RESUMEN ENTORNOS DE DESARROLLO

1. INTRODUCCIÓN A LOS VCS

Los VCS (versión control system) son aplicaciones que permiten realizar el control de modificaciones de código fuente de manera automática y eficiente:

-Estos sistemas facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas (por ejemplo,para algún cliente específico)

- Ejemplos de este tipo de herramientas son entre otros: CVS, Subversion, SourceSafe, ClearCase, Darcs,Bazaar,Plastic,SCM, Git, SCCS, Mercurial, Perforce, FossilSCM, Team Foundation Server.

2. SISTEMAS DE CONTROL DE VERSIONES OPENSOURCE

GIT: Funciona con un modelo distribuido y está licenciado bajo GNU GPL. Originalmente fue diseñado por Linus Torvalds y es utilizado para el proyecto del núcleo de Linux.

Subversión: Funciona con un modelo centralizado y está licenciado bajo Apache.

VS: Funciona con un modelo centralizado y está licenciado bajo GNU GPL

Mercurial: Funciona con un modelo distribuido y está licenciado bajo GNU GPL.

3. CARACTERÍSTICAS DE LOS VCS

Algunas de las características de los VCS son:

• **Reporte de cambios:** Cuando un archivo es modificado se guarda la fecha y hora del cambio, el autor y un mensaje opcional explicando las razones o naturaleza del cambio.

• **Sincronización:** Varias personas en el mismo proyecto que todas puedan tener la última versión

• **Backup y restauración:** Permite guardar los cambios realizados en los archivos y restaurar el archivo a cualquiera de los estados previos en los que fue guardado.

• **Crear branch:** Permite crear una branch (rama) del proyecto y trabajarla por separado sin modificar el proyecto original.

• **Realizar merge:** Permite convertir el contenido de branch al proyecto principal

• **Generación de informes:** Aunque no es estrictamente necesario, con los cambios introducidos entre dos versiones, informes de estado, marcado con versión de un conjunto de ficheros.

4. MODELOS DE CONTROL DE VERSIONES

• **Modelo local:** utiliza una copia de la base de control de versiones y una copia de los archivos del proyecto. Este tipo es el más sencillo y no es recomendable cuando se trabaja en equipo.

• **Modelo centralizado:** el control de versiones se realiza en un servidor que se encargará de recibir y dar los cambios realizados en el archivo a cada uno de los usuarios.

• **Modelo distribuido:** es el más utilizado, en este caso cada usuario tiene un control de versiones propio que a su vez son manejadas por el servidor.

MODELO CENTRALIZADO I

Usa un servidor central para almacenar el repositorio así como el control de acceso al mismo. El repositorio se mantiene separado de la copia que existe en el directorio de trabajo de cada usuario. Suele ser un servidor dedicado (o al menos un ordenador distinto al de cualquier usuario), aunque en proyectos pequeños o con un único usuario se suele asignar otra carpeta del ordenador de trabajo.

Con un sistema centralizado, la copia de trabajo sólo almacena la versión actual de los ficheros del proyecto y las modificaciones que el usuario esté realizando en ese momento. La única copia de la historia completa está en el repositorio. Cuando un usuario envía (checkin) cambios al repositorio central, en ese momento ya podrán ser descargados por el resto de usuario

MODELO CENTRALIZADO II

Todos los usuarios hacen su checkin y checkouts sobre la rama principal: u 1 añade peras y u 2 añade agua. Para que los cambios de u1 puedan ser visto por u 2 , u 1 tiene que hacer checkin al servidor.

MODELO DISTRIBUIDO I

En un sistema distribuido cada usuario tiene su propio repositorio. Los cambios que hace u1 estarán en un repositorio local que podrá compartir o no con el resto de usuarios. Todos los usuarios son iguales entre si, el concepto maestro/esclavo que aparece en los centralizados desaparece.

Aparentemente esta forma de trabajar da la sensación de ser un poco caótica y es por ello que suele ser habitual que además exista un repositorio central donde se sincronizan todos los cambios locales de cada usuario

MODELO DISTRIBUIDO II

Cada uno de los usuarios realiza los checkout y chekin con respecto a su propio repositorio. A esos cambios sólo tendrá acceso cada usuario (cada uno al suyo respectivamente) hasta que decida compartirlos con uno o con el resto de usuarios (o con el servidor) .

5.VENTAJAS E INCONVENIENTES DE LOS SISTEMAS DISTRIBUIDOS

Ventajas:

- Necesitan estar conectados menos veces a la red para hacer operaciones, esto produce una mayor autonomía y rapidez.

- La información está muy replicada y por tanto el sistema tiene menos problemas en recuperarse. Sin embargo, los backups siguen siendo necesarios para resolver situaciones en las que cierta información todavía no haya sido replicada.

- Permite mantener repositorios centrales más limpios en el sentido de que un usuario puede decidir que ciertos cambios realizados por él en el repositorio local.

- El servidor remoto requiere menos recursos que los que necesitaría un servidor centralizado ya que gran parte del trabajo lo realizan los repositorios locales.

- Al ser los sistemas distribuidos más recientes que los sistemas centralizados, y al tener más flexibilidad por tener un repositorio local y otro u otros remotos, estos sistemas han sido

diseñados para hacer fácil el uso de ramas (creación, evolución y fusión) y poder aprovechar al máximo su potencial.

Desventajas:

- No hay claramente una última versión del código.

- No existen números de revisión definidos. En un sistema distribuido en realidad no hay números de versión, ya que cada repositorio tiene sus propios números dependiendo de la cantidad de cambios que se han realizado en él.

6.TERMINOLOGIA

1.Repositorio: es el lugar en el que se almacenan los datos actualizados e históricos de cambios, a menudo en un servidor. A veces se le denomina depósito o depot. Pueden ser archivos en un disco duro, un banco de datos, etc.

2. Módulo: Conjunto de directorios y/o archivos dentro del repositorio que pertenecen a un proyecto común

3. Revisión ("version"): es una versión determinada de la información que se gestiona. Hay sistemas que identifican las revisiones con un contador (Ej. subversion).

4. Rotular ("tag"): Darle a alguna versión de cada uno de los ficheros del módulo en desarrollo en un momento preciso un nombre común ("etiqueta" o "rótulo") para asegurarse de reencontrar ese estado de desarrollo posteriormente bajo ese nombre.

5.Línea base ("baseline"): Una revisión aprobada de un documento o fichero fuente, a partir del cual se pueden realizar cambios subsiguientes.

6. Tronco o principal: línea principal de código en el repositorio.

7. Abrir rama ("branch") o ramificar o bifurcado: en un instante de tiempo de forma que, desde ese momento en adelante se tienen dos copias (ramas) que evolucionan de forma independiente siguiendo su propia línea de desarrollo. El módulo tiene entonces 2 (o más) "ramas".

8. Desplegar ("Check-out" , "checkout" , "co"): Un despliegue crea una copia de trabajo local desde el repositorio.Se puede especificar una revisión concreta, y predeterminadamente se suele obtener la última.

9. Conflicto: Un conflicto ocurre cuando el sistema no puede manejar adecuadamente cambios realizados por dos o más usuarios en un mismo archivo.

10. Resolver: El acto de la intervención del usuario para atender un conflicto entre diferentes cambios al mismo archivo.

11. Cambio ("change" , "diff" , "delta"): Un cambio representa una modificación específica a un archivo bajo control de versiones

12. Lista de cambios ("changelist" , "change set", "patch"): En muchos sistemas de control de versiones con commits multi-cambio atómicos, una lista de cambios identifica el conjunto de cambios hechos en un único commit

13. Exportación ("export"): Una exportación es similar a un check-out, salvo porque crea un árbol de directorios limpio sin los metadatos de control de versiones presentes en la copia de trabajo.

14. Importación ("import"): Una importación es la acción de copia un árbol de directorios local (que no es en ese momento una copia de trabajo) en el repositorio por primera vez.

15. Congelar: Permitir los últimos cambios (commits) para solucionar las fallas a resolver en una entrega (release) y suspender cualquier otro cambio antes de una entrega, con el fin de obtener una versión consistente.

16. Integración o fusión ("merge"): Una integración o fusión une dos conjuntos de cambios sobre un fichero o un conjunto de ficheros en una revisión unificada de dicho fichero o ficheros. Puede suceder:

• Cuando un usuario, trabajando en esos ficheros, actualiza su copia local con los cambios realizados, y añadidos al repositorio, por otros usuarios. Análogamente, este mismo proceso puede ocurrir en el repositorio cuando un usuario intenta check-in sus cambios.

• Después de que el código haya sido branched, y un problema anterior al branching sea arreglado en una rama, y se necesite incorporar dicho arreglo en la otra.

• Después de que los ficheros hayan sido branched, desarrollados de forma independiente por un tiempo, y que entonces se haya requerido que fueran fundidos de nuevo en un único trunk unificado.

17. Integración inversa: El proceso de fundir ramas de diferentes equipos en el trunk principal.

18. Actualización ("sync" o "update"): Una actualización integra los cambios que han sido hechos en el repositorio (por ejemplo, por otras personas) en la copia de trabajo local.

19. Copia de trabajo ("workspace"): es la copia local de los ficheros de un repositorio, en un momento del tiempo o revisión específicos. Conceptualmente, es un cajón de arena o sandbox.

20. La propia idiosincrasia de un sistema distribuido hace que se identifiquen determinadas acciones como:

• **Empujar (push)**: envía un cambio desde un repositorio a otro. Según cual sea la política del sistema podría ser necesario tener los permisos adecuados.

• **Extraer (pull)**: coge los cambios desde un repositorio.

• **Clonar (clone):** traer una copia exacta del proyecto desde un repositorio a otro.

7.FORMAS DE COLABORAR

Para colaborar en un proyecto usando un sistema de control de versiones lo primero que hay que hacer es crearse una copia local obteniendo información del repositorio.

Existen dos esquemas de funcionamiento:

1.De forma exclusiva: para poder realizar un cambio es necesario comunicar al repositorio el elemento que se desea modificar y el sistema se encargará de impedir que otro usuario pueda modificar dicho elemento. Una vez hecha la modificación, esta se comparte con el resto de colaboradores.

2.De forma colaborativa: cada usuario modifica la copia local y cuando el usuario decide compartir los cambios el sistema automáticamente intenta combinar las diversas modificaciones. Los sistemas Subversion o Git permiten implementar este modo de funcionamiento.

8.EJEMPLOS DE COMANDOS

Podemos ver como algunos de los comandos más utilizados actúan en un Sistema de Control de Versiones.

NOMBRE DE COMANDO	**FUNCIONAMIENTO**
ADD	Lo primero que hay que hacer es añadir lista.txt al control de versiones mediante una comando de Add. En este caso podemos ver como el usuario u 1 , desde su directorio de trabajo incluye el fichero lista.txt a la línea principal (main) del servidor donde se encuentra el sistema de control . El fichero lista .txt ya tiene un ítem (café), pero podría estar vacío.

	Automáticamente el VCS le asigna un numero de revisión (r 1).
CHECK OUT (FOR EDIT), CHECK IN (COMMIT) Y REVERT	u1 va modificando el fichero a lo largo de la semana, incluyendo nuevos ítems a la lista. Para ello, u1 realiza primero un check out for edit. Esta operación en realidad son dos operaciones: en primer lugar se actualiza el fichero lista.txt del directorio de trabajo a la última versión existente en el servidor (r1) y posteriormente se activa la posibilidad de editarlo. Una vez u1 tiene a su disposición la copia, la edita y añade un ítem (peras) y, una vez añadido, lo actualiza en el servidor mediante un chek in (o commit) creando una nueva revisión (r3)
TAGS	Puede ser interesante que a final de cada semana, antes de empezar con la lista de la semana siguiente, etiquetar cada versión del fichero para saber que fue lo que se compro en una determinada fecha.
DIFF	Uno de los usuarios tiene interés en ver en qué se ha modificado la lista entre dos versiones. Para obtener las diferencias, el VCS se hace preguntas como ¿qué debo modificar en una versión para llegar a otra?. Si nos fijamos en el ejemplo, para llegar a r 5 a partir de r4 , habría que añadir (+) vino y leche y eliminar (-) café y zumo.
BRANCH	u1 decide que quiere experimentar cocinando comida japonesa . Para ello abre una rama llamada japonesa, donde ira incluyendo ítems relacionados con ese tipo de comida, hasta que una semana decida ponerse a ello. Al abrir una rama lista.txt puede evolucionar por dos caminos distintos main y japonesa.
MERGE	u1 por fin se decide y la semana que viene está dispuesto a cocinar comida

	japonesa, por lo que ha de mezclar (merge) la lista de la rama japonesa con la rama principal que es de donde se obtiene la lista con la que se ira al mercado.

CONFLICTOS DE LOS COMANDOS:

1.Primer conflicto:

En la gran mayoría de casos el propio VCS mezcla las versiones sin mayor problema, pero puede darse casos en que esto no sea así . Tanto u 1 como u 2 deciden que quieren cambiar la lista y para ello hacen un check out de manera mas o menos simultanea. u 2 no quiere zumo y en cambio quiere agua, mientras que u 1 que tampoco quiere zumo, quiere pan en su lugar. u 2 cambia realiza el cambio y el checkin antes que u 1. Cuando u 1 va a realizar el chekin , su copia local tiene que mezclarse con la r 4 pero el sistema detecta que desde que se hizo el checkout (r 3) otro usuario ha cambiado las mismas líneas (en este caso la n ° 2) . El VCS no tiene información suficiente para decidir con cual quedarse (¿agua o pan?) y genera un conflicto.

2. Segundo conflicto:

En este caso solicita al usuario u 2 que resuelva el conflicto, es decir, que manualmente decida cual es la línea que deberá permanecer o que incluya las dos líneas. En condiciones normales la aparición de conflictos no es (o al menos no debería ser) muy habitual. Con un poco de organización, distribución ordenada del trabajo y haciendo checkin de manera mas o menos habitual, su aparición disminuye mucho. Aun así, tal como se comentó anteriormente, si se va a editar gran parte del fichero y durante mucho tiempo, puede ser recomendable que el usuario lo bloqueé para evitar la edición por parte de otros.

9.GIT

Es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. Su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.

Es una aplicación que permite a diferentes programadores trabajar en un proyecto común ocupándose cada uno de una parte y con un control centralizado. Es ideal para grandes

proyectos o también para trabajar con otras personas que se puedan encontrarlejos físicamente.

9.1 LOS TRES ESTADOS DE GIT

Cuando trabajamos con GIT, nuestra información puede estar en una de las siguientes situaciones:

• Información modificada **(modified).** Este estado implica que hemos modificado nuestra información, pero aún no está siendo trackeada por GIT.

• Información preparada **(staged).** Este estado implica que hemos marcado nuestra información para posteriormente ser confirmada y por tanto trackeada como nueva versión.
• Información confirmada **(committed).** Este estado implica que nuestra información ha sido almacenada en la base de datos local de GIT.

Estas situaciones dan lugar a lo que se conoce como los tres estados de GIT, **el working copy, el stagin area y el repositorio.**

• **Working copy**, área de trabajo en la que hacemos los cambios en nuestros ficheros.

• **Staging area,** espacio donde colocaremos aquellos ficheros listos para ser colocados en el repositorio.

• **Repositorio local,** área donde GIT irá guardando las distintas versiones de nuestra información

• **Repositorio remoto,** área en la nube (github) para guardar versiones de nuestra información.

9.2 CARACTERÍSTICAS MÁS RELEVANTES

Entre las se encuentran:

• Fuerte apoyo al desarrollo no lineal

• **Gestión distribuida**: Git le da a cada programador una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales.

• Los almacenes de información pueden publicarse por **HTTP, FTP, rsync o protocolo nativo,** ya sea a través de una conexión TCP/IP o cifrado SSH.

• Los repositorios Subversion y svk se pueden usar directamente con **git-svn.**

• Gestión eficiente de proyectos grandes.

• Los **renombrados** se trabajan basándose en similitudes entre ficheros, y evita posibles coincidencias de ficheros diferentes en un único nombre.

• El **realmacenamiento periódico** en paquetes (ficheros).

9.3. GITHUB

GitHub es una plataforma de desarrollo colaborativo para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de ordenador. El software que opera GitHub fue escrito en Ruby on Rails

Ruby on Rails, es un framework de aplicaciones web de código abierto escrito en el lenguaje de programación Ruby, siguiendo el paradigma del patrón Modelo Vista Controlador (MVC).

9.4 FLUJO DE TRABAJO

Un **flujo de trabajo** de Git es una fórmula o una recomendación acerca del uso de Git para realizar trabajo de forma uniforme y productiva. Los flujos de trabajo más populares son:

1.GIT-FLOW:

-Creado en 2010 por Vincent Driessen.

-Es el flujo de trabajo más conocido.

-Basado en dos grandes ramas con infinito tiempo de vida (ramas master y develop) y varias ramas de apoyo:

* **Feature:** desarrollo de nuevas funcionalidades.

* **Hotfix:** arreglo de errores

* **Release:** preparación de nuevas versiones de producción.

2. GITHUB-FLOW:

Creado en 2011 por GitHub y se ajusta a sus funcionalidades. Está centrado en un modelo de desarrollo iterativo y de despliegue constante.

Principios básicos:

• Todo lo que está en **la rama master está listo para ser puesto en producción.**

• Para trabajar en algo nuevo, debes **crear una nueva rama a partir de la rama master** con un nombre descriptivo.

• Requerir información o integración de una rama local en la rama master, se debe abrir una p**ull request (solicitud de integración de cambios).**

• Revisión de los **cambios para fusionar con la rama master.**

3. GITLAB FLOW:

Creado en 2014 por Gitlab. Es una especie de extensión de GitHub Flow, pero trata de estandarizar aún más el proceso.

Además introduce otros tipos de ramas:

• **Ramas de entorno.** Por ejemplo pre-production production. Se crean a partir de la rama master cuando estamos listos para implementar nuestra aplicación.

• **Ramas de versión**. Por ejemplo 1.5-stable 1.6-stable.

4.ONE FLOW:

Creado en 2015 por Adam Ruka.

• Cada nueva versión de producción está basada en la versión previa de producción.

• La mayor diferencia entre One Flow y Git Flow es que One Flow no tiene rama de desarrollo.