

UD 3 - Optimización y documentación

C.F.G.S. Desarrollo de Aplicaciones Web

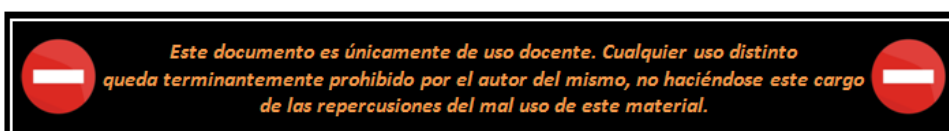
Francisco Jesús Delgado Almirón

Contenido

1.- Refactorización de código.....	2
1.1.- Limitaciones de la refactorización.....	3
1.2.- Patrones de refactorización más habituales.....	3
1.3.- Analizadores de código.....	4
1.4.- Refactorización y pruebas.....	5
1.5.- Herramientas de ayuda a la refactorización.....	6
2.- Control de versiones.....	7
2.1.- Estructura de herramientas de control de versiones.....	7
2.2.- Repositorio.....	8
2.3.- Herramientas de control de versiones.....	9
2.4.- Planificación de la gestión de configuraciones.....	10
2.5.- Gestión del cambio.....	10
2.6.- Gestión de versiones y entregas.....	12
2.7.- Herramientas CASE para la gestión de configuraciones.....	13
3.- Control de versiones en NetBeans: GIT.....	14
3.1.- ¿Qué es GitHub?.....	14
3.2.- Crear repositorio GitHub.....	15
3.3.- Inicializar un repositorio Git en un proyecto existente de Netbeans.....	17
3.4.- Subiendo nuevos archivos y cambios a nuestro repositorio Github.....	18
3.5.- Bajar cambios hechos por otros a nuestro repositorio local.....	22
3.6.- Clonar un repositorio Github desde Netbeans.....	22
4.- Documentación.....	24
4.1.- Uso de comentarios.....	24
4.2.- Documentación del código.....	25
4.3.- Herramientas.....	27
4.4.- Generar JavaDoc en NetBeans.....	27
Resumen de conceptos.....	30
Bibliografía.....	31



Esta obra está bajo una [Licencia Creative Commons Atribución- NoComercial-CompartirIgual 4.0 Internacional](https://creativecommons.org/licenses/by-nc-sa/4.0/).



1.- Refactorización de código

La refactorización es una disciplina técnica, que consiste en realizar pequeñas transformaciones en el código de un programa, para mejorar la estructura sin que cambie el comportamiento ni funcionalidad del mismo. Su objetivo es mejorar la estructura interna del código. Es una tarea que pretende limpiar el código minimizando la posibilidad de introducir errores.

Con la refactorización se mejora el diseño del software, hace que el software sea más fácil de entender, hace que el mantenimiento del software sea más sencillo, la refactorización nos ayuda a encontrar errores y a que nuestro programa sea más rápido.

Cuando se refactoriza se está mejorando el diseño del código después de haberlo escrito. Podemos partir de un mal diseño y, aplicando la refactorización, llegar a un código bien diseñado. Cada paso es simple, por ejemplo mover una propiedad desde una clase a otra, convertir determinado código en un nuevo método, etc. La acumulación de todos estos pequeños cambios pueden mejorar de forma ostensible el diseño.

Puedes visitar la siguiente página web (en inglés), donde se presenta el proceso de refactorización de aplicaciones Java con Netbeans.

<http://wiki.netbeans.org/Refactoring>

El concepto de refactorización de código, se base en el concepto matemático de factorización de polinomios.

Podemos definir el concepto de refactorización de dos formas:

- **Refactorización:** Cambio hecho en la estructura interna del software para hacerlo más fácil de entender y fácil de modificar sin modificar su comportamiento.
Ejemplos de refactorización es “Extraer Método” y “Encapsular Campos”. La refactorización es normalmente un cambio pequeño en el software que mejora su mantenimiento.
- **Campos encapsulados:** Se aconseja crear métodos getter y setter, (de asignación y de consulta) para cada campo que se defina en una clase. Cuando sea necesario acceder o modificar el valor de un campo, basta con invocar al método getter o setter según convenga.
- **Refactorizar:** Reestructurar el software aplicando una serie de refactorizaciones sin cambiar su comportamiento.

El propósito de la refactorización es hacer el software más fácil de entender y de modificar. Se pueden hacer muchos cambios en el software que pueden hacer algún pequeño cambio en el comportamiento observable. Solo los cambios hechos para hacer el software más fácil de entender son refactorizaciones.

Hay que diferenciar la refactorización de la optimización. En ambos procesos, se pretende mejorar la estructura interna de una aplicación o componente, sin modificar su comportamiento. Sin embargo, cuando se optimiza, se persigue una mejora del rendimiento, por ejemplo mejorar la velocidad de ejecución, pero esto puede hacer un código más difícil de entender.

Hay que resaltar que la refactorización no cambia el comportamiento observable del software. El software sigue cumpliendo la misma función que hacía antes. Ningún usuario, ya sea usuario final u otro programador, podrá determinar qué cosas han cambiado.

1.1.- Limitaciones de la refactorización

La refactorización es una técnica lo suficientemente novedosa para conocer cuáles son los beneficios que aporta, pero falta experiencia para conocer el alcance total de sus limitaciones. Se ha constatado que la refactorización presenta problemas en algunos aspectos del desarrollo.

Un área problemática de la refactorización son las bases de datos. Una base de datos presenta muchas dificultades para poder ser modificada, dado la gran cantidad de interdependencias que soporta. Cualquier modificación que se requiera de las bases de datos, incluyendo modificación de esquema y migración de datos, puede ser una tarea muy costosa. Es por ello que la refactorización de una aplicación asociada a una base de datos, siempre será limitada, ya que la aplicación dependerá del diseño de la base de datos.

Otra limitación, es cuando cambiamos interfaces. Cuando refactorizamos, estamos modificando la estructura interna de un programa o de un método. El cambio interno no afecta al comportamiento ni a la interfaz. Sin embargo, si renombramos un método, hay que cambiar todas las referencias que se hacen a él. Siempre que se hace esto se genera un problema si es una interfaz pública. Una solución es mantener las dos interfaces, la nueva y la vieja, ya que si es utilizada por otra clase o parte del proyecto, no podrá referenciarla.

Hay determinados cambios en el diseño que son difíciles de refactorizar. Es muy difícil refactorizar cuando hay un error de diseño o no es recomendable refactorizar, cuando la estructura a modificar es de vital importancia en el diseño de la aplicación.

Hay ocasiones en las que no debería refactorizar en absoluto. Nos podemos encontrar con un código que, aunque se puede refactorizar, sería más fácil reescribirlo desde el principio. Si un código no funciona, no se refactoriza, se reescribe.

1.2.- Patrones de refactorización más habituales

En el proceso de refactorización, se siguen una serie de patrones preestablecidos, los más comunes son los siguientes:

- **Renombrado (rename)**: Este patrón nos indica que debemos cambiar el nombre de un paquete, clase, método o campo, por un nombre más significativo.
- **Sustituir bloques de código por un método**: Este patrón nos aconseja sustituir un bloque de código, por un método. De esta forma, cada vez que queramos acceder a ese bloque de código, bastaría con invocar al método.
- **Campos encapsulados**: Se aconseja crear métodos getter y setter, (de asignación y de consulta) para cada campo que se defina en una clase. Cuando sea necesario acceder o modificar el valor de un campo, basta con invocar al método getter o setter según convenga.
- **Mover la clase**: Si es necesario, se puede mover una clase de un paquete a otro, o de un proyecto a otro. La idea es no duplicar código que ya se haya generado. Esto impone la actualización en todo el código fuente de las referencias a la clase en su nueva localización.
- **Borrado seguro**: Se debe comprobar, que cuándo un elemento del código ya no es necesario, se han borrado todas las referencias a él que había en cualquier parte del proyecto.
- **Cambiar los parámetros del proyecto**: Nos permite añadir nuevos parámetros a un método y cambiar los modificadores de acceso.
- **Extraer la interfaz**: Crea una nueva interfaz de los métodos public non-static seleccionados en una clase o interfaz.
- **Mover del interior a otro nivel**: Consiste en mover una clase interna a un nivel superior en la jerarquía.

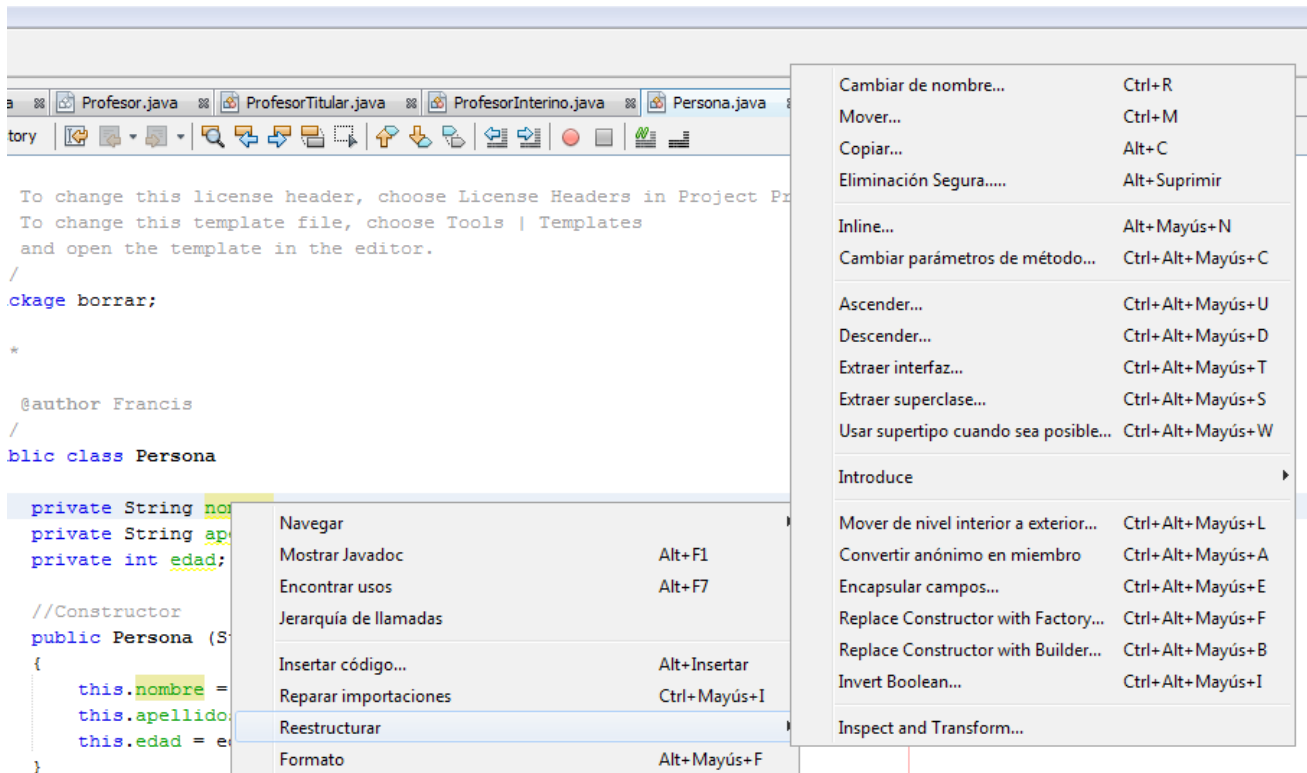


Ilustración 1: Refactorización de una variable en NetBeans

1.3.- Analizadores de código

El análisis estático de código, es un proceso que tiene como objetivo, evaluar el software, sin llegar a ejecutarlo.

Esta técnica se va a aplicar directamente sobre el código fuente, para poder obtener información que nos permita mejorar la base de código, pero sin que se modifique la semántica.

Los analizadores de código, son las herramientas encargadas de realizar esta labor. El analizador estático de código recibirá el código fuente de nuestro programa, lo procesará intentando averiguar la funcionalidad del mismo, y nos dará sugerencias, o nos mostrará posibles mejoras.

Los analizadores de código incluyen analizadores léxicos y sintácticos que procesan el código fuente y de un conjunto de reglas que se deben aplicar sobre determinadas estructuras. Si el analizador considera que nuestro código fuente tiene una estructura mejorable, nos lo indicará y también nos comunicará la mejora a realizar.

Las principales funciones de los analizadores es encontrar partes del código que puedan reducir el rendimiento, provocar errores en el software, tener una excesiva complejidad, complicar el flujo de datos, crear problemas de seguridad.

El análisis puede ser automático o manual. El automático, los va a realizar un programa, que puede formar parte de la funcionalidad de un entorno de desarrollo, por ejemplo el FindBugs en NetBeans, o manual, cuando es una persona.

El análisis automático reduce la complejidad para detectar problemas de base en el código, ya que los busca siguiendo una serie de reglas predefinidas. El análisis manual, se centra en apartados de nuestra propia aplicación, como comprobar que la arquitectura de nuestro software es correcta.

Tomando como base el lenguaje de programación Java, nos encontramos en el mercado un conjunto de analizadores disponibles:

- **PMD**. Esta herramienta basa su funcionamiento en detectar patrones, que son posibles errores en tiempo de ejecución, código que no se puede ejecutar nunca porque no se puede llegar a él, código que puede ser optimizado, expresiones lógicas que pueden ser simplificadas, malos usos del lenguaje, etc.
- **CPD**. Forma parte del PMD. Su función es encontrar código duplicado.

1.4.- Refactorización y pruebas

En la actualidad, la refactorización y las pruebas, son dos aspectos del desarrollo de aplicaciones, que por sus implicaciones y su interrelación, se han convertido en conceptos de gran importancia para la industria. Muchas cuestiones y problemas siguen sin ser explorados en estos dos ámbitos. Uno de los enfoques actuales, que pretende integrar las pruebas y la refactorización, es el Desarrollo Guiado por Pruebas (TDD, Test Driven Development).

Con el Desarrollo Guiado por Pruebas (TDD), se propone agilizar el ciclo de escritura de código, y realización de pruebas de unidad.

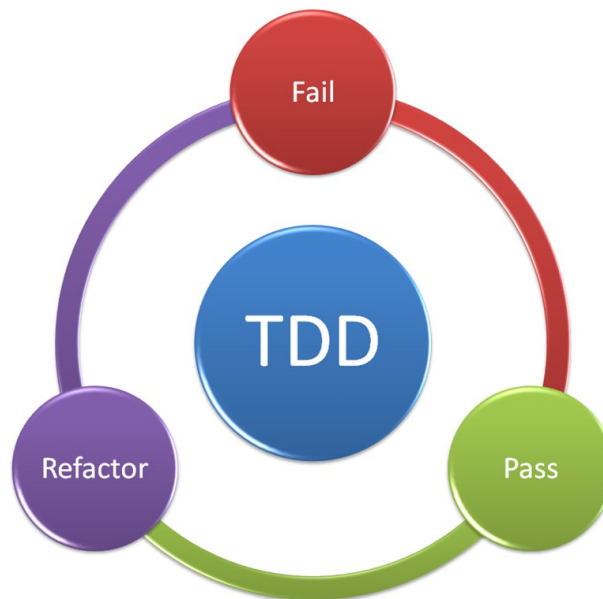


Ilustración 2: Desarrollo guiado por pruebas TDD

Cabe recordar, que el objetivo de las pruebas de unidad, es comprobar la calidad de un módulo desarrollado. Existen utilidades que permiten realizar esta labor, pudiendo ser personas distintas a las que los programan, quienes los realicen. Esto provoca cierta competencia entre los programadores de la unidad, y quienes tienen que realizar la prueba. El proceso de prueba, supone siempre un gasto de tiempo importante, ya que el programador realiza revisiones y depuraciones del mismo antes de enviarlo a prueba.

Durante el proceso de pruebas hay que diseñar los casos de prueba y comprobar que la unidad realiza correctamente su función. Si se encuentran errores, éstos se documentan, y son enviados al programador para que lo subsane, con lo que debe volver a sumergirse en un código que ya había abandonado.

Con el Desarrollo Guiado por Pruebas, la propuesta que se hace es totalmente diferente. El programador realiza las pruebas de unidad en su propio código, e implementa esas pruebas antes de escribir el código a ser probado.

Cuando un programador recibe el requerimiento para implementar una parte del sistema, empieza por pensar el tipo de pruebas que va a tener que pasar la unidad que debe elaborar, para que sea correcta. Cuando ya tiene claro la prueba que debe de pasar, pasa a programar las pruebas que debe pasar el código que debe de programar, no la unidad en sí. Cuando se han implementado las pruebas, se comienza a implementar la unidad, con el objeto de poder pasar las pruebas que diseñó previamente.

Cuando el programador empieza a desarrollar el código que se le ha encomendado, va elaborando pequeñas versiones que puedan ser compiladas y pasen por alguna de las pruebas. Cuando se hace un cambio y vuelve a compilar también ejecuta las pruebas de unidad. Y trata de que su programa vaya pasando más y más pruebas hasta que no falle en ninguna, que es cuando lo considera listo para ser integrado con el resto del sistema.

Para realizar la refactorización siguiendo TDD, se refactoriza el código tan pronto como pasa las pruebas para eliminar la redundancia y hacerlo más claro. Existe el riesgo de que se cometan errores durante la tarea de refactorización, que se traduzcan en cambios de funcionalidad y, en definitiva, en que la unidad deje de pasar las pruebas. Tratándose de reescrituras puramente sintácticas, no es necesario correr ese riesgo: las decisiones deben ser tomadas por un humano, pero los detalles pueden quedar a cargo de un programa que los trate automáticamente.

1.5.- Herramientas de ayuda a la refactorización

Los entornos de desarrollo actuales, nos proveen de una serie de herramientas que nos facilitan la labor de refactorizar nuestro código. En puntos anteriores, hemos indicado algunos de los patrones que se utilizan para refactorizar el código. Esta labor se puede realizar de forma manual, pero supone una pérdida de tiempo, y podemos inducir a redundancias o a errores en el código que modificamos.

En el Entorno de Desarrollo NetBeans, la refactorización está integrada como una función más, de las utilidades que incorpora.

A continuación, vamos a usar los patrones más comunes de refactorización, usando las herramientas de ayuda del entorno.

- **Renombrar**. Ya hemos indicado en puntos anteriores, que podemos cambiar el nombre de un paquete, clase, método o campo para darle un nombre más significativo. NetBeans nos permite hacerlo, de forma que actualizará todo el código fuente de nuestro proyecto donde se haga referencia al nombre modificado.
- **Introducir método**. Con este patrón podemos seleccionar un conjunto de código, y reemplazarlo por un método.
- **Encapsular campos**. NetBeans puede automáticamente generar métodos getter y setter para un campo, y opcionalmente actualizar todas las referencias al código para acceder al campo, usando los métodos getter y setter.

2.- Control de versiones

Con el término versión, se hace referencia a la evolución de un único elemento, o de cada elemento por separado, dentro de un sistema en desarrollo.

Siempre que se está realizando una labor, sea del tipo que sea, es importante saber en cada momento de que estamos tratando, qué hemos realizado y qué nos queda por realizar.

En el caso del desarrollo de software ocurre exactamente lo mismo. Cuando estamos desarrollando software, el código fuente está cambiando continuamente, siendo esta particularidad vital. Esto hace que en el desarrollo de software actual, sea de vital importancia que haya **sistemas de control de versiones**. Las ventajas de utilizar un sistema de control de versiones son múltiples. Un sistema de control de versiones bien diseñado facilita al equipo de desarrollo su labor, permitiendo que varios desarrolladores trabajen en el mismo proyecto (incluso sobre los mismos archivos) de forma simultánea, sin que se pisen unos a otros. Las herramientas de control de versiones proveen de un sitio central donde almacenar el código fuente de la aplicación, así como el historial de cambios realizados a lo largo de la vida del proyecto. También permite a los desarrolladores volver a una versión estable previa del código fuente si es necesario.

Una versión, desde el punto de vista de la evolución, se define como la forma particular de un objeto en un instante o contexto dado. Se denomina revisión, cuando se refiere a la evolución en el tiempo. Pueden coexistir varias versiones alternativas en un instante dado y hay que disponer de un método, para designar las diferentes versiones de manera sistemática u organizada.

En los entornos de desarrollo modernos, los sistemas de control de versiones son una parte fundamental, que van a permitir construir técnicas más sofisticadas como la Integración Continua.

En los proyectos Java, existen dos sistemas de control de versiones de código abierto, CVS y Subversion. La herramienta CVS es una herramienta de código abierto que es usada por gran cantidad de organizaciones. Subversion es el sucesor natural de CVS, ya que se adapta mejor que CVS a las modernas prácticas de desarrollo de software.

Para gestionar las distintas versiones que se van generando durante el desarrollo de una aplicación, los IDE, proporcionan herramientas de Control de Versiones y facilitan el desarrollo en equipo de aplicaciones.

2.1.- Estructura de herramientas de control de versiones

Las herramientas de control de versiones, suelen estar formadas por un conjunto de elementos, sobre los cuales, se pueden ejecutar órdenes e intercambiar datos entre ellos. Como ejemplo, vamos a analizar la herramienta CVS.

Una herramienta de control de versiones, como CVS, es un sistema de mantenimiento de código fuente (grupos de archivos en general) extraordinariamente útil para grupos de desarrolladores que trabajan cooperativamente usando alguna clase de red. CVS permite a un grupo de desarrolladores trabajar y modificar concurrentemente ficheros organizados en proyectos. Esto significa que dos o más personas pueden modificar un mismo fichero sin que se pierdan los trabajos de ninguna. Además, las operaciones más habituales son muy sencillas de usar.

CVS utiliza una arquitectura cliente-servidor: un servidor guarda la versión actual del proyecto y su historia, y los clientes conectan al servidor para sacar una copia completa del proyecto, trabajar en esa copia y entonces ingresar sus cambios. Típicamente, cliente y servidor conectan utilizando Internet, pero cliente y servidor pueden estar en la misma máquina. El servidor normalmente utiliza un sistema operativo similar a

Unix, mientras que los clientes CVS pueden funcionar en cualquier de los sistemas operativos más difundidos.

Los clientes pueden también comparar diferentes versiones de ficheros, solicitar una historia completa de los cambios, o sacar una "foto" histórica del proyecto tal como se encontraba en una fecha determinada o en un número de revisión determinado. Muchos proyectos de código abierto permiten el "acceso de lectura anónimo", significando que los clientes pueden sacar y comparar versiones sin necesidad de teclear una contraseña; solamente el ingreso de cambios requiere una contraseña en estos escenarios. Los clientes también pueden utilizar el comando de actualización con el fin de tener sus copias al día con la última versión que se encuentra en el servidor. Esto elimina la necesidad de repetir las descargas del proyecto completo.

El sistema de control de versiones está formado por un conjunto de componentes:

- **Repositorio:** Es el lugar de almacenamiento de los datos de los proyectos. Suele ser un directorio en algún ordenador.
- **Módulo:** En un directorio específico del repositorio. Puede identificar una parte del proyecto o ser el proyecto por completo.
- **Revisión:** Es cada una de las versiones parciales o cambios en los archivos o repositorio completo. La evolución del sistema se mide en revisiones. Cada cambio se considera incremental.
- **Etiqueta:** Información textual que se añade a un conjunto de archivos o a un módulo completo para indicar alguna información importante.
- **Rama:** Revisiones paralelas de un módulo para efectuar cambios sin tocar la evolución principal. Se suele emplear para pruebas o para mantener los cambios en versiones antiguas.

Las órdenes que se pueden ejecutar son:

- **Checkout:** obtiene una copia del trabajo para poder trabajar con ella.
- **Update:** actualiza la copia con cambios recientes en el repositorio.
- **Commit:** almacena la copia modificada en el repositorio.
- **Abort:** abandona los cambios en la copia de trabajo.

2.2.- Repositorio

El repositorio es la parte fundamental de un sistema de control de versiones. Almacena toda la información y datos de un proyecto.

El repositorio es un almacén general de versiones. En la mayoría de las herramientas de control de versiones, suele ser un directorio.

El repositorio centraliza todos los componentes de un mismo sistema, incluyendo las distintas versiones de cada componente. Con el repositorio, se va a conseguir un ahorro de espacio de almacenamiento, ya que estamos evitando guardar por duplicado, los elementos que son comunes a varias versiones. El repositorio nos va a facilitar el almacenaje de la información de la evolución del sistema, ya que, aparte de los datos en sí mismo, también almacena información sobre las versiones, temporización, etc.

El entorno de desarrollo integrado NetBeans usa como sistema de control de versiones CVS. Este sistema, tiene un componente principal, que es el repositorio. En el repositorio se deberán almacenar todos los ficheros de los proyectos, que puedan ser accedidos de forma simultánea por varios desarrolladores.

Cuando usamos un sistema de control de versiones, trabajamos de forma local, sincronizándonos con el repositorio, haciendo los cambios en nuestra copia local, realizando el cambio, se acomete el cambio en el repositorio. Para realizar la sincronización, en el entorno NetBeans, lo realizamos de varias formas:

- Abriendo un proyecto CVS en el IDE.
- Comprobando los archivos de un repositorio.
- Importando los archivos hacia un repositorio.

Si tenemos un proyecto CVS versionado, con el que hemos trabajado, podemos abrirlo en el IDE y podremos acceder a las características de versionado. El IDE escanea nuestros proyectos abiertos y si contienen directorios CVS, el estado del archivo y la ayuda-contextual se activan automáticamente para los proyectos de versiones CVS.

2.3.- Herramientas de control de versiones

Durante el proceso de desarrollo de software, donde todo un equipo de programadores están colaborando en el desarrollo de un proyecto software, los cambios son continuos. Es por ello necesario que existan en todos los lenguajes de programación y en todos los entornos de programación, herramientas que gestionen el control de cambios.

Si nos centramos en Java, actualmente destacan dos herramientas de control de cambios: CVS y Subversion. CVS es una herramienta de código abierto ampliamente utilizada en numerosas organizaciones. Subversion es el sucesor natural de CVS, está rápidamente integrándose en los nuevos proyectos Java, gracias a sus características que lo hacen adaptarse mejor a las modernas prácticas de programación Java. Estas dos herramientas de control de versiones, se integran perfectamente en los entornos de desarrollo para Java, como NetBeans y Eclipse.

Otras herramientas de amplia difusión son:

- **SourceSafe:** Es una herramienta que forma parte del entorno de desarrollo Microsoft Visual Studio.
- **Visual Studio Team Foundation Server:** Es el sustituto de Source Safe. Es un productor que ofrece control de código fuente, recolección de datos, informes y seguimiento de proyectos, y está destinado a proyectos de colaboración de desarrollo de software.
- **Darcs:** Es un sistema de gestión de versiones distribuido. Algunas de sus características son: la posibilidad de hacer commits locales (sin conexión), cada repositorio es una rama en sí misma, independencia de un servidor central, posibilidad de renombrar ficheros, varios métodos de acceso como local, ssh, http y ftp, etc.
- **Git:** Esta herramienta de control de versiones, diseñada por Linus Torvalds,
- **Mercurial:** Esta herramienta funciona en Linux, Windows y Mac OS X, Es un programa de línea de comandos. Es una herramienta que permite que el desarrollo se haga distribuido, gestionando de forma robusta archivos de texto y binarios. Tiene capacidades avanzadas de ramificación e integración. Es una herramienta que incluye una interfaz web para su configuración y uso.

2.4.- Planificación de la gestión de configuraciones.

Cuando se habla de la gestión de configuraciones, se está haciendo referencia a la evolución de todo un conjunto de elementos. Una configuración es una combinación de versiones particulares de los componentes que forman un sistema consistente. Desde el punto de vista de la evolución en el tiempo, es el conjunto de las versiones de los objetos componentes en un instante dado.

Una configuración puede cambiar porque se añaden, eliminan o se modifican elementos. También puede cambiar, debido a la reorganización de los componentes, sin que estos cambien.

Como consecuencia de lo expuesto, es necesario disponer de un método, que nos permita designar las diferentes configuraciones de manera sistemática y planificada. De esta forma se facilita el desarrollo de software de manera evolutiva, mediante cambios sucesivos aplicados a partir de una configuración inicial hasta llegar a una versión final aceptable del producto.

La Gestión de Configuraciones de Software se va a componer de cuatro tareas básicas:

- **Identificación.** Se trata de establecer estándares de documentación y un esquema de identificación de documentos.
- **Control de cambios.** Consiste en la evaluación y registro de todos los cambios que se hagan de la configuración software.
- **Auditorías de configuraciones.** Sirven, junto con las revisiones técnicas formales para garantizar que el cambio se ha implementado correctamente.
- **Generación de informes.** El sistema software está compuesto por un conjunto de elementos, que evolucionan de manera individual, por consiguiente, se debe garantizar la consistencia del conjunto del sistema.

La planificación de la Gestión de Configuraciones de Software va a comprender diferentes actividades:

- Introducción (propósito, alcance, terminología).
- Gestión de GCS (organización, responsabilidades, autoridades, políticas aplicables, directivas y procedimientos).
- Actividades GCS (identificación de la configuración, control de configuración, etc.).
- Agenda GCS (coordinación con otras actividades del proyecto).
- Recursos GCS (herramientas, recursos físicos y humanos).
- Mantenimiento de GCS.

2.5.- Gestión del cambio

Las herramientas de control de versiones no garantizan un desarrollo razonable, si cualquier componente del equipo de desarrollo de una aplicación puede realizar cambios e integrarlos en el repositorio sin ningún tipo de control. Para garantizar que siempre disponemos de una línea base para continuar el desarrollo, es necesario aplicar controles al desarrollo e integración de los cambios. El control de cambios es un mecanismo que sirve para la evaluación y aprobación de los cambios hechos a los elementos de configuración del software.

Pueden establecerse distintos tipos de control:

- **Control individual:** antes de aprobarse un nuevo elemento. Cuando un elemento de la configuración está bajo control individual, el programador responsable cambia la documentación cuando se requiere. El cambio se puede registrar de manera informal, pero no genera ningún documento formal.
- **Control de gestión u organizado:** conduce a la aprobación de un nuevo elemento. Implica un procedimiento de revisión y aprobación para cada cambio propuesto en la configuración. Como en el control individual, el control a nivel de proyecto ocurre durante el proceso de desarrollo pero es usado después de que haya sido aprobado un elemento de la configuración software. El cambio es registrado formalmente y es visible para la gestión.
- **Control formal:** se realiza durante el mantenimiento. Ocurre durante la fase de mantenimiento del ciclo de vida software. El impacto de cada tarea de mantenimiento se evalúa por un Comité de Control de Cambios, el cuál aprueba las modificaciones de la configuración software.

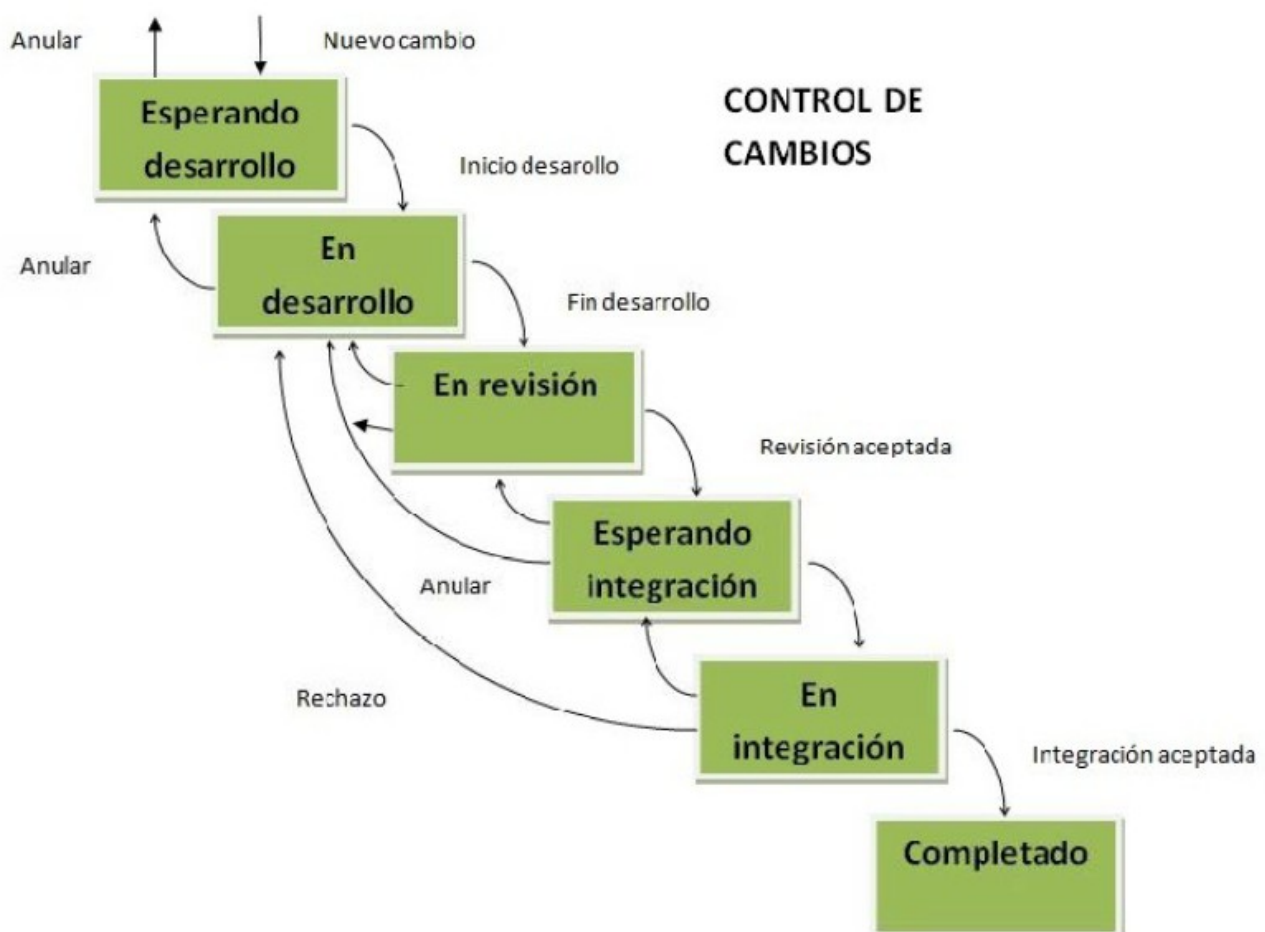


Ilustración 3: Control de cambios

2.6.- Gestión de versiones y entregas

Las versiones hacen referencia a la evolución de un único elemento, dentro de un sistema software. La evolución puede representarse en forma de grafo, donde los nodos son las versiones y los arcos corresponden a la creación de una versión a partir de otra ya existente.

Grafo de evolución simple: Las revisiones sucesivas de un componente dan lugar a una simple secuencia lineal. Esta evolución no presenta problemas en la organización del repositorio y las versiones se designan mediante números correlativos.

Variantes: En este caso, existen varias versiones del componente. El grafo ya no es una secuencia lineal, si no que adopta la forma de un árbol. La numeración de las versiones requerirá dos niveles. El primer número designa la variante (línea de evolución) y el segundo la versión particular (revisión) a lo largo de dicha variante.

La terminología que se usa para referirse a los elementos del grafo son:

- **Tronco (trunk)**: Es la variante principal.
- **Cabeza (head)**: Es la última versión del tronco.
- **Ramas (branches)**: Son las variantes secundarias.
- **Delta**: Es el cambio de una revisión respecto a la anterior.

Propagación de cambios: Cuando se tienen variantes que se desarrollan en paralelo, suele ser necesario aplicar un mismo cambio a varias variantes.

Fusión de variantes: En determinados momentos puede dejar de ser necesario mantener una rama independiente. En este caso se puede fundir con otra (MERGE).

Técnicas de almacenamiento: Como en la mayoría de los casos, las distintas versiones tienen en común gran parte de su contenido, se organiza el almacenamiento para que no se desaproveche espacio repitiendo los datos en común de varias versiones.

- **Deltas directos**: Se almacena la primera versión completa, y luego los cambios mínimos necesarios para reconstruir cada nueva versión a partir de la anterior.
- **Deltas inversos**: Se almacena completa la última versión del tronco y los cambios necesarios para reconstruir cada versión anterior a partir de la siguiente. En las ramas se mantiene el uso de los deltas directos.
- **Marcado selectivo**: Se almacena el texto refundido de todas las versiones como una secuencia lineal, marcando cada sección del conjunto con los números de versiones que corresponde.

En cuanto a la gestión de entregas, en primer lugar definimos el concepto de entrega como una instancia de un sistema que se distribuye a los usuarios externos al equipo de desarrollo.

La planificación de la entrega se ocupa de cuándo emitir una versión del sistema como una entrega. La entrega está compuesta por el conjunto de programas ejecutables, los archivos de configuración que definan como se configura la entrega para una instalación particular, los archivos de datos que se necesitan para el funcionamiento del sistema, un programa de instalación para instalar el sistema en el hardware de destino, documentación electrónica y en papel, y, el embalaje y publicidad asociados, diseñados para esta entrega. Actualmente los sistemas se entregan en discos ópticos (CD o DVD) o como archivos de instalación descargables desde la red.



Ilustración 4: Gestión del control de versiones

2.7.- Herramientas CASE para la gestión de configuraciones

Los procesos de gestión de configuraciones están estandarizados y requieren la aplicación de procedimientos predefinidos, ya que hay que gestionar gran cantidad de datos. Cuando se construye un sistema a partir de versiones de componentes, un error de gestión de configuraciones, puede implicar que el software no trabaje correctamente. Por todo ello, las herramientas CASE de apoyo son imprescindibles para la gestión de configuraciones.

Las herramientas se pueden combinar con entornos de trabajo de gestión de configuraciones. Hay dos tipos de entornos de trabajo de Gestión de Configuraciones:

- **Entornos de trabajo abiertos:** Las herramientas de cada una de Gestión de Configuraciones son integradas de acuerdo con procedimientos organizacionales estándar. Nos encontramos con bastantes herramientas de Gestión de Configuraciones comerciales y open-source disponibles para propósitos específicos. La gestión de cambios se puede llevar a cabo con herramientas de seguimiento (bug-tracking) como Bugzilla, la gestión de versiones a través de herramientas como RCS o CVS, y la construcción del sistema con herramientas como Make o Imake. Estas herramientas son open-source y están disponibles de forma gratuita.
- **Entornos integrados:** Estos entornos ofrecen facilidades integradas para gestión de versiones, construcción del sistema o seguimiento de los cambios. Por ejemplo, está el proceso de control de Cambios Unificado de Rational, que se basa en un entorno de Gestión de Configuraciones que incorpora ClearCase para la construcción y gestión de versiones del sistema y ClearQuest para el seguimiento de los cambios. Los entornos de Gestión de Configuraciones integrados, ofrecen la ventaja de un intercambio de datos sencillos, y el entorno ofrece una base de datos de Gestión de Configuraciones integrada.

3.- Control de versiones en NetBeans: GIT

3.1.- ¿Qué es GitHub?

GitHub es una plataforma de desarrollo colaborativo de software para alojar proyectos utilizando el sistema de control de versiones Git.

El código se almacena de forma pública, aunque también se puede hacer de forma privada, creando una cuenta de pago.

¿Para qué sirve?

GitHub aloja tu repositorio de código y te brinda herramientas muy útiles para el trabajo en equipo, dentro de un proyecto.

Además de eso, puedes contribuir a mejorar el software de los demás. Para poder alcanzar esta meta, GitHub provee de funcionalidades para hacer un fork y solicitar pulls.

Realizar un fork es simplemente clonar un repositorio ajeno (genera una copia en tu cuenta), para eliminar algún bug o modificar cosas de él. Una vez realizadas tus modificaciones puedes enviar un pull al dueño del proyecto. Éste podrá analizar los cambios que has realizado fácilmente, y si considera interesante tu contribución, adjuntarlo con el repositorio original.

¿Qué herramientas proporciona?

En la actualidad, GitHub es mucho más que un servicio de alojamiento de código. Además de éste, se ofrecen varias herramientas útiles para el trabajo en equipo. Entre ellas, cabe destacar:

- Una wiki para el mantenimiento de las distintas versiones de las páginas.
- Un sistema de seguimiento de problemas que permiten a los miembros de tu equipo detallar un problema con tu software o una sugerencia que deseen hacer.
- Una herramienta de revisión de código, donde se pueden añadir anotaciones en cualquier punto de un fichero y debatir sobre determinados cambios realizados en un commit específico.
- Un visor de ramas donde se pueden comparar los progresos realizados en las distintas ramas de nuestro repositorio.



Ilustración 5: Logo GitHub

3.2.- Crear repositorio GitHub

El IDE Netbeans, cuenta con soporte nativo para Git en todos los proyectos que manejamos dentro del IDE, una vez que comenzamos a versionar un proyecto, Netbeans nos mostrará cambios, eliminaciones y archivos o componentes agregados al proyecto. Para utilizar Git desde Netbeans podemos inicializar un repositorio en algún proyecto existente, o clonar un proyecto que ya está versionado desde Netbeans.

El primer paso será crearnos una cuenta en github donde podremos subir nuestros proyectos. La web de github es la siguiente:

<https://github.com/>

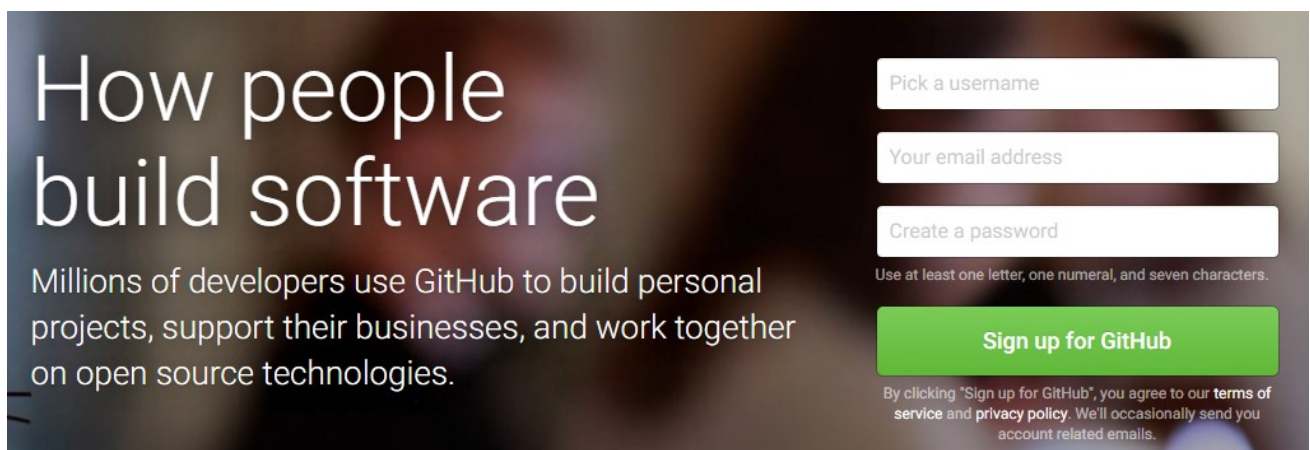


Ilustración 6: Registro en GitHub

Una vez registrados tendremos que crear un repositorio para ahí poder guardar nuestro proyecto, pulsamos en *New repository*:

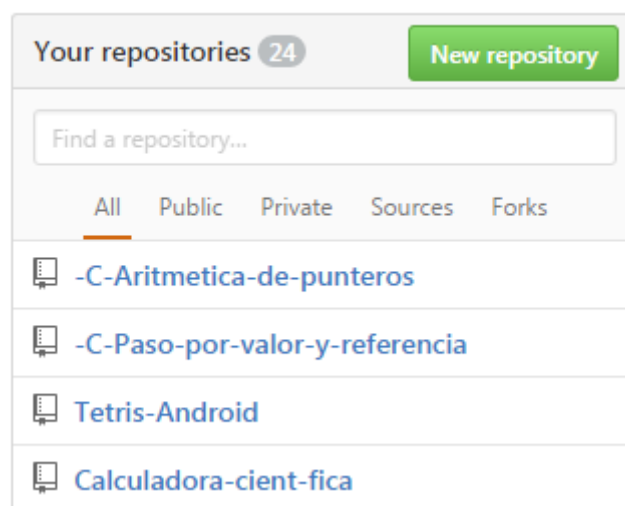


Ilustración 7: Nuevo Repositorio


Nos aparecerá una pantalla como la siguiente:

Create a new repository

A repository contains all the files for your project, including the revision history.

Owner


Repository name

 galleta ▾


 /

Great repository names are short and memorable. Need inspiration? How about **literate-goggles**.

Description (optional)

☒  Public

Anyone can see this repository. You choose who can commit.

☐  Private

You choose who can see and commit to this repository.


☐ Initialize this repository with a README

This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: None ▾

 |

Add a license: None ▾



Create repository

Ilustración 8: Creando un nuevo repositorio

Rellenamos el nombre de repositorio, elegimos si queremos que sea público o privado y pulsamos *Create repository*.

Una vez hecho todo estoy ya tenemos nuestro repositorio creado y funcionando, ahora copiamos la dirección del repositorio que nos hará falta, en este caso la dirección es: <https://github.com/galleta/pruebaNetBeans> .

3.3.- Inicializar un repositorio Git en un proyecto existente de Netbeans

Si ya tenemos iniciado algún proyecto en Netbeans y queremos comenzar a versionarlo con Git, simplemente hacemos clic derecho sobre nuestro proyecto (En el explorador de proyectos), a continuación hacemos clic en Versioning –> Initialize Git Respository.

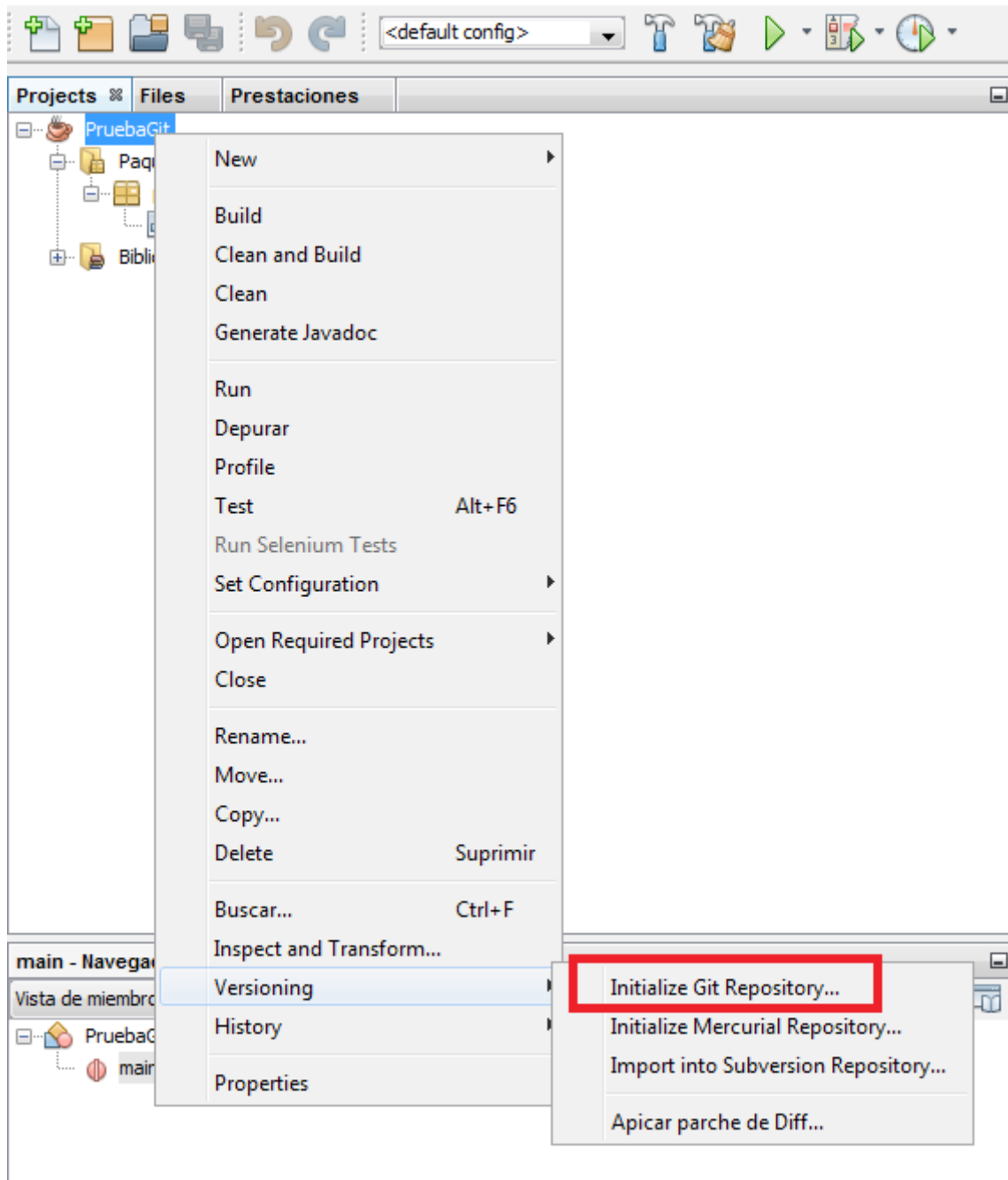


Ilustración 9: Inicializar repositorio

Luego de esto Netbeans nos pedirá que seleccionemos un directorio para el repositorio, por default nos pondrá el directorio en el que se almacena nuestro proyecto, preferentemente utilizaremos este directorio.

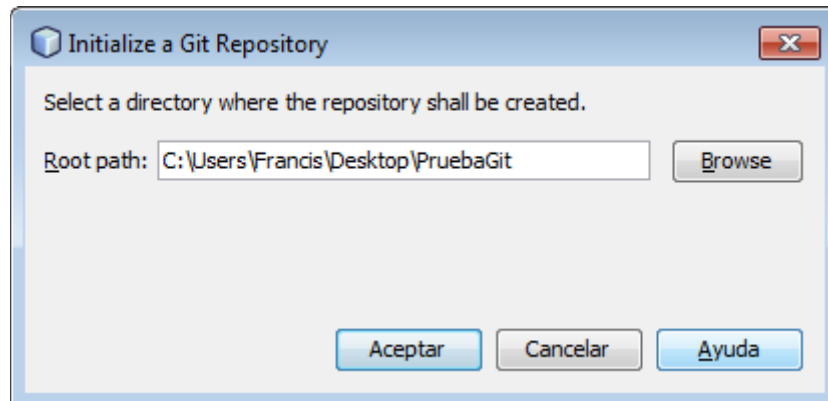


Ilustración 10: Selección del directorio del repositorio

Listo, ahora nuestro proyecto será versionado por Git aprovechando las herramientas de análisis de versiones que Netbeans provee.

Si todo ha ido correctamente nuestro proyecto deberá verse más o menos de la siguiente forma:

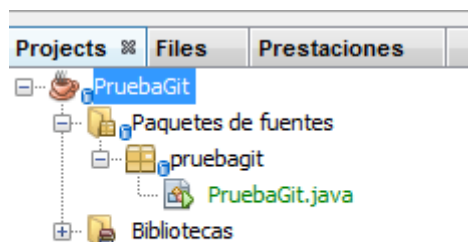


Ilustración 11: Proyecto enlazado a GitHub correctamente

3.4.- Subiendo nuevos archivos y cambios a nuestro repositorio Github

Ahora supongamos que hemos pasado un tiempo trabajando en nuestro proyecto, hemos hecho algunos cambios, resuelto algunos bugs y agregado nuevos componentes. Podremos ver que netbeans nos resalta los cambios en color azul y los nuevos archivos en verde en el explorador de proyectos. Para subir nuevos cambios seguiremos siempre dos sencillos pasos: **Commit** y **Push**. Para el primero, hacemos clic derecho sobre nuestro proyecto y elegimos Git -> Commit, en la ventana que aparece podemos introducir algún comentario que distinguirá este commit de los demás. También podemos elegir los archivos nuevos o con modificaciones que se enviarán al repositorio. Por default todos los archivos nuevos o con modificaciones aparecen seleccionados.

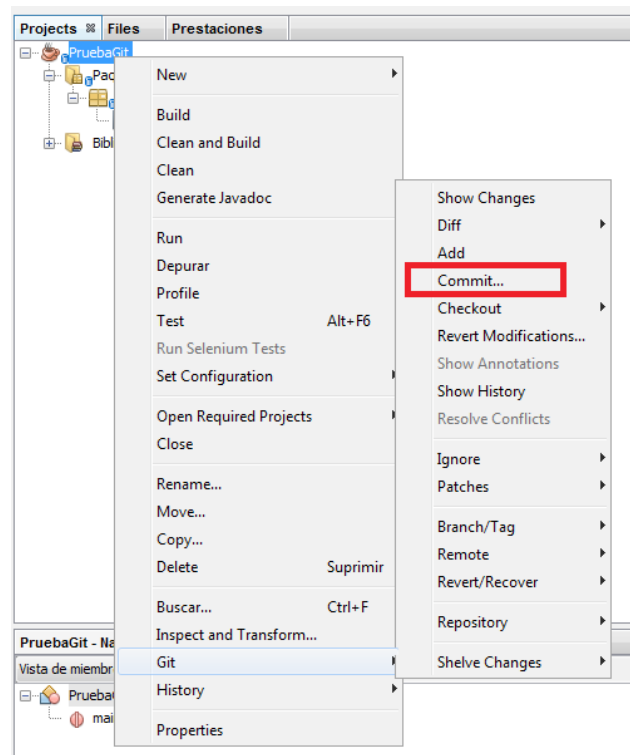


Ilustración 12: Comando Commit

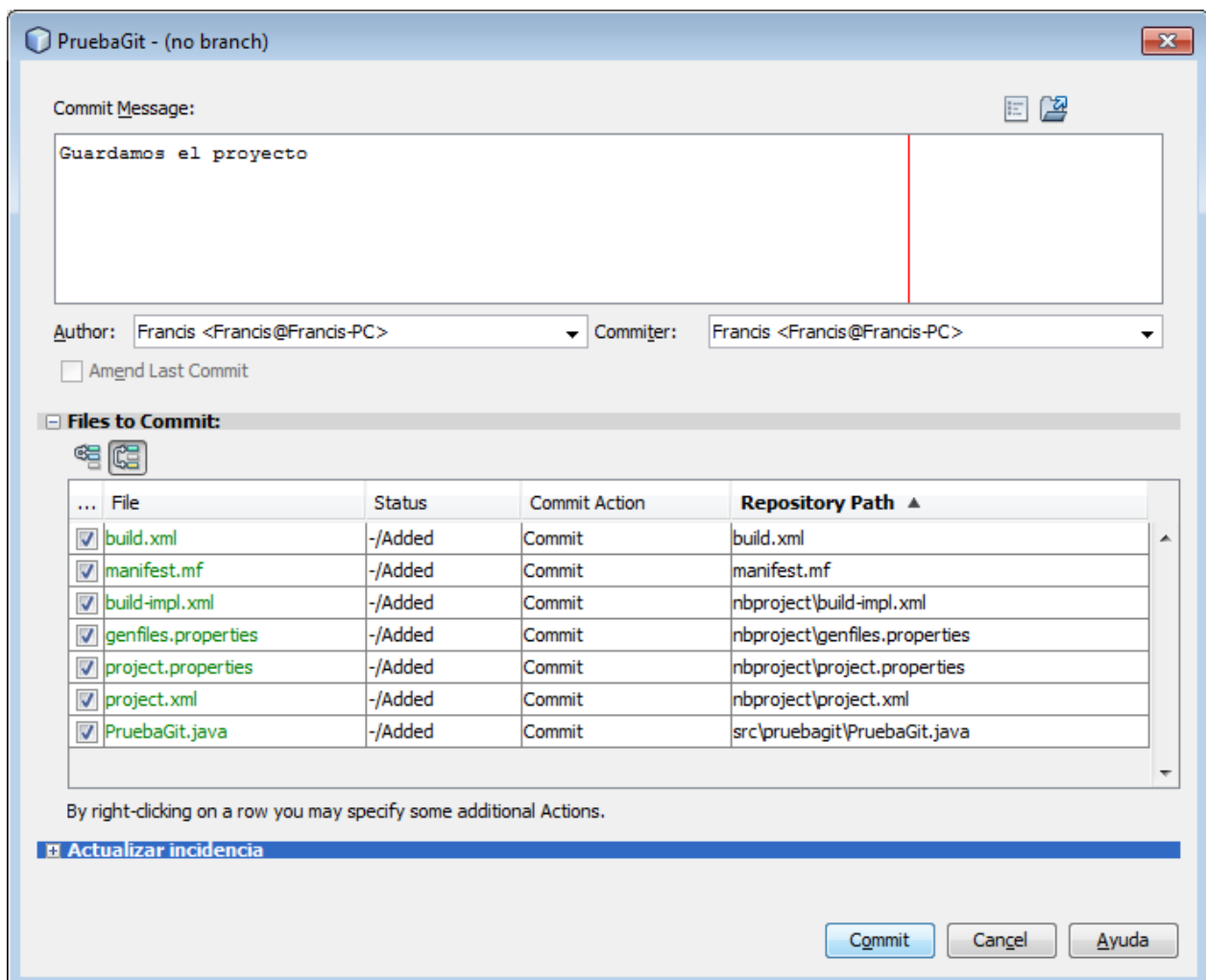


Ilustración 13: Pantalla del comando Commit

Hacemos clic en el botón 'commit' y listo, nuestros cambios se han enviado al repositorio local. Ahora para subirlos al repositorio Github, nuevamente hacemos clic con el botón secundario sobre nuestro proyecto y elegimos: 'Git —> Remote —> Push...'

En la ventana emergente debemos introducir el repositorio remoto, nuestro usuario y contraseña de Github, lo cual Netbeans ya hace por nosotros, así que en la mayoría de los casos, nos limitaremos a hacer clic en 'Next'.

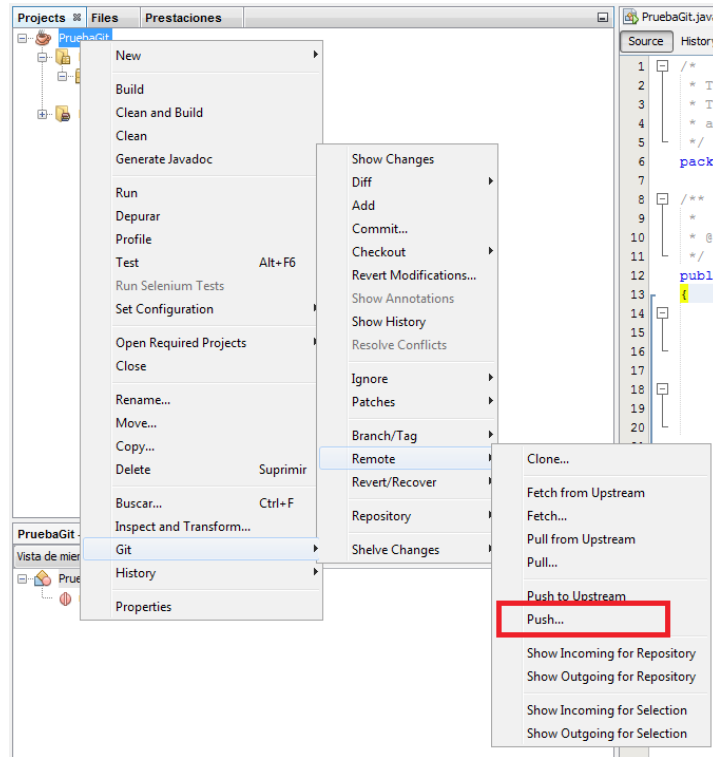


Ilustración 14: Comando Push

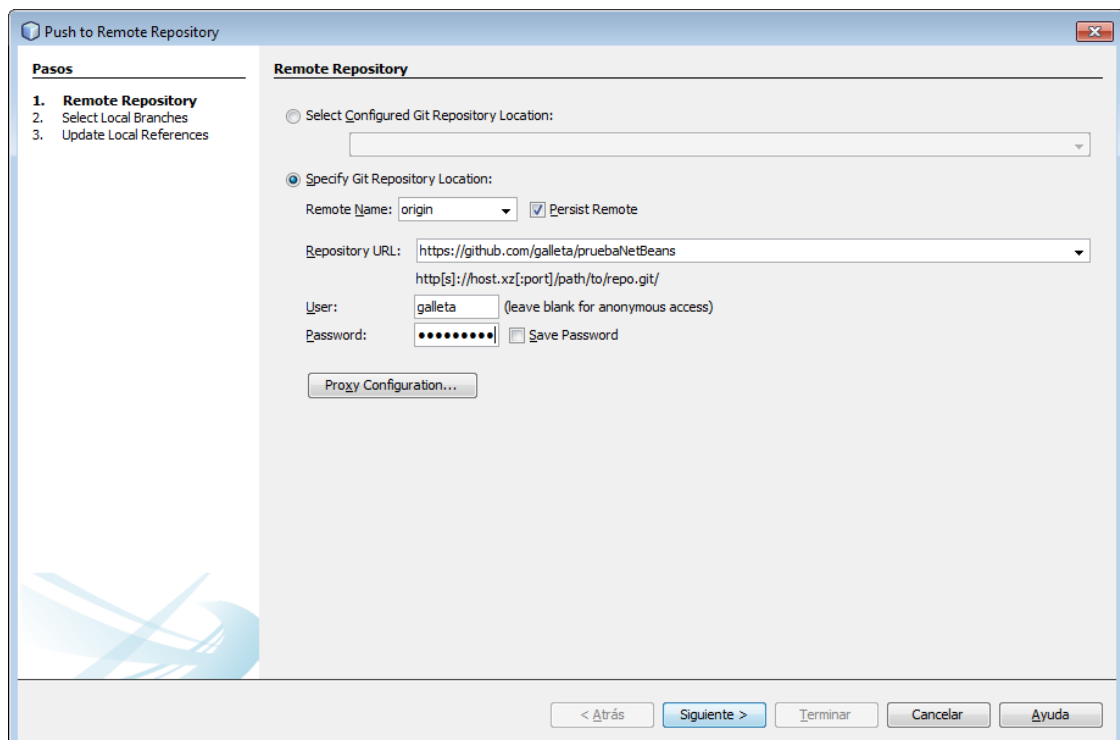


Ilustración 15: Pantalla del comando Push

Seleccionamos el Branch Master.

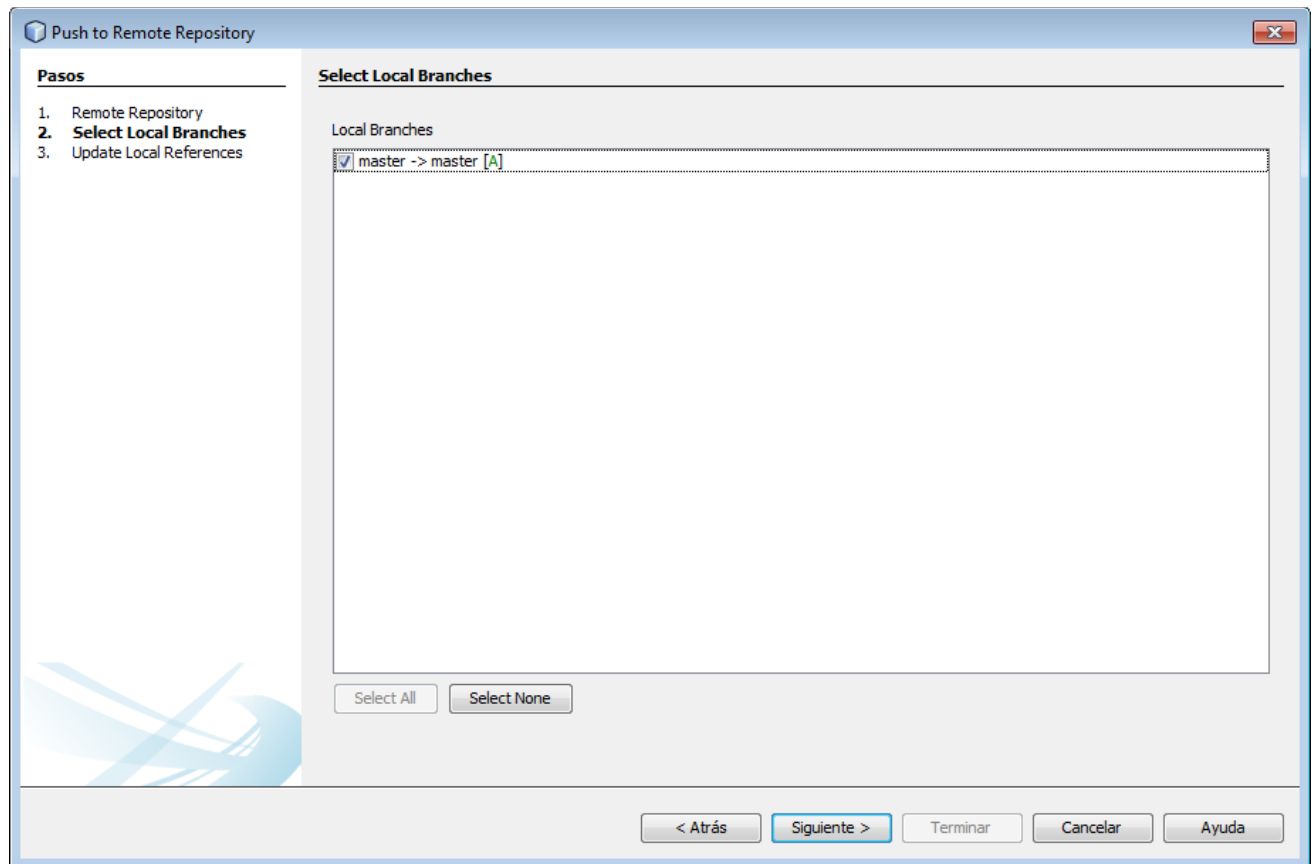


Ilustración 16: Selección del Branch para hacer el Push

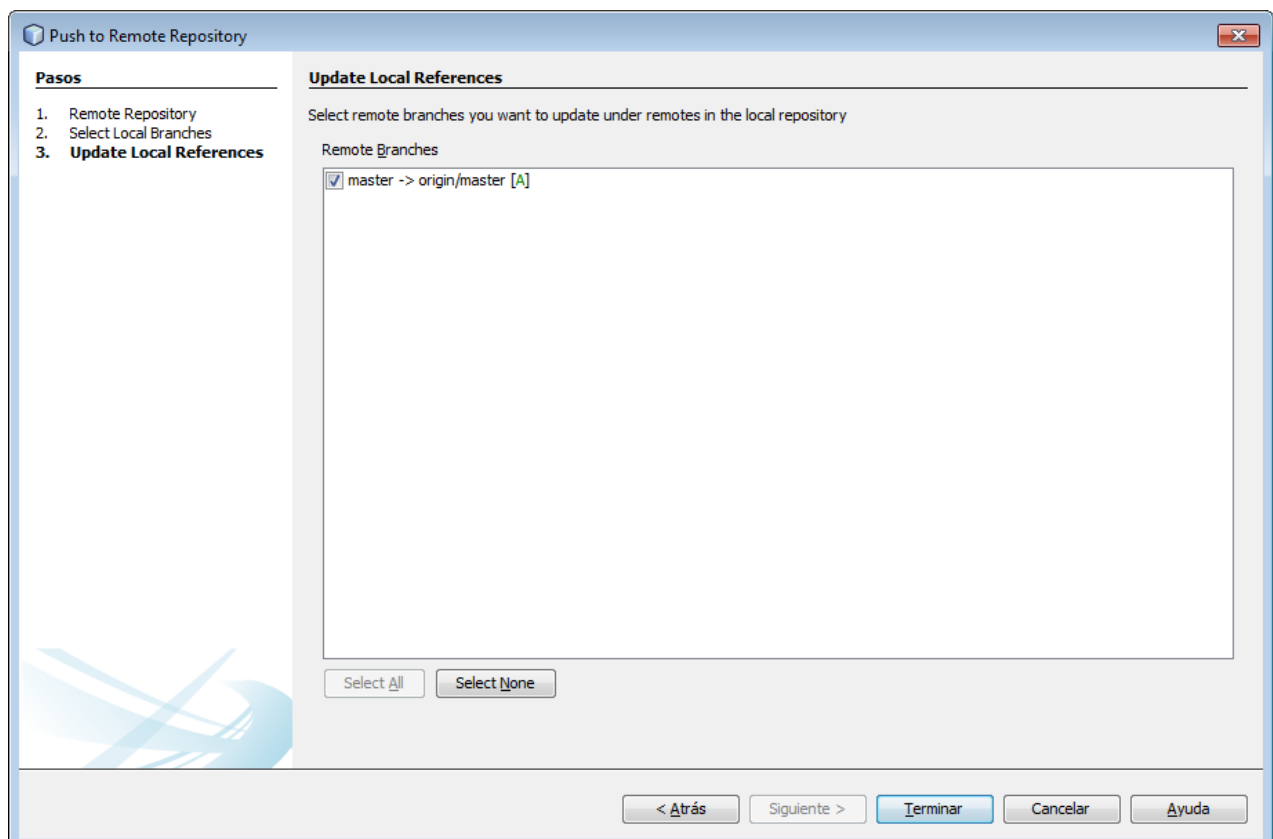


Ilustración 17: Selección del Branch para hacer el Push II

Si ahora nos vamos a nuestro git veremos que se han efectuado los cambios y está guardado nuestro proyecto de NetBeans.

Repositorio de prueba para un proyecto de NetBeans — Edit

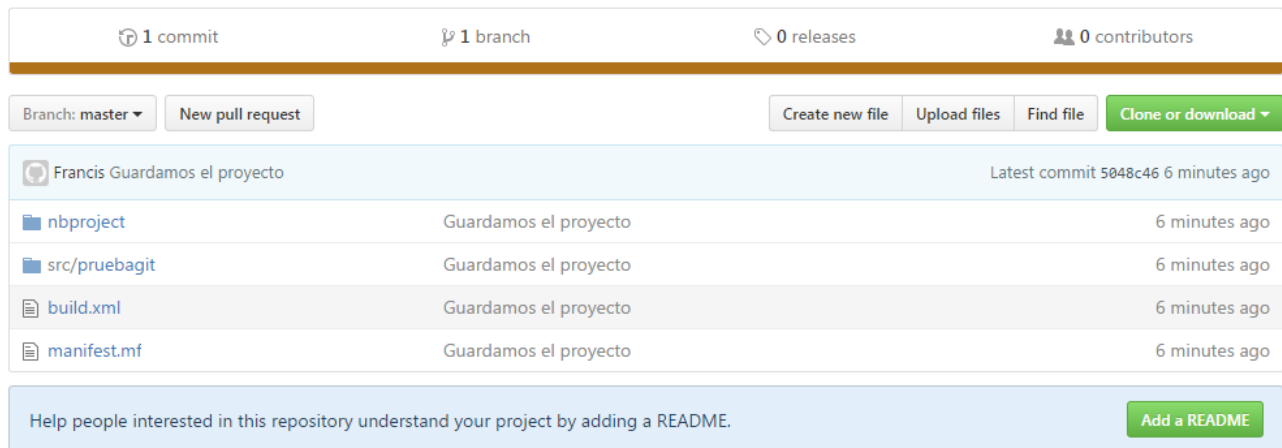


Ilustración 18: Repositorio creado correctamente y actualizado

3.5.- Bajar cambios hechos por otros a nuestro repositorio local

En caso de que alguien más haya subido cambios al repositorio Github o en cualquier caso queramos actualizar nuestro repositorio local con el de Github, debemos hacer un pull para lo cual nuevamente hacemos clic derecho sobre nuestro proyecto y elegimos: 'Git → Remote → Pull...' y se lanzará un ventana con los datos del repositorio y nuestro usuario – contraseña de github. Hacemos clic en Net , a continuación elegimos la rama que queremos revisar y hacemos clic en Finish. Con esto nuestro proyecto local se actualizara con los últimos cambios encontrados en Github.

3.6.- Clonar un repositorio Github desde Netbeans

Si antes de utilizar git dentro de Netbeans, ya tenemos algún proyecto alojado en Github o si queremos descargar el source de algun proyecto de código libre que utilice git para versionar su código podemos hacerlo sin problemas desde Netbeans. Para ello vamos a el menu: Team –> Git –> Clone...

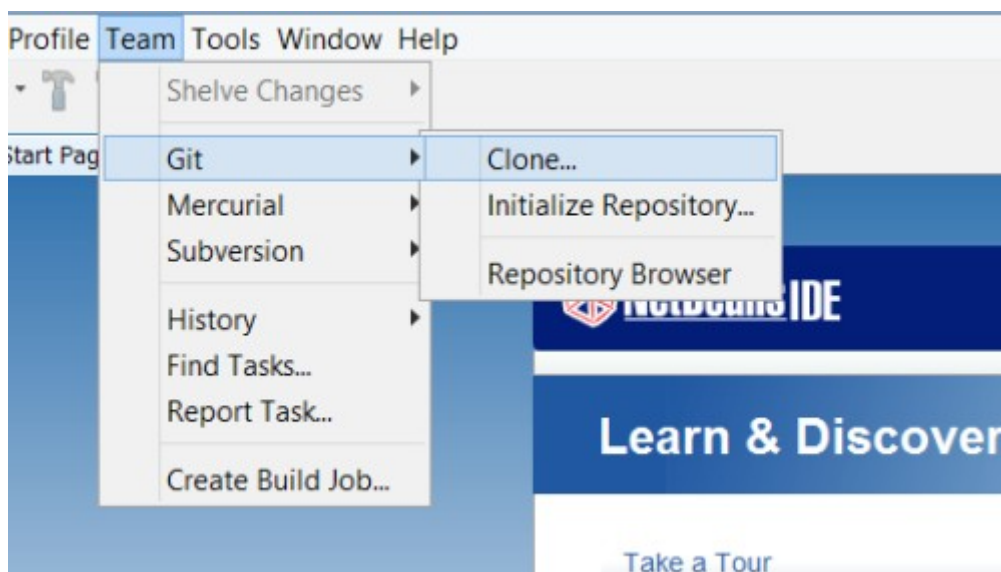


Ilustración 19: Comando Clone

A continuación Netbeans nos pedirá la URL del repositorio que pretendemos clonar.

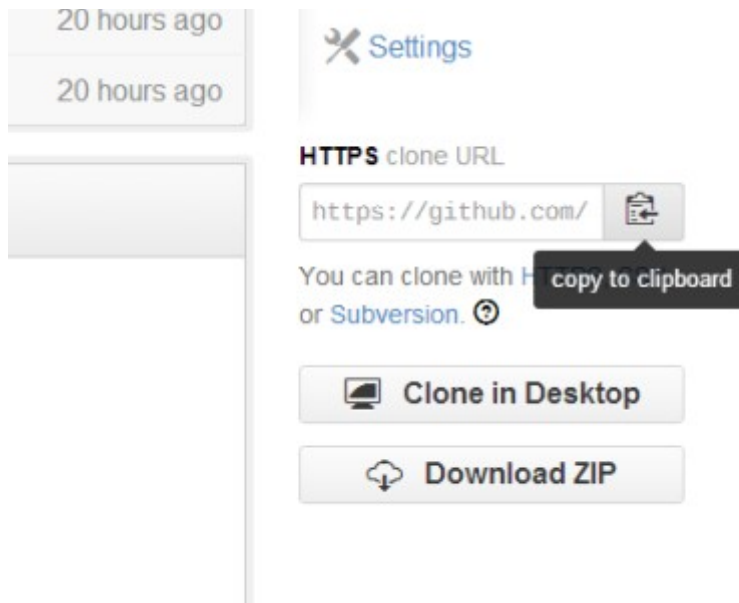


Ilustración 20: Copiar URL repositorio Git

Además de pegar la dirección del repositorio, también introducimos nuestro usuario y contraseña de Github.

Posteriormente hacemos clic en Next y a continuación elegimos la rama que deseamos clonar, en este caso la rama 'master' y hacemos clic en 'Next'.

En la siguiente ventana, elegimos en que directorio deseamos que se clone el proyecto, así como el nombre que le daremos a nuestro repositorio (De preferencia dejar el que viene por default), activamos la casilla que dice: 'Scan for netbeans projects after Clone' y hacemos clic en 'Finish'.

Con esto se habrá clonado el proyecto y podremos abrirlo desde Netbeans.

4.- Documentación

El proceso de documentación de código, es uno de los aspectos más importantes de la labor de un programador.

Documentar el código nos sirve para explicar su funcionamiento, punto por punto, de forma que cualquier persona que lea el comentario, puede entender la finalidad del código.

La labor de documentación es fundamental para la detección de errores y para su mantenimiento posterior, que en muchos casos, es realizado por personas diferentes a las que intervinieron en su creación. Hay que tener en cuenta que todos los programas tienen errores y todos los programas sufren modificaciones a lo largo de su vida.

La documentación añade explicaciones de la función del código, de las características de un método, etc. Debe tratar de explicar todo lo que no resulta evidente. Su objetivo no es repetir lo que hace el código, sino explicar por qué se hace.

La documentación explicará cual es la finalidad de un clase, de un paquete, qué hace un método, para que sirve una variable, qué se espera del uso de una variable, qué algoritmo se usa, por qué hemos implementado de una manera y de otro, qué se podría mejorar en el futuro, etc.

El siguiente enlace nos muestra el estilo de programación a seguir en Java, así como la forma de documentar y realizar comentarios de un código. (En inglés)

<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

4.1.- Uso de comentarios

Uno de los elementos básicos para documentar código, es el uso de comentarios. Un comentario es una anotación que se realiza en el código, pero que el compilador va a ignorar, sirve para indicar a los desarrolladores de código diferentes aspectos del código que pueden ser útiles. En principio, los comentarios tienen dos propósitos diferentes:

- **Explicar el objetivo de las sentencias:** De forma que el programador o programadora, sepa en todo momento la función de esa sentencia, tanto si lo diseñaron como si son otros los que quieren entenderlo o modificarlo.
- **Explicar qué realiza un método, o clase, no cómo lo realiza:** En este caso, se trata de explicar los valores que va a devolver un método, pero no se trata de explicar cómo se ha diseñado.

En el caso del lenguaje Java, C# y C, los comentarios, se implementan de forma similar. Cuando se trata de explicar la función de una sentencia, se usan los caracteres `//` seguidos del comentario, o con los caracteres `/*` y `*/`, situando el comentario entre ellos: `/* comentario */`

Otro tipo de comentarios que se utilizan en Java, son los que se utilizan para explicar qué hace un código, se denominan comentarios **JavaDoc** y se escriben empezando por `/**` y terminando con `*/`, estos comentarios pueden ocupar varias líneas. Este tipo de comentarios tienen que seguir una estructura prefijada.

Los comentarios son obligatorios con JavaDoc, y se deben incorporar al principio de cada clase, al principio de cada método y al principio de cada variable de clase. No es obligatorio, pero en muchas situaciones es conveniente, poner los comentarios al principio de un fragmento de código que no resulta lo

suficientemente claro, a la largo de bucles, o si hay alguna línea de código que no resulta evidente y pueda llevarnos a confusión.

Hay que tener en cuenta, que si el código es modificado, también se deberán modificar los comentarios.

4.2.- Documentación del código

Las clases que se implementan en una aplicación, deben de incluir comentarios. Al utilizar un entorno de programación para la implementación de la clase, debemos seguir una serie de pautas, muchas de las cuales las realiza el IDE de forma transparente, en el diseño y documentación del código. Cuando se implementa una clase, se deben incluir comentarios. En el lenguaje Java, los criterios de documentación de clases, son los establecidos por Javadoc.

Los comentarios de una clase deben comenzar con `/**` y terminar con `*/` . Entre la información que debe incluir un comentario de clase debe incluirse, al menos las etiquetas **@autor** y **@version**, donde **@autor** identifica el nombre del autor o autora de la clase y **@version**, la identificación de la versión y fecha.

Con el uso de los entornos de desarrollo, las etiquetas se añaden de forma automática, estableciendo el **@autor** y la **@version** de la clase de forma transparente al programador-programadora. También se suele añadir la etiqueta **@see**, que se utiliza para referenciar a otras clases y métodos.

Dentro de la la clase, también se documentan los constructores y los métodos. Al menos se indican las etiquetas:

- **@param**: seguido del nombre, se usa para indicar cada uno de los parámetros que tienen el constructor o método.
- **@return**: si el método no es void, se indica lo que devuelve.
- **@exception**: se indica el nombre de la excepción, especificando cuales pueden lanzarse.
- **@throws**: se indica el nombre de la excepción, especificando las excepciones que pueden lanzarse.

Los campos de de una clase, también pueden incluir comentarios, aunque no existen etiquetas obligatorias en Javadoc.

Un ejemplo de una clase documentada es el siguiente:

```
/**
 * Clase que representa a un profesor.
 * Esta clase hereda de la clase Persona
 * @author Francis
 * @version 1.0
 * @see Persona
 */
public abstract class Profesor extends Persona
{
    // Identificador del profesor
    private String idProfesor;

    /**
     * Constructor de la clase Profesor
     * @param nombre Nombre del profesor
     * @param apellidos Apellidos del profesor
     * @param edad Edad del profesor
     * @param idProfesor Identificador del profesor
     */
    public Profesor (String nombre, String apellidos, int edad, String idProfesor)
    {
        super(nombre, apellidos, edad);
        this.idProfesor = idProfesor;
    }

    /**
     * Modifica el identificador del profesor
     * @param IdProfesor Nuevo identificador para el profesor
     */
    public void setIdProfesor (String IdProfesor)
    {
        this.idProfesor = IdProfesor;
    }

    /**
     * Devuelve el identificador del profesor
     * @return Identificador del profesor
     */
    public String getIdProfesor ()
    {
        return idProfesor;
    }

    /**
     * Calcula el importe de la nómina del profesor. Es un método abstracto
     * @return Importe total de la nómina del profesor
     */
    abstract public float importeNomina();

    @Override
    /**
     * Método toString de la clase Profesor
     */
    public String toString()
    {
        return("Profesor de nombre: " + getNombre() + " " + getApellidos() +
            " con Id de profesor: " + getIdProfesor() );
    }
}
```

Ilustración 21: Clase Profesor

4.3.- Herramientas

Los entornos de programación que implementa Java, como Eclipse o Netbeans, incluyen una herramienta que va a generar páginas HTML de documentación a partir de los comentarios incluidos en el código fuente. La herramienta ya se ha indicado en los puntos anteriores, y es JavaDoc. Para que JavaDoc pueda generar las páginas HTML es necesario seguir una serie de normas de documentación en el código fuente, estas son:

- Los comentarios JavaDoc deben empezar por `/**` y terminar por `*/`.
- Los comentarios pueden ser a nivel de clase, a nivel de variable y a nivel de método.
- La documentación se genera para métodos public y protected.

Se puede usar tag para documentar diferentes aspectos determinados del código, como parámetros. Los tags más habituales son los siguientes:

Tag	Formato	Descripción
Todos	@see	Permite crear una referencia a la documentación de otra clase o método.
Clases	@version	Comentario con datos indicativos del número de versión.
Clases	@author	Nombre del autor.
Clases	@since	Fecha desde la que está presente la clase.
Métodos	@param	Parámetros que recibe el método.
Métodos	@return	Significado del dato devuelto por el método.
Métodos	@throws	Comentario sobre las excepciones que lanza.
Métodos	@deprecated	Indicación de que el método es obsoleto.

Tabla 1: Tags más habituales para JavaDoc

4.4.- Generar JavaDoc en NetBeans

Una vez tengamos todas nuestras clases documentadas podemos crear el JavaDoc del proyecto.

Para ello nos vamos al menú *Run* y seleccionamos *Generate Javadoc*.

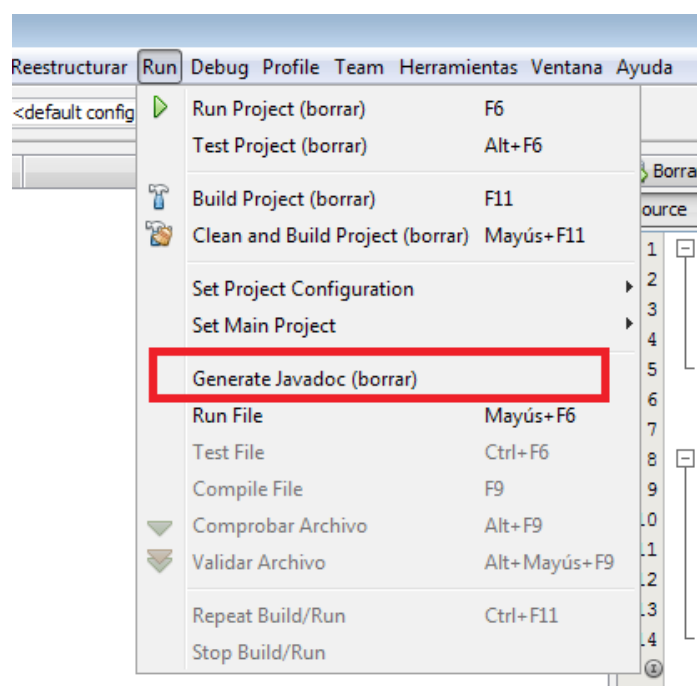
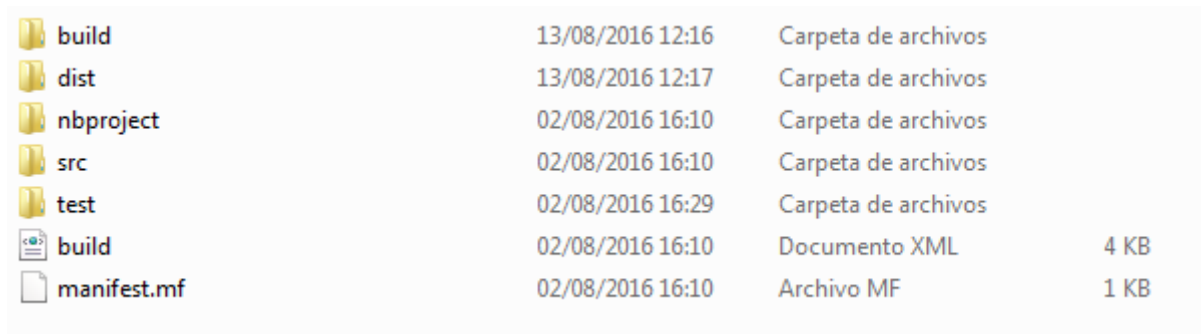


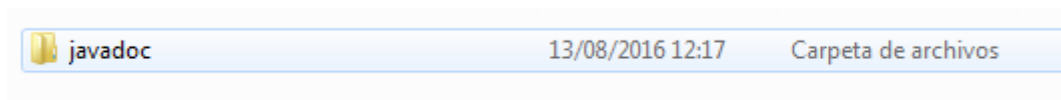
Ilustración 22: Generar JavaDoc

NetBeans creará la documentación Javadoc y la guardará en una carpeta en nuestro proyecto que se llama javadoc, la cual está dentro de la carpeta dist.



build	13/08/2016 12:16	Carpeta de archivos	
dist	13/08/2016 12:17	Carpeta de archivos	
nbproject	02/08/2016 16:10	Carpeta de archivos	
src	02/08/2016 16:10	Carpeta de archivos	
test	02/08/2016 16:29	Carpeta de archivos	
build	02/08/2016 16:10	Documento XML	4 KB
manifest.mf	02/08/2016 16:10	Archivo MF	1 KB

Ilustración 23: Carpeta dist



javadoc	13/08/2016 12:17	Carpeta de archivos	
---------	------------------	---------------------	--

Ilustración 24: Carpeta javadoc

Dentro de la carpeta javadoc tendremos toda la documentación generada de todas las clases en formato html.

Class Profesor

```
java.lang.Object
  borrar.Persona
    borrar.Profesor
```

Direct Known Subclasses:

ProfesorInterino, ProfesorTitular

```
public abstract class Profesor
extends Persona
```

Clase que representa a un profesor. Esta clase hereda de la clase Persona

See Also:

Persona

Constructor Summary

Constructors

Constructor and Description

Profesor(java.lang.String nombre, java.lang.String apellidos, int edad, java.lang.String idProfesor)
Constructor de la clase Profesor

Method Summary

Methods

Modifier and Type	Method and Description
java.lang.String	getIdProfesor() Devuelve el identificador del profesor
abstract float	importeNomina() Calcula el importe de la nómina del profesor.
void	setIdProfesor(java.lang.String idProfesor) Modifica el identificador del profesor
java.lang.String	toString()

Ilustración 25: JavaDoc de la clase Profesor

Resumen de conceptos

Refactorización: es una disciplina técnica, que consiste en realizar pequeñas transformaciones en el código de un programa, para mejorar la estructura sin que cambie el comportamiento ni funcionalidad del mismo.

Control de versiones: Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo. Una versión, revisión o edición de un producto, es el estado en el que se encuentra el mismo en un momento dado de su desarrollo o modificación.

Repositorio: El repositorio es la parte fundamental de un sistema de control de versiones. Almacena toda la información y datos de un proyecto.

Documentación de código: Documentar el código nos sirve para explicar su funcionamiento, punto por punto, de forma que cualquier persona que lea el comentario, puede entender la finalidad del código.

Comentario: Un comentario es una anotación que se realiza en el código, pero que el compilador va a ignorar, sirve para indicar a los desarrolladores de código diferentes aspectos del código que pueden ser útiles.

JavaDoc: Javadoc es una utilidad de Oracle para la generación de documentación de APIs en formato HTML a partir de código fuente Java. Javadoc es el estándar de la industria para documentar clases de Java. La mayoría de los IDEs los generan automáticamente. Javadoc también proporciona una API para crear doclets y taglets, que le permite analizar la estructura de una aplicación Java.

Bibliografía

administrador. (s.f.). *Introducción*. Recuperado el 12 de Agosto de 2016, de <http://conociendogithub.readthedocs.io/en/latest/data/introduccion/>

Cabeza, J. L. (s.f.). *DISEÑO Y REALIZACIÓN DE PRUEBAS*. Recuperado el 2016 de Julio de 25, de <http://www.sitiolibre.com>: <http://www.sitiolibre.com/curso/pdf/ED3.pdf>

Cabeza, J. L. (s.f.). *OPTIMIZACIÓN Y DOCUMENTACIÓN*. Recuperado el 11 de Agosto de 2016, de <http://www.sitiolibre.com>: <http://www.sitiolibre.com/curso/pdf/ED4.pdf>

DIGANMEGIOVANNI. (19 de Enero de 2014). *Integrar Netbeans con Github*. Recuperado el 11 de Agosto de 2016, de <https://lineaporlinea.wordpress.com>: <https://lineaporlinea.wordpress.com/2014/01/19/integrar-netbeans-con-github/>

Wikipedia, C. d. (s.f.). *Control de versiones*. Recuperado el 11 de Agosto de 2016, de Wikipedia: https://es.wikipedia.org/wiki/Control_de_versiones

Wikipedia, C. d. (s.f.). *GitHub*. Recuperado el 12 de Agosto de 2016, de Wikipedia: <https://es.wikipedia.org/wiki/GitHub>

Wikipedia, C. d. (s.f.). *Javadoc*. Recuperado el 13 de Agosto de 2016, de Wikipedia: <https://es.wikipedia.org/wiki/Javadoc>

Wikipedia, C. d. (s.f.). *Refactorización*. Recuperado el 11 de Agosto de 2016, de Wikipaida: <https://es.wikipedia.org/wiki/Refactorizaci%C3%B3n>