

Versión 3.0

Plastic SCM **Introducción**

Una guía introductoria donde se explican los conceptos de Plastic SCM

© 2006-2010 Codice Software

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Codice Software cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

**Contenidos**

[1. Introducción 1](#_Toc266201861)

[1.1. ¿Qué es Plastic SCM? 1](#_Toc266201862)

[1.2. Problemas sin un sistema de control de versiones 1](#_Toc266201863)

[1.3. Componentes 2](#_Toc266201864)

[2. Conceptos básicos de Plastic SCM 4](#_Toc266201865)

[2.1. Repositorio 4](#_Toc266201866)

[2.2. Ítem o elemento 5](#_Toc266201867)

[2.3. Revisiones 6](#_Toc266201868)

[2.4. Rama (branch) 7](#_Toc266201869)

[El árbol de versiones 10](#_Toc266201870)

[El explorador de ramas 11](#_Toc266201871)

[2.5. Ramas inteligentes (smart) 12](#_Toc266201872)

[2.6. Workspace (espacio de trabajo) 13](#_Toc266201873)

[2.7. Etiqueta (Label) 15](#_Toc266201874)

[3. Operaciones de Plastic SCM 16](#_Toc266201875)

[3.1. Creando un repositorio 17](#_Toc266201876)

[3.2. Crear espacios de trabajo 17](#_Toc266201877)

[3.2.1. Espacios de trabajo compartidos 18](#_Toc266201878)

[3.3. Add (Añadir) 19](#_Toc266201879)

[3.3.1. Cómo ignorar determinados ítems 19](#_Toc266201880)

[3.4. Check in (proteger) 22](#_Toc266201881)

[3.5. Update (obtener) 23](#_Toc266201882)

[Ítems modificados en el espacio de trabajo 23](#_Toc266201883)

[Ítems marcados como .unloaded 23](#_Toc266201884)

[Resguardo automático de desprotecciones (Checkout shelving) 24](#_Toc266201885)

[Update paralelo 25](#_Toc266201886)

[3.5.1. Cloaked ítems 25](#_Toc266201887)

[3.6. Check out (desproteger) 27](#_Toc266201888)

[Detalles internos de la operación de check out 28](#_Toc266201889)

[3.7. Viendo las protecciones pendientes 29](#_Toc266201890)

[3.8. Uncheckout (cancelar una desprotección) 30](#_Toc266201891)

[3.9. Crear una rama 31](#_Toc266201892)

[3.10. Merge (integración) 31](#_Toc266201893)

[Tipos de conflicto de merge 32](#_Toc266201894)

[Contribuidores 33](#_Toc266201895)

[Merges desde la misma rama 34](#_Toc266201896)

[Merges desde ramas diferentes 35](#_Toc266201897)

[Merge desde una rama y una etiqueta 36](#_Toc266201898)

[Merge desde una etiqueta 37](#_Toc266201899)

[Merge desde un changeset 38](#_Toc266201900)

[Merge desde una revisión 38](#_Toc266201901)

[Cherry Picking 39](#_Toc266201902)

[Cherry Picking desde rama 41](#_Toc266201903)

[Merge de un intervalo de cambios 42](#_Toc266201904)

[3.11. Crear una etiqueta 42](#_Toc266201905)

[3.12. Aplicar una etiqueta 43](#_Toc266201906)

[3.13. Poner código existente bajo el control de versiones 44](#_Toc266201907)

[4. Empezando un proyecto 45](#_Toc266201908)

[4.1. Creando un repositorio 45](#_Toc266201909)

[4.2. Creación de un espacio de trabajo 46](#_Toc266201910)

[4.3. Poner código existente bajo el control de versiones 47](#_Toc266201911)

[4.4. Empezando a trabajar: Actualizar 48](#_Toc266201912)

[4.5. Desprotegiendo ítems para trabajar con ellos 48](#_Toc266201913)

[4.6. Mostrando las protecciones pendientes 49](#_Toc266201914)

[5. Selectores en profundidad 50](#_Toc266201915)

[5.1. Aspecto de un selector 50](#_Toc266201916)

[5.2. Definición del selector 51](#_Toc266201917)

[5.3. Reglas de path 52](#_Toc266201918)

[5.4. La opción norecursive de la regla path 53](#_Toc266201919)

[5.5. Comodines en la regla de selección de path 54](#_Toc266201920)

[5.6. Obteniendo una revisión específica de un ítem 55](#_Toc266201921)

[5.7. Cómo utilizar varios servidores de repositorios 55](#_Toc266201922)

[6. Especificaciones de objetos 57](#_Toc266201923)

[6.1. Especificaciones de revisión 57](#_Toc266201924)

[6.2. Especificaciones de rama 58](#_Toc266201925)

[6.3. Especificaciones de ítem 58](#_Toc266201926)

[6.4. Especificaciones de etiqueta 58](#_Toc266201927)

[6.5. Especificaciones de repositorio 58](#_Toc266201928)

[6.6. Especificaciones de espacio de trabajo 59](#_Toc266201929)

[6.7. Especificaciones de servidor de repositorios 59](#_Toc266201930)

[7. Gestionando proyectos con Plastic SCM 60](#_Toc266201931)

[7.1. Trabajando en la rama principal 60](#_Toc266201932)

[7.2. Rama por desarrollador 61](#_Toc266201933)

[7.3. Rama por tarea 62](#_Toc266201934)

[8. Sistema de consultas 64](#_Toc266201935)

[8.1. Lenguaje de consulta 64](#_Toc266201936)

[8.2. Ejemplos de uso 68](#_Toc266201937)

[8.3. La gramática del lenguaje de consultas 70](#_Toc266201938)

[9. Sistema de consultas avanzado (AQS) 71](#_Toc266201939)

[9.1. Lenguaje de consulta 71](#_Toc266201940)

[9.2. Ejemplos de uso 75](#_Toc266201941)

**Prólogo**

**Una breve introducción a la gestión de configuración de software**

Una solución de gestión de configuración de software suele contener los siguientes componentes:

* Identificación de la configuración, por lo que se puede saber en qué módulo de código fuente se está trabajando
* Control de configuración, que implica controlar el ciclo de entrega (*release)* de un producto y sus cambios a lo largo de su ciclo de vida, de modo que se pueden crear líneas base consistentes del producto
* Auditoría de estados: se pueden registrar y crear informes sobre el estado de los componentes y las peticiones de cambio para ayudar a medir algunos aspectos sobre la calidad y el proceso de cambio
* Revisiones, para validar el grado de finalización de un producto y ayudar a mantener la consistencia entre los componentes
* Gestión de compilaciones (build management), que facilita la creación y mantenimiento de un proceso de construcción repetible
* Gestión de procesos, asegura que aquellos que producen y entregan el software siguen los procesos de desarrollo de la organización.

Sin embargo, el tener todas estas piezas no significa que la solución SCM (Software Configuration Management) vaya a ser efectiva. Para tener una solución efectiva (al contrario que completa) es necesario entender que hay más de una manera de implementar todos los elementos de la lista anterior y que el modo de hacerlo variará de una organización a otra. Es también imprescindible ver que los elementos enumerados se enfocan a muy alto nivel y que para que el proceso funcione día a día es necesario centrarse también en micro actividades: todo lo que ocurre en el espacio de trabajo del desarrollador debe ser correcto. Hay que recordar que la razón de tener procesos y herramientas SCM es la comunicación y poder favorecer el trabajo en equipo.

Un proceso SCM efectivo ha de estar integrado completamente con la arquitectura, con la organización estructural de la empresa y por supuesto, con su cultura. Un proceso SCM adecuado para una empresa de nueva creación de 3 personas creando un nuevo sistema web no tiene por qué ser válido para un gran proyecto con gente trabajando desde varios países. Lo que ambas organizaciones deben compartir es que su proceso sea fácil de usar y que permita *repetitividad*. Es aquí donde el buen uso de patrones entra en juego.

Uno de los elementos claves que el proceso SCM debería permitir, pero que normalmente no suele tenerse en consideración, es la posibilidad de que los desarrolladores puedan crear *espacios de trabajo privados* que contengan la versión de la aplicación con la que tienen que trabajar. También la capacidad de que los desarrolladores puedan compilarla completamente antes de integrar sus cambios y de realizar pruebas para verificar que todos los cambios cumplen con las expectativas.

La razón fundamental de tener un proceso SCM efectivo es facilitar la comunicación. Un buen SCM permitirá que todos los miembros del equipo trabajen juntos.

Stephen P. Berczuk - Octubre 2006

**Acerca de este manual**

Este manual describe el uso normal del producto Plastic SCM desarrollado por Códice Software. En él se presentarán características y técnicas para obtener el máximo rendimiento de la herramienta.

**A quién está dirigido**

Esta guía está dirigida al desarrollador medio, asumiendo que el lector está familiarizado con conceptos básicos del sistema operativo, algún lenguaje de programación y entornos de desarrollo como Microsoft Visual Studio o Eclipse.

**Documentación online**

Además de este manual y el resto de guías, Plastic SCM proporciona una referencia online a través de su página web y su blog.

Desde la línea de comandos (tanto Windows como Linux) es posible teclear:

cm help

Para obtener información acerca de todos los comandos disponibles, y

cm help command

Para obtener información acerca de un comando específico.

Desde las herramientas gráficas es posible acceder a la ayuda Online en el menú de ayuda.

**Errores en la documentación**

Si encuentra algún problema en cualquiera de las guías, o en alguna parte de la ayuda online, por favor, notifíquelo a la siguiente dirección de correo electrónico:

[support@codicesoftware.com](mailto:support@codicesoftware.com)

# Introducción

## ¿Qué es Plastic SCM?

Plastic SCM es un completo sistema de Gestión de Configuración del Software (SCM) capaz de gestionar la evolución y cambios del código fuente en el tiempo. Mejora la visibilidad y la gestión en el desarrollo paralelo, asegurando una colaboración óptima de los equipos, y añadiendo novedosas fórmulas de visualización en una interfaz amigable.

## Problemas sin un sistema de control de versiones

Los grupos de desarrollo que no usan control de versiones están muy posiblemente familiarizados con los siguientes problemas:

* La versión actual del código se sobrescribe accidentalmente con otra más antigua.
* Un cambio crítico se descarta erróneamente de una entrega a un cliente.
* Se hacen cambios a una versión incorrecta del sistema.
* *Bugs* arreglados reaparecen de nuevo.
* Es difícil (o imposible) determinar qué versiones de cada fichero están en una *entrega* concreta. Este problema puede tener diferentes niveles de gravedad: desde no saber exactamente la versión de ciertos elementos hasta un total *agujero negro* en el que no hay manera fácil de conocer a qué fuentes se corresponden los binarios en producción.
* Las compilaciones no son reproducibles. No hay una manera fácil de reconstruir una entrega antigua, porque los fuentes y sus dependencias (librerías, componentes) no están correctamente gestionados.
* Enorme sobrecarga cuando se mantienen diferentes *releases* en producción. Es un escenario muy común para una empresa, ya sea una organización centrada en proyectos o en producto, tener más de una *versión* de su software en producción. Sin la ayuda de un SCM adecuado, gestionar correcciones, cambios y mejoras es una auténtica pesadilla.
* No hay una forma sencilla de comprobar los cambios hechos al código o a recursos asociados (imágenes, ayudas, manuales).
* Los jefes de proyecto tienen problemas graves para medir el avance.
* Pobre comunicación entre los miembros del equipo.

Prácticamente todos los problemas descritos pueden resolverse haciendo un buen uso de herramientas SCM. Este manual mostrará cómo Plastic SCM puede ser la clave para afrontar toda esa problemática.

## Componentes

Plastic SCM está diseñado en torno a una arquitectura de cliente y servidor.

* El servidor es responsable de almacenar toda la información de los proyectos, y arbitra el acceso de los clientes a dicho almacén.
* Los clientes son ejecutados por cada desarrollador en su equipo, y son responsables de comunicar al servidor las operaciones que hacen los desarrolladores. Los clientes básicos y sus plataformas soportadas son los siguientes:
  + Cliente de línea de comandos (CLI): da acceso a todas las operaciones a través de comandos desde el *Shell* del sistema.
  + Cliente gráfico: da acceso a todas las operaciones de manera gráfica e intuitiva. Es capaz de mostrar diagramas que no aparecen en la línea de comandos, como el Explorador de Ramas.
  + Integración con Microsoft Visual Studio: da acceso a las operaciones más comunes como proteger/desproteger desde el entorno de desarrollo, de forma cómoda y eficiente.
  + Integración con Eclipse: da acceso a las operaciones más comunes como proteger/desproteger desde el entorno de desarrollo, de forma cómoda y eficiente.
  + Integración con IntelliJ: permite realizar un buen número de operaciones en Plastic SCM desde el entorno de programación.

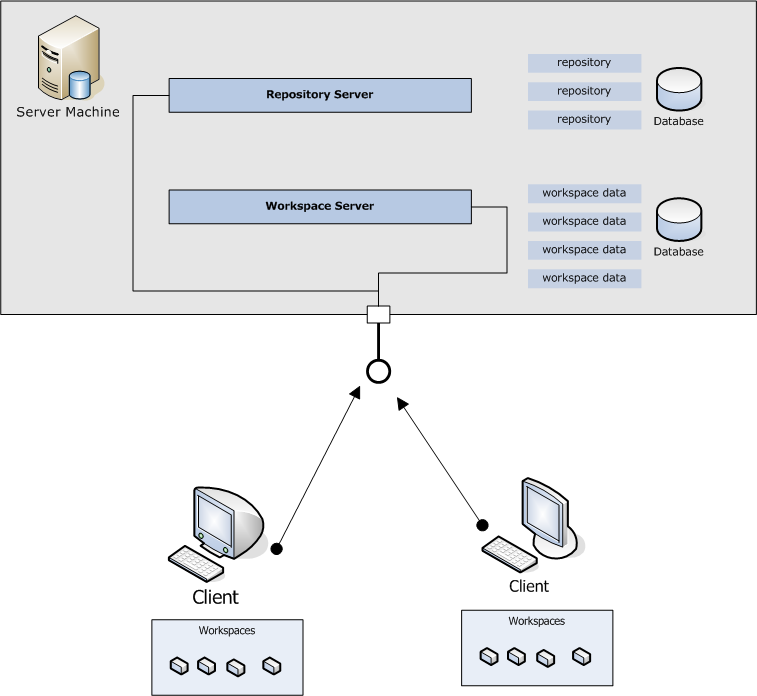


Figura 1. Componentes y estructura de Plastic SCM

La siguiente tabla resume las plataformas soportadas por los diferentes componentes:

|  |  |  |  |
| --- | --- | --- | --- |
| Componente | MS Windows | Linux | Mac OS X |
| Servidor | X | X | X |
| Cliente CLI | X | X | X |
| Cliente Gráfico | X | X | X |
| Integración con Visual Studio | X |  |  |
| Integración con Eclipse | X | X | X |
| Integración con JDeveloper | X | X | X |
| Integración con IntelliJ | X | X |  |

# Conceptos básicos de Plastic SCM

Este capítulo hará un recorrido en detalle por los diferentes elementos que forman la plataforma Plastic SCM y servirá de base para describir las operaciones del sistema en capítulos posteriores.

## Repositorio

El repositorio es el almacenamiento principal de Plastic SCM, donde se guardan todas las revisiones de los ficheros y carpetas, el momento en que se crearon esas versiones, quién las creó, y las relaciones que existen entre ellas. La Figura 2 muestra todos los tipos de objetos que contiene un repositorio, y que serán descritos en las secciones siguientes.

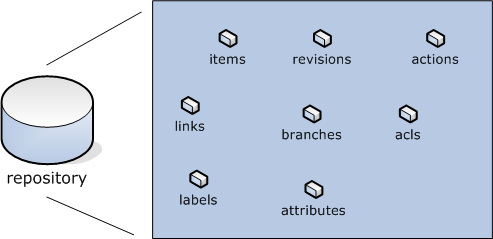


Figura 2. Objetos almacenados en un repositorio

Un servidor de Plastic SCM puede tener tantos repositorios como se precise, limitado únicamente por el espacio en disco.

La primera vez que el servidor de Plastic SCM arranca tras la instalación creará automáticamente un repositorio llamado *default* si no se detectan repositorios ya creados previamente. De esta forma el usuario puede empezar a trabajar de manera inmediata en ese primer repositorio, y crear más con posterioridad.

Para una referencia completa sobre las operaciones que se pueden hacer con los repositorios o procedimientos de copia de seguridad, consulte la Guía de Administración de Plastic SCM.

## Ítem o elemento

Un ítem es cualquier fichero o directorio cuya evolución en el tiempo se gestionará con Plastic SCM. Un proyecto de software a menudo está compuesto por múltiples ficheros y directorios. Cuando esta estructura se pone bajo control de versiones, esos ficheros y directorios se denominan ítems, indicando que será posible ver su historial de cambios y recuperar un estado anterior. Toda la información sobre los ítems se almacena en el repositorio (su nombre, tipo, quién lo ha creado y todos los detalles de su historia)

Plastic SCM soporta el versionado de directorios igual que soporta el de ficheros, característica muy importante para poder reconstruir correctamente el estado de un proyecto en el pasado.

No todos los ficheros y directorios que existen en un proyecto deben ser puestos bajo el control de código fuente. Por ejemplo, no tiene sentido poner los productos de la compilación, o ficheros de configuración específicos de cada usuario, ya que éstos son generados por los entornos de desarrollo o los compiladores. En este sentido, Plastic SCM denomina **ítems** a los elementos que están bajo control de versiones, y **elementos privados** a aquellos que están junto a éstos pero no están bajo el control de versiones. Estos elementos no existen, a efectos prácticos, para Plastic SCM pero pueden ser de utilidad para otras herramientas como compiladores y entornos de desarrollo. La figura siguiente muestra cómo conviven ítems con ficheros y directorios privados en el espacio de trabajo:



Figura 3: Ejemplo de ítems bajo control de versiones y elementos privados

## Revisiones

A medida que los ítems evolucionan en el tiempo, se van creando nuevas revisiones de los mismos. Como es lógico, algunos ítems sufrirán más cambios que otros a lo largo del desarrollo. Plastic SCM almacena todas las revisiones de cada ítem, de manera que es posible volver al estado de uno de estos elementos en un momento dado. Esta evolución suele representarse como muestra la Figura 4.

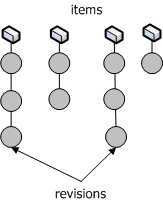


Figura 4. Ítems y revisiones

La creación de revisiones ocurre a través de las operaciones de desprotección (check-out) y protección (check-in). Cuando se va a hacer una modificación en un fichero, se *desprotege* para editarlo, y este hecho queda registrado en el servidor de Plastic SCM.

Para crear una nueva revisión del ítem de forma que se pueda recuperar su estado en este punto, se *protege*, lo que indica al servidor de Plastic SCM que debe almacenar los nuevos contenidos del ítem. Éste almacenamiento esta optimizado de forma que las revisiones son comprimidas de forma transparente en el repositorio para evitar que este crezca desmesuradamente.

Las revisiones se almacenan en el repositorio, junto al resto de información del proyecto. Entre los datos almacenados se encuentra el número de revisión, la fecha de creación, el autor de la revisión y otros detalles que se enumerarán más adelante en la sección de ramas.

Al conjunto de revisiones de un ítem se le denomina **historia** de ese ítem, y resume la evolución de ese ítem en el tiempo. La siguiente figura muestra un ejemplo de historia de un ítem:

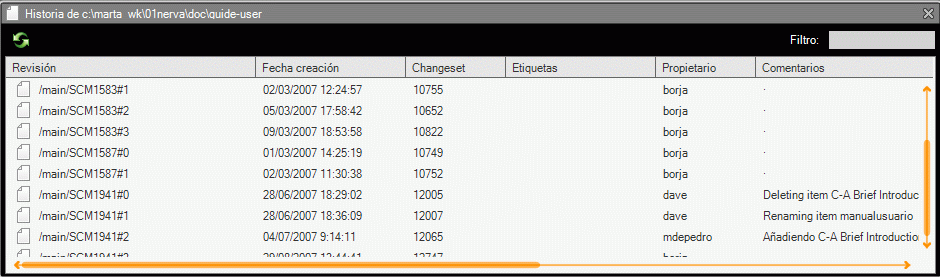


Figura 5: Ejemplo de historial de un ítem

## Rama (branch)

Las rama o *branch* es uno de los elementos claves en Plastic SCM, y una de las características de diseño que le confieren una enorme potencia. En la sección sobre revisiones, se hacía una representación de la evolución de un ítem como ésta:



Figura 6: Evolución (historia) de un ítem

En esta representación se está empleando el concepto de rama de manera implícita. Una rama es un contenedor de revisiones, capaz de almacenar la evolución de los ítems, como en la siguiente figura:



Figura 7: Revisiones de un ítem contenidas en una rama

Cuando se crea un nuevo repositorio, por defecto incluye una rama denominada “main”, de manera que todos los ítems que se añaden al repositorio van creando sus revisiones en ella, salvo que se indique otra cosa.

Ya se ha mencionado anteriormente que las ramas pueden contener revisiones de **más de un ítem**. Ésta es, de hecho, la situación más habitual y es normal que la rama “main” contenga muchas revisiones de prácticamente todos los ítems, como se observa en la figura siguiente:

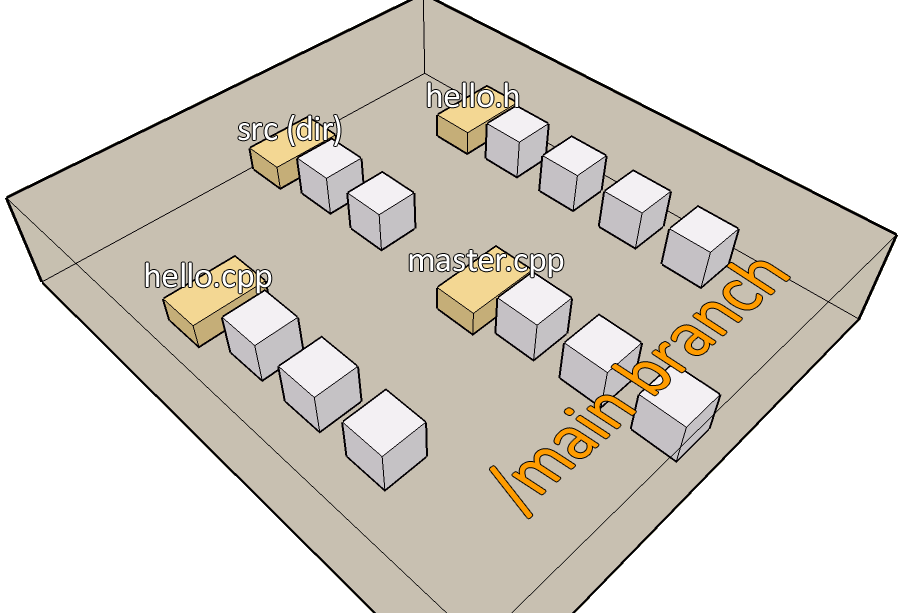


Figura 8: Representación de una rama

En la Figura 8: Representación de una rama se observa cómo la rama es un contenedor de los diferentes ítems, así como de sus revisiones (representados como cubos blancos debajo de los amarillos, que representan al ítem), determinando su evolución temporal. La historia de la estructura del proyecto queda determinada por las revisiones de los directorios en el repositorio. De esta forma, cuando un elemento se mueve de un directorio a otro, se crean revisiones del directorio origen y del destino, pero no se altera el propio ítem que se mueve. En el ejemplo de la figura, el elemento ‘master.cpp’ puede moverse del directorio ‘src’ a otro, y sus 3 revisiones seguirán siendo las mismas. En base a este principio Plastic SCM soporta de forma eficaz y transparente las operaciones de refactor, sin perder la historia de los ítems que se mueven o renombran.

Uno de los conceptos más importantes en Plastic SCM es su soporte para una jerarquía de ramas, y la selección de contendidos que se produce entre ellas. Este concepto es lo que se denomina Herencia de Ramas, y su comportamiento se describe en la figura siguiente:

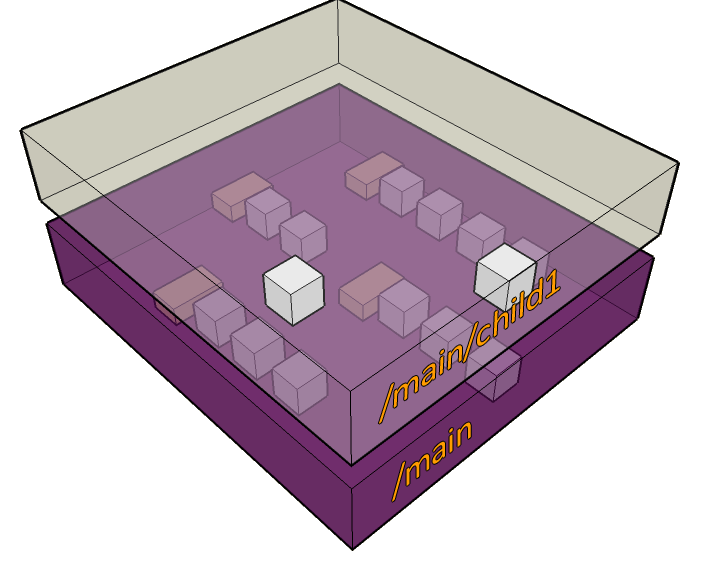


Figura 9: Representación de la herencia de ramas

Continuando con el ejemplo de la Figura 8: Representación de una rama, en la Figura 9: Representación de la herencia de ramas se muestra una rama hija de la rama principal (de ahí que se denomine ‘/main/child1’). Ésta rama contiene únicamente revisiones de dos ítems. Esto indica que sólo esos elementos (hello.cpp y master.cpp) han sido modificados en la rama, y por tanto la rama sólo contiene esas revisiones.

Todas las piezas encajan cuando se mapean revisiones al espacio de trabajo, que es donde el usuario le indica al servidor sobre qué rama (o ramas) desea trabajar. Si el usuario desea trabajar con la rama ‘/main/child1’, cuando el espacio de trabajo se actualice, para cada elemento, tomará la última revisión que se encuentre en la rama ‘/main/child1’ y, si no existiera revisión en esa rama, la de su rama padre, es decir, ‘/main’. Se puede ver una representación de cómo se mapean las revisiones del repositorio a las revisiones que se copian al espacio de trabajo en la figura siguiente:

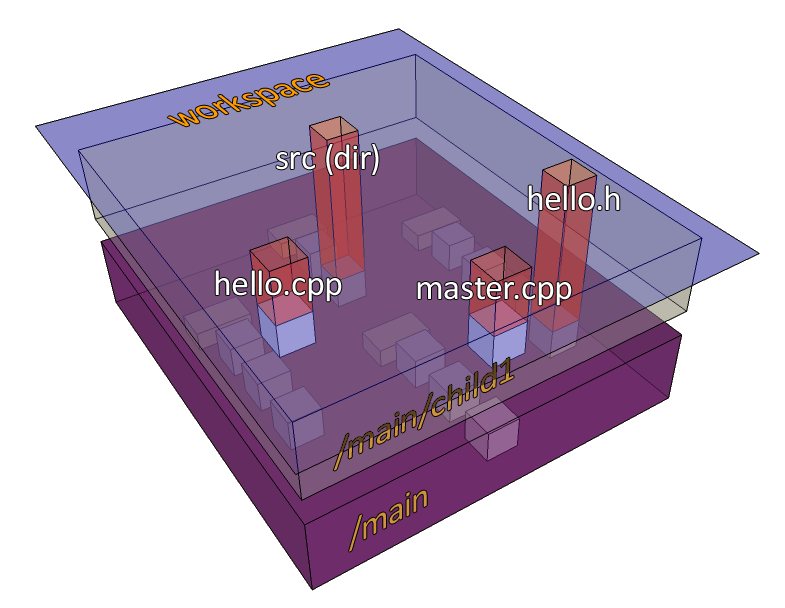


Figura 10: Cómo el espacio de trabajo carga unas revisiones en particular

El espacio de trabajo (workspace), en la Figura 10: Cómo el espacio de trabajo carga unas revisiones en particular debe seleccionar una revisión de cada ítem, y la regla es: “seleccionar la última revisión en la rama ‘/main/child1’”. Como la Herencia de Ramas está presente en todo momento, al final el contenido del espacio de trabajo es:

* master.cpp: se toma la última revisión en la rama /main/child1, como indica la regla.
* hello.h: se toma la última revisión en la rama /main/child1, como indica la regla.
* src: como no existen revisiones en /main/child1, se toma la última revisión en /main, por la herencia de ramas.
* hello.cpp: como no existen revisiones en /main/child1, se toma la última revisión en /main, por la herencia de ramas.

Un elemento puede tener revisiones en una jerarquía de ramas más complicada, permitiéndose múltiples niveles de herencia y, sobre todo, que varios de desarrolladores trabajen simultáneamente sobre los mismos elementos en ramas separadas, de manera que no interfieran entre sí. En secciones posteriores se describirá cómo volver a integrar estas ramas, de forma que los cambios de múltiples autores queden fundidos para producir *releases* estables del producto.

En Plastic SCM crear una rama no implica hacer una copia de todos los ítems del proyecto en esa nueva rama, ni siquiera se crean enlaces. Tan sólo se crea una entrada de rama, por lo que la creación de ramas es una operación extremadamente eficiente y que no afecta al tamaño del repositorio.

Las representaciones anteriores resultan útiles para introducir los conceptos de rama y espacio de trabajo al usuario que no las conoce. Sin embargo, para el trabajo diario se emplean otro tipo de representaciones más adecuadas, como son el árbol de versiones de un ítem concreto, o la jerarquía de ramas y sus relaciones que ofrece el explorador de ramas.

### El árbol de versiones

El árbol de versiones es la representación de la evolución de un único ítem. La Figura 11: El árbol de versiones en la estructura del repositorio muestra dónde encaja el árbol de versiones desde las representaciones que se han utilizado como ejemplo:

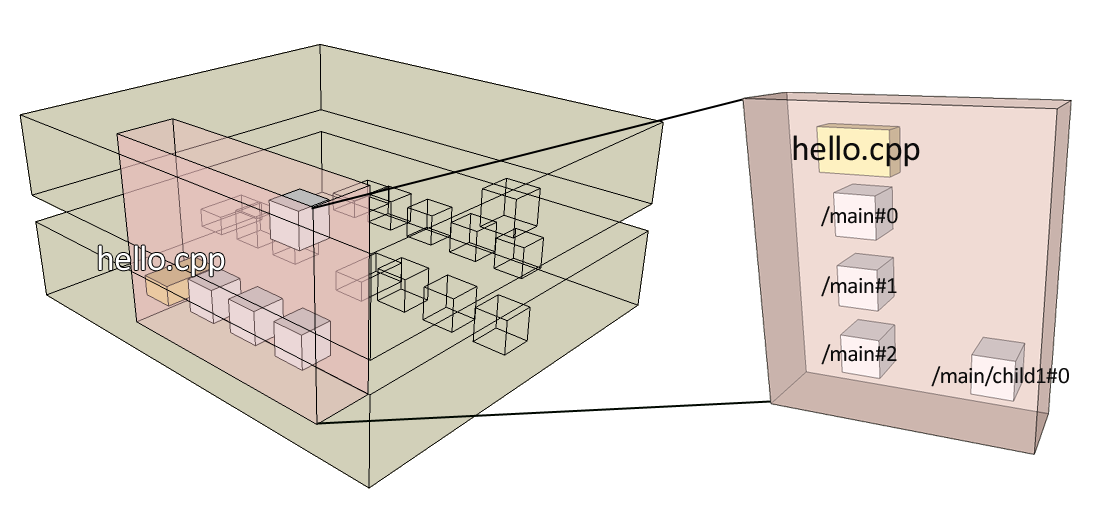


Figura 11: El árbol de versiones en la estructura del repositorio

Ese árbol de versiones, calculado por Plastic SCM, se muestra en la figura siguiente:

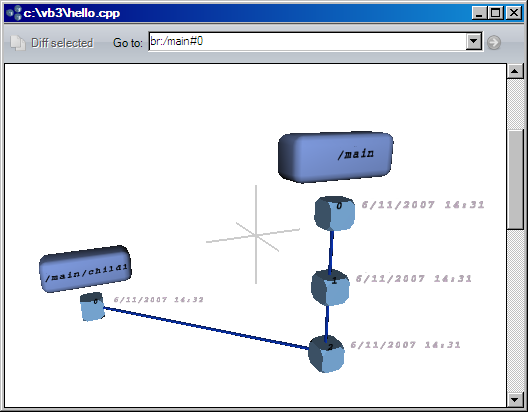


Figura 12: El árbol de versiones del ítem hello.cpp calculado por Plastic SCM

Debido a que el propio árbol de versiones de un ítem puede crecer a lo largo de muchas ramas, Plastic SCM ofrece una representación tridimensional del mismo, como se muestra en esta figura:

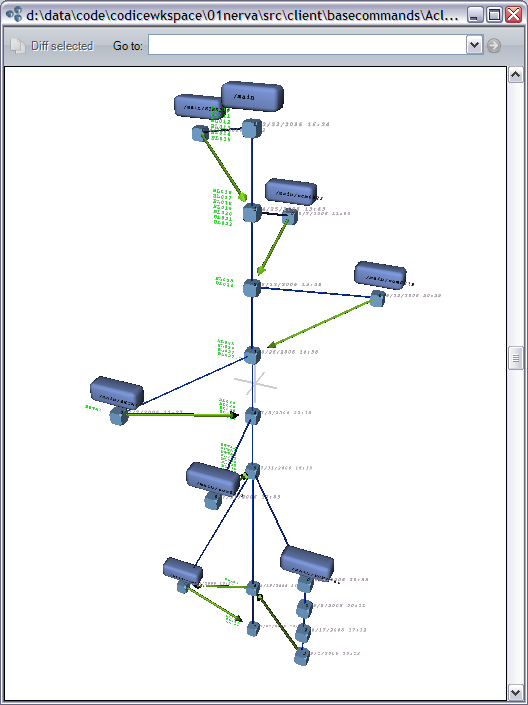


Figura 13: Ejemplo de árbol de versiones de un elemento

### El explorador de ramas

El explorador de ramas ofrece una visión temporal de la evolución de las ramas y sus relaciones. El explorador de ramas presenta una imagen global de la evolución del proyecto, frente a la visión mucho más localizada del árbol de versiones, en la que únicamente se muestra la historia de un ítem.

El explorador de ramas dibuja una representación de todas las ramas presentes en un repositorio, mostrando cuándo fueron creadas y qué periodo de vida han tenido, que es el intervalo de fechas en el que se han creado revisiones dentro de esa rama. Un ejemplo de este diagrama se encuentra en la figura siguiente:

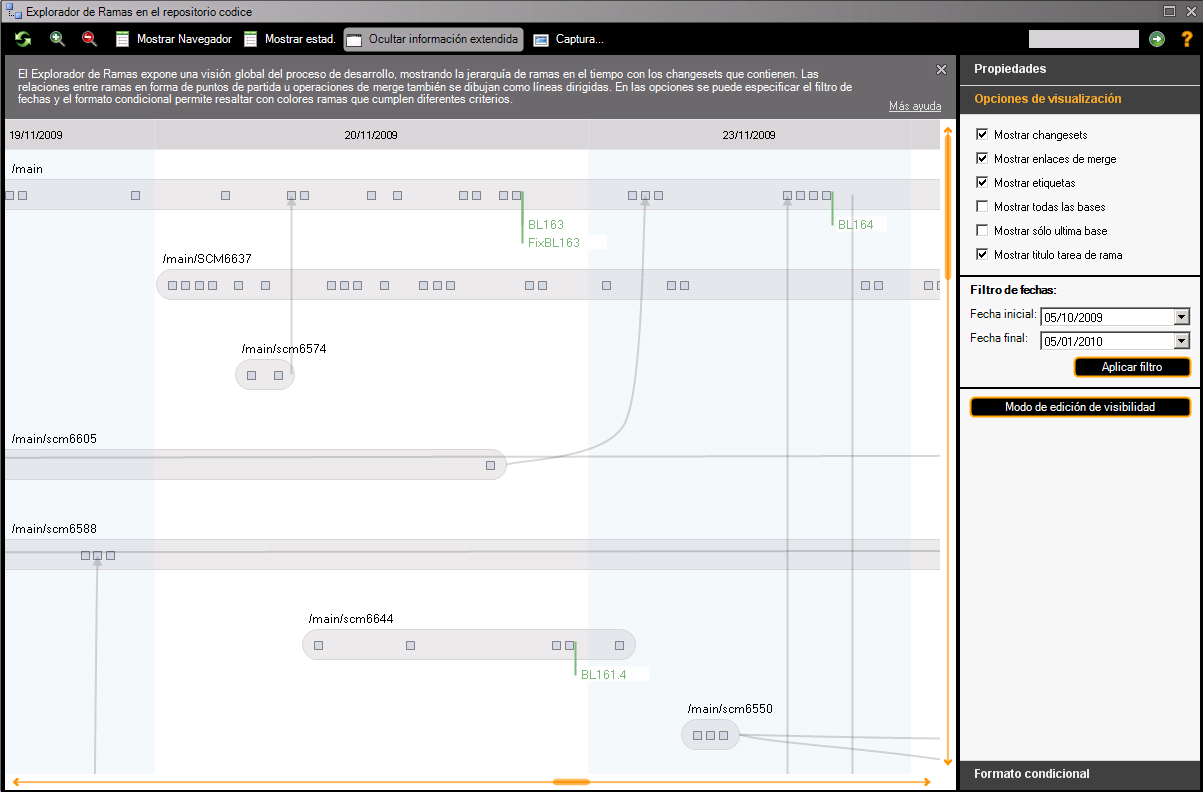


Figura 14: Ejemplo del explorador de ramas

En él aparecen las ramas como barras horizontales. Las columnas corresponden a fechas específicas en las que ha ocurrido algún evento (se ha creado la rama, es el último día en que una rama tiene revisiones, o hay una integración –merge- que se describirá en secciones posteriores). Las relaciones de parentesco entre ramas aparecen como líneas rectas desde el padre hasta el inicio de las ramas hijas. Las integraciones aparecen como líneas curvas. Para ayudar a determinar la dirección esta operación, las líneas de merge siempre parten horizontales del origen y van curvándose hasta el destino, donde terminan en vertical y en un círculo.

Cuando se selecciona un elemento éste cambia de color, y se resaltan sus relaciones con otras ramas, para ayudar a localizarlas más fácilmente.

## Ramas inteligentes (smart)

El soporte de ramas inteligentes aumenta las capacidades de herencia de ramas de Plastic SCM incluyendo la posibilidad de definir de manera dinámica las jerarquías de ramas.

La herencia de ramas básica está muy unida al *selector del espacio de trabajo*, por lo que dependiendo de la manera en la que se configure el selector, así se comportará la herencia de ramas.

Anteriormente, al introducer la herencia de ramas se decía que cuando un desarrollador está trabajando en una rama específica y una revision no está en la rama, Plastic cargará esa revisión de la rama padre. La pregunta aquí es: ¿qué revisión se utilizará de ese ítem? Esta pregunta la responderá el desarrollador al seleccionar una etiqueta como base de la rama en su espacio de trabajo a través del selector de espacio de trabajo. Pero las *ramas inteligentes* introducen una solución más sencilla: al crear una rama el usuario puede especificar el punto (etiqueta, changeset o rama) que se utilizará como base de la nueva rama, por lo que cada vez que Plastic necesite localizar una revisión de un ítem en concreto en la jerarquía de ramas, irá al lugar exacto.

Las ramas inteligentes suponen un paso más en la herencia de ramas de Plastic y una mejora en las ramas, de tal manera que son capaces de recordar qué rama específica, changeset o etiqueta es su padre en un momento concreto.

La Figura 15 muestra un ejemplo de scenario de ramas inteligentes en el cuál una misma rama tiene diferentes *bases* o *puntos de herencia* ya que va evolucionando en el tiempo.

La rama realmente *recuerda* cada una de las configuraciones por lo que es possible volver a un changeset en concreto y reconstruir completamente la configuración deseada.



Figura 15: Ramas definiendo la jerarquía dinámica de ramas

## Workspace (espacio de trabajo)

El espacio de trabajo, o workspace, es la zona sobre la que Plastic SCM puede mantener ítems bajo control de versiones. Sólo los ficheros y directorios que se encuentran en el espacio de trabajo son susceptibles de ser controlados por Plastic SCM.

A efectos prácticos, el espacio de trabajo no es más que otro directorio en el disco del desarrollador. Para que un usuario pueda empezar a trabajar con sus proyectos en el control de versiones, siempre necesita haber definido al menos un espacio de trabajo. En el espacio de trabajo Plastic SCM mantendrá **copias** de los diferentes ítems que componen un proyecto.

El espacio de trabajo siempre tiene una configuración que determina cuáles son sus contenidos, marcando qué versiones de los ítems se copian desde el servidor para trabajar sobre ellas. Normalmente indica a qué repositorio o repositorios se conectará, y para cada uno de esos repositorios, qué ramas y puntos de partida se emplean. Por ejemplo, esta configuración puede marcar una regla para copiar desde el servidor al espacio de trabajo un proyecto tal como estaba cuando fue entregado por primera vez al cliente (lo que podría ser una versión 1.0). La figura siguiente muestra cómo el espacio de trabajo selecciona un conjunto de versiones concreto, que son las que se copian desde el servidor.

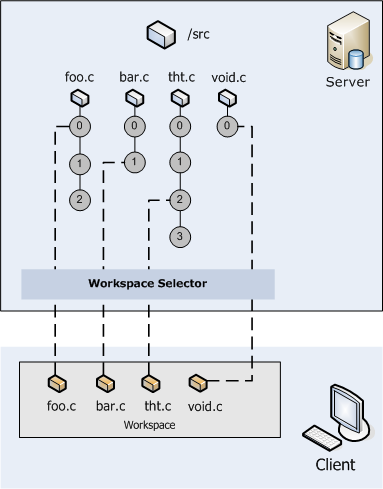


Figura 16. Cómo se mapean revisiones a workspaces

Cuando se crea un espacio de trabajo, se asume una configuración por defecto, que puede variarse más tarde desde los distintos clientes. Esta configuración por defecto dicta que el espacio de trabajo use el repositorio *default* (creado durante el primer arranque del servidor) y tome la última versión de sus contenidos.

Es posible, y de hecho habitual, que un desarrollador tenga más de un espacio de trabajo en la misma máquina e incluso repartidos entre varias máquinas, a menudo con diferentes configuraciones para trabajar de forma paralela en diferentes tareas.

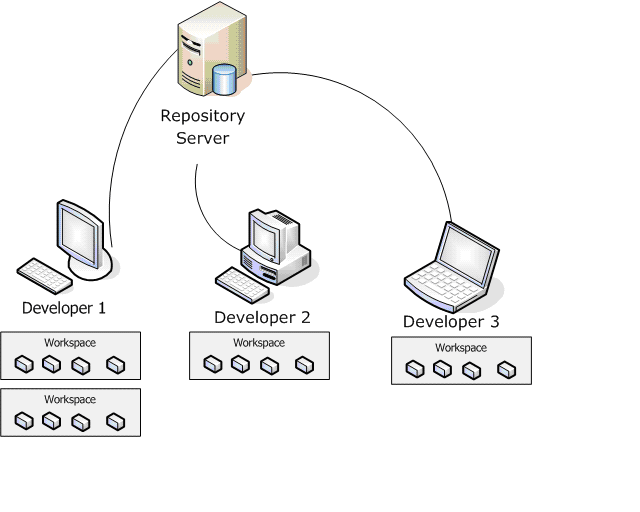


Figura 17. Cada desarrollador puede tener varios workspaces

Los espacios de trabajo en Plastic se almacenan localmente en el cliente, en una carpeta oculta *.plastic* situada en la raíz del espacio de trabajo que contiene los siguientes ficheros:

* *plastic.selector*, que almacena el contenido del selector que tiene el cliente en el momento actual.
* *plastic.workspace*, que almacena el nombre del espacio de trabajo y su GUID (identificador único de objeto).
* *plastic.wktree*, un fichero binario que indica las revisiones cargadas en el espacio de trabajo en este momento.

Además, en la carpeta local del usuario (en Windows XP Documents and Settings\<user>\Configuración Local\Datos de Programa\plastic) se almacena un fichero *plastic.workspaces* con información de los workspaces gestionados por el cliente de Plastic. El formato de dicho fichero es como sigue:

// Known Plastic SCM workspaces

// id name path

6fc4ed7d-d2ee-49c3-b9ec-980aab89f429 wkspace c:\wkspace

## Etiqueta (Label)

Una etiqueta es el modo de marcar revisiones para poder agruparlas según un cierto criterio, que normalmente fija el usuario. Cuando se aplica una etiqueta, se crea una instantánea de la situación de los ítems en el tiempo. Más tarde esa instantánea puede ser referenciada con facilidad para identificar ese momento específico. Una etiqueta es, en definitiva, un nombre más fácil de recordar que se asigna a un conjunto particular de revisiones.

Las etiquetas se aplican siempre a las revisiones de los ítems que se encuentren actualmente en el espacio de trabajo, y pueden aplicarse a todos los ítems, o sólo a algunos de ellos, según las necesidades.

La Figura 18 muestra dos labels: *Label 0* y *Label 1*. Ambas etiquetas se aplican a diferentes revisiones de diferentes ítems. La etiqueta *Label 0* simplemente marca la revisión 0 del primer ítem. Sin embargo *Label 1* se usa para agrupar juntas varias revisiones de diferentes ítems.

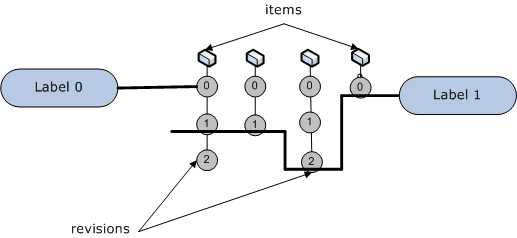


Figura 18. Etiquetas aplicadas a revisiones

Para usar una etiqueta, el primer paso consiste en crearla, indicando en qué repositorio y el nombre que usará.

Las etiquetas se usan por muchos motivos diferentes. Se pueden agregar a ciertas revisiones simplemente para resaltar algo, o pueden usarse para gestionar el proceso de desarrollo, ayudando a crear *entregas* (releases) y líneas base (baselines).

# Operaciones de Plastic SCM

En este capítulo se describirán las principales operaciones de control de versiones que Plastic SCM puede realizar. Estando familiarizado con otros SCM o sistemas de control de versiones, muchas de las operaciones resultarán familiares, quizá con algunos cambios de nombre.

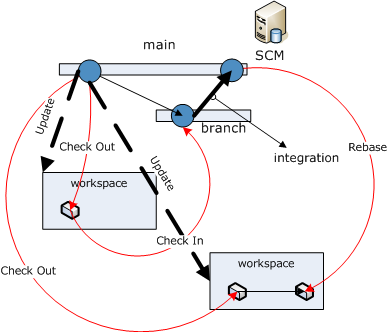


Figura 19. Operaciones básicas de control de versiones

Por cada operación se muestran algunos ejemplos de uso desde la interfaz de línea de comandos. Se puede obtener una ayuda más detallada de cada comando a través del modificador ‘--help’:

cm help

cm help *comando*

cm *comando* --help

## Creando un repositorio

Un repositorio llamado *default* se crea durante el primer arranque del servidor. Usando este repositorio los usuarios pueden empezar a trabajar más fácilmente ya que sólo necesitarán crear un espacio de trabajo. Aquí se creará uno nuevo y es el que se usará en el resto del ejemplo.

#### Desde la interfaz gráfica

1. En el menú Repositorios / Crear nuevo repositorio…
2. Introducir un nombre descriptivo, y el servidor en el que se creará.

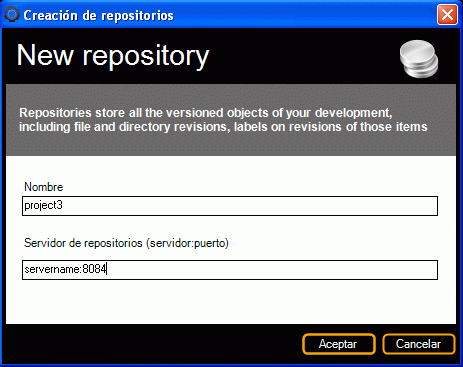


Figura 20: Diálogo de creación de repositorio

#### Desde la línea de comandos

cm mkrep servername:8084 project3

## Crear espacios de trabajo

La primera vez que un desarrollador arranca la herramienta GUI, si no se detecta ningún espacio de trabajo, aparecerá un diálogo de creación, como muestra la Figura 48: Creando un espacio de trabajo. Por defecto los espacios de trabajo apuntan al repositorio *default*, y a la rama *main*.

#### Desde la interfaz gráfica

1. En el menú Workspaces / Crear nuevo workspace.
2. Introducir un nombre descriptivo, la ruta y el servidor en el que se creará.

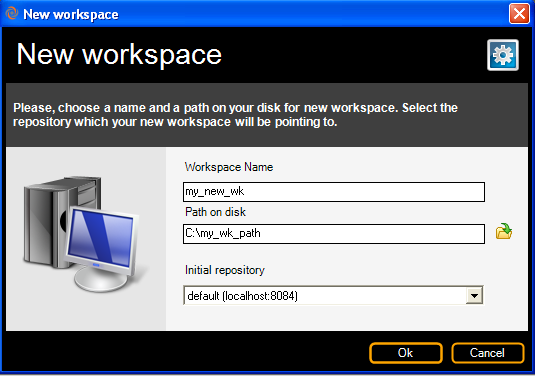


Figura 21: Creando un espacio de trabajo

#### Desde la línea de comandos

cm mkwk dave\_view c:\projects\view1

### Espacios de trabajo compartidos

Plastic SCM permite compartir espacios de trabajo mediante la utilización de carpetas o directorios compartidos, montados mediante UNC, unidades de disco virtuales o vínculos simbólicos (esto último solamente en sistemas basados en UNIX).

De este modo, los usuarios podrán manejar las revisiones cargadas en su espacio de trabajo desde cualquier máquina que pueda conectarse a la máquina donde se encuentra el recurso compartido.

Para utilizar esta funcionalidad, simplemente cree una carpeta (en terminología Windows) o directorio (en terminología UNIX) compartido y a continuación cree un espacio de trabajo en él. Posteriormente, desde otra máquina, monte el recurso compartido y trate de crear un espacio de trabajo en él. Plastic detectará automáticamente que el espacio de trabajo ya existe y simplemente lo conectará para que pueda utilizarlo como si fuera local. Cada cambio que lleve a cabo en el espacio de trabajo compartido estará disponible desde cualquier ubicación que utilice dicho espacio de trabajo.

**Aviso:** Para poder utilizar sin problemas el workspace compartido, debe cuidar especialmente las credenciales de usuario que utiliza para conectarse, ya que de otro modo puede obtener errores como: acceso denegado a la ruta, no tiene permisos para realizar dicha operación, etc.

## Add (Añadir)

Una vez que un repositorio ha sido creado (un repositorio denominado *default* se crea siempre durante la instalación), la primera operación será añadir ficheros y directorios al control de versiones.

La operación de *añadir* (add) le indica al sistema que un conjunto de ficheros y directorios serán controlados por Plastic SCM. La operación de añadir crea la información necesaria en el repositorio para que esos ítems puedan ser utilizados posteriormente.

Cuando se inicia una operación *add*, todos los ítems **se añaden en la rama que se encuentre configurada en ese momento en las reglas del selector**.

### Cómo ignorar determinados ítems

Plastic SCM puede ignorar determinados patrones de ficheros o directorios para que no sean añadidos al control de versiones cuando se realiza un add. Estos patrones se definen en un fichero llamado *ignore.conf*. Este fichero se puede colocar en varias ubicaciones y dependiendo de donde se ubique, diferirá su ámbito de aplicación.

* Si se coloca en la raíz del espacio de trabajo, los patrones de exclusión se aplicaran sólo en ese espacio de trabajo y no se tendrán en cuenta ninguno de los ficheros de exclusión que se encuentren en las demás ubicaciones.
* Si se coloca en el directorio de configuración de plastic de un usuario (leer más abajo), se tendrá en cuenta para todos los espacios espacios de trabajo de ese usuario (no se tendrá en cuenta para aquellos espacios de trabajo que ya tenga un fichero de exclusiones propio).

El contenido del fichero ignore.conf consta de un patrón por línea. En los patrones se pueden especificar caracteres comodín (\* y ?) con el mismo significado que en la línea de comandos. Las líneas que empiezan por el carácter # se interpretan como un comentario.

Cada entrada en este fichero especifica una ruta completa al ítem que se ha de ignorar. Si se desean ignorar todas las ocurrencias de una entrada, es necesario utilizar caracteres comodín.

Además de esto se pueden incluir excepciones a las reglas de ignorado; para ello la línea deberá ir precedida de una exclamación (!). Esto permite una mayor potencia en el manejo de ficheros de ignorados, permitiendo que solamente ciertos ficheros que cumplan una determinada condición sean añadidos o ignorados. Un poco más adelante se muestra un ejemplo de este caso.

El directorio de configuración de un usuario se encuentra en:

Para systemas Windows:

C:\Documents and Settings\user\Configuración Local\Datos de Programa\plastic

Para sistemas no Windows:

$HOME/. plastic

Este es un ejemplo del fichero *ignore.conf* file (El carácter # indica comentario):

#Patrones ignorados por Plastic SCM

\*/bin (ignora todos los directorios “bin”)  
  
\*processes.conf (ignora todos los ficheros ubicados en el workspace cuyo nombre es processes.conf)

\*.suo (ignora todos los ficheros con extensión .suo)  
  
!\*/result.\* (añade todos los ficheros cuyo nombre sea “result”, sin importar la extensión que tengan).

\*private\* (ignora todos los ficheros y directorios ubicados en el workspace que contienen “private” en sus nombres.)

**Notas:**

- Si se incluye una línea como “\*/bin” en el archivo ignore.conf, solamente se ignorarán los directorios /bin; cuando se trate de realizar un add recursivo, se tratará de agregar los ítems contenidos en /bin y Plastic SCM se quejará. La forma correcta de ignorar un directorio y todo su contenido es: \*/<dir\_name>\*; así por ejemplo: \*/bin\*.

- Pueden utilizarse paths en formato UNIX (“/”) o Windows (“\”) independientemente de la plataforma en la cual se está trabajando.

#### Tipos de items (binario/texto)

Plastic SCM reconoce automáticamente el tipo de un fichero, por su extensión. Pero algunas ocasiones, tipos de ficheros menos comunes son interpretados como contenido binario cuando en realidad son texto. El usuario puede asociar extensiones de fichero a un tipo fijo (binario o texto). Para ello ha de definir un fichero llamado *filetypes.conf* y colocarlo en el directorio de configuración del usuario (ver punto anterior)

Este es un ejemplo del contenido de un fichero *filetypes.conf* file. (El character # indica comentario):

#Tipos de fichero definidos para PlasticSCM.

#Sintaxis: <extension>:<type>. Type can be 'txt' or 'bin'.

#Examples:

.java:txt

.cpp:txt

.jpg:bin

.psd:bin

Tras una operación de añadir, tanto ficheros como directorios permanecen desprotegidos (checked out). Por tanto la operación *añadir* no provoca que los datos viajen a través de la red, simplemente añade *metadatos* al repositorio. La operación de *añadir* puede realizarse tanto desde la línea de comandos, como desde la herramienta GUI o desde las integraciones con los diferentes entornos de desarrollo (IDE). El último caso presenta la ventaja de que el entorno se encarga de añadir sólo los ficheros correctos, dejando fuera elementos temporales, configuraciones locales, binarios generados y elementos intermedios de compilación. En la figura siguiente, se han añadido algunos ítems desde Microsoft Visual Studio. Los elementos en verde con un punto rojo se encuentran añadidos y desprotegidos. Cuando se protejan aparecerán en color verde sin el punto.

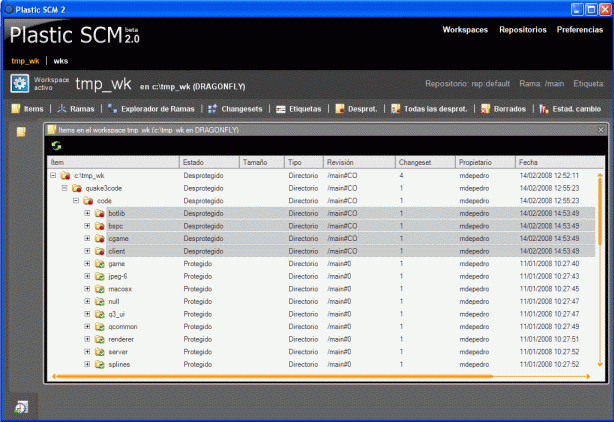


Figura 22: Elementos recién añadidos desde Visual Studio

#### Añadiendo ítems desde la línea de comandos

Para importar un proyecto en Plastic SCM, copie los contenidos dentro del directorio del workspace, y añádalos de manera recursiva con el comando:

cm add –R \*

Para añadir un elemento aislado, simplemente:

cm add *nombre\_del\_ítem*

Para añadir un elemento, haciendo checkout del padre de manera automática:

cm add –-coparent *nombre\_del\_ítem*

## Check in (proteger)

La operación proteger o check-in sirve para almacenar los cambios realizados en un ítem en el servidor. La protección siempre se realiza después de una operación de *check out* o de *add* (es necesario recordar que tras una operación de añadido, *add*, los ítems siguen en estado *check out*).

Durante la operación de protección el sistema comprobará si los datos del fichero o directorio han cambiado. Si el usuario intenta proteger un fichero o un directorio que no han sido modificados, el sistema emitirá un aviso. Por defecto será necesario hacer modificaciones antes de hacer *check in*. En caso de que se desee crear una revisión igual a la anterior, se podrá usar la opción de *check in forzado*.

La operación proteger crea la revisión en la misma rama en la que se especificó que ocurriría la desprotección, y es una operación atómica. Esto significa que la operación o bien se completa satisfactoriamente protegiendo todos los elementos que el usuario había indicado, o bien no se protege ninguno, quedando pendientes de protección.

Así se garantiza que un error en algún punto de la protección no crea un estado inconsistente, ya que todos los ítems que son protegidos en una única operación de check-in comparten un identificador denominado **número del conjunto de cambios** o **changeset number**. Este identificador es impreso al final de una protección tanto en la interfaz de línea de comandos como en la ventana de log de la interfaz gráfica. Este número es una fórmula más para agrupar cambios que Plastic SCM pone a disposición del usuario. Dado que normalmente un cambio (por ejemplo un bugfix) afecta a varios ítems, el changeset permite agruparlos.

Es posible ver qué ítems hay en un *changeset* concreto a través del sistema de consultas, descrito en una sección posterior.

#### Desde la interfaz gráfica

Cualquier ítem desprotegido se puede proteger desde su menú de contexto, tanto en una vista de ítems como en la vista de desprotecciones, seleccionando la opción “proteger”.

#### Protegiendo ítems desde la línea de comandos

Para proteger un ítem que se encuentra desprotegido (en checkout) desde la línea de comando, emplee el comando ‘checkin’, o abreviadamente, ‘ci’

cm ci *ítem\_desprotegido*

Es posible proteger recursivamente todo un directorio y sus contenidos, aunque la operación mostrará un aviso para aquellos que no están desprotegidos o no están bajo control de versiones

cm ci –R \*

Para proteger un ítem, desactivando la comprobación de contenido

cm ci --nchk *ítem\_desprotegido*

Para listar todos los ítems contenidos en un changeset:

cm find revision where changeset = número\_de\_*changeset*

## Update (obtener)

La operación de update[[1]](#footnote-2) descarga ficheros y directorios a un espacio de trabajo teniendo en cuenta sus reglas de selección. **Cuando se acaba de crear un espacio de trabajo nuevo, se hace un update para poblarlo por primera vez**. En condiciones normales, se hace un update para copiar al espacio de trabajo las últimas versiones de los ítems que corresponden a su configuración. Cuando se cambia la configuración del espacio de trabajo (alterando el selector, o cambiando la rama o etiqueta a las que apunta), se realiza un update para que el sistema calcule qué revisiones han de descargarse.

La operación de update recorre toda la estructura del proyecto según el selector actual, y calcula si necesita traer versiones diferentes de las que hay al espacio de trabajo. Se puede hacer que la operación de update siempre sobrescriba las versiones existentes en el espacio de trabajo, independientemente de sus contenidos, a través del modificador

cm update **–-forced** .

### Ítems modificados en el espacio de trabajo

En ocasiones, la operación de update puede mostrar un mensaje como el siguiente:

El ítem **xxx s**e ha modificado en el espacio de trabajo actual. No se sobrescribirá.

Esto es debido a que los contenidos del ítem en cuestión han sido modificados sin haber indicado a Plastic SCM que hiciera una desprotección, y la operación de update por defecto no sobrescribe el fichero, ya que podría contener cambios importantes. Si quiere que esos cambios formen parte del historial del ítem, desproteja el ítem y protéjalo con los cambios. Si sabe con certeza que no ha habido ningún cambio en el ítem que está dando el aviso, y quiere evitarlo en el futuro, puede forzar el update, a través del modificador --forced que se ha introducido en el apartado anterior, sobrescribiendo el ítem con la versión que le correspondería en todo caso.

### Ítems marcados como .unloaded

A consecuencia de la operación de update, puede ocurrir que en las nuevas versiones que se están cargando al espacio de trabajo se hayan eliminado items. En este caso, Plastic SCM sigue la regla de no borrar nada, por lo que en vez de borrar los elementos que se perderían, los renombra con la extensión .unloaded con lo que se convierten en elementos privados. De esta manera el contenido no se pierde, por si hay algo importante, pero con un nombre diferente para que no entren en conflicto con otras herramientas.

### Resguardo automático de desprotecciones (Checkout shelving)

Plastic SCM introduce una característica muy conveniente cuando se trabaja con varias ramas en el mismo espacio de trabajo. En el caso de que tras un cambio del selector de un espacio de trabajo una operación de update necesite sobrescribir un ítem que se encuentra desprotegido, Plastic SCM salvará automáticamente el contenido de esa desprotección en el servidor antes de sobrescribirla, de forma que sus contenidos son preservados. No implica el que se haga una protección ni se cree una nueva revisión como tal. Más tarde esos ítems pueden ser recuperados.

Imagine que está trabajando sobre la rama ‘/main/tarea4’, en la que tiene algunos ítems desprotegidos.

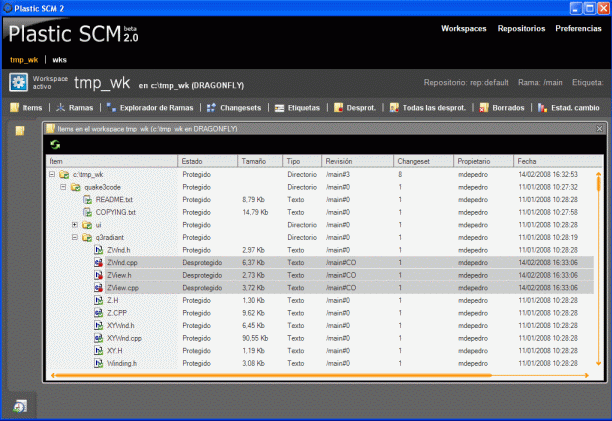


Figura 23: Ficheros desprotegidos antes del update

Si cambia la configuración de su espacio de trabajo para que apunte a ‘/main/tarea6’, la operación de update normalmente traerá las versiones que están en main de los ficheros que usted había modificado en ‘/main/tarea4’, ya que esos ficheros no están en ‘/main/tarea6’. Esta operación se ve reflejada en el log.

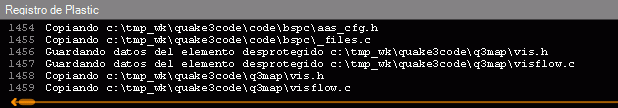


Figura 24: Protección de ficheros en check out

Plastic SCM guardará esas desprotecciones en el servidor mientras usted continúa trabajando en ‘/main/tarea6’. Puede comprobar que sus desprotecciones aún están pendientes a través de la vista de ‘protecciones pendientes’ en la interfaz gráfica. Aparecen con un signo naranja, indicando que son desprotecciones que no están en el espacio de trabajo.

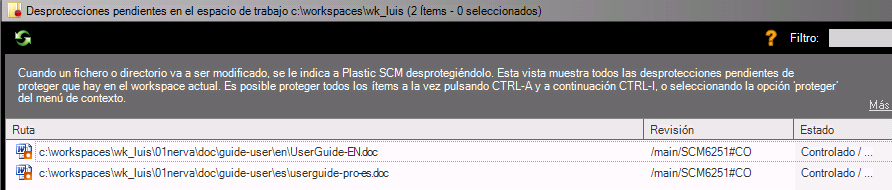


Figura 25: Ficheros desprotegidos en la rama

Cuando vuelva a apuntar su espacio de trabajo a ‘/main/tarea4’, la operación de update recuperará los contenidos de sus desprotecciones y las indicará como tal, ahora en verde con un punto rojo, porque ya se encuentran en el espacio de trabajo.

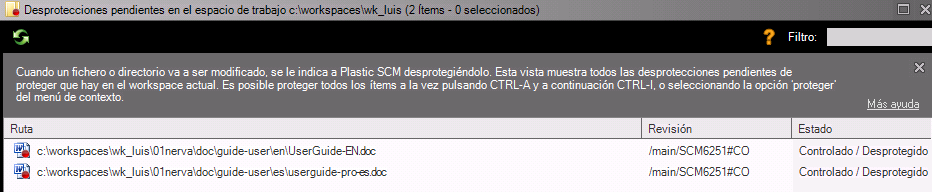


Figura 26: Recuperación de contenidos del check out

### Update paralelo

La operación de update emplea varios hilos o threads. Mientras que este modo aumenta el rendimiento en la mayoría de escenarios, existen unos pocos en los que puede resultar interesante emplear un solo hilo para reducir, por ejemplo, la cantidad de conexiones que realiza el cliente, quizá en entornos con poco ancho de banda entre cliente y servidor. Para llevar a cabo una operación de update que no emplee varios hilos se emplea el modificador –noparallel desde la interfaz de línea de comandos.

cm update . **–noparallel**

### Cloaked ítems

Es posible especificar que no se desea actualizar determinados ficheros o directorios cada vez que se camba la configuración del espacio de trabajo. Hay diversas razones para querer hacer esto:

* Archivos que se modifican con muy poca frecuencia;
* Archivos muy grandes que requieren un tiempo considerable para ser descargados;
* Archivos que no afectan a los procesos de trabajo habituales.

Ejemplos de estos casos serían programas de testing, programas de terceros o ficheros binarios de diversa índole con propósitos determinados.

De este modo puede lograr una ganancia en el tiempo de actualización del espacio de trabajo y por consiguiente trabajar de un modo más cómodo.

Para indicar qué ficheros/directorios debe Plastic ignorar, debe crear un fichero *cloaked.conf*, que tendrá una sintaxis muy similar a la del fichero *ignore.conf* (consulte la sección de Add en este mismo manual):

#File patterns cloaked by Plastic SCM  
  
\*/bin (cloak todos los directorios bin)

\*processes.conf (cloak todos los ficheros con nombre processes.conf del espacio de trabajo)

\*.suo (cloak todos los ficheros con extensión .suo)  
  
!\*/result.\* (excluye del cloak a todos los ficheros cuyo nombre es “result”, la extensión del fichero no importa)

\*private\* (cloak de todos los ficheros que contienen “private” en su nombre, que estén en el espacio de trabajo)

El fichero *cloaked.conf* puede ser colocado en el directorio local del usuario (esto es, /home/<user>/.plastic en Linux o C:\Documents and Settings\<user>\Configuración Local\Datos de programa\plastic en Windows XP o C:\Users\<user>\AppData\Local\plastic en Vista) o en la raíz del espacio de trabajo. Si se coloca en la primera ubicación indicada, las reglas del fichero se aplicarán a todo el espacio de trabajo; en el segundo caso se aplicará solamente al espacio de trabajo donde el fichero está colocado.

De ese momento en adelante, cuando abra la interfaz de Plastic GUI, verá los elementos cloak indicados en la vista de ítems; si realiza un update del espacio de trabajo dichos ítems serán ignorados; si se hace un checkout recursivo de un directorio que contiene elementos cloak, dichos elementos serán saltados.

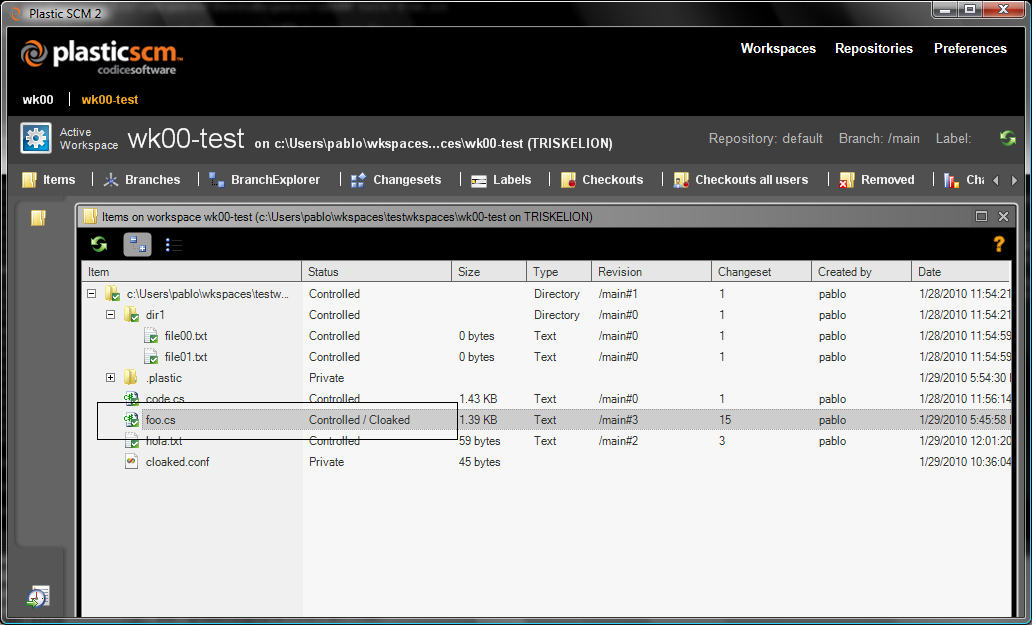


Figura 27: Ejemplo de ítems ‘cloaked’

## Check out (desproteger)

La operación desproteger sobre un ítem indica al control de versiones que ese ítem va a ser modificado. Se desprotegen los ficheros cuando se van a hacer cambios en su contenido, y se desprotegen los directorios cuando se van a añadir, mover o eliminar ítems que contienen.

Cuando un ítem se desprotege, aparece la marca ‘CO’ (de check out) en la interfaz de línea de comandos, y un punto rojo en el icono del ítem en el caso de la interfaz gráfica. En caso de que el espacio de trabajo apunte a una rama hija (por ejemplo, /main/tarea4, hija de /main) y no exista previamente una revisión de ese ítem en esta rama hija, la desprotección creará la primera en esa rama, **desde la revisión actual en el espacio de trabajo**. En el siguiente ejemplo se han desprotegido los ítems Form1.cs y Form1.Designer.cs, que se encontraban en la rama /main, con el espacio de trabajo apuntando a la rama /main/tarea4:

11/06/2007 19:46 dir br:/main#0 .  
11/06/2007 19:46 txt br:/main/tarea4#CO CO Form1.cs  
11/06/2007 19:47 txt br:/main/tarea4#CO CO Form1.Designer.cs  
11/06/2007 19:46 txt br:/main#0 Program.cs  
11/06/2007 19:46 dir br:/main#0 Properties  
11/06/2007 19:46 txt br:/main#0 WindowsApplication1.csproj

Cuando un fichero es *desprotegido* el atributo de *sólo lectura* es eliminado, de modo que el usuario puede escribir sobre él. Este atributo es restablecido cuando más tarde se proteja de nuevo.

#### Desde la interfaz gráfica

Cualquier ítem protegido se puede desproteger desde su menú de contexto desde la vista de ítems, seleccionando la opción “desproteger”.

#### Desprotegiendo ítems desde la línea de comandos

Para desproteger un ítem cualquiera:

cm co *nombre\_del\_ítem nombre\_de\_otro\_ítem*

Para desproteger de manera recursiva un directorio y todos sus contenidos:

cm co –R directorio/\*

Desproteger todos los ítems leídos desde la entrada estándar:

cm co –

#### Desde la integración con Visual Studio

La integración con Visual Studio desprotege de manera automática los ítems en el momento en que empiezan a modificarse. Cuando un ítem está desprotegido, el candado al lado de ese ítem en el Explorador de Soluciones se reemplaza por una marca de edición.

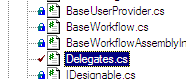


Figura 28: Detalle de elemento desprotegido en Visual Studio

Visual Studio puede configurarse para preguntar antes de desproteger un ítem, desde el menú de opciones. Cuando esta preferencia esta activa, al desproteger un ítem aparece este diálogo

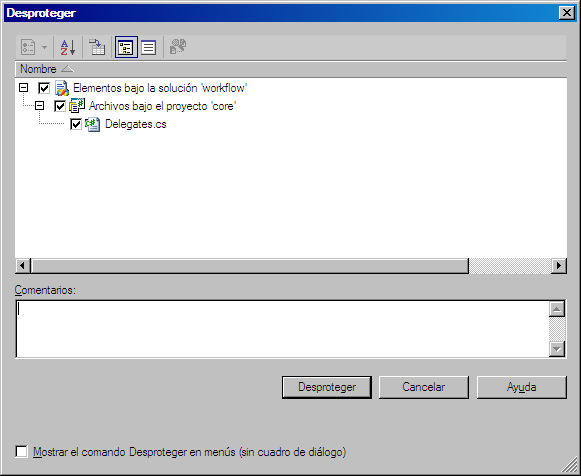


Figura 29: Diálogo de desprotección de Visual Studio 2003

#### Desde la integración con Eclipse

Por defecto el Plugin de Eclipse está configurado para desproteger los elementos de manera automática cuando empiezan a modificarse, o al hacer una operación de refactor.

#### Desde la integración con IntelliJ

Consulte el capítulo que explica la integración con IntelliJ para más información.

### Detalles internos de la operación de check out

A nivel interno, desproteger un elemento tiene el efecto de crear una nueva revisión especial y vacía en el servidor. La Figura 30 muestra cómo esta revisión se crea cuando se ejecuta una operación de check out. Al hacer posteriormente la *protección* o check in, la revisión se *convertirá* en una revisión convencional que tendrá datos asociados.

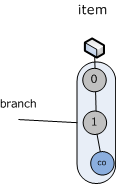


Figura 30. Creación de una revisión durante la operación de check out

El árbol de versiones tridimensional muestra las desprotecciones como un cubo morado, frente a los cubos azules que representan revisiones normales.

Este mecanismo hace posible trazar dónde se están realizando las desprotecciones de cada usuario y es la base para el *check out shelving* que se ha descrito en el apartado dedicado al update.

## Viendo las protecciones pendientes

Plastic SCM controla qué ítems se encuentran desprotegidos para cada usuario en cada espacio de trabajo, y es posible listarlos desde los diferentes clientes.

#### Desde la interfaz gráfica

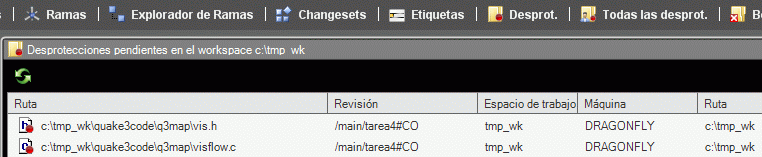


Figura 31. Listar las revisiones *desprotegidas*

#### Desde la línea de comandos

Mostrar los ítems desprotegidos del espacio de trabajo actual

cm findcheckouts  
cm fco

Mostrar los ítems desprotegidos de todos los espacios de trabajo

cm fco --all

Mostrar sólo el nombre de los ítems desprotegidos sin información de ramas o de otro tipo

cm fco –format={4}

## Uncheckout (cancelar una desprotección)

Cuando se ha desprotegido un ítem, es posible cancelar esa desprotección. Internamente, la operación *uncheckout* realiza una cancelación del checkout, y un update de ese ítem. Cuando se trata de un fichero, esto implica que se restaura el fichero a la revisión que selecciona reglas del espacio de trabajo. Cuando se trata de un directorio ocurre lo mismo, aunque en este caso sólo se actualizan los nombres de los ítems que contiene, es decir, se actualizan los posibles renombrados, y se manejan los borrados o añadidos consecuencia de la operación de update a la revisión que corresponde.

#### Desde la interfaz gráfica

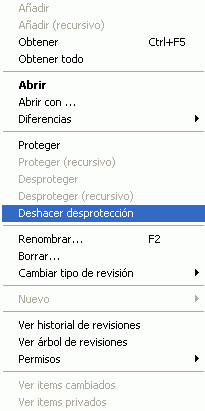


Figura 32: Cancelar una desprotección desde la línea de comandos

#### Cancelando desprotecciones desde la línea de comandos

Cancelar la desprotección de un ítem cualquiera:

cm unco nombre\_del\_ítem

Cancelar las desprotecciones de todos los elementos pendientes de protección (cancelar todos los checkouts en el espacio de trabajo actual:

cm fco --format={4} | cm unco –

El comando ‘findcheckouts’ o, abreviadamente ‘fco’, imprime información de los checkouts del espacio de trabajo actual, o de todos los espacios de trabajo. El modificador *--format* permite especificar el formato de lo que se imprimirá, en este caso {4} indica que sólo se imprima el nombre del ítem. El – le indica al comando uncheckout que lea los ítems desde la entrada estándar.

## Crear una rama

Como ya se ha mencionado con anterioridad, Plastic SCM crea por defecto una rama llamada ‘/main’ en cada repositorio. Para poder usar otras ramas, es necesario crearlas primero.

#### Crear una rama desde la interfaz gráfica

Desde la vista de ramas, se selecciona la rama padre en la que se creará la nueva rama, botón derecho y “Crear rama hija”:

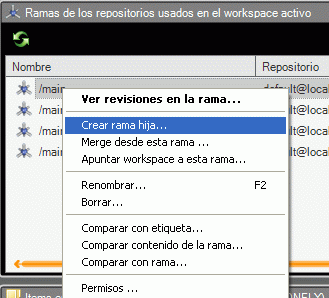


Figura 33: Creando una rama hija desde la interfaz gráfica

#### Crear una rama desde la línea de comandos

Las ramas son creadas con el comando ‘mkbranch’ como en el siguiente ejemplo, que crea una rama llamada ‘child1’ cuyo padre es la rama ‘/main’.

cm mkbranch br:/main/child1

Las barras ‘/’ separan los niveles de la jerarquía de ramas. Una rama tiene una sola rama padre y puede tener varias ramas hijas.

## Merge (integración)

La operación de merge o integración es la encargada de mezclar contenidos que han evolucionado por separado en Plastic SCM. Por ejemplo, cuando dos desarrolladores han modificado los ítems en ramas separadas y se pretende juntarlas de nuevo o cuando dos desarrolladores trabajan sobre la misma rama pero han modificado el mismo fichero por separado y hay que reconciliar los cambios de ambos.

La operación de merge es tradicionalmente una de las más complejas en los sistemas de control de versiones. Sin embargo, Plastic SCM trata de mantener esta complejidad alejada del usuario al máximo, resolviendo internamente tantos conflictos como le es posible.

**Una operación de merge comprobará qué diferencias hay desde una rama (que se denomina origen) con los contenidos actuales del espacio de trabajo (que será el destino).** Si el origen del merge es una rama, se tomarán las últimas versiones de los ítems en la rama para calcular cuáles de ellos necesitan ser mezclados. En la operación de merge, por tanto, siempre se elige un **origen**, y se compara con el contenido actual del espacio de trabajo, que se denomina **destino**. En Plastic SCM, el destino siempre es el contenido (las versiones de los ítems) que se encuentran en el espacio de trabajo.

Plastic SCM determinará con precisión qué ítems han sido modificados en el origen respecto al destino, y propondrá una lista con ellos. Un ejemplo de la lista de candidatos a merge puede verse en la figura siguiente:

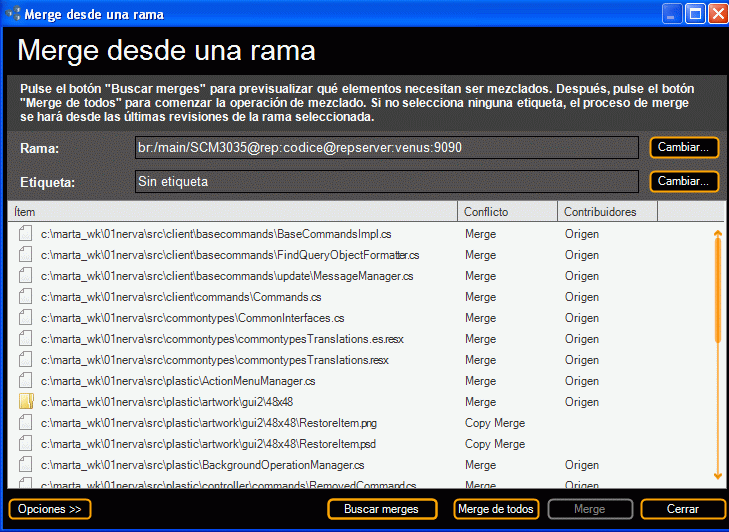


Figura 34: Ejemplo de ítems que necesitan merge

En esta lista aparecen los nombres de los ítems que necesitan merge, tanto ficheros como directorios, el tipo de conflicto y los contribuidores implicados. Las secciones siguientes explicarán cada uno de estos puntos en detalle.

### Tipos de conflicto de merge

Cuando se está realizando una operación de merge, el objetivo es reconciliar cambios que han ocurrido a la base de código por caminos separados. En este contexto se pueden dar dos tipos de conflictos:

* Copy merge: el ítem se ha añadido por primera vez al control de versiones en el origen, y el merge no necesita comprobar los contenidos, tan sólo copiarlo al destino.
* Merge: el ítem ya existía tanto en el origen como en el destino de la operación de merge, y hay que reconciliar cambios. En este caso, se examina de dónde vienen los cambios, que es lo que indica la columna contribuidores.

### Contribuidores

Los contribuidores son las revisiones del ítem que intervienen en el *merge*. Siempre hay un origen del merge, que es la revisión de la rama que se está integrando. Un destino, que es la revisión que hay en el espacio de trabajo actual, y una base, que es la revisión de la que partieron originalmente las dos anteriores. La siguiente figura muestra de dónde viene cada revisión:



Figura 35: Revisiones implicadas en una operación de merge

En esta figura, el espacio de trabajo del usuario apunta a la rama br:/main y se pretende incorporar a esa rama los contenidos de la rama br:/main/child1. Los contribuidores serán:

* Origen (source): /main/child1#1
* Destino (destination): /main#5
* Base: /main#4

En la Figura 35: Revisiones implicadas en una operación de merge también se representa que el resultado de la operación de merge creará una nueva revisión /main#6, una vez se haga el merge, y se proteja el resultado.

Teniendo claros estos conceptos, la columna ‘Contributors’ del diálogo de merge puede tener éstos valores:

* Source: los cambios vienen únicamente del origen. No hay revisiones nuevas en el destino después de la revisión base. Este tipo se resuelve de manera automática aplicando todos los cambios del origen en el destino.
* Destination: los cambios vienen únicamente del destino. Es el caso opuesto al anterior, y también se resuelve de manera automática.
* Both (ambos): han aparecido revisiones después de la base tanto en la rama de origen como en la de destino. Este tipo de merge puede ser automático o necesitar la intervención del usuario, según éstos criterios:
  + Si se trata de un fichero y es de texto, los conflictos se resolverán automáticamente cuando los cambios ha ocurrido en zonas que no se solapan entre sí. Si se han cambiado líneas coincidentes en ambas revisiones, aparecerá la herramienta gráfica de merge, que permite reconciliar los cambios. Esta herramienta se describe en una sección posterior.
  + Si se trata de un fichero de tipo binario, aparecerá en todo caso la herramienta de merge binario, que permitirá elegir cuál de las revisiones implicadas se guarda como resultado del merge. Es posible definir herramientas de merge asociadas a diferentes extensiones de ficheros en la configuración de cada cliente.
  + Si se trata de un directorio, el sistema intentará reconciliar los cambios que han ocurrido en las dos revisiones de origen y destino. Normalmente esta reconciliación es automática, aunque existen cuatro escenarios en los que se requerirá la intervención del usuario, al que se presentará un asistente para reconciliar los cambios. Excepto estos cuatro casos, todo el merge de directorios ocurre de forma automática.
    - Dos ítems se han renombrado al mismo nombre en el origen y el destino
    - Un ítem se ha renombrado de manera diferente en el origen y el destino
    - Se han añadido dos nuevos ítems con el mismo nombre en el origen y en el destino
    - Un ítem se ha renombrado con el mismo nombre con el que se ha añadido uno nuevo en origen o destino.

### Merges desde la misma rama

Es posible que un merge sea necesario usando la misma rama como origen y destino. Esto sucede cuando un fichero ha sido modificado desde dos espacios de trabajo diferentes apuntando a la misma rama.

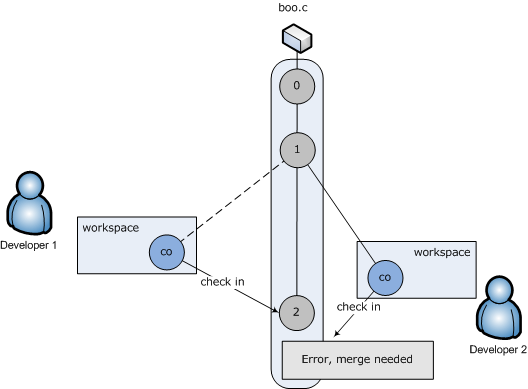


Figura 36. Situación que necesitará un merge desde la misma rama

En este caso será necesario que ‘developer2’ primero mezcle sus cambios con los que hay en la rama, introducidos por ‘developer1’ y de los que ‘developer2’ no ha sabido nada hasta la hora de proteger su trabajo.

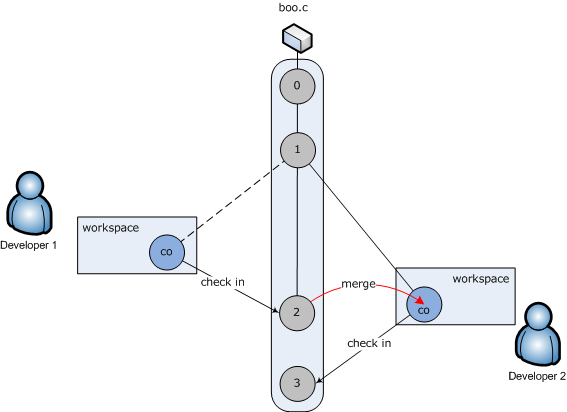


Figura 37. Ejemplo básico de merge, merge terminado

Una vez que ‘developer2’ ha completado el merge de los ítems necesarios, puede proteger sus cambios manteniendo la integridad de la base de código.

### Merges desde ramas diferentes

El caso de merge más habitual es, no obstante, aquel en que la rama de origen es diferente de la rama de destino (que es a la que apunta el espacio de trabajo). En la figura siguiente se muestra un posible escenario, en el que dos desarrolladores han creado ramas para trabajar en paralelo (/main/branch001 y /main/branch002) partiendo de un mismo punto en la rama principal (la revisión /main#1). La figura representa únicamente un ítem en particular (boo.c), si bien es corriente al trabajar con ramas que se creen revisiones de numerosos ítems.

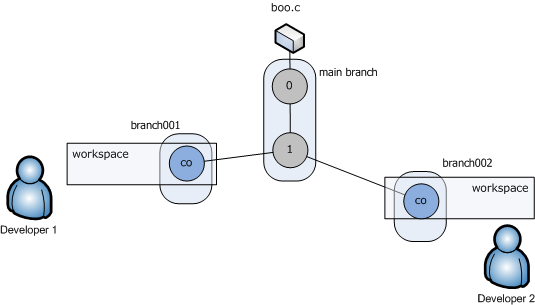


Figura 38: Escenario de desarrollo paralelo

La Figura 39. muestra cómo los desarrolladores van trabajando por separado, haciendo modificaciones sobre el ítem, cada uno en su rama, sin interferir con el otro. Pueden hacer tantas protecciones intermedias como deseen ya que sus cambios sólo son vistos por ellos mismos.

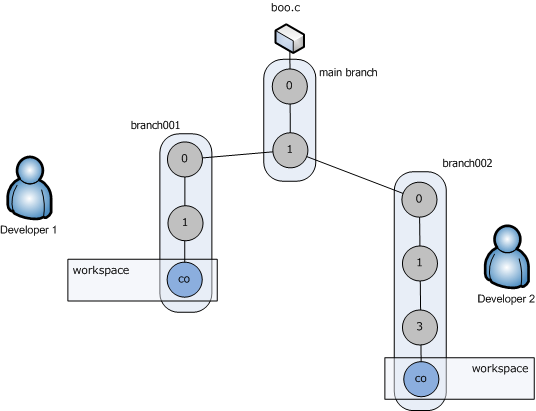


Figura 39. Los desarrolladores evolucionan las revisiones

Cuando el desarrollador termina sus modificaciones, los cambios están listos para ser integrados en la rama principal, de manera que serán visibles por los demás desarrolladores.

A la hora de hacer la integración, se parte de un espacio de trabajo que apunta a ‘br:/main’ (ya que el destino de la operación de merge siempre es el contenido actual del espacio de trabajo), y se incorporan las ramas una a una. De esta forma se pueden incorporar tantas como se desee. El resultado para el ítem ‘boo.c’ aparece representado en la figura siguiente.

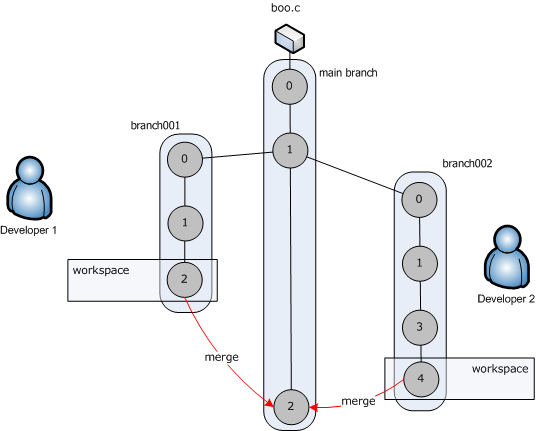


Figura 40: Las dos ramas han sido integradas

En esta representación no se ha hecho checkin después de cada operación de merge, por lo que todos los merges han ido a la misma revisión (en este caso, /main#2).

### Merge desde una rama y una etiqueta

Aunque posiblemente el caso de merge más habitual sea desde ramas distintas, cuando se vio se describía como los últimos cambios de la rama origen eran incorporados en el espacio de trabajo. En ocasiones puede que no interese hacer el merge del último contenido de la rama sino de aquel que previamente se ha etiquetado. Esto sucede cuando deseamos incorporar los cambios de una rama en la que se ha seguido, o se sigue, trabajando y queremos un contenido concreto que se sabe que funciona justo como se quiere.

Para conseguir este se puede particularizar el merge de una rama, para que en lugar de tomar la última revisión de cada elemento coja una revisión concreta que se encuentra etiquetada en la rama.

La figura representa únicamente un ítem en particular (boo.c), sobre el que se ha seguido trabajando en la rama y del cual queremos recuperar la versión etiquetada.

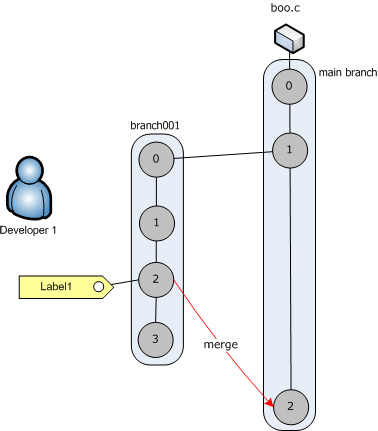


Figura 41: La revisiones etiquetadas en la rama han sido integradas

Tal y como puede verse, la revisión que es incorporada en el merge no es la última de la rama, cómo ocurría en el caso de del merge desde ramas diferentes, sino que es la revisión que estaba etiquetada.

### Merge desde una etiqueta

Hasta ahora hemos visto como incorporar los cambios de una rama, en sus distintas versiones, en el espacio de trabajo. Puede darse el caso que interese incorporar los cambios que previamente han sido etiquetados con independencia de en que rama se encuentren dichos cambios. La diferencia entre este caso y el merge desde una rama y una etiqueta es que mientras en el anterior sólo se cogían aquellos elementos etiquetados en esa rama, aquí se cogen todos los elementos con esa etiqueta con independencia de la rama en la que se encuentren.

Trabajando con múltiples ramas, puede resultar muy útil cada vez que se genera una nueva release. Cuando se genera una release, es aconsejable que sea etiquetada para poder volver a ella. Tras concluir la release y una vez esta probada y etiquetada, aquellos desarrolladores que estén trabajando deberán incorporar los cambios realizados en la release en su rama. Para hacerlo se hará un merge desde la etiqueta de la release.

 A esta operación que consiste en actualizar el contenido de la rama de trabajo para qué deje de basarse en la release antigua y pase a basarse en la nueva release se le denomina rebase.

También puede interesar incorporar un cambio concreto que debido a su importancia u a otras razones ha sido etiquetado.

### Merge desde un changeset

Cuando se describió el funcionamiento de la operación de check in, se habló de que todos aquellos elementos que se protegían al mismo tiempo recibían un identificador denominado changeset. Por medio te este changeset podríamos hacer referencia a todos aquellos elementos que formaban parte de un cambio.

Gracias a que el cambio puede ser identificado de esta manera es posible incorporar ese cambio si se hace un merge de ese changeset. Al igual que en el caso del merge desde una etiqueta, no se depende de la rama donde se encuentra el contenido del changeset. Dado que lo único que necesitamos para acceder a todos los elementos que forman ese cambio sin ambigüedad es el changeset.

 Para que el merge desde un changeset tenga sentido, cuando se realice un cambio (se creo nueva funcionalidad, se corrigió,…) todos los elementos que formaban parte de él deben de protegerse a la vez y no ir protegiendo los distintos elementos que los componen de forma separada. Si un cambio está dividido entre varios changeset sería necesario hacer el merge de todos y cada uno de ellos, por lo que esta opción pierde su utilidad.

### Merge desde una revisión

Hasta ahora siempre que se ha hablado del origen del merge este contenía más de un elemento, o al menos podía contenerlos. Si sólo se quiere incorporar los cambios de un elemento, en concreto el contenido que tenía un elemento en una revisión concreta se emplearía este método.

A diferencia de los anteriores con este método sólo se incorporará el contenido de la revisión indicada por el usuario. Por tanto será el usuario el que decida que elemento ha de ser mezclado en lugar de ser el sistema el que los busque en base a alguno de los criterios dado (changeset, rama,…).

Si se quisiese hacer de más de un elemento, o se usa uno de los métodos anteriores, o se hace el merge de manera individual para cada uno de esos elementos. Por lo tanto para hacer el merge de 2 elementos habría que hacer el merge dos veces de forma independiente, para 3 elementos 2 y así sucesivamente. Salvo que el merge sea de un elemento o de muy poco elementos, o bien que los elementos de los que se desea hacer el merge no este relacionados de ninguna manera, éste método no es el más conveniente.

#### Haciendo merge desde la línea de comandos

La operación de merge siempre toma como destino el contenido del espacio de trabajo. Por eso es interesante hacer una operación de update antes de comenzar con un *merge* para asegurar que los contenidos del espacio de trabajo son los apropiados.

Para actualizar el contenido del espacio de trabajo a las versiones que deben tener, se usa este comando, desde la raíz del espacio de trabajo:

cm update .

El comando ‘mergebranch’ toma como argumento principal una *especificación de rama*. Aunque las especificaciones de objetos del control de versiones se describen en una sección posterior, sirva como adelanto que las especificaciones de rama tienen la forma:

br:/nombre\_de\_rama[#especificación\_de\_etiqueta]

Para ver los ítems que necesitan ser mezclados desde la rama ‘/main/child1’ al contenido actual del espacio de trabajo:

cm mergebranch br:/main/child1

El comando ‘mergebranch’ con una especificación de rama únicamente muestra los ítems que necesitan merge. Para llevar a cabo efectivamente la operación de merge, hay que proporcionar el modificador ‘--merge’

cm mb --merge br:/main/child1

La operación de merge deja todos los elementos que ha mezclado desprotegidos, para que puedan ser revisados y es el usuario el que debe protegerlos explícitamente una vez decide que todo está correcto. Este comando hace un checkin de todos los ítems desprotegidos como en el ejemplo presentado en la sección de uncheckout:

cm fco –format={4} | cm ci –

### Cherry Picking

El cherry picking es una forma diferente de ver el merge. Hasta ahora cuando hablábamos de hacer un merge, hablábamos de mezclar el contenido de las revisiones que conformaban el origen del merge con el contenido del espacio de trabajo. Al hablar de cherry picking esto cambia.

El cherry picking es un merge, mediante el cual, un cambio/s en el contenido de un elemento (delta/s), que ha sido introducido por una revisión, es aplicado en el espacio de trabajo. En este caso no se hace el merge del contenido del elemento en el origen, sino de los cambios que este ha incorporado en esa revisión respecto a su revisión anterior.

En el siguiente ejemplo podrá verse más claramente la diferencia entre ambos. Se tiene un fichero, en el que se ha añadido un método como parte de una nueva funcionalidad que se está desarrollando. Más tarde en ese mismo fichero se modifica uno de los métodos existentes para corregir un error que se ha detectado. Si se hace un merge de este fichero como resultado se incorporará la corrección en el método, pero también se incorporará el nuevo método que se había desarrollado. En cambio si hace un cherry picking solamente se incorporará la corrección en el método, pero no la nueva funcionalidad que se había incorporado.

Es importante señalar que para que una modificación en un elemento sea incorporada mediante el cherry picking toda ella debe de haberse introducido mediante un solo cambio (una única operación de proteger) y no mediante cambios sucesivos. Si se ha definido una función en un cambio, luego se han ido haciendo cambios sucesivos para completar su funcionalidad y al final en un último cambio se añade el comentario explicando la función. Al hacerse un cherry picking del ultimo cambio sólo se incorporaría el cometario de la función, pero ni la funcionalidad de la función ni la definición de la misma se incorporarían al haberse realizado en cambios anteriores.

 Para que un cambio pueda ser incorporado en el espacio de trabajo, sin problemas, mediante la operación de cherry picking es necesario que la versión sobre la que se realizo dicho cambio y la versión sobre la que se desea aplicar sean similares. Si estas dos versiones son parecidas, o al menos en la región o regiones de los cambios, el merge será automático, ya que los cambios continúan siendo aplicables. Si por el contrario estas dos versiones son completamente diferentes no habrá manera de identificar donde debe ir el cambio, si es que continúa siendo aplicable en el nuevo contexto, y tendrá que ser el usuario mediante la herramienta de merge el encargado de aplicarlo.

El uso del cherry picking está más orientado al trabajo con conjuntos de cambios o changesets dentro de una misma rama que al trabajo mediante rama por tarea. En el caso de trabajar empleando una rama para cada cambio esta operación pierde su sentido, porque todo cambio que se quisiese incorporar estaría en una rama de forma aislada, por lo que puede ser incorporado sin problemas mediante un merge de esa rama.

Tal y como se ha comentado antes el cherry picking tiene sentido cuando se aplica a un cambio. Para realizar este cambio puede que sólo haya sido necesario modificar un elemento o que se hayan modificados múltiples elementos conjuntamente a través de un changeset. Por ello como orígenes para el cherry picking se soportan:

* Desde una revisión. Si lo que se quiere es incorporar un cambio en un único que elemento que ha sido realizado por una revisión, por lo que haremos un cherry picking de dicha revisión.
* Desde un cahngeset. A diferencia del caso anterior, donde el cambio afectaba a un único elemento suele suceder que para realizar un cambio sea necesario modificar más de un elemento. Por ejemplo si se modifica los parámetros de un a función es necesario modificar tanto el fichero donde se define como todos los ficheros que contienen una llamada a esa función. Si todos los elementos modificados para realizar ese cambio son protegidos a la vez, están en un mismo changeset, podemos usar este changeset para referirnos al cambio. Por ello para poder incorporar ese cambio bastaría con hacer el cherry picking del changeset en el que se hizo.

Aunque hemos estado hablando de cherry picking y no de merge, el cherry picking no deja de ser un caso particular de merge.

Para poder realizar un cherry picking desde la línea de comandos se emplearía el comando de merge con el modificador --cherrypicking, de está manera le indicamos que el merge que ha de realizar es de este tipo. Por ejemplo el comando para realizar un cherry picking de un changeset 50 quedaría de la siguiente manera:

cm merge cs:50 --cherrypicking –merge

La Figura 42 muestra un proceso de merge normal, en el que todas las revisions de la rama 001 se integran en la revisión 2 de la rama /main, pero en el caso de la integración por Cherry pick, como muestra la Figura 43, solo se integra la revisión que nos interesa, en este caso es la revisión 4 de la rama 001 la que se integra en la rama /main, pero el usuario puede seleccionar que revisión o revisiones le interesa en cada momento.

|  |  |
| --- | --- |
| Figura 42: Operación de merge | Figura 43: Cherry-pick |

### Cherry Picking desde rama

Un caso particular de cherry picking es aquél que se realiza entre ramas. Como antes comentamos, tendría poco sentido aplicar un cherry picking cuando se trabaja en rama por tarea, pues normalmente se desea incorporar todos los cambios introducidos en las ramas.

El caso del cherry picking desde rama es un poco especial, ya que cambia la idea de “coger los cambios introducidos por el último changeset o la última revisión” a “coger los cambios introducidos por la última rama”.

Supongamos el siguiente caso:

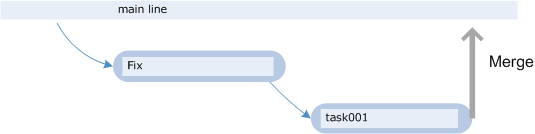


Figura 44: Cherry-pick desde rama

En el cual tenemos una rama principal de la que parte una rama de Fix a partir de una etiqueta y de ésta parte una rama que implementa una tarea determinada (task001).

Si a continuación realizáramos un merge normal de la rama task001 a la rama principal, nos traeríamos todos los cambios introducidos por la rama de la tarea y **también** todos los cambios introducidos por la rama de Fix.

En el caso de aplicar un cherry picking desde task001 a la rama principal, solamente incorporaríamos los cambios introducidos por dicha rama task001, desechando los cambios introducidos por la rama de Fix. El concepto es similar, pero adaptado a la circunstancia especial de las ramas.

Para realizar esta operación en Plastic, ejecute un comando como el siguiente:

cm merge <branch\_from> --cherrypicking –merge

Donde <branch\_from> es la rama desde la que se desea realizar el cherry pick.

Desde la GUI, haga clic derecho sobre la rama desde la que desea realizar la operación y seleccione la opción ‘Cherry pick desde esta rama’.

### Merge de un intervalo de cambios

Antes cuando se ha hablado del cherry picking se ha explicado que este permite incorporar un cambio en el espacio de trabajo. Pero tal, y como veíamos mientras hablábamos del cherry picking pueden darse casos en los que una modificación no se haya realizado mediante un único cambio sino mediante cambios sucesivos. En estas ocasiones lo que nos interesa no es incorporar el último de esos cambios, sino todos ellos.

El merge de un intervalo lo que permite es justamente eso, incorporar todos los cambios existentes entre dos revisiones dadas. En este caso lo que estamos haciendo es el merge de una revisión, pero teniendo en cuenta sólo aquello que ha modificado desde una revisión dada.

Al igual que sucedía con el cherry picking, la incorporación de los cambios realizados entre dos revisiones en el espacio de trabajo es otro caso particular de merge, en realidad es una generalización del cherry picking.

Por ejemplo si se tiene una rama br:/main/fix y se quiere incorporar los cambios realizados en el fichero “test” desde la revisión 5 a la 15, el comando de merge quedaría de la siguiente manera:

cm merge rev:test#br:/main/fix#15 --ancestor=rev:test#br:/main/fix#5 --merge

También en este caso si estamos usando un patrón de desarrollo de rama por tarea, este tipo de merge pierde su utilidad. Para ver más información sobre el uso de los diferentes patrones para la creación de ramas puede consultar el sitio web de Códice: ([www.codicesoftware.com](http://www.codicesoftware.com)).

## Crear una etiqueta

#### Desde la interfaz gráfica

Desde el menú de etiqueta: Crear Nueva Etiqueta.

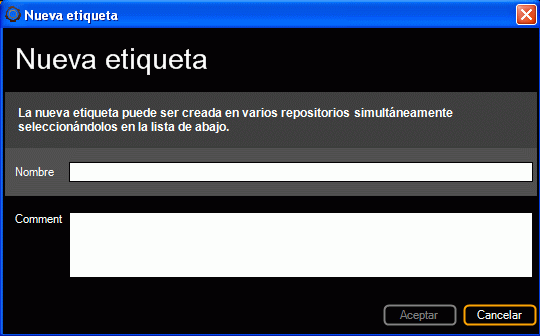


Figura 45: Creación de una nueva etiqueta

#### Desde la línea de comandos

Para crear una etiqueta se emplea el comando mklabel:

cm mklabel rel1.0

## Aplicar una etiqueta

#### Desde la línea de comandos

Aplicar una etiqueta implica indicar la etiqueta y la revisión a la que aplicarla. Para aplicar la etiqueta ‘rel1.0’ a la revisión que se encuentra actualmente en el espacio de trabajo del ítem ‘hello.cpp’ se emplea este comando:

cm label lb:rel1.0 hello.cpp

‘lb:rel1.0’ es la especificación de la etiqueta. Una sección posterior contiene la referencia de todas las especificaciones de objeto que se pueden suministrar al control de versiones.

Es normal aplicar una etiqueta a más de un ítem. El comando label admite el modificador –R para hacerlo recursivo. Este comando etiqueta todos los ítems que encuentran por debajo del directorio actual (**sin incluir el propio directorio .**) con la etiqueta ‘rel1.0’

cm label lb:rel1.0 –R \*

Es posible, asimismo, aplicar una etiqueta a una revisión específica, suministrando una especificación de revisión:

cm label lb:rel1.0 hello.cpp#br:/main/child1#4

Que aplicará la etiqueta ‘rel1.0’ a la revisión 4 del elemento ‘hello.cpp’ en la rama ‘/main/child1’ .

#### Desde la interfaz gráfica

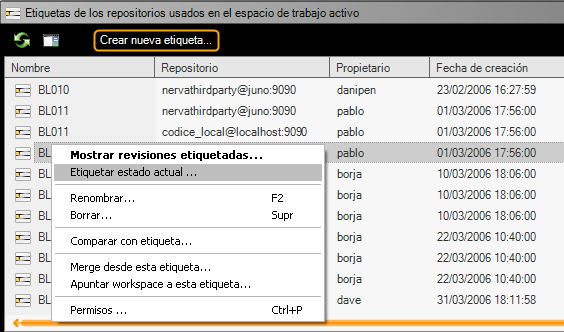


Figura 46: Aplicar etiqueta al contenido del espacio de tabajo

## Poner código existente bajo el control de versiones

Plastic SCM sólo puede controlar los ficheros y directorios que se encuentran en el espacio de trabajo, por lo que el paso previo para poner algo bajo el control de versiones consiste en tenerlo dentro de un espacio de trabajo. Lo normal es copiar los ficheros y directorios que componen el proyecto que se va a poner bajo control de versiones al directorio del espacio de trabajo. Es posible añadir el proyecto desde la interfaz gráfica, desde la línea de comandos, y desde las integraciones con los entornos de desarrollo.

#### Desde la interfaz gráfica

1. En la raíz del espacio de trabajo, botón derecho, opción de menú “Añadir recursivamente”.
2. Botón “Desprot.” en la barra de herramientas, o menú “Ver / Mostrar desprotecciones”.
3. Seleccionar todos los ítems con Ctrl-A
4. Botón “Checkin” en la barra de herramientas

#### Desde la línea de comandos

cm add –R \*  
cm ci –R \*

# Empezando un proyecto

Plastic SCM ha sido diseñado para permitir una puesta en marcha rápida y sencilla. Esta sección mostrará los pasos necesarios para empezar a poner un proyecto bajo el control de versiones tanto desde la interfaz gráfica como desde la línea de comandos.

Se mostrarán únicamente los pasos necesarios para completar las operaciones. La mayoría de las operaciones ya han sido descritas con detalle en las secciones anteriores.

Se asume que el usuario ha completado la instalación y configurado correctamente el modo de autentificación en su cliente.

## Creando un repositorio

Un repositorio denominado ‘default’ se crea durante el primer arranque del servidor. Usando ese repositorio los usuarios pueden empezar a trabajar más fácilmente, porque sólo necesitarán crear un espacio de trabajo, aquí se creará uno nuevo y es el que se usará en el resto del ejemplo.

#### Desde la interfaz gráfica

1. En el menú Repositorios / Crear nuevo repositorio
2. Introducir un nombre descriptivo y el servidor en el que se creará.

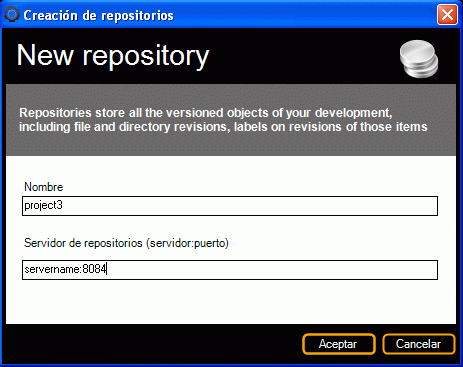


Figura 47: Diálogo de creación de repositorio

#### Desde la línea de comandos

cm mkrep servername:8084 project3

## Creación de un espacio de trabajo

La primera vez que un desarrollador arranca la herramienta GUI, si no se detecta ningún espacio de trabajo, aparecerá un diálogo de creación, como muestra la Figura 48. Por defecto, los espacios de trabajo apuntan al repositorio *default* y a la rama *main.* Se podrá cambiar dependiendo de la estrategia de branching utilizada.

#### Desde la interfaz gráfica

1. En el menú Workpaces / Crear nuevo workspace
2. Introducir un nombre descriptivo, la ruta y el servidor en el que se creará.

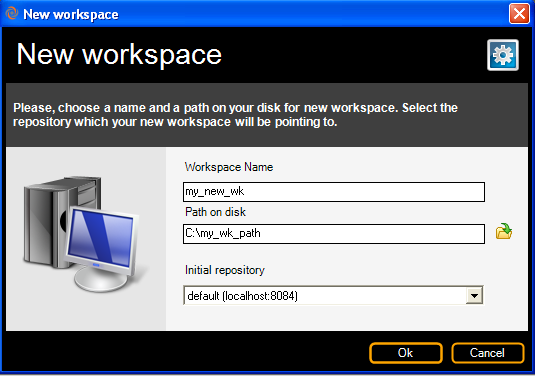


Figura 48: Creando un espacio de trabajo

#### Desde la línea de comandos

cm mkwk dave\_view c:\projects\view1

## Poner código existente bajo el control de versiones

En el espacio de trabajo recién creado (c:\projects\view1\), copiar los ficheros y directorios que componen el proyecto que se va a poner bajo control de versiones. Es posible añadir el proyecto desde la interfaz gráfica, desde la línea de comandos, y desde las integraciones con los entornos de desarrollo.

#### Desde la interfaz gráfica

1. En la raíz del espacio de trabajo, botón derecho, opción de menú “Añadir recursivamente”.
2. Botón “Desprot.” en la barra de herramientas, o menú “Ver / Mostrar desprotecciones”.
3. Seleccionar todos los ítems con Ctrl-A
4. Botón “Checkin” en la barra de herramientas

#### Desde la línea de comandos

cm add –R \*  
cm ci –R \*

## Empezando a trabajar: Actualizar

Una vez que el código ha sido importado dentro del repositorio, el resto de los desarrolladores puede obtener una copia local en su propio espacio de trabajo: Este proceso se denomina hacer un update.

#### Desde la interfaz gráfica

1. Botón “Obtener” en la barra de herramientas.

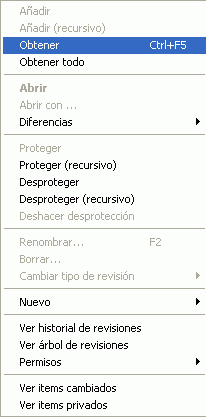


Figura 49. Realizando una operación de *Obtener*

#### Desde la línea de comandos

cm update .

## Desprotegiendo ítems para trabajar con ellos

Para realizar un cambio a cualquier ítem bajo el control de versiones hay que desprotegerlo.

#### Desde la interfaz gráfica

1. Seleccionar el ítem que se quiere desproteger.
2. Botón derecho, en el menú, opción “Desproteger”, o bien botón “Desproteger” en la barra de herramientas.

#### Desde la línea de comandos

Para desproteger un ítem cualquiera:

cm co *nombre\_del\_ítem nombre\_de\_otro\_ítem*

Para desproteger de manera recursiva un directorio y todos sus contenidos:

cm co –R directorio/\*

Desproteger todos los ítems leídos desde la entrada estándar:

cm co -

## Mostrando las protecciones pendientes

#### Desde la interfaz gráfica

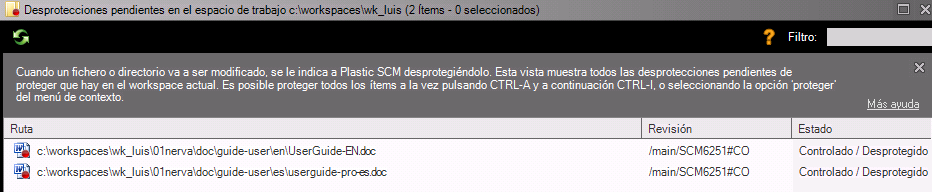


Figura 50. Listar las revisiones *desprotegidas*

#### Desde la línea de comandos

cm findcheckouts  
cm fco

# Selectores en profundidad

Los selectores son la pieza central de los espacios de trabajo porque contienen el conjunto de reglas que determinan sus contenidos. Configurando el selector un usuario podrá indicar con precisión y comodidad los ítems sobre los que va a trabajar. Este punto describe cómo crear y configurar selectores.

## Aspecto de un selector

Un selector es una especificación de qué revisiones de qué ítems serán descargados al espacio de trabajo en el disco del usuario. Indica también en qué rama se crearán las desprotecciones (*checkouts)*, es decir, las nuevas revisiones.

Un selector muy común será el que muestra la Figura 51. En primer lugar el selector indica sobre qué repositorio se trabajará, mediante la regla de selección de repositorio.

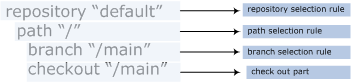


Figura 51. Apariencia del selector por defecto

Una vez que se selecciona un repositorio el siguiente paso es decir exactamente qué descargar. En el ejemplo la *regla de selección de path* indica que se descargará todo aquello por debajo del directorio raíz.

Un selector **siempre debe indicar una regla para descargar el directorio raíz**, de otro modo Plastic lanzará un error. Para obtener los ficheros y directorios desde el repositorio hasta el espacio de trabajo, Plastic necesita *resolver* los paths desde el directorio *raíz* hacia abajo, según las reglas del selector, y por eso es necesario que todos los ítems puedan encontrar una regla adecuada.

Dentro de la *regla de selección de path* hay otra especificación que indica la rama de origen y la rama sobre la que hacer *check outs*.

## Definición del selector

La muestra la definición completa del selector. Hay muchas posibilidades disponibles para poder indicar exactamente qué debe descargarse al espacio de trabajo.

Un selector consiste en una o varias reglas de carga de repositorio, cada una de las cuales define uno o más *paths* que deberán procesarse, y en qué lugar estarán las revisiones de los ítems (branches, etiquetas, changesets o números de revisión).

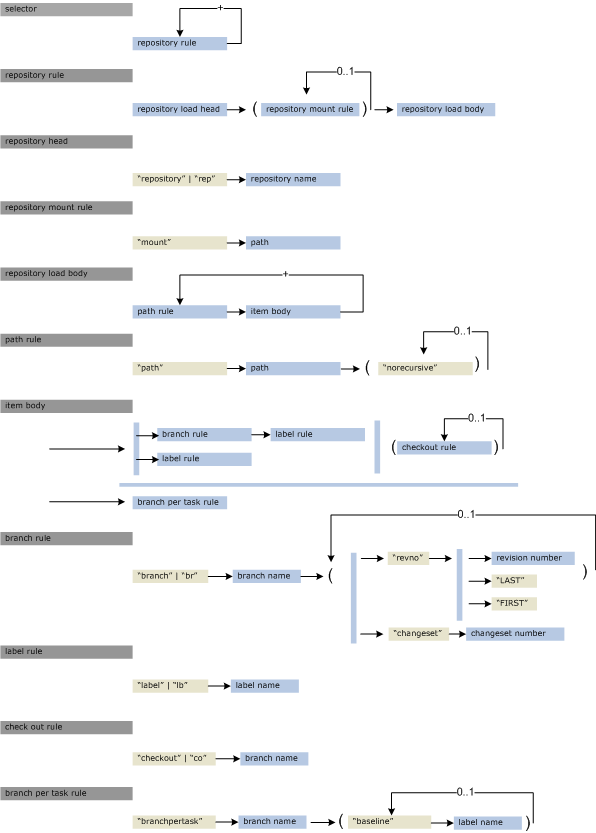


Figura 52. Definición de la *gramática* del selector

## Reglas de path

Las reglas de path son las que definen exactamente qué se va a descargar a un path determinado. Se ejecutan *en orden*, lo que significa que si una regla es capaz de cargar un *ítem*, las reglas siguientes no serán usadas.

El siguiente ejemplo muestra cómo funcionan las reglas de path:

repository "codice"  
 path "/01nerva/doc"  
 br "/main/SCM0841" co "/main/SCM0841"  
 path "/"  
 br "/main"

El selector indica que se cargarán las revisiones de los ítems dentro del path /01nerva/doc que están en la rama /main/SCM0841 y los checkouts que se produzcan irán a la misma rama. Todo lo demás, incluyendo el directorio raíz, se descargará desde la rama *main*.

Pero, ¿qué ocurriría con el siguiente selector?:

repository "codice"  
 path "/"  
 br "/main"  
 path "/01nerva/doc"  
 br "/main/SCM0841" co "/main/SCM0841"

En el segundo selector las reglas de path han sido intercambiadas. Ahora la regla del path “/” será evaluada primero, y como todos los ítems pueden ser resueltos con esa regla, la segunda no se usará nunca.

## La opción norecursive de la regla path

La opción *norecursive* significa que la regla se aplicará al directorio, pero no a los ítems que contiene.

Si se dispone de una estructura como la ilustrada en la y sólo se quiere trabajar con los contenidos del directorio /01nerva/doc, no es necesario descargar toda la estructura de directorios.

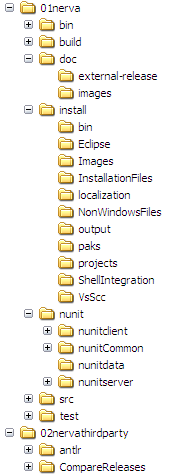


Figura 53. Estructura de repositorio de ejemplo

Para descargar únicamente el directorio deseado se usará un selector como:

repository "codice"  
 path "/01nerva/doc"  
 lb "BL034"  
 path "/01nerva" norecursive  
 lb "BL034"  
 path "/" norecursive  
 br "/main"

El selector sólo *descargará* las revisiones de los ítems dentro del path requerido, cogiendo las revisiones etiquetadas con BL034.

## Comodines en la regla de selección de path

En ciertas situaciones sólo es necesario descargar ciertos elementos al espacio de trabajo. Para descargar sólo los ficheros *.doc* en el ejemplo anterior se usará un selector como:

repository "codice"  
 path "/01nerva/doc/\*.doc"  
 lb "BL034"  
 path "/01nerva/doc" norecursive  
 lb "BL034"  
 path "/01nerva" norecursive  
 lb "BL034"  
 path "/" norecursive  
 br "/main"

## Obteniendo una revisión específica de un ítem

Si necesita descargar una revisión específica de un ítem concreto, la opción *revno* de la regla de branch del selector será útil. *Revno* puede ser un número o los nombres *especiales* *LAST* y *FIRST*. *LAST* indica la última revisión en la rama y *FIRST* la primera.

Volviendo al ejemplo anterior se verá un selector capaz de descargar una revisión concreta el fichero /01nerva/doc/manual.doc:

repository "codice"  
 path "/01nerva/doc/manual.doc"  
 br "/main" revno "5"  
 path "/01nerva/doc"  
 lb "BL034"  
 path "/01nerva" norecursive  
 lb "BL034"  
 path "/" norecursive  
 br "/main"

## Cómo utilizar varios servidores de repositorios

Para poder configurar un espacio de trabajo de modo que pueda utilizar varios repositorios ubicados en distintos servidores, es necesario registrarlos. Para ello, cree un fichero *plastic.servers* en su carpeta plastic en el directorio local del usuario (esto es, /home/<user>/.plastic directory en Linux o C:\Documents and Settings\<user>\Configuración Local\Datos de Probrama\plastic en Windows XP o C:\Users\<user>\AppData\Local\plastic en Vista). El siguiente ejemplo muestra los contenidos que podría tener dicho fichero:

london:8084  
mymachine:8084  
 …………  
mainmachine:8084

Como puede observar, cada línea contiene un servidor y un puerto.

De este modo, es posible utilizar dichos repositorios en el espacio de trabajo. Así, cuando cree un nuevo espacio de trabajo, podrá seleccionar uno de esos repositorios en la lista de repositorios que aparece.

Además, puede utilizar los repositorios registrados en los espacios de trabajo existentes mediante la modificación del selector. Veamos un ejemplo:

repository "remote\_rep1”

path "/"

label "BL172.2"

repository "rep:default@remote\_machine2:8084”

path "/"

br “/main”

co “/main”

repository "mylocalrep”

path "/"

label "BL132"

En este ejemplo estamos indicando que utilizaremos tres repositorios: nuestro repositorio local ‘mylocalrep’ y además otros dos repositorios remotos: ‘remote\_rep1’, el cual se encuentra en un servidor de repositorios determinado, y además ‘remote\_rep2’, que se encuentra en la máquina ‘remote\_machine2’.

Como puede observar, para especificar ‘remote\_rep1’ se ha utilizado simplemente el alias del repositorio, sin embargo para especificar el repositorio remoto ‘default’ se ha utilizado la especificación completa del repositorio, incluyendo máquina y puerto. ¿Por qué?

Si tenemos dos repositorios registrados que tienen el mismo nombre, Plastic decidirá utilizar aquél especificado en el fichero client.conf. Para poder utilizar dichos repositorios que tienen un nombre idéntico, es necesario indicar la especificación completa de dichos repositorios que no aparecen en el client.conf.

# Especificaciones de objetos

En Plastic SCM todos los objetos pueden ser referenciados mediante *especificaciones de objetos*. Los especificadores no son muy comunes cuando se usa la herramienta gráfica, pero es interesante conocer cómo el sistema referencia los objetos.

## Especificaciones de revisión

Para referenciar una revisión específica de un ítem, se puede usar uno de los siguientes formatos:

path#branch name#revision number

path#label name

Por ejemplo, es posible referenciar la revisión 23 del fichero /src/main.java en la rama main con la siguiente sintaxis:

/src/main.java#br:/main#23

Otra posibilidad sería acceder a una revisión etiquetada:

/src/main.java#lb:RELEASE001

## Especificaciones de rama

Los especificadores de rama se usan durante el merge o al fijar selectores y tienen la sintaxis siguiente:

**br:***branch\_name*

Ejemplos:

**br:**/main

**br:**/main/release00

## Especificaciones de ítem

Las especificaciones de ítem serán:

**item:***path*

Ejemplo:

**item:**/src/hello.java

## Especificaciones de etiqueta

Una especificación de etiqueta tendrá el siguiente formato:

**lb:***label\_name*

Ejemplo:

**lb:**LABEL001

## Especificaciones de repositorio

La especificación de un repositorio tendrá el formato:

**rep:***repository\_name*@*repository\_server*

Ejemplo:

**rep:**myrep@REPSERVER:8084

## Especificaciones de espacio de trabajo

Para referenciar un workspace la sintaxis será:

**wk:***workspace\_name*

Ejemplo:

**wk:**integrator\_wk

## Especificaciones de servidor de repositorios

La sintaxis de una especificación de servidor de repositorios será:

**repserver:***servername***:***portnumber*

Ejemplo:

**repserver:**mainRepServer:8084

# Gestionando proyectos con Plastic SCM

Disponer de un sistema SCM no es suficiente para poder gestionar correctamente un proyecto. Será necesario también conocer cómo estructurar el trabajo de desarrollo.

En este punto se introducirán diferentes métodos de trabajo, estrictamente relacionados con diferentes patrones de *branching*.

## Trabajando en la rama principal

Se trata de un patrón muy común, y es el que se emplea con sistemas de control de versiones como Subversion, CVS o SourceSafe. Es muy sencillo de configurar, pero tiene muchos inconvenientes.

La ilustra el proceso. Cada desarrollador trabaja en su propio espacio de trabajo (*workspace*). Cada uno de ellos puede desproteger (*check out*) tantos ficheros y directorios como necesite, pero cuando tenga que protegerlos (*check in*) los cambios irán directamente a la rama br:/main. Los programadores, por tanto, tendrán que ser muy cuidadosos con sus *check outs*.

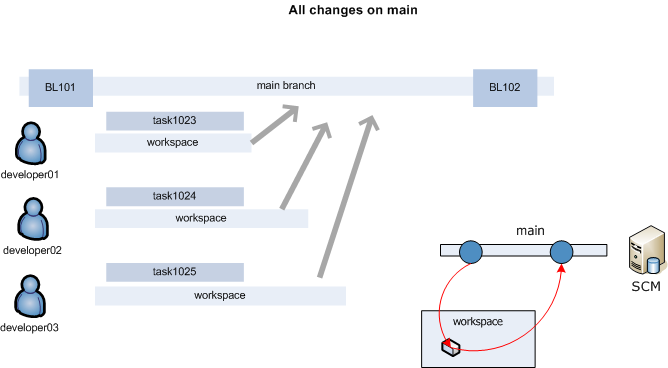


Figura 54: Todos los cambios en la rama principal

Si un programador protege unos cambios (tanto ficheros como directorios) y otro desarrollador actualiza inmediatamente su *workspace*, recibirá las modificaciones recién *subidas*. Si había algo erróneo en el *check in* del primer programador, el problema se propagará al trabajo del segundo.

El principal problema con este modelo es que no hay *lugares intermedios* en los que almacenar los cambios antes de pasar a la línea principal. Por tanto los *check ins* son menos frecuentes, dejando que el código pase mucho tiempo fuera del control de versiones. Los desarrolladores no usan el SCM como una herramienta para ayudarles en su trabajo diario, sólo para enviar trabajo terminado.

Se desaconseja el uso de este modelo.

#### Cómo hacerlo con Plastic SCM

Para usar este modelo los programadores tendrán que usar el siguiente selector:

repository "default"  
 path "/"  
 br "/main"  
 co "/main"

## Rama por desarrollador

Usando este patrón los desarrolladores dispondrán de un mayor aislamiento. De este modo podrán hacer tantas protecciones intermedias como necesiten, sin comprometer la integridad de la rama principal. La muestra cómo funciona el modelo.

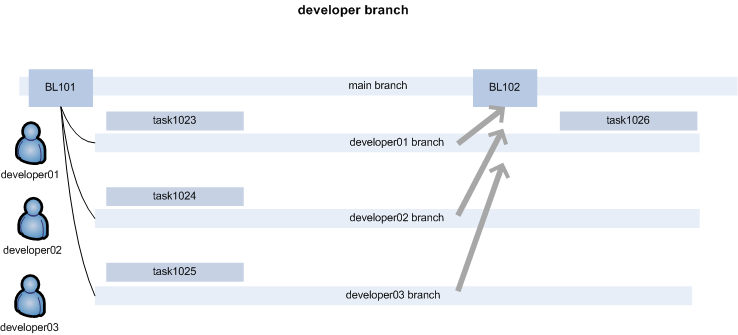


Figura 55: Patrón de rama por desarrollador

El trabajo podrá ser dividido en tareas, controladas por un sistema de gestión de tareas o defectos. Cada desarrollador implementará las tareas que se le asignen en su propia rama.

Una vez que varias tareas hayan sido completadas y con una frecuencia que será inversamente proporcional al número de desarrolladores, se creará una nueva *release*, integrando todas las ramas en la línea principal. La integración podrá ser realizada por cada desarrollador o por un *integrador* responsable del paso a la línea principal.

Una vez que la *integración* (que se realizará mediante operaciones de *merge* desde cada una de las ramas) haya sido completada se creará una nueva *baseline* etiquetando los nuevos contenidos. Después cada desarrollador realizará un *rebase* (un *merge* desde br:/main) a su rama para continuar trabajando con los últimos contenidos.

#### Cómo hacerlo con Plastic SCM

Para implementar este modo de trabajo cada desarrollador tendrá un selector como el siguiente:

repository "default"  
 path "/"  
 br "/main/*developerbranch*"  
 co "/main/*developerbranch*"

Donde *developerbranch* será el nombre o identificador del desarrollador, según la convención de nombres que se elija.

Para integrar el trabajo hacia *main* cada desarrollador, o el integrador, usará el siguiente selector:

repository "default"  
 path "/"  
 br "/main"  
 co "/main"

## Rama por tarea

Usando el patrón de rama por tarea se creará una nueva rama por cada una de las tareas asignadas a un desarrollador.

Se trata del patrón que mayor aislamiento y control proporciona, pero requiere que el SCM soporte un fuerte trabajo con ramas. Plastic SCM es capaz de implementar el patrón sin ningún impacto en el rendimiento o integridad del sistema, a diferencia de lo que ocurriría con otros SCMs.

Cada desarrollador trabajará en una rama, aislando su trabajo, y cambiará a una nueva rama cada vez que complete una tarea.

El integrador creará nuevas *releases* y *baselines* con una frecuencia fija, integrando las ramas de tarea que hayan sido terminadas y probadas.

El patrón de rama por tarea posibilita un máximo aislamiento entre desarrolladores, y facilita mantener la estabilidad del producto en su línea principal. Mejora la productividad de los programadores permitiendo que usen el SCM para ayudarles en su trabajo diario.

La ilustra el patrón.

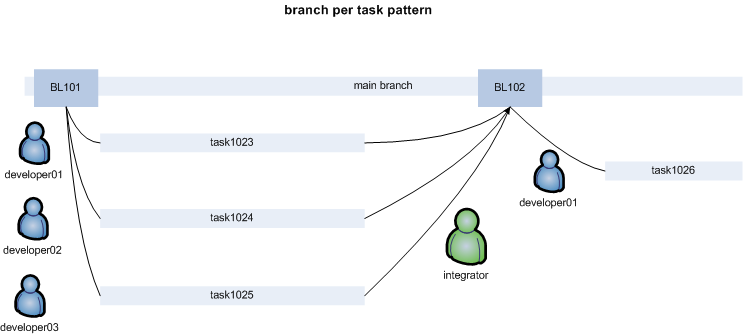


Figura 56: El patrón de rama por tarea

#### Cómo hacerlo con Plastic SCM

Un posible selector para los desarrolladores será:

repository "default"  
 path "/"  
 br "/main/*taskXXX*" label “BL101”  
 co "/main/*taskXXX*"

Donde *taskXXX* será reemplazado por el número de tarea.

Existe un modo de simplificar el selector:

repository "default"  
 path "/"  
 branchpertask “/main/taskXXX” baseline “BL101”

El integrador trabajará en la rama principal con el siguiente selector:

repository "default"  
 path "/"  
 br "/main"  
 co "/main"

# Sistema de consultas

Plastic SCM proporciona un mecanismo de consultas e informes que ofrece la base para implementar una amplia variedad de estadísticas sobre el desarrollo. Se trata de un mecanismo de consultas simplificado, que permite recuperar información de los objetos del repositorio pudiendo especificar una amplia variedad de filtros y sus combinaciones.

A través del sistema de consultas, Plastic SCM puede responder a preguntas como:

* ¿Qué ítems están en la etiqueta xx y no están en la etiqueta yy? (o lo que es lo mismo ¿qué ha cambiado entre estas dos etiquetas?)
* ¿Qué revisiones ha creado el usuario ‘xx’ en la rama ‘rr’ en la última semana?
* ¿Qué revisiones hay en el changeset xx?

Adicionalmente es posible exportar cualquier consulta en formato XML, gracias a lo que se abren numerosas vías de integración con herramientas de terceros.

## Lenguaje de consulta

Plastic SCM ofrece un lenguaje propio de consultas para especificar el tipo de objetos a devolver como resultado, y los parámetros de filtrado de esos objetos. Todas las consultas ocurren a través del comando ‘find’.

**cm find** *objeto* **[where** *condiciones***]****[on repositories** *repositorios***]**

Los tipos de objeto posibles son:

Attribute

Attributetype

Branch

Changeset

Link

Linktype

Marker

Merge

Moved

Removed

Revision

User

Wkrepository

Workspace

Las consultas solo pueden realizarse sobre un tipo de objeto a la vez, aunque sí es posible realizar esa búsqueda sobre diferentes repositorios simultáneamente.

Las condiciones de la consulta se especifican a través de la cláusula *where*. Es posible agrupar varias condiciones a través de los operadores *and* y *or*, y es obligatorio encerrar los valores de cadena entre comillas simples cuando se utilizan para especificar filtros. Por ejemplo, obtener todas las revisiones que se encuentran en la rama ‘/main/child1’:

cm find “revision where branch = ’br:/main/child1’”

Cada entidad posee una serie de atributos que pueden emplearse para especificar las condiciones de filtrado, así como el formato de salida.

Los siguientes atributos son comunes para todas las entidades del sistema, con la excepción de los tipos *user, wkrepository* y *workspace*:

Id: Identificador único del objeto.  
Owner: Nombre del usuario que creó el objeto.  
Date: Fecha en que se creó el objeto.  
Comment: Comentario asociado a la creación del objeto.

Para cada uno de los objetos se puede especificar el usuario “ME” y la consulta buscará los objetos creados por el usuario que está en ese momento logeado en el sistema.

Los siguientes apartados describen los atributos específicos de cada uno de los tipos de objeto anteriores.

#### Attribute

Srcobj: Objeto origen

Type: Tipo de objeto

Value: Valor dado

#### Attributetype

Name: Nombre del atributo

Source: Atributos aplicados al objeto

Value: Valor aplicado al objeto, el resultado es el atributo que contiene ese valor

#### Branch

Name: Nombre de la rama

Parent: Especificación de la rama padre

Attribute: Atributo que se le da a la rama

Attrvalue: Valor del atributo que se le da a la rama

#### Changeset

Attribute: Atributo que se le da al changeset

Attrvalue: Valor del atributo que se le da al changeset

Changesetid: ID del changeset

#### Link

Type: Tipo de *links* disponibles

Dstobj: Objeto de destino

Srcobj: Objeto origen

#### Linktype

Name: Nombre del *link*  
Source: Especificación del objeto origen del *link*  
Destination: Especificación del objeto destino del *link*

#### Marker

Attribute: Atributo dado a la etiqueta

Attrvalue: Valor del atributo dado a la etiqueta

Name: Nombre de la etiqueta  
Revision: Especificación de la revisión a la que se aplica la etiqueta

#### Merge

Dstbranch: Especificación de la rama de destino

Dstrev: Especificación de la revisión de destino

Dstchangeset: Changeset de la revisión de destino

Srcbranch: Especificación de la rama origen

Srcrev: Especificación de la revisión origen

Srcchangeset: Changeset de la revision de origen

#### Moved

Dstbranch: Especificación de la rama de la revisión de destino

Dstchangeset: Changeset de la revision destino

Dstdirrev: Especificación de la revisión de destino

Item: Especificación del item que fue movido

Itemid: Id del item que fue movido

Srcbranch: Especificación de la rama de la revision de origen

Srcchangeset: Changeset de la revision origen

Srcdirrev: Especificación de la revisión origen

#### Removed

Branch: Especificación de la rama en que el item fue borrado.

Changeset: Changeset de la revisión en que el item fue borrado

Dirrev: Especificación de la revisión del directorio en la cual se borro el item.

Item: Especificación del item que fue borrado

Itemid: Id del item que fue borrado

#### Revision

Branch: Especificación de la rama a la que pertenece la revisión.  
Changeset: Número de changeset asociado a la revisión.  
Item: Especificación del ítem al que pertenece la revisión.  
Marker: Nombre de la etiqueta asociada a la revisión.  
Parent: Especificación de la revisión padre de la que puede partir otra revisión.  
RevNo: Número de la revisión.  
Size: Tamaño en bytes de la revisión.

Label: Nombre de la etiqueta asociada con la revisión  
Type: Tipo de la revisión. Puede ser de tipo texto, binaria o directorio.

Attribute: Atributo dado a la revisión

Attrvalue: Valor del atributo dado a la revisión

#### User

Name: Nombre del usuario.  
Code: SID de identificación del usuario. Depende del sistema de autenticación configurado.  
Active: Booleano que identifica si el usuario esta activo o no.  
Group: Booleano que identifica si corresponde con un grupo de usuarios.

#### Wkrepository

Alias: Alias dado al repositorio

Id: ID del objeto

Name: Nombre dado al repositorio

Owner: Nombre del usuario que creó el repositorio

Repid: ID del repositorio en el servidor de repositorios

Server: Servidor en el cual está localizado el repositorio

#### Workspace

Id: ID del objeto

Machine: Máquina en la cual está situado el espacio de trabajo

Name: Nombre dado al espacio de trabajo

Owner: Nombre del usuario que creó el espacio de trabajo

Path: Ruta en la cual está localizado el espacio de trabajo

## Ejemplos de uso

Revisiones de una determinada etiqueta:

cm find “revision where marker=’BL002’”

Revisiones creadas por un usuario en un periodo de tiempo y en la rama principal:

cm find “revision where owner = ’user’ and date between ’01/01/2007’ and ’02/02/2007’ and branch = ’br:/main’”

Historial de un determinado ítem en una rama concreta:

cm find “revision where item = ’item:.’ and branch = ‘br:/main/task003’”

Revisiones que han cambiado desde la etiqueta ‘excel2’ a la etiqueta ‘excel3’:

cm find “revision where marker = 'excel3' and not marker = 'excel2' --format={item}”

Revisiones binarias de más de 1MB en un changeset concreto:

cm find “revision where type = ’bin’ and size > 1024 and changeset = 12345”

Revisiones desprotegidas en una serie de ramas:

cm find “revision where revno = ’CO’ and (branch = ’br/main’ or branch = ’br:/main/task002’)”

Etiquetas que marcan una determina revisión:

cm find “marker where revision = ’.#br:/main#2’”

Ramas hijas de ‘/main’ de un usuario concreto:

cm find “branch where parent = ’br:/main’ and owner = ‘user’”

Usuarios desactivados en varios repositorios:

cm find “user where active=’F’ on repositories ‘rep1’, ‘rep2’”

Ramas integradas por el usuario:

cm find “branches where owner= ’ME’ and attrvalue: ‘Integrated’”

Espacios de trabajo creados por el usuario en su máquina

cm find “workspaces where owner=’ME’ and machine=’mycomputer’”

Ramas marcadas con el atributo que permanecen abiertas:

cm find “branches where attribute=’Status’ and attrvalue=’Open’”

Merges realizados desde la rama ‘/main/task001’ a la rama ‘/main’

cm find “merges where srcbranch=’br:/main/task001’ and dstbranch=’br:/main’”

Revisiones creadas en ‘/main’ resultado de la integración de cualquier rama en el changeset 5

cm find “merges where dstbranch=’br:/main and dstchangeset=5’”

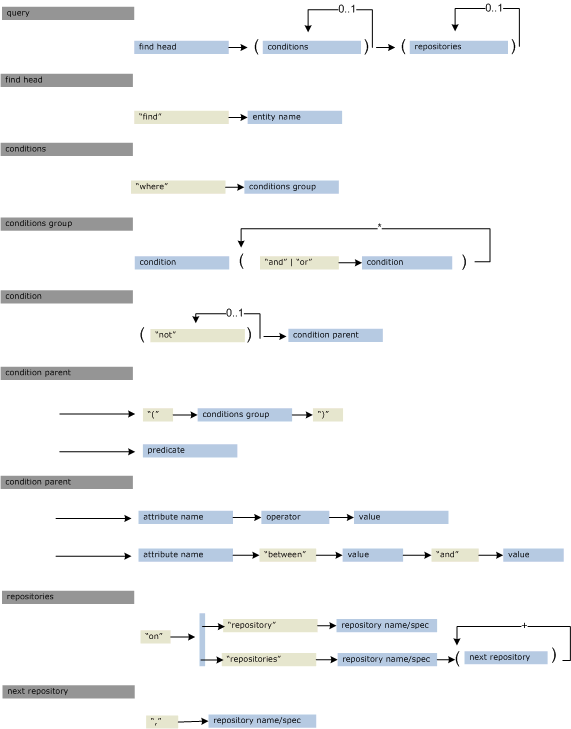
Movidos realizados de item ‘doc’ en la rama ‘/main’

cm find “moved where item=’doc’ and (srcbranch=’br:/main’ or dstbranch=’br:/main’)”

Borrados realizados en la creación de la revision ‘.#br:/main#4’

cm find “removed where dirrev=’.#br:/main#4’”

## La gramática del lenguaje de consultas



# Sistema de consultas avanzado (AQS)

Además del sistema de consultas, Plastic también proporciona un mecanismo de consultas avanzado, disponible a través de la interfaz de la línea de comandos de Plastic, y en particular en el comando cm query, que permite recuperar objetos de una o varias entidades a la vez en función de unos criterios de búsqueda. Le permite al usuario realizar cualquier tipo de consulta contra el sistema.

Las diferencias entre los dos sistemas de consultas disponibles es que el simple va a realizar las consultas directamente sobre las entidades del sistema y solo sobre una entidad a la vez, mientras que el avanzado lo realiza directamente sobre las bases de datos del servidor y se puede realizar sobre una o varias entidades a la vez, por lo que el avanzado nos va a poder proporcionar más información que el simple, pero para consultas que se puedan hacer con cualquiera de la dos se recomienda realizarlo con el sistema de consultas simple porque la consulta a realizar es mucho más sencilla. En el apartado de ejemplos se puede ver un caso realizado para los dos sistemas de consultas, para comprobarlo. Aunque hay entidades que son iguales en ambos sistemas, anotar que hay mayor número de entidades en el AQS y que algunas son distintas.

A través del sistema de consultas, Plastic SCM puede responder a preguntas como:

* ¿En qué ramas tiene alguna revisión un determinado ítem?
* ¿De qué revisiones se ha hecho merge hacia main desde una fecha dada?

## Lenguaje de consulta

Para realizar las consultas el comando cm query va a utilizar una sintaxis SQL para realizar las mismas sobre la base de datos del servidor. Se va a hacer uso de todas las posibilidades de SQL, de tal modo que hasta las funcionalidades más avanzadas de Plastic van a utilizar AQS.

El AQS está totalmente integrado con Plastic y no impone restricciones adicionales al funcionamiento del sistema. Funciona sobre todos los sistemas operativos que soporte PlasticSCM.

Para poder ejecutar una consulta de este tipo el usuario deberá disponer del permiso advanced query sobre el repositorio.

Las consultas se pueden realizar sobre una entidad o varias a la vez.

El siguiente listado muestra las entidades sobre las que se pueden realizar consultas:

Items

Revisions

Checkouts

Branches

Labels

LabeledRevisions

Links

LinkedObjects

Attributes

Objectswithatributes

Changesets

Cada entidad posee una serie de atributos que permiten definir los criterios de búsqueda, y así limitar el número de objetos recuperados de cada entidad. Los siguientes atributos son comunes para todas las entidades del sistema:

Owner: Nombre del usuario que creó el objeto en el sistema.

CreationDate: Fecha en que se creó el objeto en el sistema.

A continuación se muestran los atributos propios para cada una de las entidades mencionadas anteriormente.

#### Items

Objectid: Identificador del ítem .

#### Revisions

Objectid: Identificador de la revisión.

Sizebytes: Tamaño en bytes de la revisión

Itemid: Identificador del ítem del que queremos saber la revisión

Branchid: Identificador de la rama a la que pertenece la revisión

Revisionnumber: Número de revisión

Changeset: Número de changeset asociado a la revisión.

Comment: Campo de comentarios incluido en la revisión

#### Checkouts

Revisionid: Especificación del identificador de la revisión a la que se aplica el checkout

Clientmachine: Máquina donde está instalado el cliente

Exclusive: te dice si hay checkouts exclusivos

#### Branches

Objectid: Identificador de la rama

Name: Nombre de la rama

ParentBranchid: Identificador de la rama padre

Revision: Revisión de la rama

#### Labels

Objectid: Identificador de la etiqueta

Name: Nombre de la etiqueta

#### LabeledRevisions

Labelid: Identificador de la etiqueta

Revisionid: Identificador de la revisión

#### Links

Objectid: Identificador del link.

Name: Nombre del link

#### Linkedobjects

Linkid: Identificador del link

Sourceobjectid: Especificación del objeto origen del link

Destinationobjectid: Especificación del objeto destino del link

#### Attributes

Objectid: Identificador del atributo.

Name: Nombre del atributo

#### Objectswithattributes

Attributeid: Identificador del atributo

Sourceobjectid: Identificador del objeto fuente del atributo

Attributesvalue: Valor de los atributos

#### Changesets

Changesetnumber: Número de changeset asociado a la revisión.

Para poder gestionar la resolución de paths y de usuarios en las consultas, se van a introducir dos funciones de resolución predefinidas. Estas son:

SolvePath(path): La herramienta Plastic va a utilizar unos identificadores de item para cada objeto que esté en sus repositorios. Lo que va a hacer esta función es resolver las rutas de disco a los identificadores de ítem (itemid) que utiliza Plastic.

SolveUser(nombre\_usuario): Resuelve el nombre de usuario dado al formato que maneja PlasticSCM para identificar al mismo.

Para mostrar el resultado de las consultas de forma legible al usuario, se puede indicar al intérprete de consultas las siguientes opciones:

--solveuser=nombre\_columna: Indica al intérprete de consultas que la columna especificada contiene usuarios. Se resolverá el convertir los identificadores de usuario en nombres de usuario

--solvepath=nombre\_columna: Indica al intérprete de consultas que la columna especificada contiene identificadores de item. Se resolverá el convertir los identificadores de item a rutas de disco.

--outputfile=ruta: Escribe en un fichero el resultado de la consulta

--columnwidth=valor: Especifica la anchura de cada columna del resultado de la consulta.

Se pueden especificar varios nombres de columnas separados por comas.

Para que se entienda para qué sirven tanto las funciones predefinidas como las opciones de las consultas, y entender por qué es necesario utilizar ambas a la hora de realizar las consultas, a continuación de muestran varios ejemplos aclaratorios:

Revisiones del repositorio actual

cm query “SELECT \* FROM revisions”

Revisiones del repositorio actual del item cuyo itemid utilizado por Plastic se corresponda con la ruta especificada c:\workspace. Lo que hace la función es resolver la ruta de disco especificada al identificador de item utilizado por Plastic de la misma.

cm query “SELECT \* FROM revisions WHERE itemid=’SolvePath(c:\workspace)’”

Se obtiene lo mismo que lo anterior pero ahora la columna itemid no va a contener el identificador de item sino la ruta de disco correspondiente a ese identificador, en el caso del ejemplo, c:\workspace.

cm query “SELECT \* FROM revisions WHERE itemid=’SolvePath(c:\workspace)’” -–solvepath=itemid

Revisiones del repositorio actual del item cuyo identificador de usuario utilizado por Plastic se corresponde con el usuario especificado a la función.

cm query “SELECT \* FROM revisions WHERE owner=’SolveUser(dave)’”

So obtiene lo mismo que lo anterior pero ahora la columna owner no va a contener el identificador de usuario utilizado por Plastic sino el nombre de usuario.

cm query “SELECT \* FROM revisions WHERE owner=’SolveUser(dave)’” -–solveuser=owner

## Ejemplos de uso

Checkouts del repositorio

cm query “SELECT \* FROM checkouts

Ramas del repositorio actual creadas por el usuario especificado.

cm query “SELECT \* FROM revisions WHERE owner=´SolveUser(dave)’ ”

En qué ramas que tienen alguna revisión el item c:\workspace\file.txt

cm query “SELECT DISTINCT r.ITEMID as path, b.NAME

FROM REVISIONS r,BRANCHES b

WHERE r.BRANCHID=b.OBJECTID   
AND   
r.ITEMID=’SolvePath(c:\workspace\file.txt)’

ORDER BY r.OBJECTID"

--solvepath=path --columnwidth=50

De qué revisiones se ha hecho merge hacia main desde una fecha dada

cm query "SELECT BR1.NAME, R1.ITEMID

from REVISIONS R1, REVISIONS R2, BRANCHES BR1, BRANCHES BR2, LINKEDOBJECTS LO, LINKS L

where LO.SOURCEOBJECTID = R1.OBJECTID

and R1.BRANCHID = BR1.OBJECTID

and LO.DESTINATIONOBJECTID = R2.OBJECTID

and L.OBJECTID = LO.LINKID

and R2.BRANCHID = BR2.OBJECTID

AND BR2.Name='main'

AND L.NAME='merge'

AND R2.CREATIONDATE > '2007/07/15'"

--SolvePath=ITEMID --columnwidth=30

1. La operación de update es similar a una operación de *checkout* en sistemas CVS y Subversion. En Plastic SCM descargar y desproteger revisiones son operaciones separadas. [↑](#footnote-ref-2)