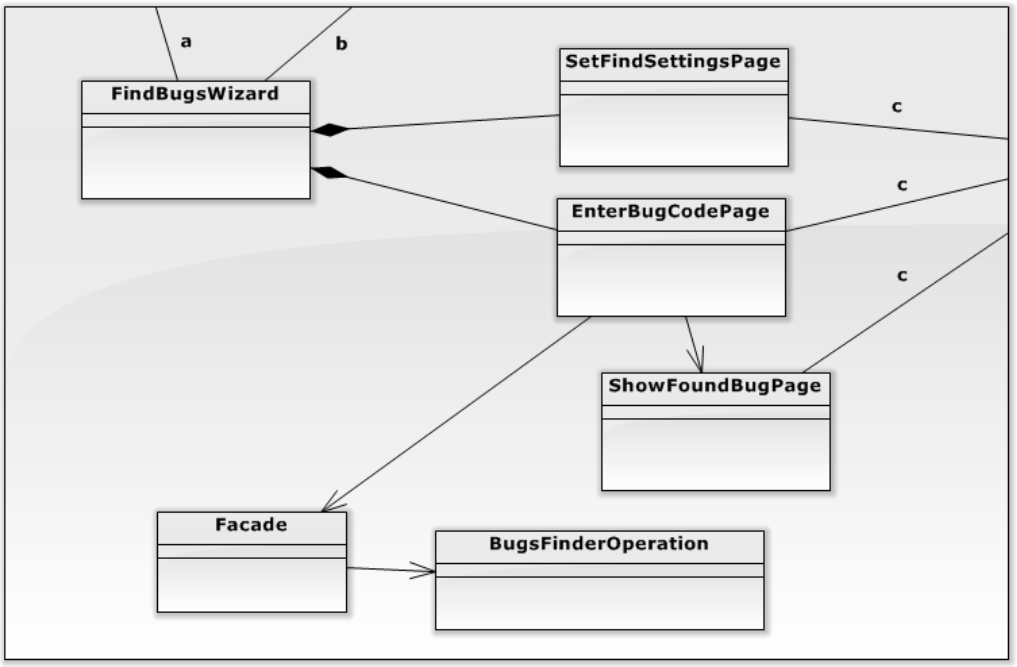


Figura 37 - Diagrama de clases: Buscar errores en el código

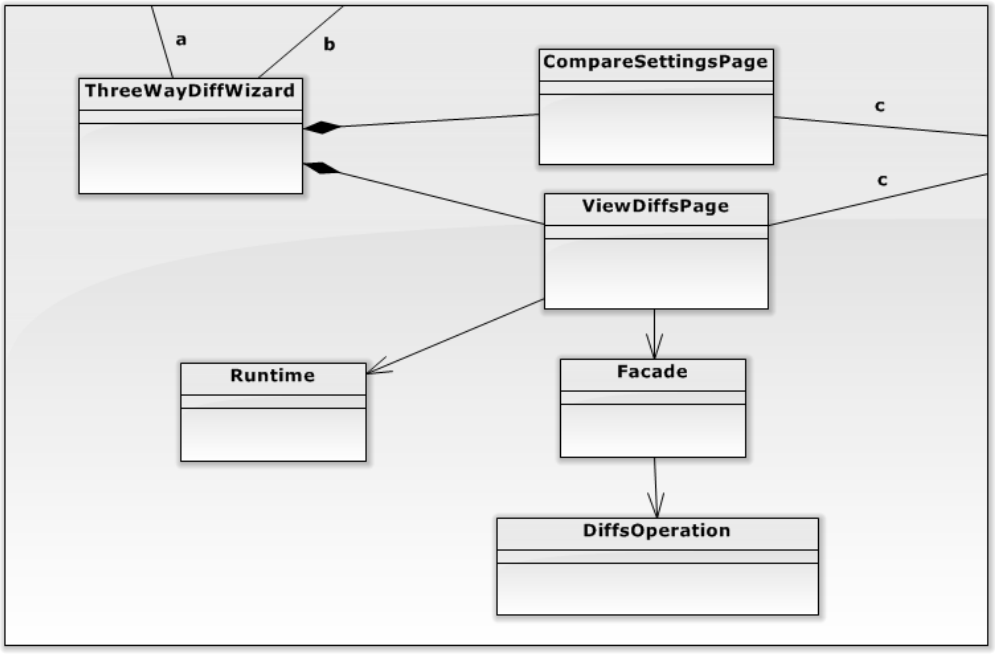


Figura 38 - Diagrama de clases: Ver diferencias a tres vías

4.4.3 Diagramas de secuencia

En esta sección se muestran los diagramas de secuencia pertenecientes a las funcionalidades implementadas en la herramienta proFlow. Los diagramas se han dibujado desde un punto de vista general sin entrar en el detalle para facilitar la compresión de las funcionalidades.

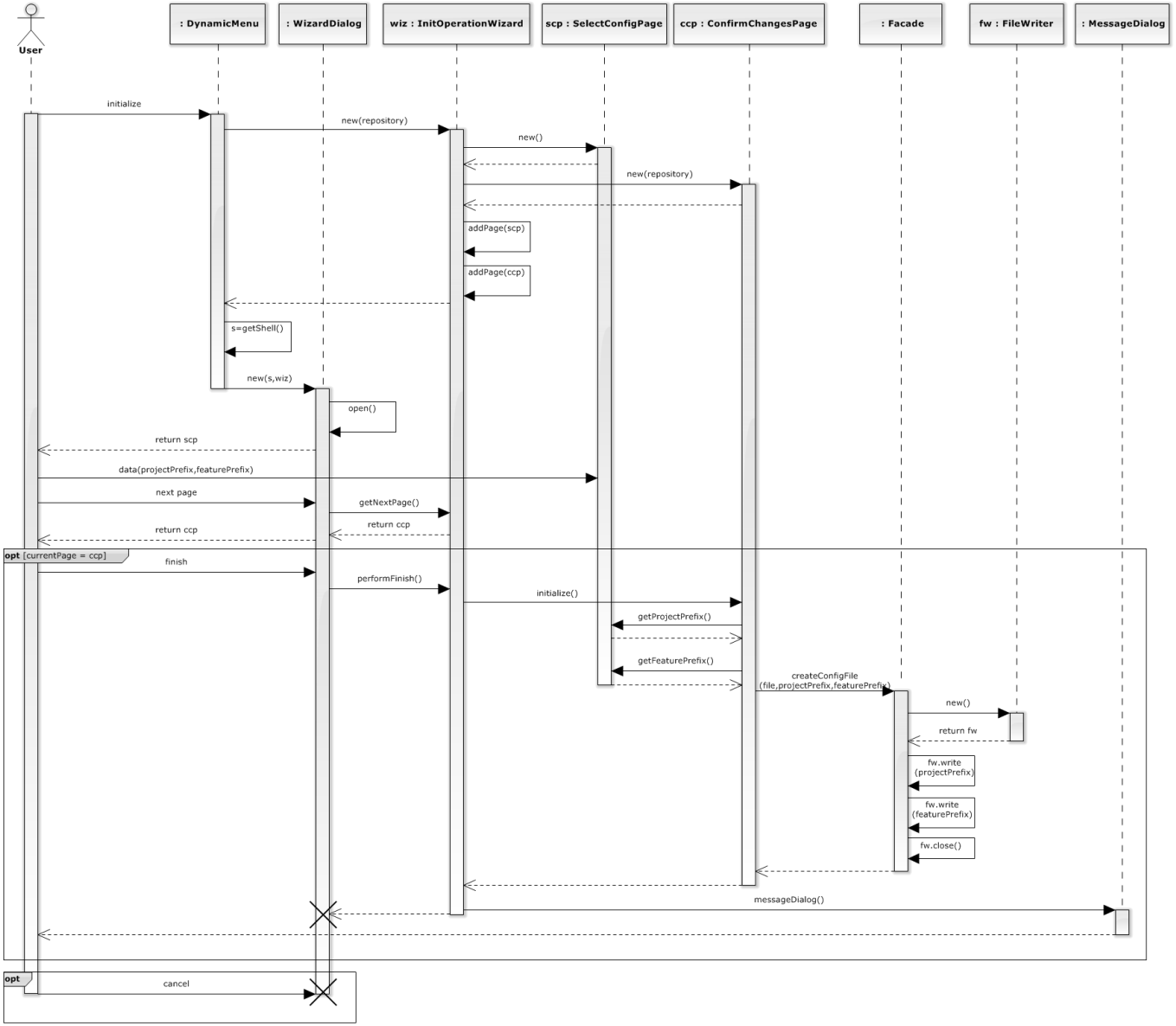


Figura 39 - Diagrama de secuencia: Inicializar proFlow

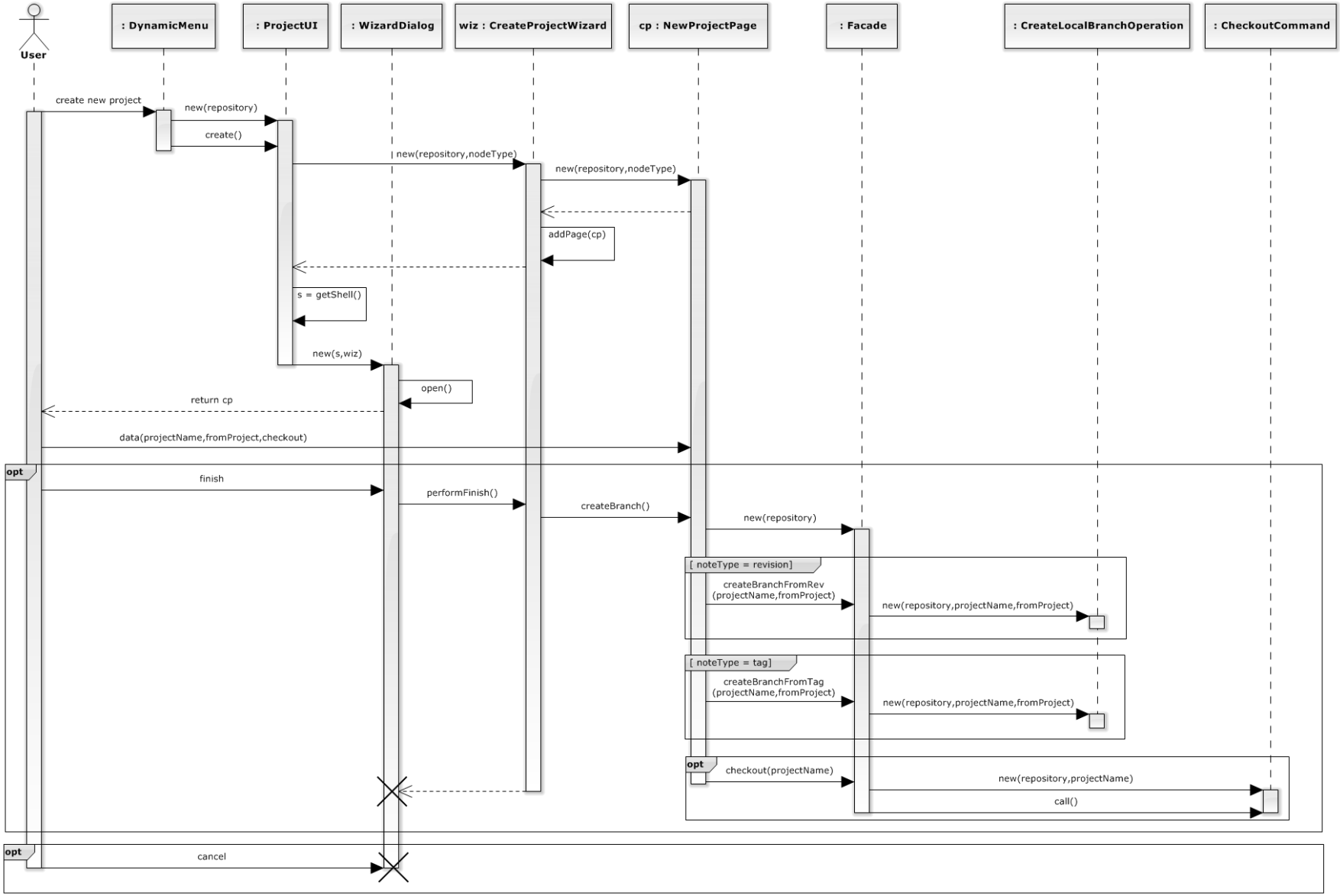


Figura 40 - Diagrama de secuencia: Crear proyecto

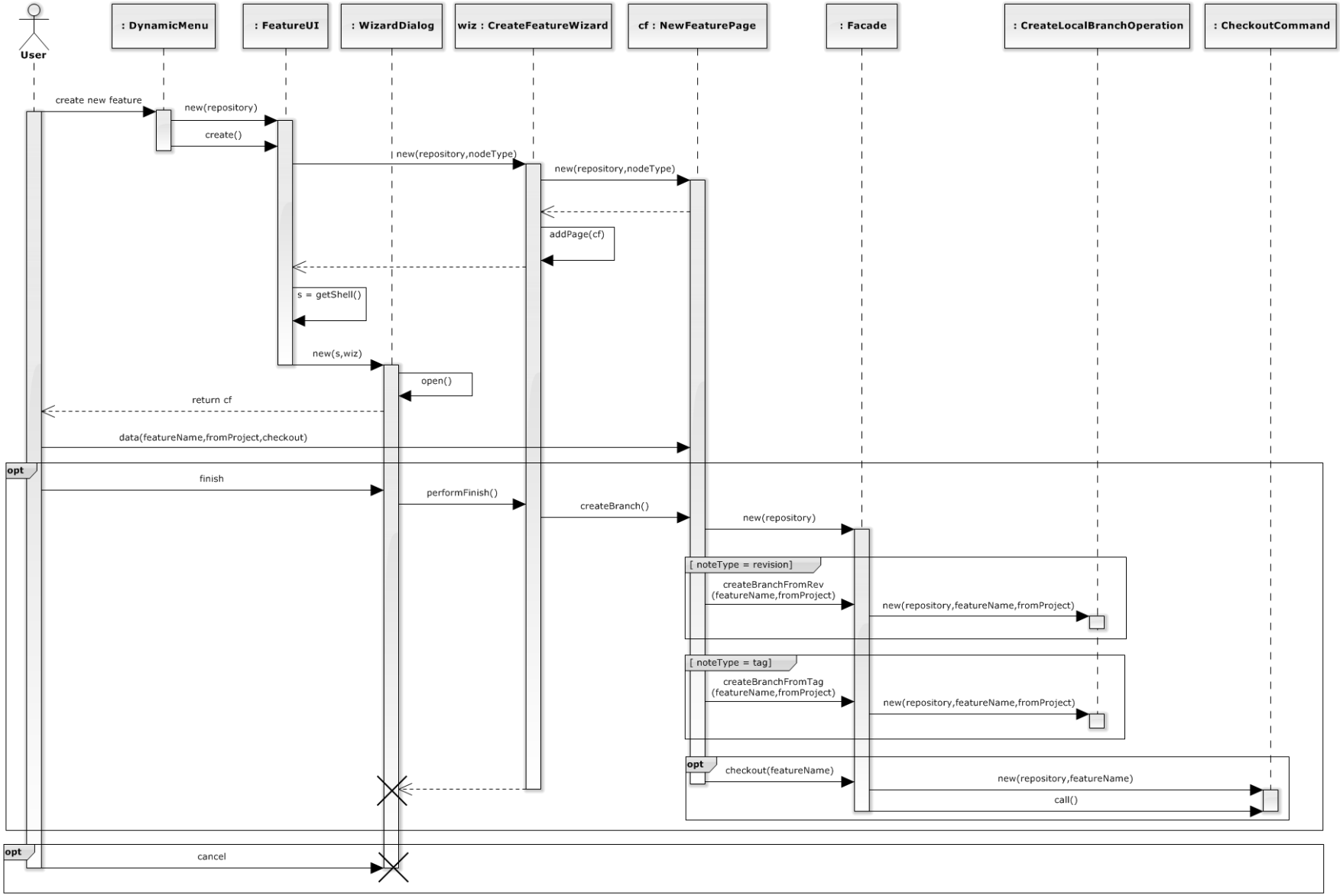


Figura 41 - Diagrama de secuencia: Crear característica

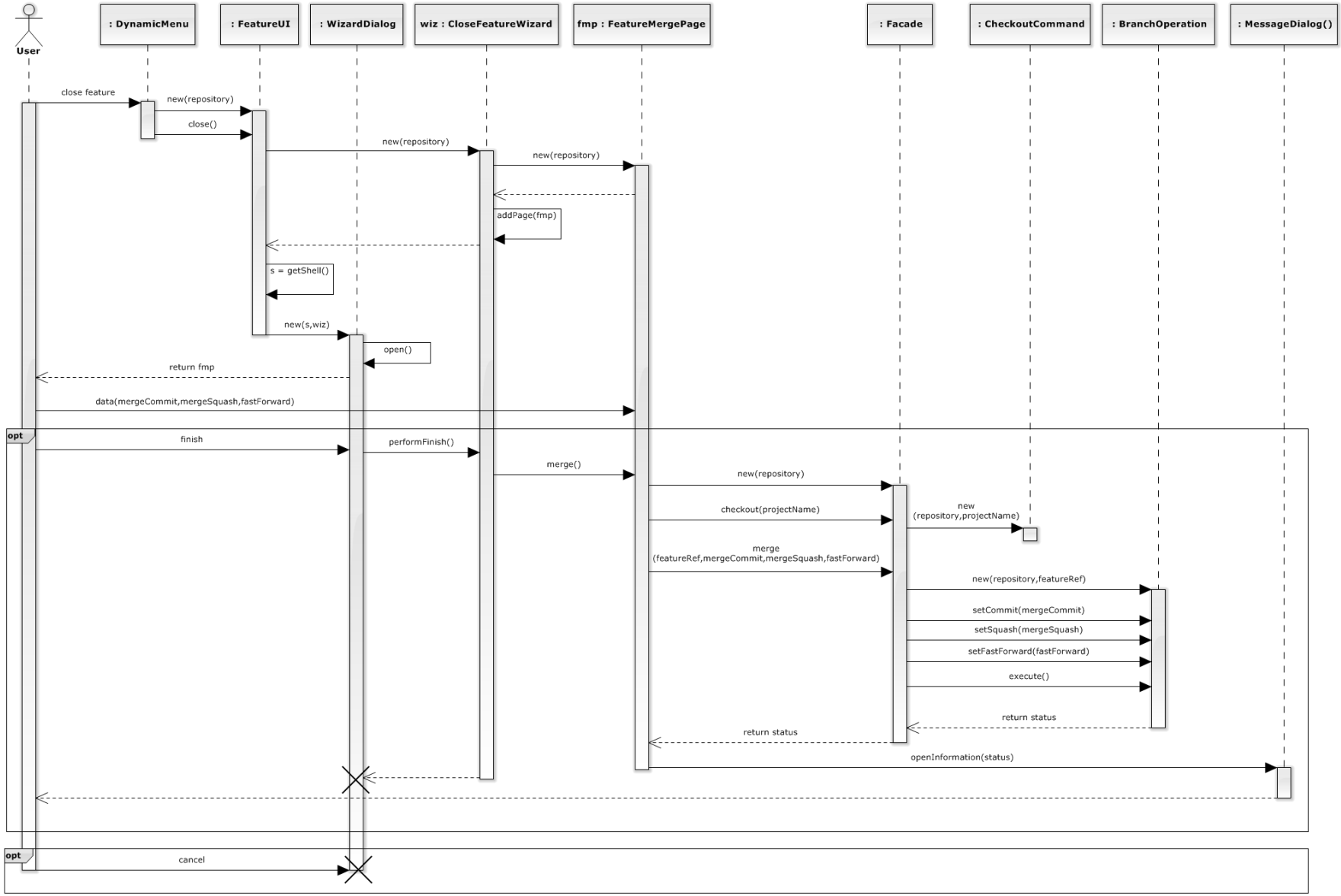


Figura 42 - Diagrama de secuencia: Cerrar característica

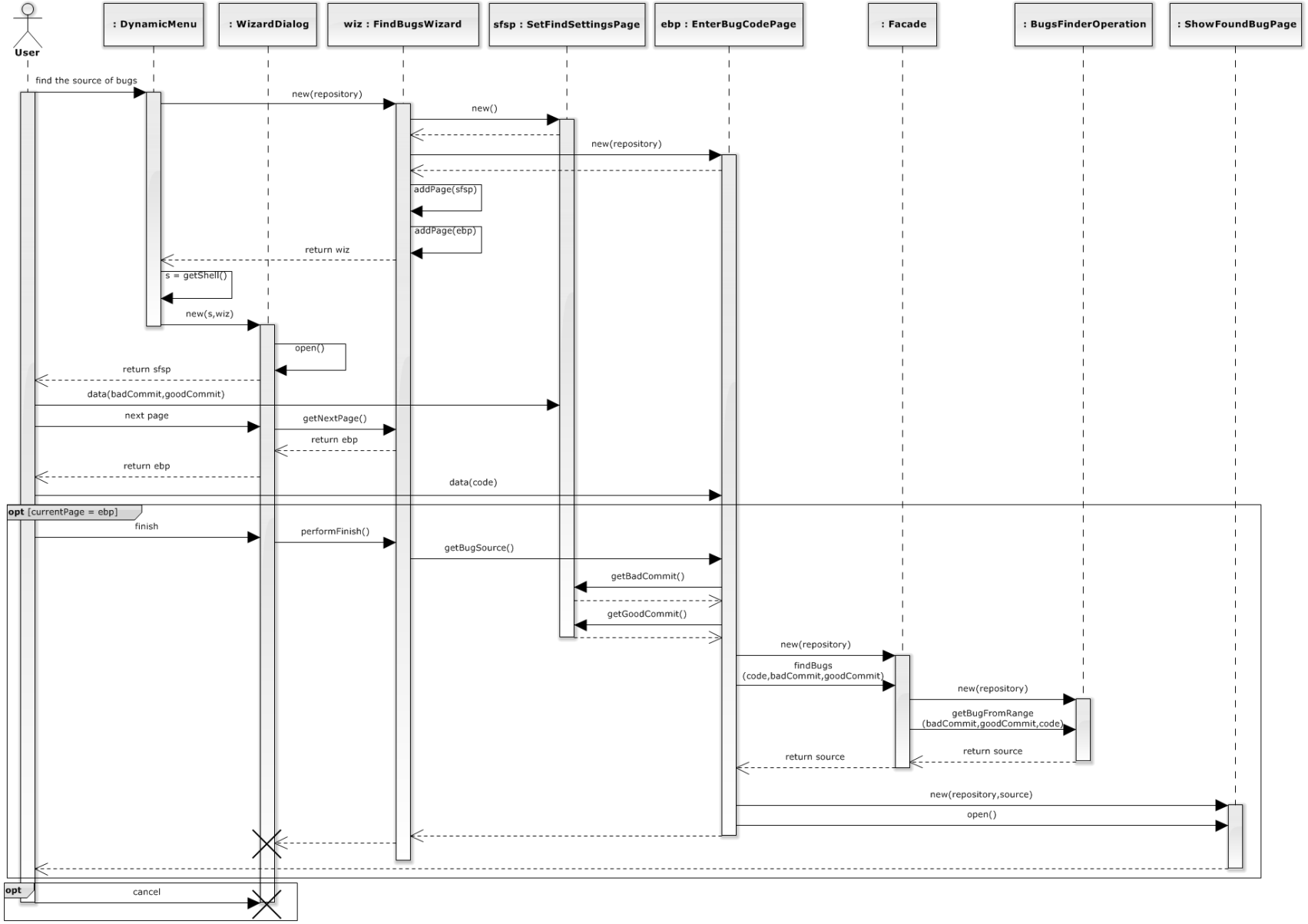


Figura 43 - Diagrama de secuencia: Buscar errores en el código

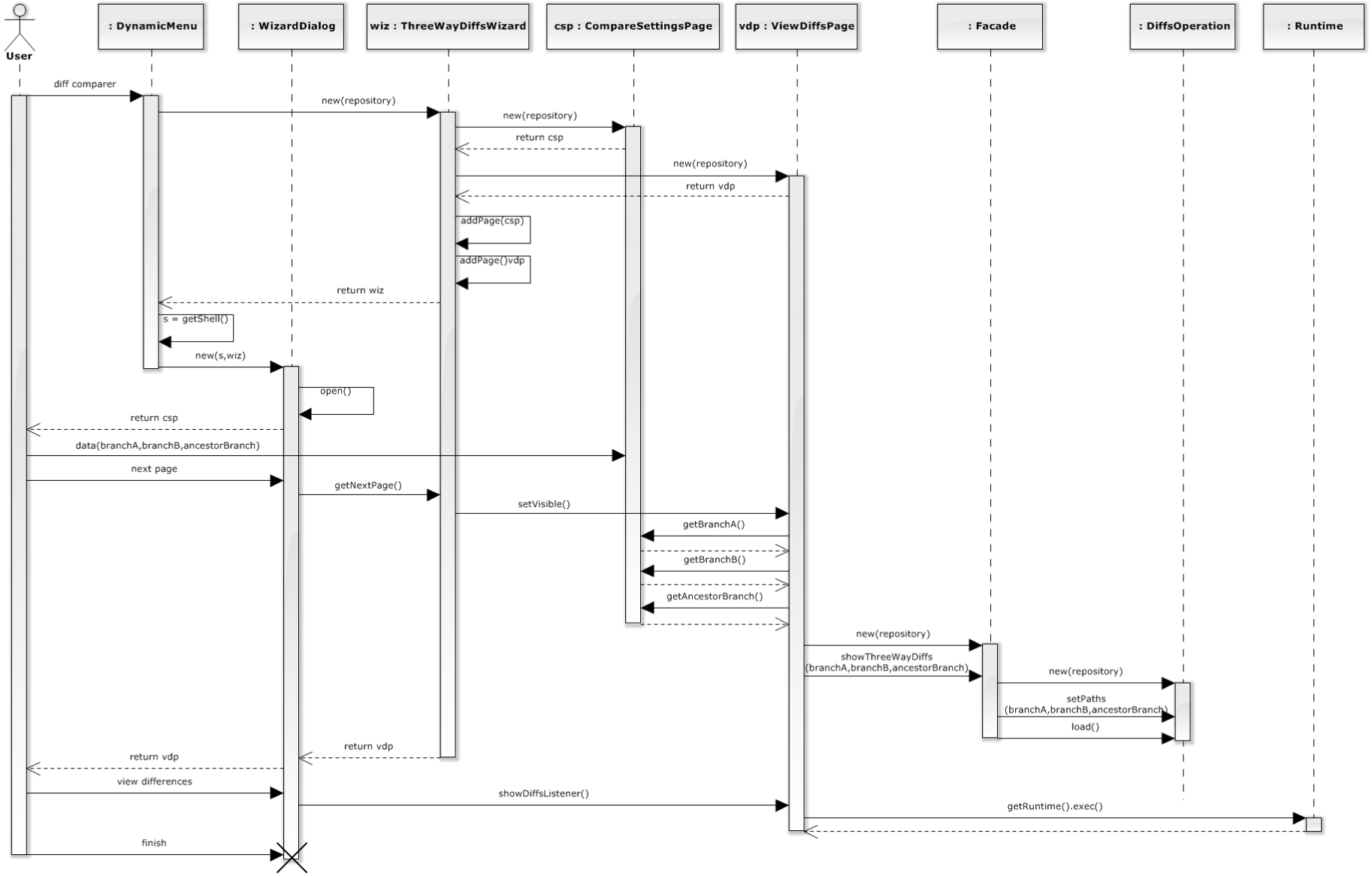


Figura 44 - Diagrama de secuencia: Ver diferencias a tres vías

4.5 Desarrollo

En esta sección se presenta de forma resumida la estructura que mantiene la herramienta proFlow para ser integrada con eGit y los pasos que se han seguido para desarrollarla al completo.

4.5.1 Estructura

La aplicación ha sido desarrollada manteniendo y haciendo uso de la composición que eGit lleva por defecto en lugar de empezar un nuevo desarrollado desde cero. Para llevar a cabo el desarrollo se ha utilizado el framework Plug- in Developnemt Enviroment (para más información ver el capítulo del estado del arte) utilizado también para desarrollar eGit. En definitiva, se han agregado nuevas funciones a eGit, de tal forma que integre las funciones necesarias que permitan aplicar en la práctica la metodología planteada en este proyecto. (Ver capítulo 3).

Arquitectura

La aplicación utiliza una arquitectura de dos capas (vista y lógica de negocio), separando la capa de presentación de la lógica de negocio en la medida de lo posible. Por defecto eGit reserva un proyecto para cada capa como se puede ver en la figura siguiente:

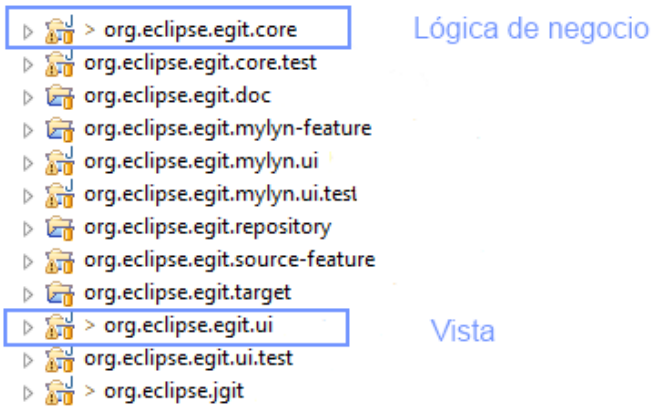


Figura 45 - eGit arquitectura

Para mantener la estructura, el código desarrollado en proFlow ha sido agregado a esos dos proyectos, respetando así la vista de la lógica.

Por un lado, para la vista se ha hecho uso de asistentes (Wizards) guiados para ir recogiendo datos en la medida que se le vayan pidiendo al usuario.

Por otro lado, para la lógica de negocio, se ha separado en diferentes clases dependiendo de la tarea a realizar. De todas formas, mediante el patrón Fachada, que permite agrupar todos los métodos de la aplicación en una misma clase, se ha conseguido que para cualquier orden a realizar se invoque siempre una clase común y sea ésta la que se encargue de desarrollar la orden.

Menú de acceso

La herramienta contiene un menú principal del cual se podrán ejecutar las distintas funcionalidades del mismo. Este menú desplegable está situado en el menú desplegable de eGit llamado Team, donde se encuentran todas las opciones relacionadas con Git. En las figuras de a continuación se observa la configuración de la extensión encargada de ello y un ejemplo del menú en ejecución.

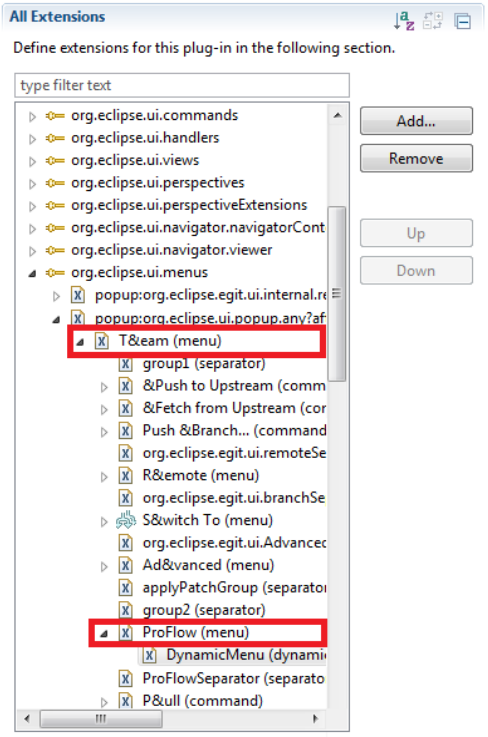


Figura 46 - Estructura del menú de la herramienta

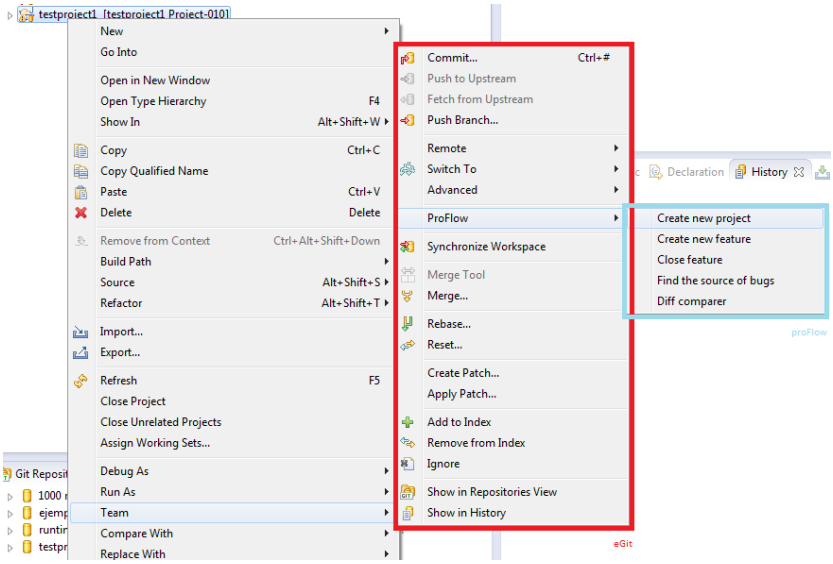


Figura 47 - Menú de la herramienta en ejecución

Cadenas de texto e imágenes

En cuanto a la organización de las cadenas de texto de la aplicación, ya sean textos de los componentes visuales, mensajes informativos, etc. se ha utilizado una clase exclusivamente para almacenarlos estáticamente. Evitando así duplicar código y consiguiendo un rápido acceso a todas las cadenas de texto y todavía más importante, preparar la herramienta al multilenguaje.

Igualmente, para las imágenes (iconos) se ha seguido el mismo patrón que el anterior, sólo que en esta ocasión se almacenan las rutas de las imágenes.

Las clases para las cadenas de texto e imágenes son, UIText.class y UIIcons.class respectivamente.

4.5.2 Implementación

En este bloque se exponen los puntos más relevantes para comprender la implementación de la herramienta proFlow.

Como ya se ha mencionado antes, proFlow está compuesto por diferentes funcionalidades que junto con las de eGit hacen que la metodología diseñada pueda implementarse correctamente. Cada funcionalidad corresponde directamente a una opción de

un menú desplegable, de modo que cuando el usuario pulse en una, inmediatamente se ejecutará la funcionalidad elegida. El menú se crea dinámicamente en base a dos estados claves, si la herramienta ha sido inicializada o no:

- ProFlow no inicializada:

  Si la herramienta no ha sido inicializada, implica que no se ha creado el fichero de configuración necesario para empezar a utilizar la herramienta. Por consiguiente, proFlow invita al usuario a inicializarla a través de una opción para ello en el menú. Dicha opción corresponde a la funcionalidad que abarca el caso de uso CU02 del apartado 4.3.1.

- ProFlow inicializada:

  Si la herramienta ha sido inicializada, significa que la herramienta está configurada correctamente para ser utilizada. Por tanto, se le muestran al usuario las opciones disponibles en el menú dinámico. Cada una de esas opciones invoca a la oportuna funcionalidad desarrollada, que implementa el caso de uso correspondiente del apartado 4.3.1. Las funcionalidades son las siguientes:

  1. Crear proyecto.
  2. Crear característica.
  3. Cerrar característica.
  4. Buscar errores en el código.
  5. Comparador de diferencias.

  En el anexo c puede verse detalladamente un el fragmento de código del menú dinámico.

Partiendo con la herramienta inicializada, el usuario se encuentra en un punto en el que debe decidir qué funcionalidad del menú ejecutar. Cada funcionalidad lanza un asistente para recoger la información necesaria y proseguir con el funcionamiento de la funcionalidad escogida.

Antes de comenzar explicando paso a paso cada funcionalidad, es necesario explicar un paso previo que sucede con las funciones relacionadas con la ramificación, como son crear proyecto, crear característica y cerrar característica. La aplicación funciona en su capa de más bajo nivel con un repositorio Git y por ello las funcionalidades se comportarán de diferente

forma dependiendo de a qué apunte el HEAD (ver apartado 2.1 para más información). De modo que puede darse diferentes estados:

- HEAD apunta a una referencia de una rama. (primer ejemplo de la figura 43)
- HEAD apunta a una revisión concreta. (segundo ejemplo de la figura 43)

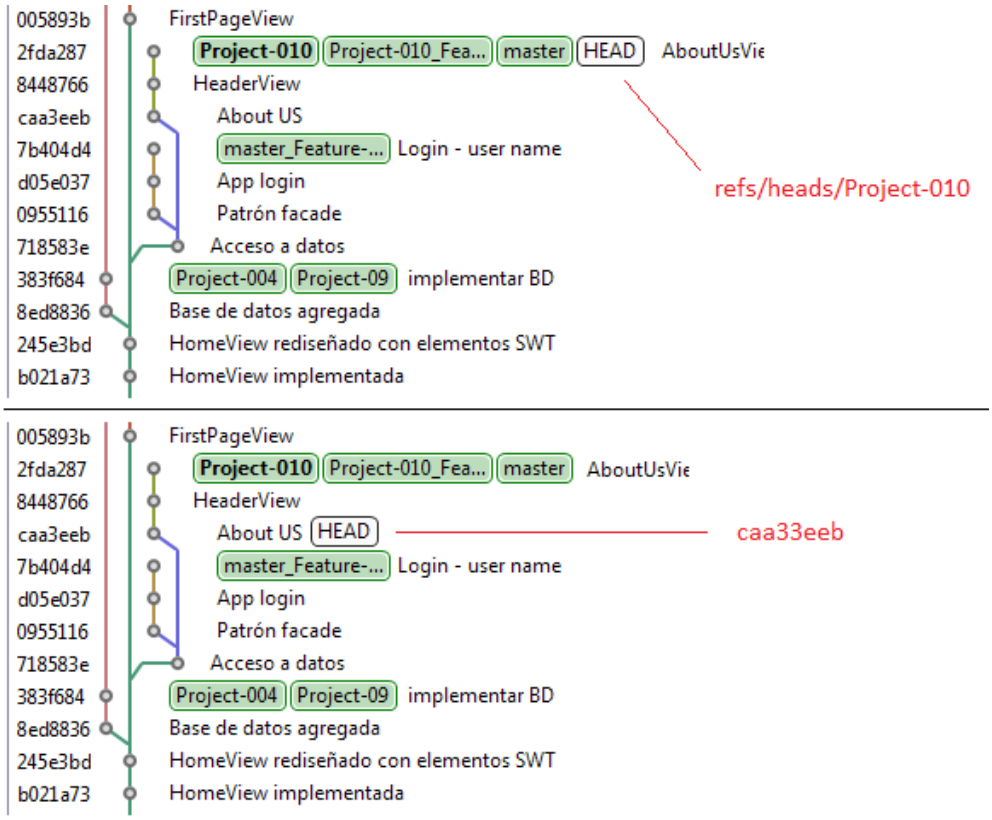


Figura 48 - eGit: Tipos de referencias

Para decidir qué acción realizar en función del HEAD se han utilizado dos clases intermediarias. La clase ProjectUI se ha usado para la funcionalidad de crear proyecto y la clase FeatureUI se ha usado para crear y cerrar características. Pueden verse en el anexo c: UI de los proyectos y UI de las características.

A continuación se describen las características más importantes de las diferentes funcionalidades de la aplicación:

Crear proyecto

El objetivo de esta funcionalidad es crear un proyecto a partir de otro proyecto, pudiendo partir desde cualquier revisión del mismo. Esta funcionalidad se extiende de una

funcionalidad genérica para crear ramas de eGit. En la sección crear proyecto del anexo C se muestran las partes de código más importantes.

Crear característica

Esta funcionalidad se encarga de crear nuevas características a partir de proyectos. Sólo pueden surgir nuevas características del último nodo de un proyecto. Al igual que la anterior ésta se extiende de una funcionalidad genérica para crear ramas de eGit. Parte del código de la funcionalidad crear característica se encuentra en el anexo c.

Cerrar característica

La funcionalidad cerrar característica consiste en fusionar el trabajo realizado en una característica sobre su rama padre, en este caso, un proyecto en concreto. La operación de fusionar ramas ya está implementada en eGit, por lo que en esta funcionalidad se hace uso de ella ajustando los datos de entrada a las características de la funcionalidad. Para visualizar el bloque de código encargado de desarrollar la operación, ver la sección cerrar característica del anexo c.

Buscar errores en el código

Esta operación permite al usuario encontrar la revisión donde se introdujeron errores de código en un pasado, y que luego fueron replicados sin conocimiento del error a lo largo del tiempo mediante las confirmaciones posteriores. Este proceso está resuelto en Git por defecto mediante el comando bisect, pero lamentablemente no existe método de eGit o de jGit que lo implemente. Por ello, mediante el uso de jGit se ha construido una funcionalidad que procede de forma parecida al git bisect. En el fragmento de código siguiente se aprecia el apartado más importante de esta funcionalidad, que corresponde al método recursivo que realiza la búsqueda binaria. En el anexo c pueden verse una parte de las líneas de código dedicadas a buscar errores en el código.

Ver diferencias a tres vías

Finalmente, esta funcionalidad se encarga de comparar las diferencias entre tres revisiones diferentes del repositorio. eGit no permite realizar comparaciones de más de dos nodos. Por este motivo, se ha hecho uso de la herramienta kdiff3 que si permite ver las diferencias entre tres directorios. En las porciones de código mostradas en ver diferencias a tres vías del anexo c puede verse la parte del proceso más importante.

4.6 Pruebas

En esta sección se realiza una breve descripción de los casos de pruebas realizados para asegurar que la aplicación funciona como se espera y también se describen las pruebas de carga efectuadas para comprobar el funcionamiento con un repositorio de muchos proyectos.

4.6.1 Pruebas de funcionalidad

Los casos de pruebas realizados para garantizar el correcto funcionamiento de la aplicación son los siguientes:

P1 Inicializar proFlow			
Genera el archivo de configuración y preparar la aplicación para su uso posterior.			
Nº	**Datos de entrada**		**Resultado obtenido**
	Prefijo de proyecto	Prefijo de característica	
1	Project	Feature	Genera un archivo de configuración almacenando los prefijos introducidos por el usuario.

Tabla 22 - Caso de prueba 1: Inicializar proFlow

P2 Crear nuevo proyecto			
Crear una rama para un nuevo proyecto a partir de otro proyecto.			
Nº	**Datos de entrada**		**Resultado obtenido**
	Origen	Proyecto	
1	Revisión de un proyecto	001	Nuevo proyecto a partir de la revisión seleccionada.
2	Último nodo de un proyecto	001	Nuevo proyecto a partir del último nodo del proyecto padre.
3	Cualquier revisión que no sea de un proyecto	001	Mensaje informativo explicando que no es un punto de comienzo válido.

Tabla 23 - Caso de prueba 2: Crear nuevo proyecto

P3 Crear nueva característica			
Crear una rama de característica a partir de un proyecto.			
Nº	**Datos de entrada**		**Resultado obtenido**
	Origen	Proyecto	
1	Último nodo de un proyecto	001	Nueva característica a partir del último nodo del proyecto.
2	Cualquier revisión que no sea el último nodo de un proyecto	001	Mensaje informativo explicando que no es un punto de comienzo válido.

Tabla 24 - Caso de prueba 3: Crear nueva característica

P4 Cerrar característica		
Fusiona los cambios realizados a lo largo del desarrollo de una característica con los de su proyecto padre.		
Nº	**Datos de entrada**	**Resultado obtenido**
	Origen	

1	Última revisión de la rama de característica	Fusiona la rama de característica con la del proyecto al que pertenece.
2	Cualquier otra revisión que no sea la anterior	Mensaje informativo explicando que no es un punto válido para realizar la fusión.

Tabla 25 - Caso de prueba 4: Cerrar característica

P5	**Buscar errores en el código**		
A partir de un rango de revisiones del repositorio, realiza una búsqueda binaria para encontrar el origen donde se produjo el error de código dado.			
Nº	**Datos de entrada**		**Resultado obtenido**
	Rango de revisiones	Error de código	
1	Rango válido (El primer commit es mayor en el tiempo que el segundo commit)	Se encuentra en el rango propuesto.	Localiza la revisión donde se cometió el error de código y la muestra por pantalla.
2	Rango válido (El primer commit es mayor en el tiempo que el segundo commit)	No hay error	Al no haber error que buscar, como es lógico advierte al usuario de que no ha encontrado ningún error en el rango de revisiones.
3	Rango inválido(cualquier rango que no respete el rango válido)	Indiferente	Evidentemente no encuentra nada en el repositorio, y así se lo indica al usuario.

Tabla 26 - Caso de prueba 5: Buscar errores en el código

P6 Ver diferencias a tres vías		
Compara las diferencias entre tres puntos diferentes: una rama, su rama padre y la rama en común de ambas ramas. La revisión en común (C) se calcula automáticamente.		
Nº	**Datos de entrada** Revisiones A y B	**Resultado obtenido**
1	Tanto A como B se encuentran en diferentes ramas.	Compara las ramas A, B y C y muestra las diferencias al usuario.
2	Las revisiones A y B son las mismas.	Aunque no tiene sentido, muestra las diferencias de cada rama. En este caso las tres ramas son la misma, de modo que muestra tres directorios idénticos.
3	Tanto A como B son diferentes revisiones pero se encuentran en la misma rama.	Al igual que en el caso anterior no tiene sentido realizar esta comparación. Pero igualmente compara dos ramas iguales y una diferente.

Tabla 27 - Caso de prueba 6: Ver diferencias a tres vías

4.6.2 Pruebas de carga

Con el fin de probar como responde la aplicación en escenarios de gran escala, se ha han realizado diferentes casos de prueba de acorde a un caso real. Para ello se ha partido con un escenario de 1000 ramas abiertas, que en la realidad son el número de proyectos distintos que podrían generase a lo largo de un año en un caso real. El escenario ha sido generado mediante un script para generar ramas automáticamente.

Para comprobar la respuesta de la aplicación, se ha probado con las siguientes operaciones.

- Crear nueva rama:

Se ha creado una nueva rama partiendo de cualquier parte del repositorio.

Resultado

Se ha creado correctamente y no se ha notado diferencia en cuanto al tiempo demorado en la creación en comparación a repositorios de menor escala.

- Cambiar de rama:

Se ha probado a cambiar de rama para comprobar el comportamiento del directorio, si se actualiza correctamente y si lo hace rápidamente.

Resultado

Se ha notado que invierte un poco más de tiempo en cambiar de una rama a otra, pero se considera normal para el número de ramas que hay.

- Fusionar rama:

La fusión es una tarea algo más compleja y que permite ver claramente si el repositorio reacciona bien. Se ha probado a fusionar la rama creada anteriormente.

Resultado

La respuesta ha sido positiva, apenas se ha notado diferencia en comparación a una fusión en un repositorio menor.

- Desplazamiento por el árbol de revisiones

Se ha probado como responde la visualización del árbol de revisiones a medida que se desplaza por él, para experimentar posibles retrasos o congelamientos de imagen.

Resultado

De vez en cuando se puede notar un pequeño retraso al desplazarse verticalmente por el árbol, sobre todo en la primera carga del repositorio. En cualquier caso, el resultado obtenido es aceptable considerando el volumen de carga del repositorio.

- Buscar errores en el código:

Mediante esta operación se quiere comprobar cómo responde el repositorio frente a una búsqueda completa. Se ha probado con uno de los peores casos, es decir, se ha propuesto un ejemplo de prueba que realice los máximos ciclos de búsqueda posible.

Resultado

El resultado ha sido el esperado. Se ha demorado un tiempo de 30 segundos en encontrar la revisión origen donde se introdujo el error, pero teniendo en cuenta la envergadura del repositorio se ve razonable. Pese haber empleado más tiempo en la búsqueda, la funcionalidad ha realizado su función correctamente.

5 CONCLUSIONES

Para concluir este proyecto de fin de grado, en este capítulo se plasman la contribución realizada, las conclusiones obtenidas y a partir de estas líneas de trabajo futuro. Se plantean además, las lecciones aprendidas por el alumno a lo largo del proyecto.

5.1 Contribución

Después de un largo proceso de análisis de la problemática del proyecto y del estudio posterior de las diferentes tecnológicas, metodologías y herramientas del mercado, se ha desarrollado una solución que da respuesta a la problemática que la gestión de una familia de productos de software conlleva.

En primer lugar, partiendo de una metodología diferente a la convencional se ha desarrollado una estrategia que facilita la solución a las dificultades encontradas en diferentes escenarios de una serie de proyectos de la empresa ULMA PACKAGING.

En segundo lugar, se ha diseñado una herramienta que facilita el uso de la estrategia al usuario desde el IDE Eclipse.

5.2 Conclusiones

Pese a haber desarrollado una herramienta que facilita en gran medida la solución a la problemática mediante la estrategia orientada a la gestión de familias de productos de software, es importante tener en cuenta que no sólo con eso basta. El factor humano es una dificultad extra a considerar, precisamente si el uso que se les da a las herramientas es incorrecto. En este caso, llevar una gestión correcta de un grupo de productos significa tener que implantar unas reglas para llevar un orden. Pero lo más importante de todo para que esto cobre sentido, es inculcar estas normas en las personas. Si bien es cierto que es complicado diseñar un modelo que gestione de forma óptima una familia de productos, más complejo es aún que los usuarios lo interioricen y lo utilicen de forma correcta.

Por otro lado, los sistemas de control de versiones son herramientas que facilitan en gran medida la gestión de los proyectos. Son herramientas muy potentes pero complejas al mismo tiempo, debido a que permiten una gran variedad de operaciones. Son necesarios unos

conocimientos básicos para desenvolverse con soltura y utilizar las herramientas beneficiosamente, lo que implica la necesidad de invertir en formación de los desarrolladores. Esto puede ser un punto en contra de sus inquietudes, pues es probable que a los desarrolladores les interese más formarse en otro tipo de lenguajes/tecnologías. Sin embargo, no hay que descuidar que aprender a utilizar un sistema de control de versiones puede ser muy positivo de cara a futuros proyectos. Un conocimiento básico de un SCV agiliza el proceso de desarrollo de los proyectos y los mantiene organizados.

Además, este tipo de herramientas fomentan el trabajo colaborativo, al dar la oportunidad a cada desarrollador de aportar sus cambios en proyectos en común. Al fin y al cabo se puede tomar como una manera de compartir proyectos entre un grupo de desarrolladores, de tal modo que también mejoren en productividad.

En cuanto a la herramienta desarrollada en este proyecto, su cometido es facilitar al usuario la implementación de la estrategia que gestiona familias de productos de software en escenarios complejos. En este proyecto se han tratado con familias de más de 1000 productos que es lo que en un caso real puede encontrarse. Hoy en día las herramientas disponibles no están diseñadas para grupos de tal envergadura, están pensadas para gestionar un único proyecto en lugar de un grupo de proyectos. Por ello se considera que la herramienta desarrollada facilita la gestión de los repositorios con gran cantidad de proyectos.

5.3 Líneas futuras

La herramienta proFlow cumple los objetivos marcados en el proyecto. Aun así se han identificado posibles mejoras de ciertos casos puntuales de la aplicación. A continuación se expone una lista con las posibles mejoras:

- Visualización del árbol de revisiones:

  La funcionalidad de eGit encargada de visualizar el árbol de revisiones del repositorio a la hora de tratar con gran cantidad ramas presenta limitaciones. No presenta la información en un orden correcto, confundiendo al usuario, y por tanto no siendo muy amigable para él. Por ello se ha tenido que hacer uso de herramientas externas de visualización (GUIs, estado del arte) que presenten un árbol más intuitivo a simple vista. Por tanto, mejorar esta funcionalidad sería de gran ayuda para el usuario.

- Diferencias a tres vías:

  La aplicación consta de una funcionalidad para ver diferencias a tres vías. Pero que para funcionar ha tenido que utilizar una herramienta externa ya que eGit no tiene tal posibilidad. Solamente es capaz de comparar diferencias de dos ramas y no tres. Por ello, se cree conveniente extender el comparador del que ya dispone.

- Ventanas no modales:

  Los asistentes de configuración de las funcionalidades utilizan ventanas modales obligando al usuario a centrarse en el asistente. Como posible mejora de estos podría ser, utilizar ventanas no modales que permitiesen realizar cualquier otra opción manteniendo el asistente en ejecución.

Por otro lado, el campo de los sistemas de control de versiones es muy amplio en cuanto a la gran variedad de herramientas que existen. Pero en general todas funcionan de manera similar.

Como se ha comentado, estas herramientas están todas enfocadas a la gestión de un proyecto, con sus variaciones, evolución, etc. En el presente proyecto se ha optado por la adaptación de uno de estos sistemas para la gestión de familias de productos. Un enfoque alternativo podría ser definir un VCS para gestionar familias de productos desde "cero", es decir, con esta problemática específica (multiproyectos, ramas abiertas,….) abordada desde la concepción misma de la aplicación.

5.4 Lecciones aprendidas

En lo que respecta al desarrollo del proyecto en general, he aprendido que en proyectos de investigación se demora mucho tiempo en el estudio de herramientas, tecnologías, lenguajes, etc. hasta que se comienza con el diseño. Hasta que no se hace un análisis de las diferentes vías y se ve con claridad cómo llevar a cabo un desarrollo de garantías no se comienza con el diseño.

Asimismo, realizar el proyecto de fin de carrera me ha servido para tener una visión más completa acerca del proceso de desarrollo y gestión de proyectos. Unos de los puntos más

problemáticos ha sido el establecer el orden de la estructura de la memoria y decidir el nivel de detalle con el que escribir puntos como el estado del arte o el de implementación.

A nivel profesional, a pesar de haber invertido gran parte del tiempo en conocer los SCVs a fondo, pensando que no me aportarían grandes habilidades como desarrollador, lo cierto es que se trata de un área que está en auge y cada vez es más reclamado por las empresas. Disponer de personas que conozca los SCVs es importante para la gestión y el desarrollo de sus proyectos, por lo que estos conocimientos acerca de Git podrían ser utilizados a modo de consultor para ayudar a otras personas o empresas.

En lo que a la empresa respecta, el hecho de realizar el proyecto en la empresa IK4-Ikerlan ha sido una oportunidad para compartir ideas y opiniones con personas cualificadas del sector y poder aprender de ellas. Igualmente, me ha servido para involucrarme en el mundo laboral y conocer el día a día de una empresa.

6 BIBLIOGRAFÍA

Control de versiones

[1] http://es.wikipedia.org/wiki/Control_de_versiones

[2] http://en.wikipedia.org/wiki/Comparison_of_revision_control_software

[3] http://es.wikipedia.org/wiki/Revision_Control_System

[4] http://www.codeproject.com/Articles/431125/Choosing-a-Version-Control-System-A-Beginners-Tour

[5] http://betterexplained.com/articles/a-visual-guide-to-version-control/

Git

[6] http://thkoch2001.github.io/whygitisbetter/

[7] http://git-scm.com/documentation

[8] http://try.github.io/

[9] http://marklodato.github.io/visual-git-guide/index-en.html

[10] http://huntingbears.com.ve/el-oraculo-git-del-desarrollador.html

[11] http://aprendegit.com/

Ramificación

[12] http://nvie.com/posts/a-successful-git-branching-model/

jGit

[13] http://www.eclipse.org/jgit/

[14] https://github.com/centic9/jgit-cookbook

eGit

[15] http://www.eclipse.org/egit/

[16] http://wiki.eclipse.org/EGit/User_Guide

[17] http://wiki.eclipse.org/EGit/Contributor_Guide

[18] http://eclipsesource.com/blogs/tutorials/egit-tutorial/

PDE

[19] http://www.eclipse.org/articles/article.php?file=Article-action-contribution/index.html

Vogella

[20] http://www.vogella.com/tutorials/

Gitflow

[21] https://github.com/nvie/gitflow

SWT Layouts

[22] http://www.eclipse.org/articles/article.php?file=Article-Understanding-Layouts/index.html

Seguimiento de incidencias

[23] http://es.wikipedia.org/wiki/Sistema_de_seguimiento_de_incidentes

Jira

[24] https://www.atlassian.com/es/software/jira

Kdiff3

[25] http://kdiff3.sourceforge.net/

SemanticMerge

[26] http://www.semanticmerge.com/

Revisión de código

[27] http://es.wikipedia.org/wiki/Revisi%C3%B3n_de_c%C3%B3digo

7 ANEXOS

7.1 Anexo A: Fundamentos básicos de Git

Este apartado contiene toda la información necesaria para conocer el funcionamiento de Git [7][10] y comenzar a utilizarlo sin problemas. Se explican, desde los conceptos a tener en cuenta hasta las operaciones más básicas.

7.1.1 Conceptos

Como en toda tecnología actual, es recomendable realizar un estudio previo de los conceptos para preparar y familiarizarse con el entorno. Con que para ello, a continuación se listan los más importantes:

- Instantáneas, no diferencias

La principal diferencia entre Git y otros VCS como puede ser Subversion, es la manera de modelar sus datos. Conceptualmente, la mayoría de los sistemas de control de versiones almacenan la información como una lista de cambios en los archivos. Estos sistemas modelan la información que almacenan como un conjunto de archivos en los que se encuentran el archivo de partida y las modificaciones realizadas a lo largo del tiempo. De modo, que en conjunto pueda construirse una determinada versión del fichero. La siguiente figura lo ilustra.

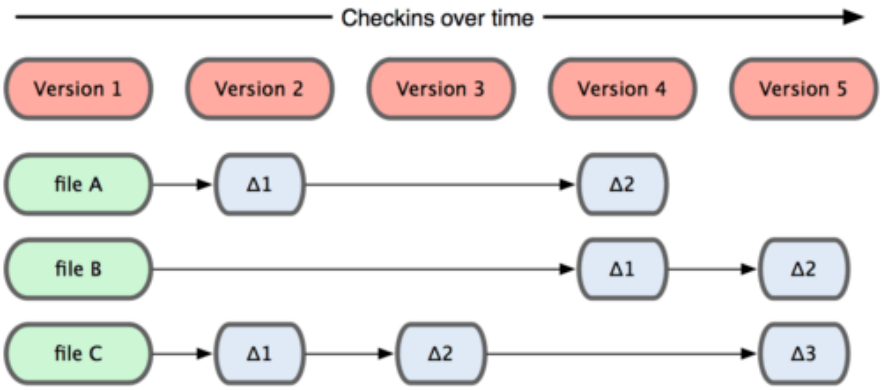


Figura 49 - Modelado de datos por diferencias

En cambio, Git no modela ni almacena sus datos de esta forma. Modela los datos como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que se confirma un cambio, guarda el estado del proyecto. Básicamente hace una foto del aspecto de los

archivos en ese instante, y guarda una referencia a esa instantánea. En la figura de a continuación puede verse.

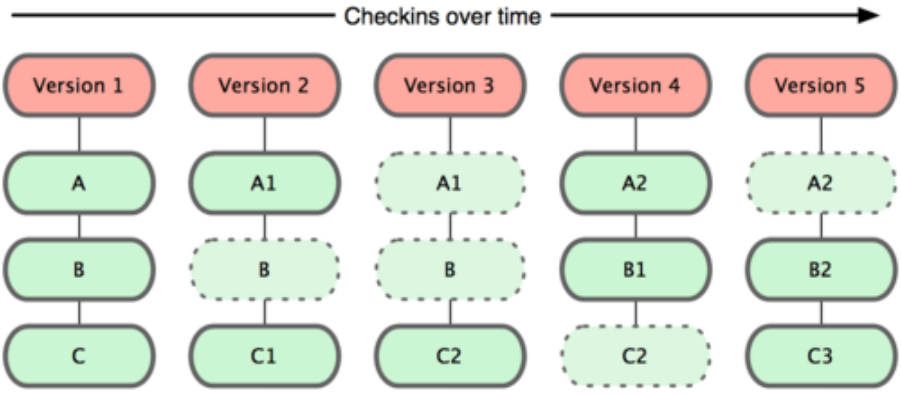


Figura 50 - Modelado de datos por instantáneas

- Revisiones

Una revisión es el estado en el que se encuentra un proyecto en un determinado momento. Dicha revisión está compuesta por los cambios realizados, el autor que los realizó, la fecha cuando se realizaron, el código identificativo, etc. Las revisiones son almacenadas en orden cronológico y en conjunto construyen un repositorio. Para poder navegar entre ellas, cada revisión apunta a la revisión que le precede.

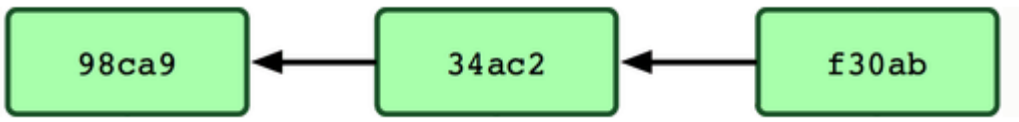


Figura 51 - Ejemplo de revisiones

- Integridad de los datos

Toda revisión en Git es identificada mediante una suma o comprobación, que se genera antes de almacenar los cambios realizados. Esto implica que es imposible modificar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No es posible perder información durante una transmisión o sufrir una corrupción de archivos sin que Git lo detecte.

El mecanismo utilizado para generar esta suma de comprobación se conoce como SHA-1. Se trata de una cadena de 40 caracteres hexadecimales, y se calcula en base a los contenidos del repositorio. Una cadena SHA-1 es de la siguiente forma:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

- Ramas y apuntadores

Hoy en día cualquier SCV moderno dispone de los mecanismos necesarios para soportar distintas líneas de desarrollo o ramas. Ramificar, significa separar en desarrollos diferentes el trabajo que se está realizando. Las ramas otorgan movilidad, pues permiten dedicar ramas para diferentes cometidos de un mismo proyecto, por ejemplo para desarrollar funcionalidades, reparar errores, etc.

En muchos sistemas de control de versiones ramificar es un proceso costoso, pues a menudo requiere hacer una copia del código en desarrollo, lo cual puede tomar demasiado tiempo si se trata de proyectos grandes. Esto no es un problema para Git, ya que como se ha comentado, hace uso de una serie de instantáneas para almacenar los datos.

Una rama Git es simplemente un apuntador móvil apuntando a una de las revisiones del repositorio. La rama por defecto de Git es la rama “master”. Con la primera confirmación de cambios que se realice, se creará esta rama principal “master” apuntando a dicha confirmación. En cada confirmación de cambios realizada, la rama irá avanzando automáticamente. Y la rama “master” apuntará siempre a la última confirmación realizada.

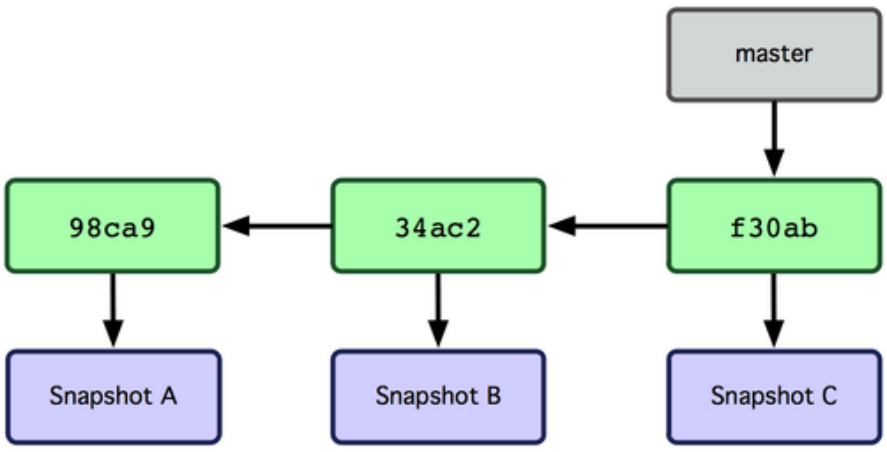


Figura 52 - Apuntadores en Git

Cuando se crea una nueva rama, simplemente se crea un nuevo apuntador en la revisión que encuentre en ese momento. Con el comando git branch se puede crear una nueva rama. Por ejemplo, para crear una nueva rama denominada “testing”:

```
$ git branch testing
```

En esta posición, con dos ramas creadas, es importante explicar cómo sabe Git en que rama está situado. Utiliza un apuntador especial denominado HEAD, que apunta a la rama local donde se encuentre el usuario. En este caso el HEAD estará apuntando al master, puesto que el comando git branch solamente crea una nueva rama, y no salta a dicha rama.

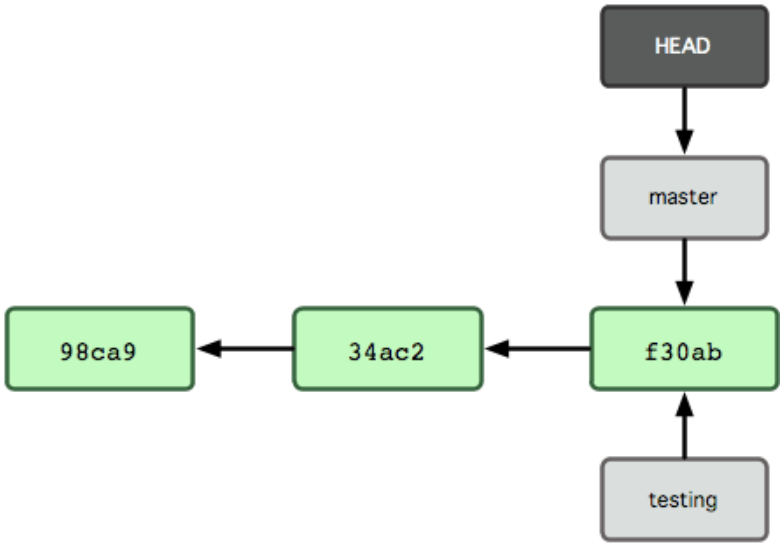


Figura 53 - HEAD en Git

Cada vez que se confirmen cambios en el repositorio, las revisiones generadas se añadirán a la rama en la que se encuentre. Por ejemplo, si se realiza una confirmación, la rama “master” será la que avanzará. De modo, que el apuntador “master” apuntará la nueva revisión, y a su vez el HEAD también.

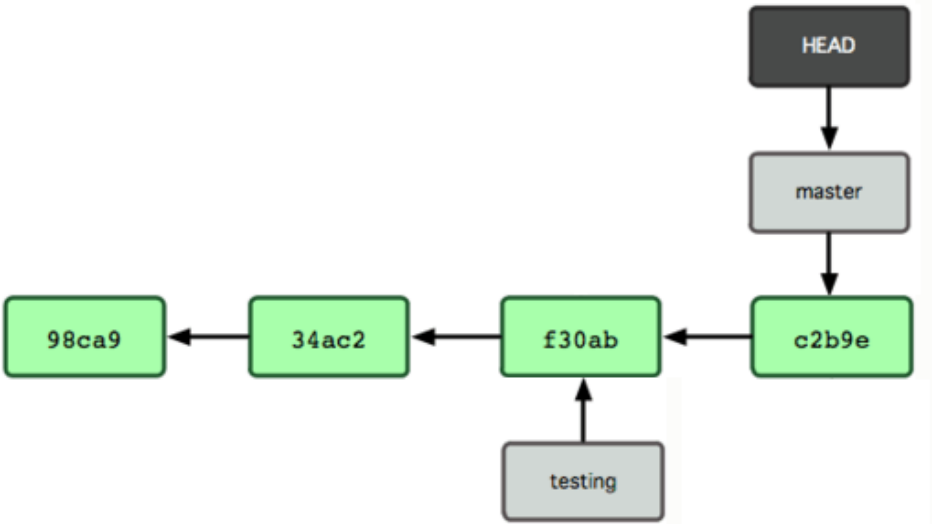


Figura 54 - Avance de los apuntadores en Git

Finalmente, para cambiar entre ramas existe el comando git checkout [rama]. Este comando realiza dos acciones: Mueve el apuntador HEAD de nuevo a la rama especificada, y revierte los archivos de tu directorio de trabajo; dejándolos tal y como estaban en la última instantánea confirmada en dicha rama.

7.1.2 Operaciones

A continuación se describen las operaciones más básicas a utilizar en Git.

Obtener un repositorio

Antes de comenzar a utilizar las funciones de Git, lo primero es preparar el directorio de trabajo. Para ello existen dos maneras: generar un nuevo repositorio u obtener una copia de uno existente.

- Inicializar nuevo repositorio

  Consiste en desplegar la estructura base de Git dentro de un directorio existente para comenzar a utilizar ese directorio con las funciones de Git. El comando a utilizar es el siguiente:

  ```
  $ git init
  ```

  Una vez ejecutado el comando añadirá un directorio .git al ya existente, que almacenará toda la información necesaria acerca del repositorio.

- Clonar un repositorio existente

  En las ocasiones en las que en lugar de comenzar un proyecto nuevo, se necesita contribuir en otro ya existente, aparece la necesidad de copiar el trabajo realizado para continuar desde ese punto. Este otro método realiza una copia exacta de otro repositorio Git existente en el directorio elegido. Para ello se usa el siguiente comando:

  ```
  $ git clone git://git.eclipse.org/r/egit.git
  ```

  Esto crea un directorio llamado "egit", inicializa un directorio .git en su interior, descarga toda la información de ese repositorio, y saca una copia de trabajo de la última versión. En el interior del nuevo directorio, estarán los archivos del proyecto, listos para ser utilizados.

  Adicionalmente, Git permite utilizar diferentes protocolos de transmisión. En el ejemplo anterior se ha utilizar el protocolo git://, pero también admite protocolos http(s):// o SSH.

Guardar cambios en el repositorio

Una operación fundamental para que el repositorio crezca en contenido es la de agregar nuevos archivos al área de preparación. Para ello Git tiene que ser capaz de detectar cuales son los ficheros a almacenar y en qué estado se encuentran. Git categoriza los archivos en dos estados diferentes: Archivos *bajo seguimiento* (tracked), que son aquellos que existían en la última instantánea, y archivos *sin seguimiento* (untracked), que son todos los demás.

Esto es así porque sólo guarda verdaderamente aquellos archivos que estén *bajo seguimiento* en el área de preparación. Para ello git dispone de dos comandos diferentes: git status, utilizado para consultar el estado en el que se encuentran los archivos, y git add, que sirve para cambiar el estado de los archivos al de *bajo seguimiento*. A continuación se muestra un ejemplo:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
```

```
#
#   new file:   README
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

Mediante el comando git add es posible añadir o cambiar de estado el fichero benchmarks.rb para que pueda ser guardado.

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

Ignorar archivos

De vez en cuando puede darse el caso de tener que ignorar ciertos ficheros o carpetas, que no interesan almacenar en el repositorio, como archivos de log, archivos generados por el compilador, etc. Para ello se da la posibilidad de crear un archivo llamado .gitignore, en el que se listarán los patrones a ignorar por Git. Por ejemplo:

```
$ cat .gitignore
*.metadata
```

En el ejemplo anterior indica, que Git debe ignorar archivos que termine en .metadata. Además permite utilizar patrones glob, expresiones regulares simplificadas orientadas a terminales de comandos, para indicar que archivos o carpetas excluir.

Confirmar cambios

Una vez preparados los archivos del repositorio, ya es posible confirmar los cambios. Todo aquel archivo que haya sido añadido (git add) será reflejado en la confirmación. Hay dos formas de realizar una confirmación: por un lado, con el comando git commit sin parámetros se lanzará el editor configurado en el terminal para que el usuario pueda introducir el mensaje de la confirmación. Por otro lado, se podrá realizar un commit ejecutando el comando y agregado el mensaje en el propio comando. El comando en cuestión es el siguiente.

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
 2 files changed, 3 insertions(+), 0 deletions(-)
        create mode 100644 README
```

Opcionalmente, git dispone de un comando para saltar del área de preparación directamente al paso de confirmación. Para ello es necesario hacer uso del parámetro - a del comando git commit. Por ejemplo:

```
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
        1 files changed, 5 insertions(+), 0 deletions(-)
```

Eliminar archivos

Al igual que Git permite preparar archivos para su posterior confirmación, en determinados momentos interesa eliminarlos del área de preparación o cambiarlos al estado de sin seguimiento. El comando git rm se encarga de realizar esta operación. Un ejemplo:

```
$ git rm grit.gemspec
rm 'grit.gemspec'
$ git status
# On branch master
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       deleted:    grit.gemspec
#
```

Excepcionalmente, si se elimina el archivo del directorio manualmente, sin utilizar el comando, no quedará reflejado en la próxima confirmación. De modo, que será necesario lanzar el comando, ya sea antes o después.

Histórico de revisiones

En un momento dado cuando el repositorio vaya obteniendo cierto volumen de revisiones probablemente interesa consultar las modificaciones que se han llevado a cabo a lo largo del tiempo. El comando dedicado a esta función es git log, a continuación se muestra un ejemplo.

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

          first commit
```

Como se puede ver en el fragmento de código superior, sin pasar ningún argumento, git log lista las confirmaciones con su código SHA-1, dirección de correo del autor, la fecha y el mensaje de la confirmación. Por defecto, git log lista las revisiones hechas sobre el repositorio en orden cronológico inverso. Es decir, las revisiones más recientes se muestran al principio.

Pese a que git log es una herramienta muy potente, conviene utilizar interfaces graficas para ver visualmente el árbol de revisiones, mejora notablemente la visualización y sin necesidad de conocer determinadas opciones del comando.

Repositorios remotos

En Git para colaborar en cualquier proyecto, es necesario gestionar repositorios remotos. Los repositorios remotos son versiones del proyecto del desarrollador que se encuentran alojados en Internet o en algún punto de la misma red., de modo que diferentes usuarios de un mismo proyecto puedan sincronizar sus cambios en un repositorio en común. Se pueden tener varios, pudiendo configurar permisos de sólo lectura o de lectura/escritura. Todo esto implica conocer la forma de añadir repositorios nuevos, eliminar aquello no válidos, etc.

- Mostrar repositorios remotos

Los repositorios remotos ya configurados se pueden ver ejecutar el comando git remote. Muestra únicamente una lista con los nombres de los repositorios. Con el parámetro –v muestra la URL asociada además de los nombres. Un ejemplo de ello:

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
   origin
```

Con las URLs asociadas:

```
$ git remote -v
   origin  git://github.com/schacon/ticgit.git
```

Por defecto, si el repositorio de trabajo ha sido clonado, debería verse por lo menos el repositorio “origin”, que corresponde al nombre predeterminado que Git da al servidor del que se realizo el clonado.

- Añadir repositorio remotos

Además de clonar los repositorios para añadirlos, existe la posibilidad de añadirlos directamente. Para añadir un nuevo repositorio Git remoto, basta con ejecutar el comando

git remote add [nombre] [url], donde el nombre será el nombre que se le quiere dar y la url la dirección del servidor. Por ejemplo:

```
$ git remote
   origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
   origin  git://github.com/schacon/ticgit.git
   pb  git://github.com/paulboone/ticgit.git
```

- Recibiendo contenido

Una vez configurado el repositorio para trabajar con repositorios remotos, surge la necesidad de obtener datos del repositorio remoto para agregarlos al repositorio local. El comando git fetch [repositorio] recupera todos los datos del repositorio remoto especificado como argumento.

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master     -> pb/master
    * [new branch]      ticgit     -> pb/ticgit
```

Es importante tener en cuenta que el comand git fetch sólo obtiene la información, no la fusiona con el trabajo existente. Por tanto, es trabajo del usuario unir ambas ramas. No obstante existe un comando para recuperar y unir automáticamente la información. El comando en cuestión se llama git pull, que en resumidas cuentas, realiza un git pull seguido de un git merge.

- Enviar contenido

Al igual que recibir cambios, llegado el momento interesará compartir el trabajo desarrollado con un repositorio remoto. El comando que permite realizarlo es git push [nombre- remoto] [nombre- rama], que envía los cambios generados en la rama elegida al servidor remoto especificado. Un ejemplo básico:

```
$ git push origin master
```

Esta acción funciona solamente si se ha clonado de un servidor en el que se tiene permisos de escritura, y nadie ha enviado información mientras tanto. En caso de que dos personas clonen a la vez, y envíen la información al mismo tiempo, el envío más tardío será rechazado. El que haya sido rechazado tendrá que obtener los cambios obtenidos por el otro e incorporarlo a al suyo. Después podrá realizar el envío sin problemas.

- Eliminar un repositorio remoto

Si por alguna razón se ve necesario eliminar la referencia a un repositorio remoto, por que se haya dejado de contribuir, ya no se utilice o cualquier otro motivo, el comando para hacerlo es git remote rm [nombre del repositorio].

```
$ git remote rm paul
$ git remote
   origin
```

7.2 Anexo B: Manual de usuario

A continuación se presenta un breve manual de usuario para utilizar la herramienta correctamente. Para ello se ha utilizado un repositorio existente sobre el que realizar diferentes operaciones de prueba.

En primer lugar, la aplicación dispone de diferentes herramientas para utilizarla. Para el correcto funcionamiento de eGit y proFlow lo más importante son dos menús en concreto. Por un lado, cada proyecto de eclipse puede desplegar un menú de opciones pulsando el botón derecho del ratón sobre el proyecto. Éste mostrará una serie de operaciones, las que interesa para realizar acciones de Git se encuentran en el submenú Team, y para las especificas de proFlow en el submenú proFlow que se encuentra su vez en Team. Un ejemplo visual:

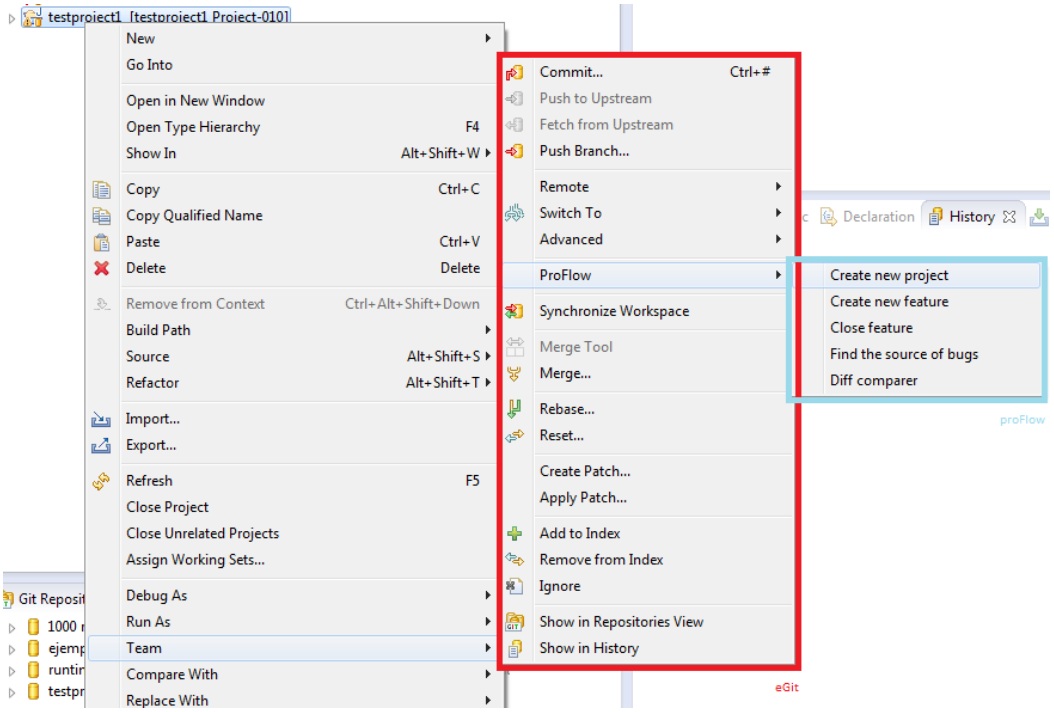


Figura 55 - Menús desplegables

Por otro lado, existe un menú desplegable al igual que el anterior para realizar las operaciones en relación con las revisiones (checkout, merge, cherry- pick, etc.). Lo único que hay que hacer es mostrar el historial de revisiones mediante la opción Show History del menú Team, y hacer click con el botón derecho del ratón en las revisiones de interés para mostrar el menú. En la figura siguiente se puede ver un ejemplo de uso:

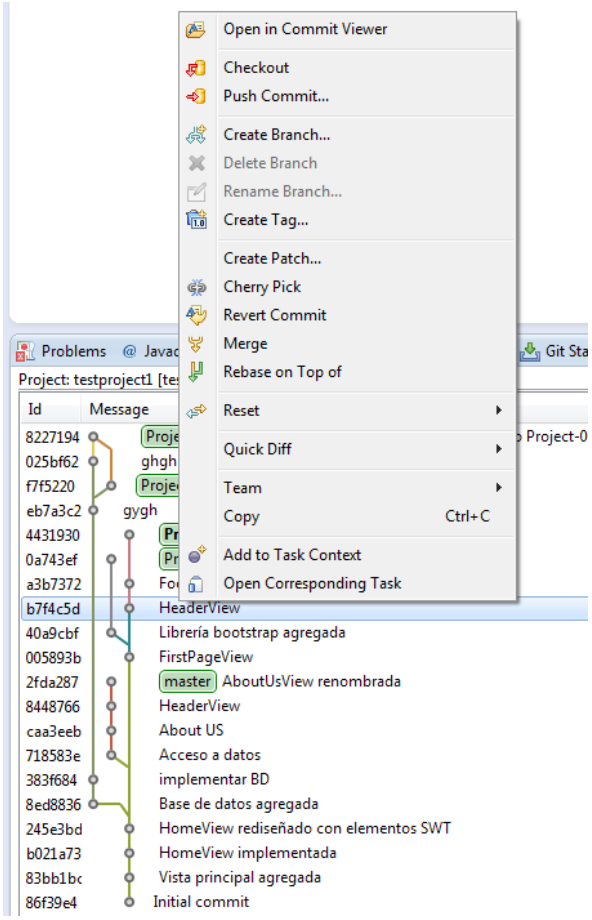


Figura 56 - Opciones del historial de revisiones

7.2.1 Inicializar repositorio

Este apartado es crucial para comenzar a utilizar las ayudas de proFlow. En todo proyecto está será la primera opción obligatoriamente. Hasta que la aplicación se haya configurado no se podrán utilizar las demás opciones. Mediante el botón del Team -> proFlow -> Init se muestra el asistente de configuración que se encarga de guardar los prefijos de proyectos y características a utilizar. El asistente, dejará introducir prefijos específicos o utilizar los prefijos por defecto. Una vez realizada una de las dos opciones se pulsará el botón next, y mostrará cómo va a ser el resultado del fichero de configuración. Al pulsar en finish, creará el fichero. En la figura siguiente se muestra un ejemplo de ello.

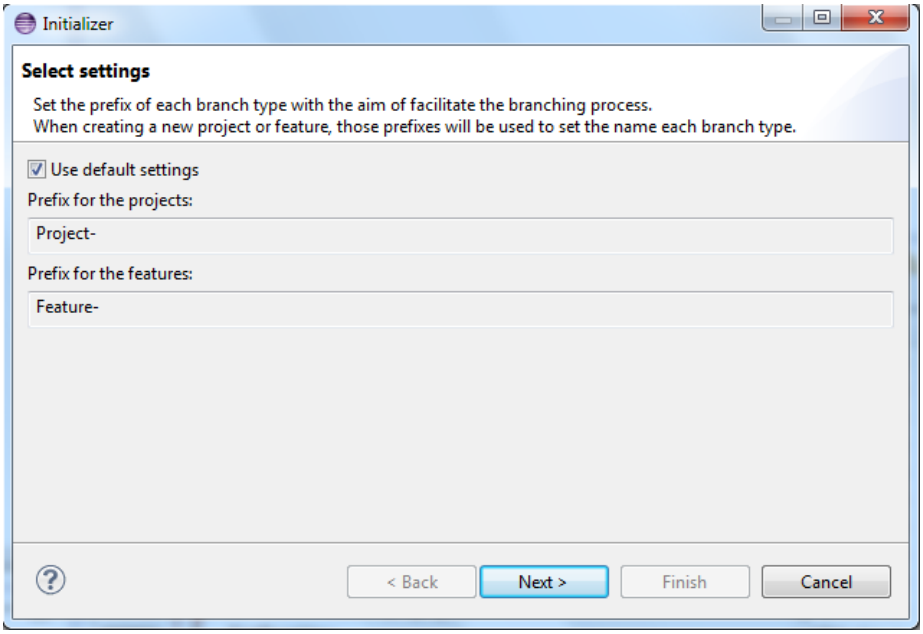


Figura 57 - Asistente para inicializar proFlow

7.2.2 Crear nuevo proyecto

Para crear nuevas líneas de desarrollo de proyectos tiene que existir un punto de comienzo para un nuevo proyecto válido. En este caso se utiliza como punto de partida la rama dedicada a otro proyecto, Project-001. Si el usuario no está colocado sobre la rama, tendrá que hacer un checkout para cambiar desde el punto en el que esté a la rama Project-001. El checkout puede realizarlo colocándose sobre la revisión de interés y pulsando en la opción checkout del menú desplegable.

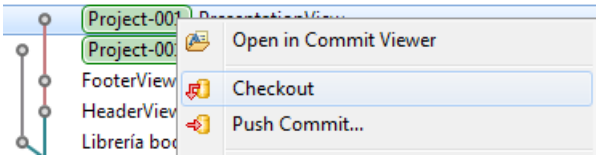


Figura 58 - Checkout de ejemplo

Después, el usuario deberá acceder el asistente de creación de proyectos siguiendo la ruta del menú Team -> proFlow -> Create new Project. El asistente solicitará el nombre del proyecto a crear. El usuario lo insertara y pulsará en finish para la nueva rama. El asistente para crear proyectos se puede visualizar en la siguiente figura.

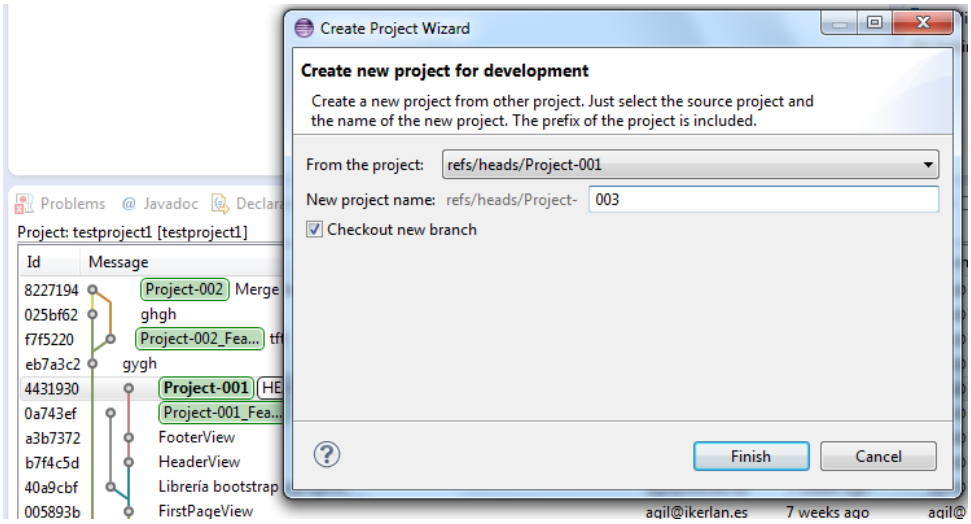


Figura 59 - Asistente para crear proyectos

El resultado, no será más que una nueva etiqueta apuntando a la revisión actual. El resultado puede verse visualmente en la siguiente figura.

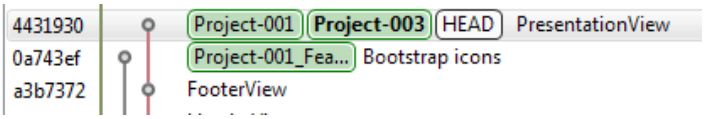


Figura 60 - Nuevo proyecto creado

A partir de este momento el usuario podrá trabajar en este nuevo proyecto partiendo con el contenido desarrollado en el Proyecto 001.

7.2.3 Crear nueva característica

Como ya se ha mencionado en ocasiones anteriores, crear nueva característica es una funcionalidad que internamente realiza el mismo proceso que la anterior. A diferencia de crear proyecto, es necesario dar un nombre a la característica a crear. Para ello, situándose en una revisión válida, el último nodo del proyecto creado en el paso anterior, es posible crear una nueva rama de característica. Teniendo el repositorio algo avanzado desde que se creó el proyecto anterior, el resultado con la nueva característica agregada sería por ejemplo el de la figura de a continuación.

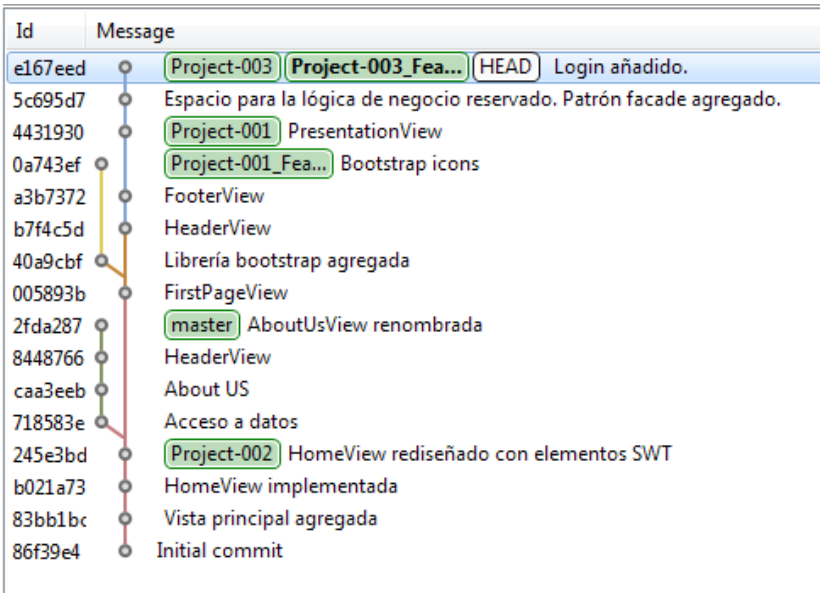


Figura 61 - Nueva característica creada

7.2.4 Cerrar característica

A menudo cuando el trabajo desarrollado en una característica está terminado, surge la necesidad de integrarlo sobre su proyecto padre. A esta acción se le ha denominado cerrar característica en la herramienta. A partir del ejemplo, con la rama característica seleccionada es posible lanzar la opción mediante la opción del menú Team -> proFlow -> Close feature. Se mostrará entonces, un asistente para seleccionar las opciones de fusionado.

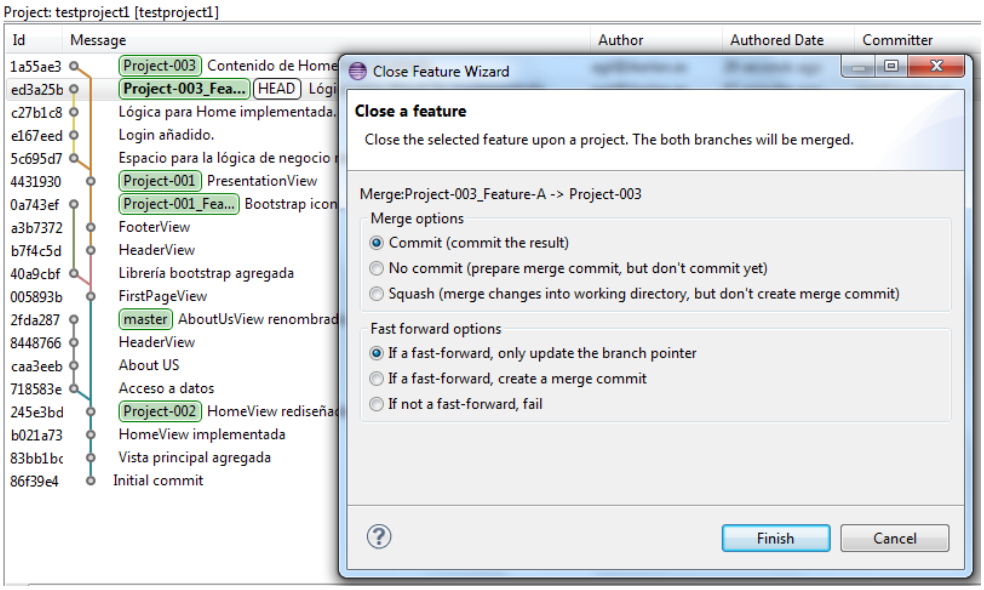


Figura 62 - Opciones de fusionado

Al pulsar en finish pueden ocurrir dos cosas dependiendo de si se han encontrado conflictos o no entre las dos ramas:

- Si no hay conflictos, la fusión se realiza correctamente.
- Si hay conflictos, no se realiza la fusión. En su defecto, marca los archivos en conflicto para que el usuario pueda solucionarlos y realizar la fusión nuevamente.

El resultado del repositorio una vez cerrada la rama será el siguiente:

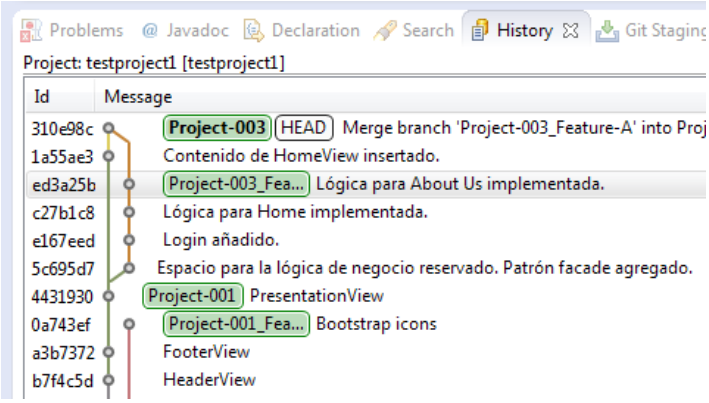


Figura 63 - Característica fusionada

7.2.5 Buscar errores en el código

Una función importante en este proyecto es la de buscar el origen de los errores generados durante el desarrollo de los proyectos. Sabiendo parte del código erróneo se puede buscar el origen del error en un rango de revisiones facilitado por el usuario. Este rango se construye a partir de dos revisiones diferentes. La primera revisión debe contener código erróneo y ser más reciente en el tiempo que la segunda. La segunda revisión no debe contener errores en el código, debe ser válida. Así se formará un rango en el que poder realizar una búsqueda binaria para ir descartando conjuntos de revisiones que no ha propagado el error. En la figura inferior puede verse un ejemplo.

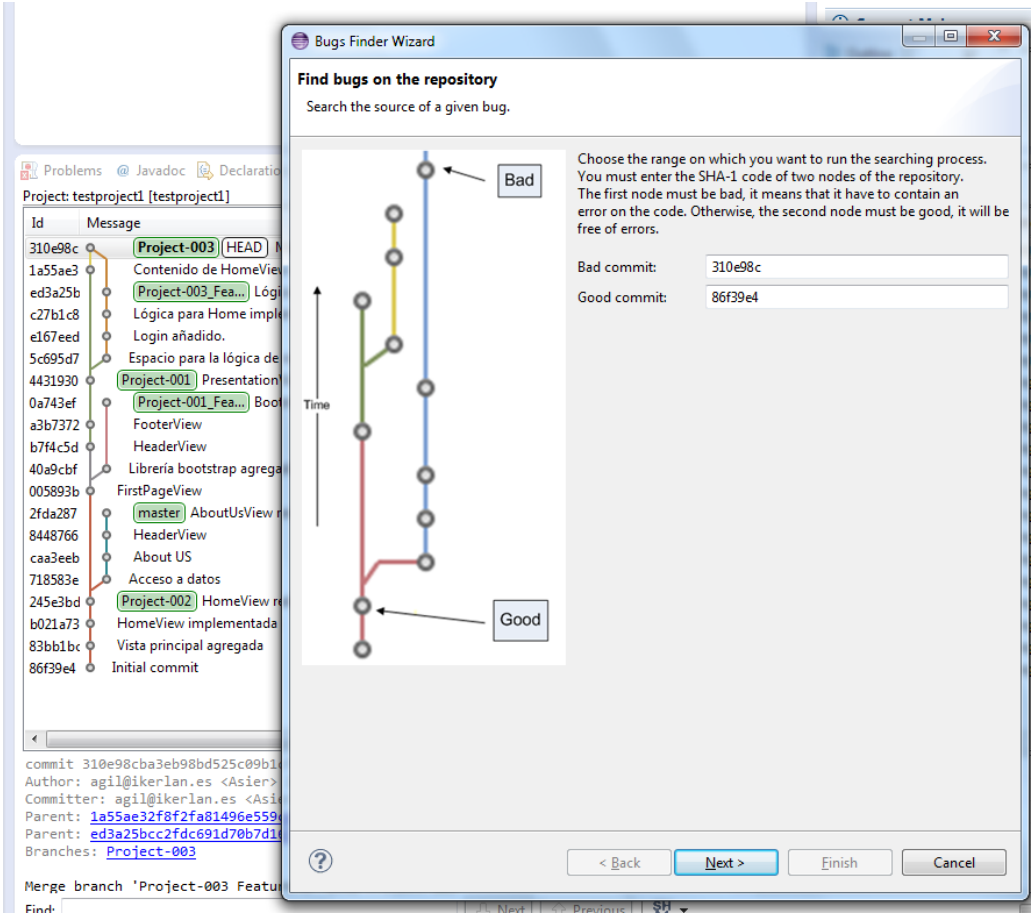


Figura 64 - Especificar rango de búsqueda

El siguiente paso, el usuario debe especificar el código a buscar en ese rango de revisiones. Una vez realizado esperará a que realice la búsqueda. Si encuentra el error lo mostrará (visualizar figura 65). En caso contrario, le hará saber al usuario que no existe ese error en el repositorio.

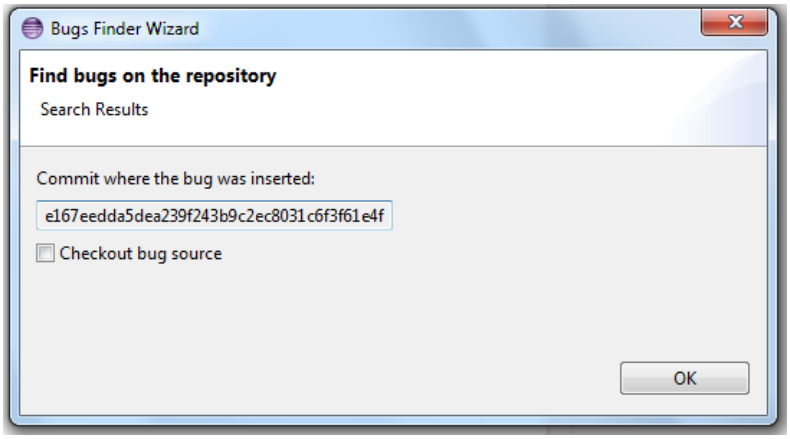


Figura 65 - Error de código encontrado

7.2.6 Ver diferencias a tres vías

En ocasiones cuando se da la necesidad de comenzar un nuevo desarrollo que parta de de un proyecto existente, es probable encontrarse con proyectos parecidos y dificulte entonces por cuál de los dos partir. Con esta herramienta, se pueden mostrar las diferencias entre dos proyectos diferentes en relación al proyecto padre de ambos. Simplemente con que el usuario indique las revisiones de cada proyecto a comparar, el sistema realizará la comparación. A continuación se ilustra un ejemplo:

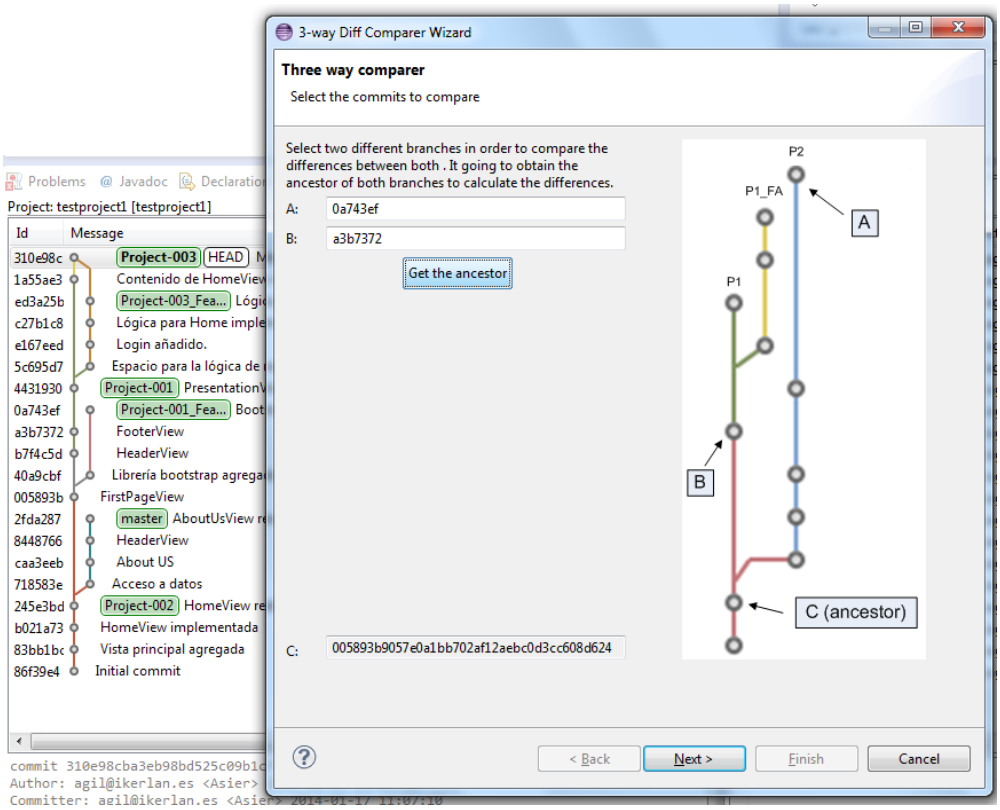


Figura 66 - Especificar revisiones a comparar

El usuario introduce cada revisión a comparar y solicita que calcule el nodo ancestro de ambas revisiones. Después pulsando next pasará a ver las diferencias entre los tres puntos, mediante la herramienta kdiff3. Una imagen de ejemplo es la siguiente:

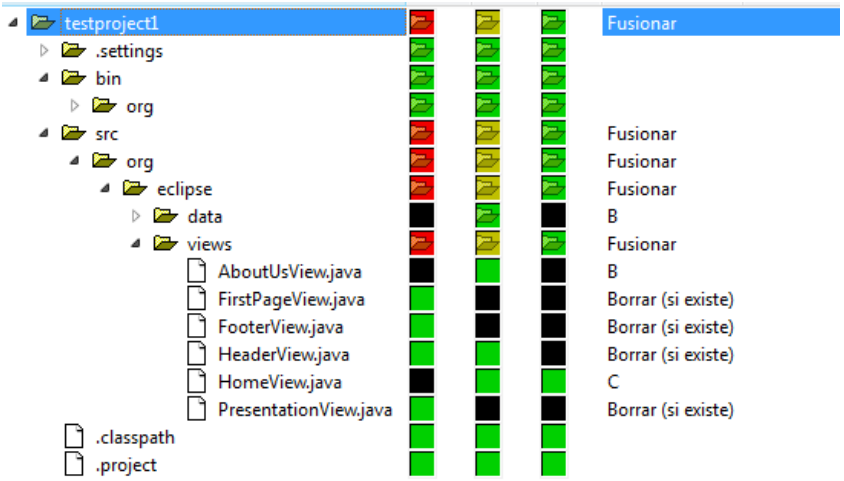


Figura 67 - Diferencias a tres vías

Para finalizar, el usuario cerrará kdiff3 y pulsará sobre el botón finish del asistente de visualización.

7.3 Anexo C: Fragmentos de código más relevantes

Este anexo muestra algunos fragmentos de código desarrollado en la herramienta para complementar el punto 4.5.2 del capítulo de implementación.

7.3.1 Menú dinámico

```
private void createDynamicMenu(Menu menu) {
        // --- MENÚ PRINCIPAL DE LA APLICACIÓN --- //
        Facade facade = new Facade(myRepository, null);

        // Estrategia no inicializada
        if (!facade.dirExist(myRepository.getDirectory().toString())) {
                MenuItem methoInit = new MenuItem(menu, SWT.PUSH);
                methoInit.setText(UIText.ProFlow_MethoInit);
                methoInit.addSelectionListener(new SelectionAdapter() {
                        public void widgetSelected(SelectionEvent e) {
                                InitOperationWizard wiz = new InitOperationWizard(myRepository);
                                new WizardDialog(getShell(), wiz).open();
                        }
                });
        } else {// Estrategia inicializada

                // --FUNCIONALIDAD: CREAR PROYECTO-- //
                MenuItem createProject = new MenuItem(menu, SWT.PUSH);
                createProject.setText(UIText.ProFlow_CreateProject);
                createProject.addSelectionListener(new SelectionAdapter() {
                        @Override
                        public void widgetSelected(SelectionEvent e) {
                                ProjectUI cUI = new ProjectUI(myRepository);
                                cUI.create();
                        }
                });

                // --FUNCIONALIDAD: CREAR CARACTERÍSTICA-- //
                MenuItem createFeature = new MenuItem(menu, SWT.PUSH);
                createFeature.setText(UIText.ProFlow_CreateFeature);
                createFeature.addSelectionListener(new SelectionAdapter() {
                        @Override
                        public void widgetSelected(SelectionEvent e) {
                                FeatureUI cUI = new FeatureUI(myRepository);
                                cUI.create();
                        }
                });

                // --FUNCIONALIDAD: CERRAR CARACTERÍSTICA-- //
                MenuItem closeFeature = new MenuItem(menu, SWT.PUSH);
                closeFeature.setText(UIText.ProFlow_CloseFeature);
                closeFeature.addSelectionListener(new SelectionAdapter() {
                        @Override
                        public void widgetSelected(SelectionEvent e) {
                                FeatureUI cUI = new FeatureUI(myRepository);
                                cUI.close();
                        }
                });

                // --FUNCIONALIDAD: BÚSQUEDA DE ERRORES EN EL CÓDIGO-- //
                MenuItem bugsFinder = new MenuItem(menu, SWT.PUSH);
                bugsFinder.setText(UIText.ProFlow_BugsFinder);
                bugsFinder.addSelectionListener(new SelectionAdapter() {
```

```
                        public void widgetSelected(SelectionEvent e) {
                                FindBugsWizard wiz;
                                try {
                                        wiz = new FindBugsWizard(myRepository);
                                        new WizardDialog(getShell(), wiz).open();
                                } catch (IOException e1) {
                                        // TODO Auto-generated catch block
                                        e1.printStackTrace();
                                }
                        }
                });
                // --FUNCIONALIDAD: COMPARACIÓN A TRES VÍAS-- //
                MenuItem threewayDiff = new MenuItem(menu, SWT.PUSH);
                threewayDiff.setText(UIText.ProFlow_DiffComparer);
                threewayDiff.addSelectionListener(new SelectionAdapter() {
                public void widgetSelected(SelectionEvent e) {
                        ThreeWayDiffWizard wiz;
                                try {
                                        wiz = new ThreeWayDiffWizard(myRepository);
                                        new WizardDialog(getShell(), wiz).open();
                                } catch (RevisionSyntaxException | IOException e1) {
                                        // TODO Auto-generated catch block
                                        e1.printStackTrace();
                                }
                        }
                });
        }
}
```

7.3.2 UI de los proyectos

```
public class ProjectUI {
        private Repository repo;
        private String base;

        private int type;
        public final static int INVALID_SOURCE = 0;
        public final static int PROJECT_REV_SOURCE = 1;
        public final static int FEATURE_REV_SOURCE = 2;
        public final static int PROJECT_SOURCE = 3;
        public final static int FEATURE_SOURCE = 4;

        public ProjectUI(Repository repo) {
                this.repo = repo;

                Facade facade = new Facade(repo, null);
                type = facade.getNodeType();
                try {
                        base = repo.getFullBranch();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }

        public void create() {
                Shell s = PlatformUI.getWorkbench().getDisplay().getActiveShell();
                switch (type) {
                case INVALID_SOURCE:
                        MessageDialog.openInformation(s,
                                        UIText.ProFlow_InvalidSource_Title,
```

```
                                        UIText.ProFlow_InvalidSource);
                        break;
                case PROJECT_SOURCE:
                        try {
                                CreateProjectWizard wiz;
                                wiz = new CreateProjectWizard(repo, repo.getRef(base));
                                new WizardDialog(s, wiz).open();
                        } catch (IOException e1) {
                                // TODO Auto-generated catch block
                                e1.printStackTrace();
                        }
                        break;
                case PROJECT_REV_SOURCE:
                        try {
                                RevCommit commit = new RevWalk(repo).parseCommit(ObjectId
                                                .fromString(base));

                                CreateProjectWizard wiz;
                                wiz = new CreateProjectWizard(repo, commit);
                                new WizardDialog(s, wiz).open();
                        } catch (IOException e1) {
                                // TODO Auto-generated catch block
                                e1.printStackTrace();
                        }
                        break;
                default:
                        MessageDialog.openInformation(s, UIText.ProFlow_NoProject_Title,
                                        UIText.ProFlow_NoProject);
                        break;

                }
        }
}
```

7.3.3 UI de las características

```
public class FeatureUI {
        private Repository repo;
        private String base;
        private int type;

        public final static int INVALID_SOURCE = 0;
        public final static int PROJECT_REV_SOURCE = 1;
        public final static int FEATURE_REV_SOURCE = 2;
        public final static int PROJECT_SOURCE = 3;
        public final static int FEATURE_SOURCE = 4;

        public FeatureUI(Repository repo) {
                this.repo = repo;

                Facade facade = new Facade(repo, null);
                type = facade.getNodeType();
                try {
                        base = repo.getFullBranch();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
```

```
        public void create() {
                Shell s = PlatformUI.getWorkbench().getDisplay().getActiveShell();
                switch (type) {
                case INVALID_SOURCE:
                        MessageDialog.openInformation(s,
                                        UIText.ProFlow_InvalidSource_Title,
                                        UIText.ProFlow_InvalidSource);
                        break;
                case PROJECT_SOURCE:
                        try {
                                CreateFeatureWizard wiz;
                                wiz = new CreateFeatureWizard(repo, repo.getRef(base));
                                new WizardDialog(s, wiz).open();
                        } catch (IOException e1) {
                                // TODO Auto-generated catch block
                                e1.printStackTrace();
                        }
                        break;
                default:
                        MessageDialog.openInformation(s, UIText.ProFlow_NoFeature_Title,
                                        UIText.ProFlow_NoFeature);
                        break;
                }
        }

        public void close() {
                Shell s = PlatformUI.getWorkbench().getDisplay().getActiveShell();
                if (type == FEATURE_SOURCE) {
                        try {
                                CloseFeatureWizard wiz;
                                wiz = new CloseFeatureWizard(repo);
                                new WizardDialog(s, wiz).open();
                        } catch (IOException e1) {
                                // TODO Auto-generated catch block
                                e1.printStackTrace();
                        }
                } else
                        MessageDialog.openInformation(s, UIText.ProFlow_Feature_Title,
                                        UIText.ProFlow_Feature);
        }
}
```

7.3.4 Crear proyecto

```
public void createBranch(IProgressMonitor monitor) throws CoreException,IOException {
        Facade facade = new Facade(myRepository, monitor);

        monitor.beginTask(UIText.CreateBranchPage_CreatingBranchMessage,
                        IProgressMonitor.UNKNOWN);

        //Se obtiene el prefijo de los proyectos
        String newRefName = facade.getPrefix("PROJECTS") + getBranchName();
        if (myBaseCommit != null
                        && this.branchCombo.getText().equals(myBaseCommit.name()))
                facade.createBranchFromRev(newRefName, myBaseCommit);
        else {
                UpstreamConfig upstreamConfig = UpstreamConfig.getDefault(
                                myRepository, myRepository.getFullBranch());
                facade.createBranchFromTag(newRefName,
                myRepository.getRef(this.branchCombo.getText()),upstreamConfig);
```

```
        }

        //Posicionarse en el nuevo proyecto
        if (checkout.getSelection())
                facade.checkout(Constants.R_HEADS + newRefName);
}
```

En el bloque de código superior se observa cómo se genera el nombre de la rama a partir del prefijo configurado en el apartado de inicialización de la herramienta, y cómo se solicita su creación. Adicionalmente, una vez creada la nueva rama, el usuario puede indicar si posicionarse directamente sobre el nuevo proyecto.

7.3.5 Crear característica

```
public void createBranch(IProgressMonitor monitor) throws CoreException,IOException {
        Facade facade = new Facade(myRepository, monitor);

        monitor.beginTask(UIText.CreateBranchPage_CreatingBranchMessage,
                                IProgressMonitor.UNKNOWN);

        String newRefName = branchCombo.getText()
                        "_" + facade.getPrefix("FEATURES") + getBranchName();
        String[] split = newRefName.split(Constants.R_HEADS);
        newRefName = split[1];

        if (myBaseCommit != null
                && this.branchCombo.getText().equals(myBaseCommit.name()))
                facade.createBranchFromRev(newRefName, myBaseCommit);
        else {
                UpstreamConfig upstreamConfig = UpstreamConfig.getDefault(
                                myRepository, myRepository.getFullBranch());
                facade.createBranchFromTag(newRefName,
                myRepository.getRef(this.branchCombo.getText()),upstreamConfig);
        }

        if (checkout.getSelection())
                facade.checkout(Constants.R_HEADS + newRefName);
}
```

El bloque de código correspondiente a crear característica muestra cómo se genera el nombre de la rama para una nueva característica y como se solicita su creación. Adicionalmente, si el usuario así lo desea puede posicionarse directamente sobre ella una vez creada.

7.3.6 Cerrar característica

```
public void merge(IProgressMonitor monitor) throws CoreException,IOException {
```

```
        Facade facade = new Facade(myRepository, monitor);

        // Cambiar de de posición el HEAD. Apuntando al proyecto que
        // integrará la rama de característica.
        facade.checkout(source);

        MergeStatus mrgStatus = facade.merge(base, isMergeCommit(),
                        fastForwardMode, isMergeSquash());

        MessageDialog.openInformation(getShell(),
        UIText.ProFlow_FeatureMergePage_InfoMsg, mrgStatus.toString());
}
```

7.3.7 Buscar errores en el código

```
public ObjectId getBugFromRange(ArrayList<RevCommit> revsList, String code) {
        boolean bugFound = false;
        ObjectId actual = null;
        ArrayList<RevCommit> aux = new ArrayList<RevCommit>();

        int pos = (revsList.size() / 2);
        actual = revsList.get(pos);
        bugFound = BugOnFile(actual, code);

        if (pos == 0)// Caso base
                if (bugFound)
                        return actual;
                else
                        return null;
        else if (bugFound) {// Caso general
                // Recorrer árbol hacia abajo
                boolean copy = false;
                for (RevCommit rev : revsList) {
                        if (rev.equals(actual))
                                copy = true;
                        if (copy)
                                aux.add(rev);
                }
        } else {
                // Recorrer árbol hacia arriba
                for (RevCommit rev : revsList) {
                        if (rev.equals(actual))
                                break;
                        aux.add(rev);
                }
        }
        return getBugFromRange(aux, code);
}
```

7.3.8 Ver diferencias a tres vías

```
public void load(ProgressBar bar) {
```

```
        //Guarda la revisión en la que se encuentra el repositorio
        String fb = null;
        try {
                fb = repository.getFullBranch();
        } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

        // Hacer un recuento de todos los archivos y carpetas a copiar. Se
        // guardará en n el número de ficheros a copiar.
        int[] maxFiles = { 0 };

        checkoutDir(commitC);
        folderNum(srcPath, filter, maxFiles);

        checkoutDir(commitB);
        folderNum(srcPath, filter, maxFiles);

        checkoutDir(commitA);
        folderNum(srcPath, filter, maxFiles);

        // Realiza una copia por cada nodo a comparar.
        // nodos: (A,B y C)
        int[] currentFiles = { 0 };
        copyFolder(srcPath, dirAPath, filter, currentFiles, maxFiles, bar);

        checkoutDir(commitB);
        copyFolder(srcPath, dirBPath, filter, currentFiles, maxFiles, bar);

        checkoutDir(commitC);
        copyFolder(srcPath, dirCPath, filter, currentFiles, maxFiles, bar);

        // Dejar el repositorio como estaba. Volver al nodo de donde se empezó.
        checkoutDir(fb);
}
```

En el primer bloque de código se muestra el proceso de copiado del contenido de cada revisión que debe realizar el sistema. Eclipse no permite hacer una integración total con kdiff3, por lo que se ha decidido copiar temporalmente el contenido en una carpeta local, para posteriormente comparar las diferencias desde kdiff3. En el estado del arte se ha redactado un párrafo en referencia a las limitaciones de kdiff3.

```
showDiffsButton = new Button(main, SWT.PUSH);
        showDiffsButton.setText(UIText.ProFlow_ViewDiffsPage_DiffsButton);
        showDiffsButton.setLayoutData(gridData2);
        showDiffsButton.addListener(SWT.Selection, new Listener() {
        public void handleEvent(Event e) {
                switch (e.type) {
                        case SWT.Selection:
                                String[] cmd = {
                                                "C:\\Program Files (x86)\\KDiff3\\kdiff3.exe",
                                                dirA.getAbsolutePath(), dirB.getAbsolutePath(),
                                                dirC.getAbsolutePath() };
                                try {
                                        Process p = Runtime.getRuntime().exec(cmd);
                                        p.waitFor();
                                } catch (IOException | InterruptedException e1) {
```

```
                                    // TODO Auto-generated catch block
                                    e1.printStackTrace();
                                }
                            break;
                    }
            }});
```

Después, en el segundo bloque de código se ejecuta kdiff3 pasándole como parámetros los directorios copiados, y así pueda mostrar las diferencias entre ellos.