Universidad Central de Venezuela

Facultad de Ciencias

Aplicaciones con la Tecnología Internet II

Semestre I-2018

Laboratorio #1

Git, debug, plug-in, & JUnit

Parte I - Teoría

1. Objetivos

- Definir los conceptos asociados con ***Git***.
- Conocer el término ***debug*** y su utilidad.
- Comprender el significado de un ***plug-in***, y cómo incorporarlos en un proyecto.
- Definir ***JUnit*** y sus funcionalidades.

2. Estructura

Este laboratorio está dividido en las siguientes secciones:

Parte I - Teoría

- **Definiciones**
  - Git.
  - Debug.
  - Plug-in.
  - JUnit.

3. Definiciones

Git

Cuando un equipo se encuentra desarrollando un determinado proyecto, surge la necesidad de mantener una sincronización entre los cambios efectuados por cada miembro del equipo, en especial si dos o más miembros realizan modificaciones sobre una misma sección del código.

Por este motivo, distintas organizaciones han desarrollado sistemas denominados Sistemas Manejadores de Versiones (SCM), así como también, herramientas para el control de versiones (VCS).

En este punto, uno de los sistemas más conocidos a nivel mundial que incluye dichas funcionalidades se conoce como ***Git*** [1].

Git es un Sistema de Control de Versiones Distribuido, libre, de código abierto diseñado para manejar cualquier aspecto proveniente de un proyecto, independientemente de su nivel de complejidad; es decir, puede ser aplicado tanto en pequeños como grandes proyectos con rapidez y eficiencia.

¿Qué es un control de versiones, y por qué debería importarte?

Un control de versiones es un sistema que registra los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.

Sistema de Control de Versiones: Centralizado vs Distribuido

Un Sistema de Control de Versiones Centralizado (CVCS por sus siglas en inglés) tiene un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos desde ese servidor central.

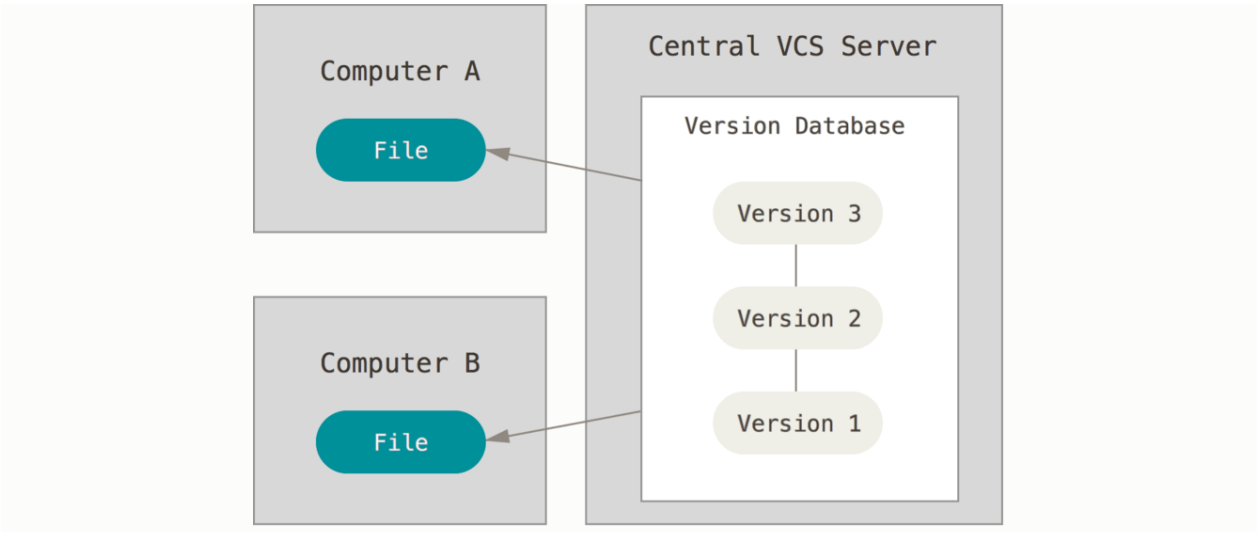

Figura 1. Sistema de Control de Versiones Centralizado

[1] Página oficial de Git: https://git-scm.com/

Para el manejo de versiones de un proyecto, este tipo de sistemas podría ser considerado una buena solución, ya que les permite a los miembros del equipo conocer hasta cierto punto en qué están trabajando el resto de los colaboradores, y no es necesario crear una base de datos local por cada cliente para almacenar la información asociada a la versión del proyecto.

Sin embargo, ese tipo de sistemas presenta una grave desventaja, lo cual se conoce como el ***cuello de botella***, el servidor central. Si existe alguna falla en el servidor central, ningún miembro del equipo podrá acceder a la versión que necesita durante el transcurso del tiempo que se presente la falla. Por otra parte, si existe una violación de seguridad en el disco duro que contiene el servidor central, o dicho disco presenta una corrupción, el riesgo de perder la información asociada a todas las versiones que contiene dicho servidor es alta.

Es por ello que se diseñaron los Sistemas de Control de Versiones Distribuidos (DVCS), donde los clientes no solo descargan la última copia instantánea de los archivos, sino que se replica completamente el repositorio. De este modo, si un servidor deja de funcionar, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor que presentó la falla con la finalidad de restaurarlo.

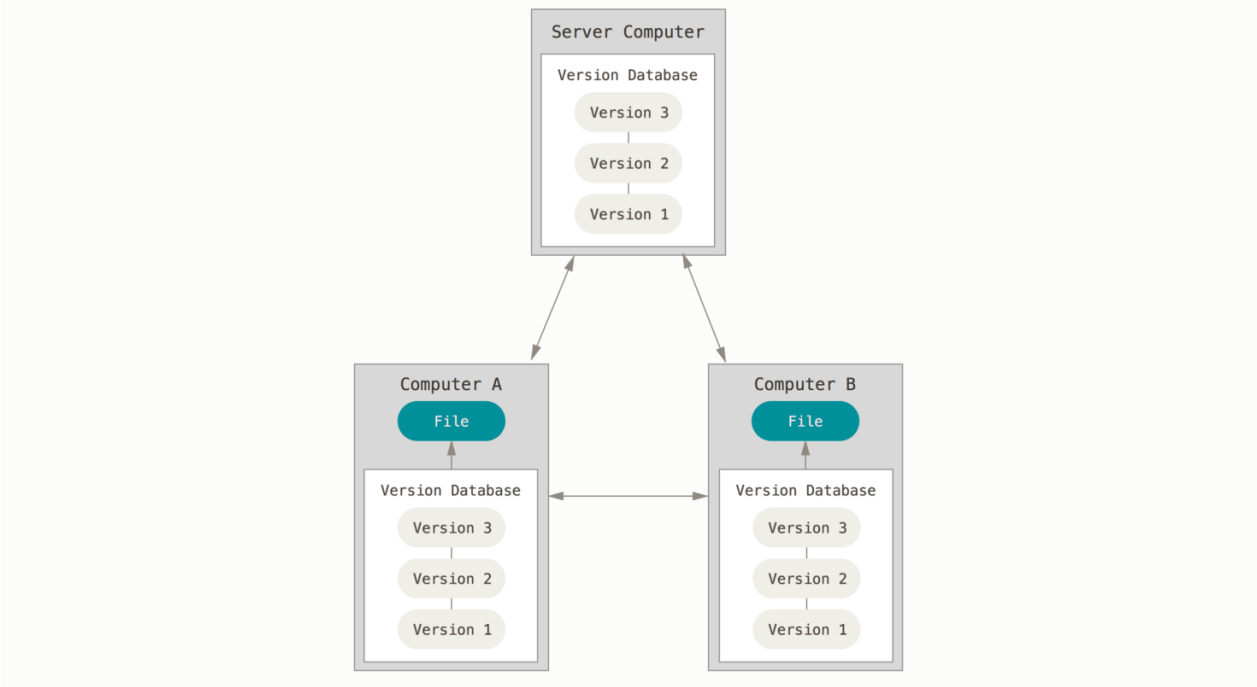

Figura 2. Sistema de Control de Versiones Distribuido

La principal diferencia entre Git y cualquier otro VCS (incluyendo Subversion y similares) es la forma en la que manejan sus datos. Conceptualmente, la mayoría de los otros sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) manejan la información que almacenan como un conjunto de archivos y las modificaciones hechas a cada uno de ellos a través del tiempo.

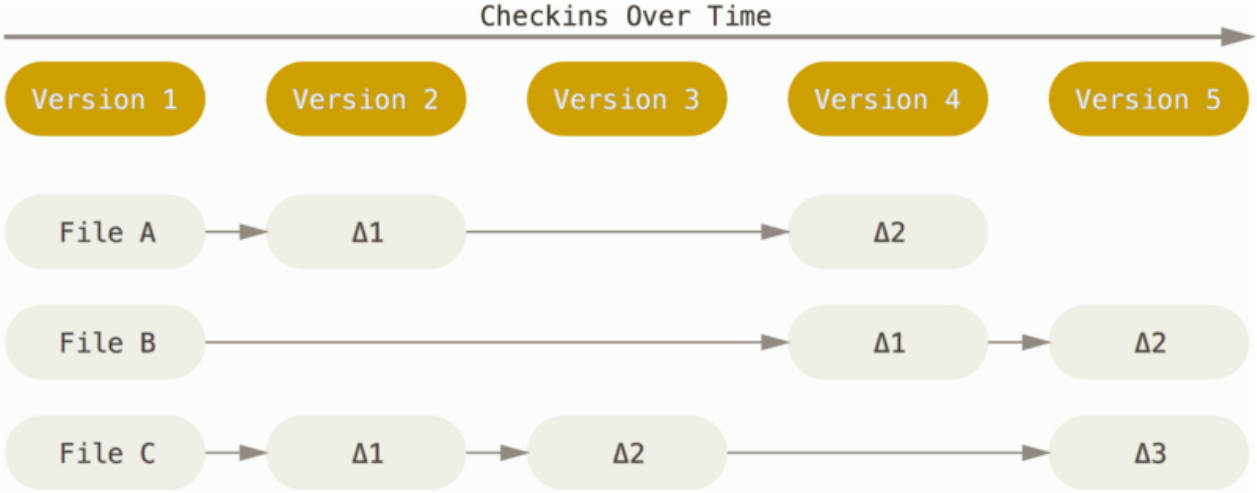

Figura 3. Almacenamiento de datos como cambios en una versión de la base de cada archivo

Git no maneja ni almacena sus datos de esta forma. Git maneja sus datos como un conjunto de copias instantáneas de un sistema de archivos miniatura. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en Git, él básicamente toma una foto del aspecto de todos tus archivos en ese momento, y guarda una referencia a esa copia instantánea. Para ser eficiente, si los archivos no se han modificado, Git no almacena el archivo de nuevo, sino un enlace al archivo anterior idéntico que ya tiene almacenado. Git maneja sus datos como una secuencia de copias instantáneas.

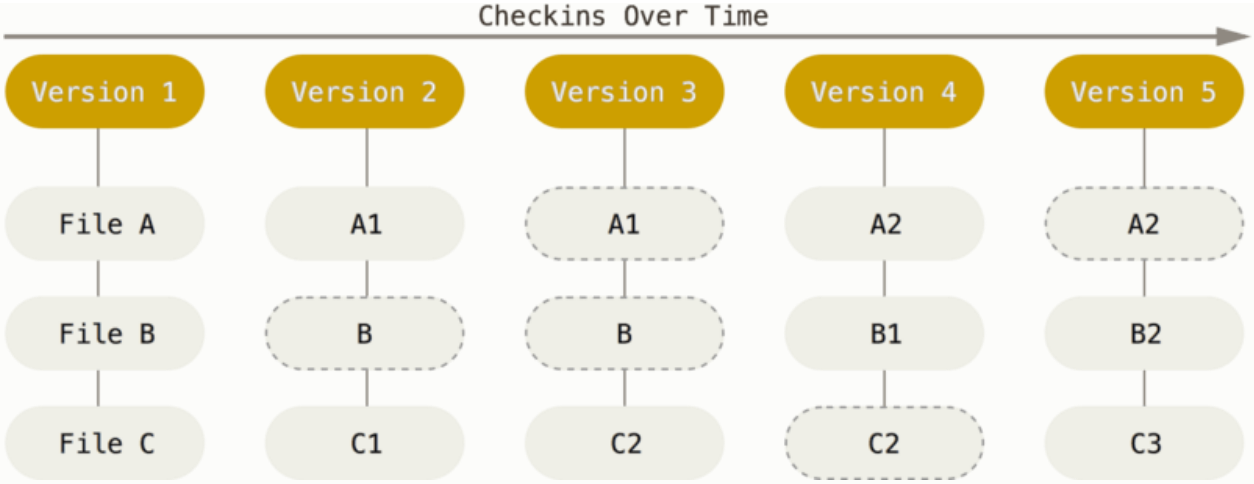

Figura 4. Almacenamiento de datos como instantáneas del proyecto a través del tiempo

Beneficios de Git

- **Casi todas las operaciones son locales:** La mayoría de las operaciones en Git sólo necesitan archivos y recursos locales para funcionar. Por ejemplo, para navegar por la historia del proyecto, Git no necesita conectarse al servidor para obtener la historia y mostrársela - simplemente la lee directamente de su base de datos local. Esto significa que es posible ver la historia del proyecto casi instantáneamente. Si se quieren ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, Git puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga u obtener una versión antigua desde la red y hacerlo de manera local.

- **Git tiene integridad:** Todo en Git es verificado mediante una suma de comprobación (checksum en inglés) antes de ser almacenado, y es identificado a partir de ese momento mediante dicha suma. Esto significa que es imposible cambiar los contenidos de cualquier archivo o directorio sin que Git lo sepa. Esta funcionalidad está integrada en Git al más bajo nivel y es parte integral de su filosofía. No puedes perder información durante su transmisión o sufrir corrupción de archivos sin que Git sea capaz de detectarlo. El mecanismo que usa Git para generar esta suma de comprobación se conoce como hash SHA-1.

Se trata de una cadena de 40 caracteres hexadecimales (0-9 y a-f), y se calcula en base a los contenidos del archivo o estructura del directorio en Git. Un hash SHA-1 se ve de la siguiente forma:

24b9da6552252987aa493b52f8696cd6d3b00373

Verás estos valores hash por todos lados en Git porque son usados con mucha frecuencia. De hecho, Git guarda todo no por nombre de archivo, sino por el valor hash de sus contenidos.

- **Git generalmente solo añade información:** Cuando realizas acciones en Git, casi todas ellas solo añaden información a la base de datos de Git. Es muy difícil conseguir que el sistema haga algo que no se pueda enmendar, o que de algún modo borre información. Como en cualquier VCS, puedes perder o estropear cambios que no has confirmado todavía. Pero después de confirmar una copia instantánea en Git es muy difícil de perderla, especialmente si envías tu base de datos a otro repositorio con regularidad.

Estados de Git

Git tiene tres estados principales en los que se pueden encontrar tus archivos: ***confirmado*** (***committed***), ***modificado*** (***modified***), y ***preparado*** (***staged***). ***Confirmado*** significa que los datos están almacenados de manera segura en tu base de datos local. ***Modificado*** significa que has modificado el archivo, pero todavía no lo has confirmado a tu base de datos. ***Preparado*** significa que has marcado un archivo modificado en su versión actual para que vaya en tu próxima confirmación.

Esto nos lleva a las tres secciones principales de un proyecto de Git: ***El directorio de Git*** (***Git directory***), ***el directorio de trabajo*** (***working directory***), y ***el área de preparación*** (***staging area***).

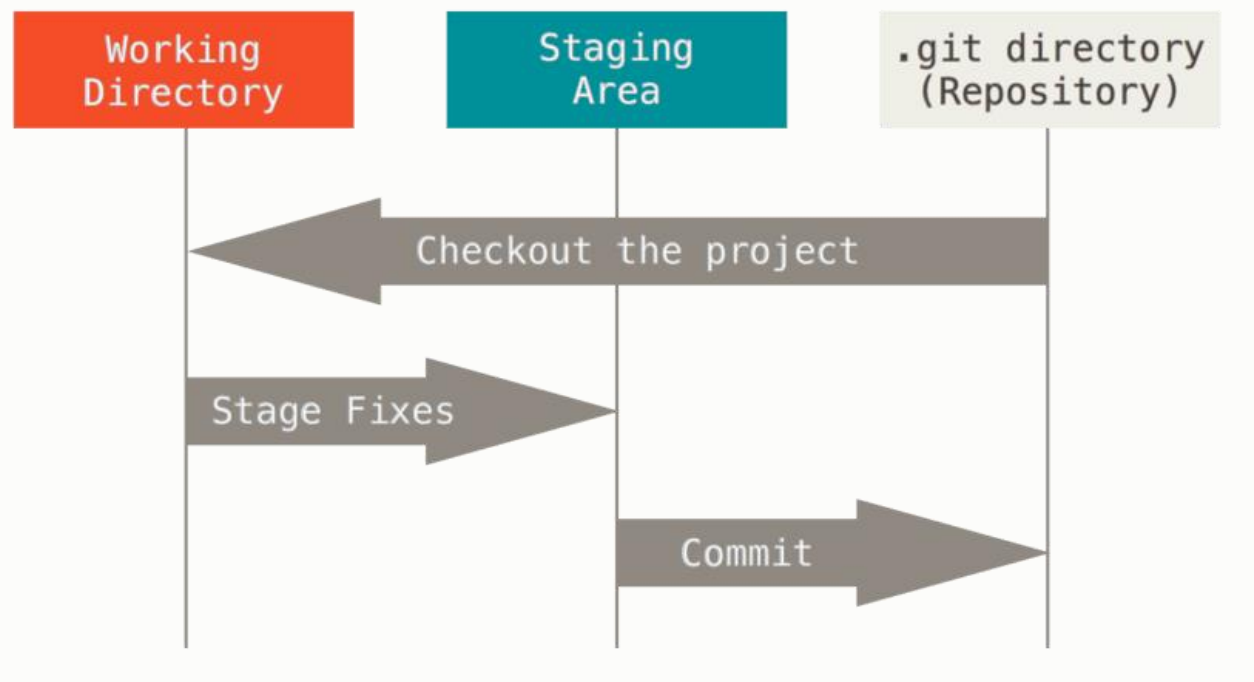

Figura 5. Directorio de trabajo, área de almacenamiento, y el directorio Git

El directorio de Git es donde se almacenan los metadatos y la base de datos de objetos para tu proyecto. Es la parte más importante de Git, y es lo que se copia cuando clonas un repositorio desde otra computadora.

El directorio de trabajo es una copia de una versión del proyecto. Estos archivos se sacan de la base de datos comprimida en el directorio de Git, y se colocan en disco para que los puedas usar o modificar.

El área de preparación es un archivo, generalmente contenido en tu directorio de Git, que almacena información acerca de lo que va a ir en tu próxima confirmación. A veces se le denomina ***índice*** (***index***), pero se está convirtiendo en estándar el referirse a ella como el área de preparación.

Flujo de trabajo básico en Git

1. Modificas una serie de archivos en tu directorio de trabajo.
2. Preparas los archivos, añadiéndolos a tu área de preparación.
3. Confirmas los cambios, lo que toma los archivos tal y como están en el área de preparación y almacena esa copia instantánea de manera permanente en tu directorio de Git.

Si una versión concreta de un archivo está en el directorio de Git, se considera confirmada (committed). Si ha sufrido cambios desde que se obtuvo del repositorio, pero ha sido añadida al área de preparación, está preparada (staged). Y si ha sufrido cambios desde que se obtuvo del repositorio, pero no se ha preparado, está modificada (modified).

Debug

Debugging es el proceso de limpieza que se realiza sobre un programa, cuya finalidad es corregir los posibles errores que se encuentren. Se suele utilizar el ***debugger*** o alguna herramienta de depuración especialmente cuando el programa logra compilar, pero su respuesta no es la esperada. En efecto, a los errores de ejecución de los programas se les conoce como ***bugs*** (insectos).

La invención del término se atribuye generalmente a la ingeniera Grace Hopper que en 1946 estaba en el laboratorio de computación de la universidad de Harvard trabajando en los ordenadores con nombre Mark II y Mark III. Los operadores descubrieron que la causa de un error detectado en el Mark II era una polilla que se había quedado atrapada entre los contactos de un relé (por aquel entonces el elemento básico de un ordenador) que a su vez era parte de la lógica interna del ordenador. Estos operadores estaban familiarizados con el término bug e incluso pegaron el insecto en su libro de notas con la anotación ***"First actual case of bug being found"*** (primer caso en el que realmente se encuentra un bug).

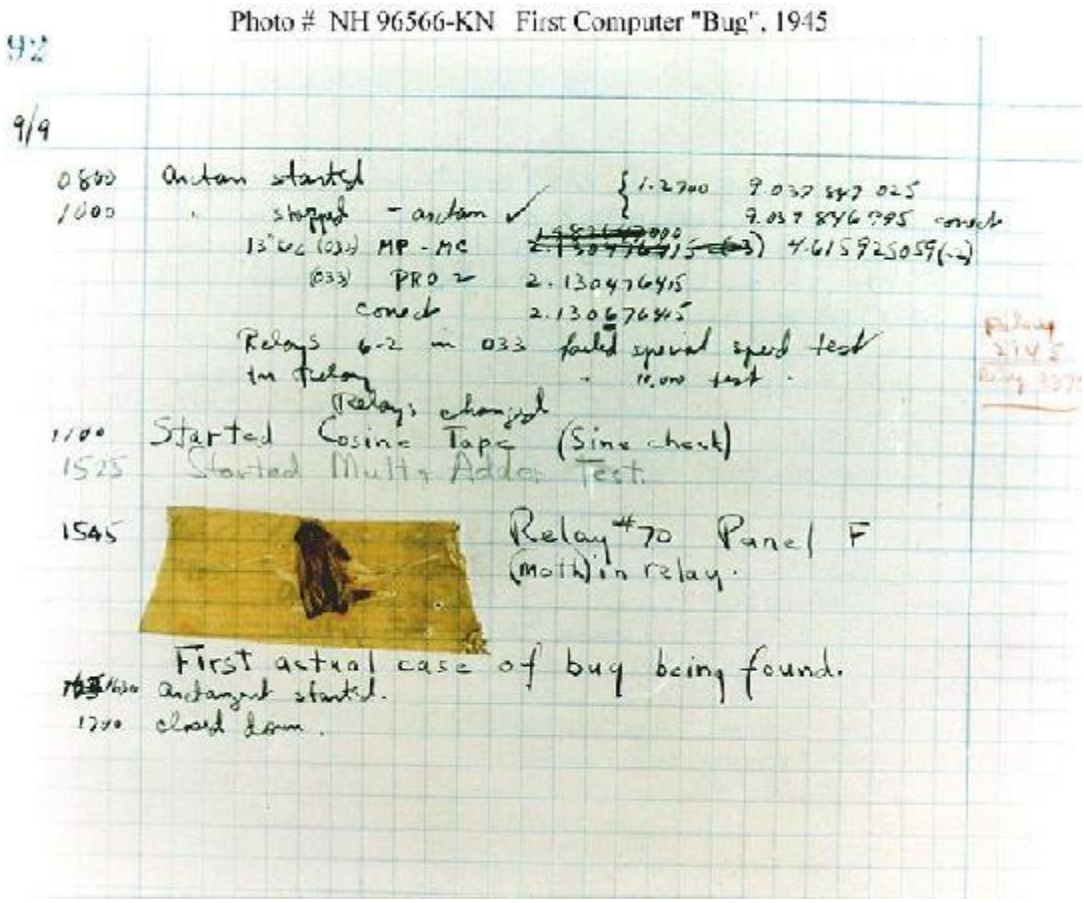

Figura 6. Primer caso en el que realmente se encuentra un bug

Como se indicó anteriormente, un ***debugger*** es un ejecutable que permite la ejecución controlada de un segundo ejecutable. Se comporta como un envoltorio dentro del cual se desarrolla una ejecución normal de un programa, pero a la vez permite realizar una serie de operaciones específicas para visualizar el entorno de ejecución en cualquier instante.

Funcionalidades del debugger

1. Permite ejecutar un programa línea a línea.
2. Detener la ejecución temporalmente en una línea de código concreta.
3. Detener temporalmente la ejecución bajo determinadas condiciones.

4. Visualizar el contenido de las variables en un determinado momento de la ejecución.
5. Cambiar el valor del entorno de ejecución para poder ver el efecto de una corrección en el programa.

Como se indicó anteriormente, el debugger es un ejecutable que permite la ejecución controlada de un segundo ejecutable. Se comporta como un envoltorio dentro del cual se desarrolla una ejecución normal de un programa, pero a la vez permite realizar una serie de operaciones específicas para visualizar el entorno de ejecución en cualquier instante.

Uno de los ***debuggers*** más utilizados en entornos Linux es ***gdb*** (Debugger de GNU). Sin embargo, también existen muchas otras alternativas, como ***pudb*** de Python, el debugger que tiene por defecto IDEs como NetBeans, Eclipse, Visual Studio Code, entre otros.

En la sección práctica del presente laboratorio se indicarán varias alternativas para utilizar los ***debuggers***, y cómo sacarles el mayor provecho durante el desarrollo de un proyecto.

Plug-in

Un ***plug-in*** es un programa que incrementa o aumenta las funcionalidades de un programa principal. Por lo general es producido por una compañía diferente a la que produjo el primer programa.

En la sección práctica del presente laboratorio se indicarán varias alternativas para incorporar los ***plug-ins***, y cómo utilizarlos.

JUnit

JUnit 5 [2] se compone de varios módulos diferentes, los cuales provienen de tres sub-proyectos.

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

La ***plataforma JUnit*** sirve como base para el lanzamiento de marcos de prueba en la JVM. También define el API ***TestEngine*** para desarrollar un marco de prueba que se ejecuta en la plataforma. Además, la plataforma proporciona un ***ConsoleLauncher*** para iniciar la plataforma desde la línea de comandos y crear complementos para ***Gradle*** y ***Maven***, así como un ***Runner*** basado en ***JUnit 4*** para ejecutar cualquier ***TestEngine*** en la plataforma.

JUnit Jupiter es la combinación del nuevo modelo de programación y el modelo de extensión para escribir pruebas y extensiones en ***JUnit 5***. El sub-proyecto de Júpiter proporciona un ***TestEngine*** de ejecución de pruebas basadas en la plataforma Júpiter.

JUnit Vintage proporciona un ***TestEngine*** de prueba para ejecutar ***JUnit 3*** y ***JUnit 4*** en la plataforma.

Compatibilidad

JUnit 5 requiere Java 8 (o superior) en tiempo de ejecución. Sin embargo, aún puede probar el código que se ha compilado con versiones anteriores del JDK.

En la sección práctica del presente laboratorio se indicará correctamente cómo utilizar ***JUnit 5***.

[2] Página oficial de JUnit: https://junit.org/junit5/