Universidad de los Andes

Ingeniería de Sistemas y Computación
ISIS1206 – Estructura de Datos
Taller 1

Taller 1: Git y Consola

Objetivos

- Entender la importancia de los repositorios de código y aprender el manejo de los mismos (concretamente GIT)
- Aprender a usar y diseñar software usando únicamente la consola de Java

Lectura Previa

Reglas de talleres y proyectos

Antes de comenzar con este taller tenga en cuenta las reglas de entrega de los talleres y proyectos en el curso (es **indispensable seguir al pie de la letra** lo dispuesto en estos links):

Entrega de talleres:
https://cursos.virtual.uniandes.edu.co/isis1206/guias-enviotalleres/

Entrega de proyectos:
https://cursos.virtual.uniandes.edu.co/isis1206/envio-proyectos/

Introducción a GIT.

Suponga que una revista reconocida a nivel mundial, le ha solicitado un artículo de interés público acerca de la Guerra Civil en Siria. Sin embargo, debido a las características propias del conflicto y la complejidad del mismo, la mesa editorial, ha decidido contactar a otros autores expertos en el tema, desconocidos para Usted, con el fin de elaborar un documento común. A continuación, la mesa editorial solicita que el documento siempre permanezca en una carpeta en el servidor central de la revista en cuestión. i.e., Los autores solo pueden subir la versión presente en su disco local y descargar la revisión presente en el servidor en el tiempo *t* en el cual realizan la operación. Además, es necesario notar que, el servidor no informa a los autores de los cambios realizados al archivo por parte de los otros autores.

Si bien, la configuración propuesta puede lucir extraña, es posible evidenciar que este procedimiento de intercambio de información es similar al proceso de intercambio de archivos a través de correo

electrónico o almacenamiento remoto en Internet, en la medida que solo es posible contar con una versión del archivo en cada instante de tiempo. Una solución a este inconveniente, consiste en guardar copias previas de cada uno de los archivos involucrados en el intercambio, sin embargo, este proceso, en ocasiones incurre en un aumento de la entropía (desorden) en el sistema de archivos de cada autor, y como resultado, el proceso de recuperación de archivos se torna complejo y difícil, en la medida que no es posible establecer el contenido de cada uno de los archivos, así como la diferencia entre los mismos.

Ahora bien, considere que hace una semana, el autor Nikolai Yezhov fue incorporado al equipo de elaboración del artículo en cuestión. Éste ha modificado algunos párrafos, eliminado un número significativo de líneas del documento. No obstante, durante el tiempo que realizó las modificaciones, no descargó de forma constante la versión actual del documento desde el servidor de almacenamiento de la revista. Tras finalizar el proceso de modificación, reemplazó el archivo presente en el servidor, eliminando de forma indiscriminada, las actualizaciones realizadas por parte de otros autores durante la última semana, y bajo el supuesto de que ningún autor cuenta con archivos de respaldo y recuperación, suprimiendo la revisión de cada uno de los autores en cuanto éstos, recuperen la nueva versión existente del archivo.

Finalmente, es posible evidenciar que la mezcla de cada una de las versiones de los archivos debe realizarse de forma manual, y debido a que los cambios realizados por cada uno de los autores, no es resaltado o indicado por parte del sistema de envío, es posible incurrir en un mayor número de errores y defectos en el documento final.

Debido a que estos inconvenientes pueden ocurrir también durante el proceso de desarrollo de componentes de Sofware a gran escala (una tarea normalmente realizada a partir del trabajo conjunto de diversos desarrolladores), se pretende mitigar sus efectos, a partir del uso de programas y rutinas establecidas para realizar el **Control de revisiones** (VC), o Software de control de revisiones (VCS) [1] . Este conjunto de herramientas, permiten resolver los inconvenientes previamente mencionados, almacenando la secuencia de cambios y estado de los archivos de forma consistente y continua, permitiendo que los equipos de desarrollo, puedan colaborar y trabajar en completa armonía. Entre algunos de los sistemas de control de revisiones, se encuentran: *Apache Subversion* (SVN)[2], *Mercurial* (Hg), *GNU Bazaar* (BZR)[3], *GIT* (dispuesto y preferido durante el proceso de desarrollo a lo largo del presente curso), entre otros.

[1] Siglas en Inglés para Version Control Software
[2] https://www.mercurial-scm.org/
[3] http://bazaar.canonical.com/en/

Lo que usted debe hacer

Parte 1 – GIT: Funcionamiento y operaciones básicas (trabajo en casa)

Git es un sistema de control de revisiones distribuido, bajo el cual, cada autor cuenta con una copia (además del historial asociado) local de los archivos almacenados en la carpeta definida (repositorio), esto permite que cada desarrollador pueda realizar cambios específicos, sin la necesidad de conocer los cambios realizados por parte de otros usuarios. Ahora bien, con el fin de sincronizar las copias locales de cada uno de los miembros asociados a un proyecto, existen diversos servicios de almacenamiento remoto con el fin de salvaguardar una copia ''central'' del repositorio de desarrollo, el cual, contiene la conjunción de cada uno de los aportes y contribuciones de los usuarios interesados o adscritos al proyecto. Actualmente, los servicios de alojamiento más usados a nivel mundial corresponden a: *Github*[4], *Gitlab*[5] y *Atlassian Bitbucket*[6]. Éste último, corresponde al servicio de alojamiento dispuesto durante el desarrollo del presente curso.

Con el fin de familiarizar al lector con el funcionamiento de GIT y la sincronización de repositorios en el servicio de alojamiento de repositorios, a continuación, se presenta un breve tutorial.

Paso 0: Creación de cuenta personal Bitbucket

Antes de comenzar con este taller, es necesario que usted cree una cuenta de Bitbucket usando su **correo uniandes**. Si usted ya tiene una cuenta Bitbucket pero con otro correo electrónico igualmente **debe crear una cuenta nueva con el correo uniandes**. Lo anterior es indispensable, puesto que los repositorios personales y grupales de cada estudiante son creados y administrados por los profesores del curso y éstos **brindan permisos administrativos a cada alumno por medio de su correo uniandes**. Para crear una nueva cuenta en Bitbucket diríjase al siguiente link y siga las instrucciones:

https://bitbucket.org/account/signup/

Nota: Recuerde que los proyectos y talleres se entregan por medio de los repositorios creados por los profesores en Bitbucket

[4] https://github.com
[5] https://about.gitlab.com
[6] https://bitbucket.org

Paso 1: Acceder a Bitbucket y obtener una copia del repositorio de talleres

En primer lugar, es necesario acceder a Bitbucket y clonar el repositorio oficial del curso. Para este fin, acceda a su cuenta asignada[7] por parte del equipo del curso a través de https://bitbucket.org, posteriormente, diríjase a la dirección https://bitbucket.org/talleres/talleres-templates-3. A continuación, proceda a realizar un **Fork**[8] del repositorio, a través de la opción presentada en el menú lateral izquierdo, como es posible apreciar en la **Figura 1**.

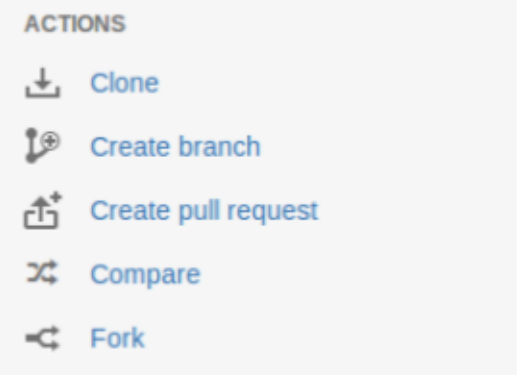

Figura 1

Paso 2: Clonar una copia del nuevo repositorio en el equipo local

Con el fin de realizar este paso, es necesario descargar e instalar un cliente de acceso Git, el cual, puede ser recuperado desde la dirección principal del cliente oficial, http://git-scm.com/ . También puede utilizar clientes gráficos para mayor facilidad, en el caso de Atlassian el cliente recomendado es **SourceTree** (https://www.sourcetreeapp.com/).

A continuación, diríjase al nuevo repositorio asignado a su cuenta, éste debe poder accederse desde la página principal de Bitbucket y proceda a copiar la dirección que se encuentra en la esquina superior derecha de la ventana actual, la dirección debería ser similar a la dirección presentada a continuación: **https://<Nombre_de_usuario>@bitbucket.org/talleres/talleres-templates-3.git**. Tenga en cuenta que de aquí en adelante vamos a usar la **terminal de comandos** de su sistema operativo.

Posteriormente, acceda al cliente de git descargado previamente[9]. Ahora, con el fin de clonar el repositorio a su equipo local, es necesario ejecutar el siguiente comando:

---

[7] Si aún no cuenta con un nombre de usuario asignado por parte del equipo del curso, puede crear una cuenta de uso temporal, con el fin de realizar el presente tutorial

[8] Copia del repositorio principal, en un repositorio asignado a la cuenta del usuario

[9] En Windows debería aparecer bajo el nombre ''Git Bash''. En sistemas UNIX y Linux es necesario ejecutar el comando **git** en una instancia del terminal virtual

```
git clone <url> #Introducir la dirección previamente recuperada en Bitbucket
```

A continuación, la utilidad procederá a solicitar las credenciales de acceso a sistema (*i.e.,* nombre de usuario y contraseña de la cuenta en Bitbucket)[10], tras realizar la descarga de los archivos presentes en el repositorio, la utilidad debería informar que la operación fue realizada de forma exitosa.

Paso 3: Evaluación del estado del repositorio

Tras completar el proceso de copia del repositorio remoto, es necesario acceder a la carpeta que almacena los archivos locales, para este fin, ejecute el comando **cd talleres-templates-3**. A continuación, es necesario conocer el estado actual del repositorio, conforme a estas necesidades, el comando **git status** comunica al desarrollador, los cambios realizados en el repositorio local, *i.e.,* sin actualización en el repositorio remoto. Debido a que ningún cambio ha sido realizado en el repositorio hasta el momento, la utilidad debería reportar el siguiente mensaje:

```
En la rama master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

Con el fin de efectuar y visualizar cambios en el repositorio, es necesario dirigirse a la carpeta que contiene el **taller1** y a continuación, la carpeta **data**. Al interior de esta carpeta, es posible encontrar un archivo comprimido llamado **data.zip** proceda a la descompresión del mismo, tras esta acción, elimine el archivo comprimido mencionado previamente.

A continuación, resulta necesario verificar el estado del repositorio, con el fin de corroborar los cambios realizados previamente, tras ejecutar el comando **git status**, el resultado debería ser similar al presentado a continuación:

[10] Si no desea introducir sus credenciales de forma permanente, es posible realizar la asociación entre el equipo local y la cuenta en Bitbucket, a partir del uso de una llave SSH, para mayor información, ver: https://confluence.atlassian.com/bitbucket/set-up-ssh-for-git-728138079.html

```
\$ git status
En la rama master
Your branch is up-to-date with 'origin/master'.
Cambios no preparados para el commit:
  (use «git add/rm <archivo>...» para actualizar lo que se ejecutará)
  (use «git checkout -- <archivo>...« para descartar cambios en le directorio de
 ↪  trabajo)

        #deleted : data/data.zip

Archivos sin seguimiento:
  (use «git add <archivo>...» para incluir lo que se ha de ejecutar)

        #data/plane.c

no hay cambios agregados al commit (use «git add» o «git commit -a»)
```

Como es posible evidenciar, el sistema de revisión dispuesto por git, ha detectado nuevos cambios en el sistema de archivos, en específico, la eliminación del contenedor comprimido y la introducción de un nuevo archivo en la carpeta **data**. Si bien, el sistema de archivos ha experimentado un cambio local, los cambios realizados no han sido almacenados aún en el repositorio local/global. Esto implica que ante un posible error o pérdida de información posterior, no es posible recuperar el estado del repositorio y retroceder hasta la versión actual (sólo es posible retroceder hasta el estado inicial del repositorio).

Con el fin de registrar los cambios locales en el repositorio global, y posteriormente, en el repositorio global, es necesario realizar un **commit**, es decir, una actualización efectiva en el repositorio local. A continuación, se procede a realizar una actualización del repositorio remoto (alojado en Bitbucket), acción conocida también como **push**.

Si bien, estas instrucciones permiten que el estado del sistema de archivos del repositorio permanezca consistente durante cada etapa de modificación y alteración de archivos a nivel local, en ocasiones, otros usuarios pueden realizar modificaciones al repositorio remoto, mientras el usuario actual realiza modificaciones al repositorio local. Con el objetivo de actualizar la versión local del repositorio, es necesario realizar una acción de **pull**. A continuación, es posible apreciar el uso de las instrucciones previamente descritas:

```
\$ git pull
Already up-to-date. #El repositorio remoto no cuenta con nuevas actualizaciones

\$ git commit -m "<Comentario relevante y conciso con respecto a la
↪  actualización>" #Se realiza una actualización al repositorio local

\$ git push #El repositorio remoto es actualizado
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 1.62 KiB   0 bytes/s, done.
Total 4 (delta 1), reused 0 (delta 0)
```

Paso 4: Uso de ramas (branches) y resolución de confictos

En ocasiones, se desean realizar modificaciones al sistema de archivos *e.g.,* Añadir, modificar o eliminar uno o más archivos, sin embargo, no se desea alterar el estado actual del repositorio remoto. Por ejemplo, durante el desarrollo de un componente de Software, es común mantener un repositorio que contenga la revisión **estable** del programa, no obstante, con el fin de desarrollar la siguiente versión del programa, es necesario declarar un sistema de revisión paralelo con el fin de no introducir posibles errores y problemas de ejecución, pruebas y otros a la versión de desarrollo principal. Este sistema de revisión paralelo debe poder ser integrado de forma eventual a la versión estable del programa en cuestión, y redistribuido bajo una actualización.

Todo sistema de revisión paralelo a la versión principal de un repositorio, se denomina ***branch*** (rama), cada branch puede contener archivos de forma independiente (*i.e.,* cada branch es independiente entre sí) y se comporta como un repositorio autónomo con respecto al repositorio principal, esto implica que todo branch[11] cuenta con dos versiones, una versión local y una versión remota (como se ha descrito a lo largo del presente documento), cada rama permite realizar las operaciones hasta el momento descritas, junto a una operación importante, conocida como **pull request**, la cual permite mezclar (**merge**) dos ramas, sí y solo sí estas no presentan conflicto entre sí. Se define un conflicto de revisión como la presencia de archivos cuyo contenido difiere de forma significativa entre dos commits realizados en una rama.

Creación y navegación entre ramas

Con el objetivo de introducir la funcionalidad y utilidad del uso de ramas como una herramienta importante al momento de realizar actualizaciones en el repositorio, además de introducir el

[11] La versión principal de desarrollo del repositorio, se encuentra almacenada (De forma común) en la rama principal del repositorio, también conocida como **master**

concepto de conflicto, además del procedimiento de solución de los mismos, resulta necesario, en primer lugar, crear dos ramas en el repositorio actual. Para este fin, el comando **git checkout <rama>** permite cambiar el estado actual del repositorio local, visualizando los archivos de una rama específica. En el presente caso, se procederá a la ejecución del comando **git checkout -b <nombre de la rama>**, el cual crea la rama solicitada, si ésta no existe aún. A continuación, es posible evidenciar un ejemplo de salida de ejecución. **Nota:** Si se desea acceder a una rama específica tras clonar un repositorio, es necesario ejecutar el comando **git fetch**, con el fin de descargar las ramas remotas existentes. Tras realizar esta operación, es posible ejecutar el comando **checkout <nombre de la rama>** sin inconveniente alguno.

Nota: Si no conoce en cuál rama se encuentra actualmente, puede ejecutar el comando **git branch**, el cual indica y resalta la rama actual de trabajo.

```
\$ git checkout -b <nombre de la rama 1>
Switched to a new branch <nombre de la rama 1> #La nueva rama local ha sido
↪  creada, el contenido es equivalente a aquel de la rama master

\$ git branch
  master
* "<nombre de la rama 1>" #El repositorio actualmente se encuentra en la rama
↪  definida previamente
```

A continuación, proceda a cambiar el estado del repositorio a la rama **master** y cree una rama adicional, finalmente, regrese a la rama **<nombre de la rama 1>**, con el fin de continuar con el siguiente paso del presente tutorial.

Resolución de conflictos entre versiones

Tras definir e inicializar dos ramas nuevas en el repositorio y dirigirse a la primera rama definida, es necesario abrir el archivo **plane.c**[12](éste debería lucir como un avión). A continuación proceda a añadir texto en líneas escogidas de forma aleatoria, es necesario recordar el número de algunas de estas líneas, con el fin de obtener de generar un conflicto, como es posible evidenciar posteriormente.

Proceda a subir los cambios efectuados a la rama actual (descritos durante el paso 3), y acceda a la interfaz principal de Bitbucket, a continuación, acceda a la opción **Pull Requests**. Como es posible apreciar en las Figuras 2, 3 y 4 presentadas a continuación, el menú dispuesto para realizar Pull Requests presenta tres grandes opciones, la primera corresponde al panel de selección de ramas y repositorios a mezclar. A continuación es posible evidenciar el panel de descripción formal de

[12] Simulador de vuelo ofuscado escrito en C, entrada ganadora del Concurso Internacional de Código Ofuscado en C, Edición 1998, ver: http://ioccc.org/

cambios, además del botón de solicitud. Finalmente, es posible apreciar el panel de descripción de cambios, el cual, compila los commits involucrados en la mezcla, así como los cambios realizados a cada uno de los archivos involucrados en el proceso.

Figure 2

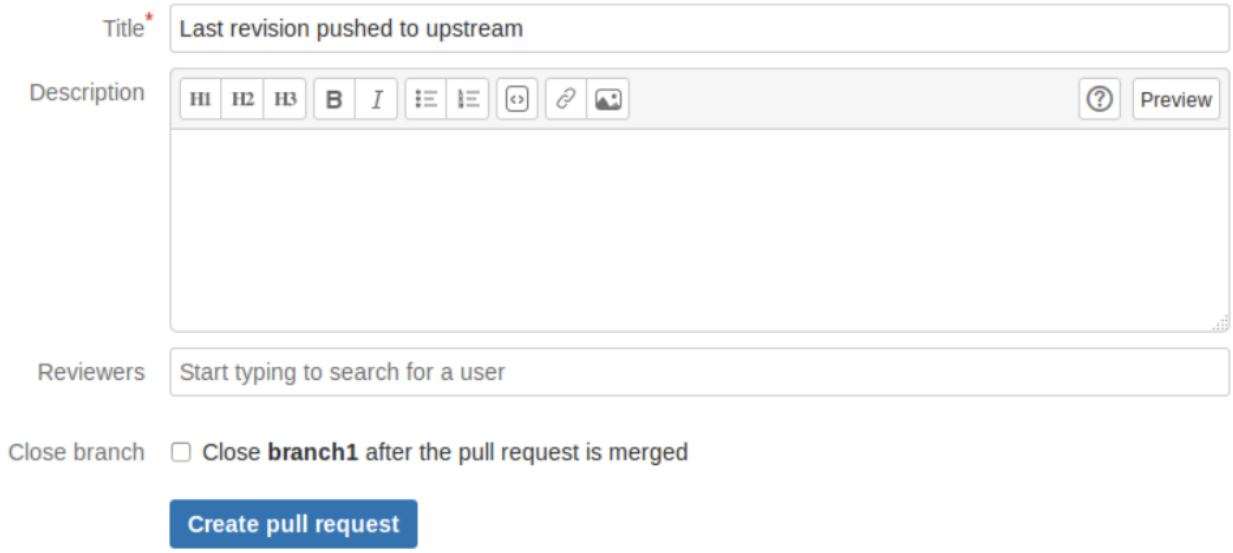

Figure 3

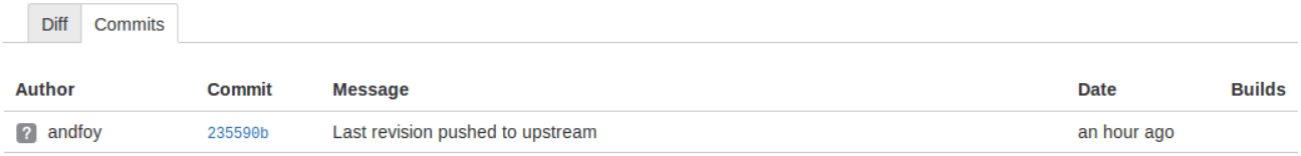

Figure 4

Tras completar el proceso de familiarización y reconocimiento de las opciones disponibles en el menú actual, proceda a realizar una solicitud (pull request), entre la rama **<nombre de la rama 1>** y la rama **<master>,** respectivamente. Si la solicitud no presenta conflicto alguno, en la esquina superior izquierda deberá aparecer un recuadro cuyo texto corresponde a **open**. Puede finalizar el proceso de mezcla, tras seleccionar la opción Merge, ubicada en la esquina superior derecha.

A continuación, diríjase a la rama principal en el repositorio local y posteriormente, recupere los cambios realizados en el repositorio remoto **pull**. Subsecuentemente, cambie el estado del repositorio a la rama **<nombre de la rama 2>,** nuevamente, acceda al archivo modificado previamente. Es posible apreciar que éste no cuenta con las modificaciones previamente realizado en la rama **<nombre de la rama 1>** y la rama **master**, respectivamente. Con el propósito de generar un conflicto entre revisiones, proceda a eliminar algunas de las líneas que fueron sujetas a modificación previamente, tras finalizar este proceso, guarde sus cambios en el repositorio local, así como en el repositorio remoto. Finalmente, proceda a crear una nueva solicitud de **Pull Request**. Sin embargo, al momento de concluir el proceso de mezcla, es posible apreciar que este no es posible, en la medida que existe un conflicto de revisiones en el archivo **plane.c.**

El conflicto ocurre en la medida que la revisión presente en la rama **<nombre de la rama 2>,** elimina contenido que fue previamente extendido en una nueva revisión realizada a la rama **master**, además debido a que la rama **<nombre de la rama 2>,** se encontraba rezagada con respecto a los cambios realizados previamente, ahora bien, debido a que el sistema de revisiones automatizadas no puede realizar el proceso de mezcla de forma automática, es necesario realizar dicho proceso de forma manual[13], Kompare (http://www.caffeinated.me.uk/kompare/), entre otros, como se explica a continuación.

Con el objetivo de resolver un conflicto de revisión de forma manual, existen dos vías alternativas: En primer lugar, es posible desplazar el estado actual del repositorio a la rama en la cual se desea hacer la mezcla y ejecutar el comando merge. En segundo lugar, es posible intentar descargar la versión remota de la rama en la cual se desea realizar la mezcla, en la rama que se encuentra en estado de conflicto. Sin importar el camino escogido, el procedimiento subsecuente es equivalente[14]. A continuación se presentan los dos comandos importantes con el fin de iniciar el proceso de resolución de conflictos.

[13] Existen diversas herramientas que permiten realizar el proceso de forma intuitiva y sencilla, por ejemplo Meld http://meldmerge.org/

[14] Si bien, ambos procedimientos son equivalentes, el primer método presentado considera el caso en el cual se desea realizar una mezcla **nombre de la rama 2> -> master** , mientras que el segundo procedimiento, considera una mezcla **master -> <nombre de la rama 2>**. Sin embargo, el primer método resulta ser el recomendado

```
#Procedimiento \# 1
\$ git checkout master #La rama sobre la cual se desean efectuar los cambios, en
↪  este caso, master
\$ git merge <nombre de la rama 2> #Se procede a mezclar la rama involucrada en
↪  el conflicto de revisiones
```

```
#Procedimiento \# 2
\$ git checkout <nombre de la rama en conflicto>
\$ git pull origin master #Se descarga la última versión de los archivos
↪  presentes en la versión remota de la rama sobre la cual se desea realizar la
↪  mezcla
```

Tras invocar alguna de las dos instrucciones presentadas previamente, la salida del programa en consola debería ser similar a esta:

```
Automezclado data/plane.c
CONFLICTO(contenido): conflicto de fusión en data/plane.c
Automatic merge failed  fix conflicts and then commit the result.
```

A continuación, es necesario abrir el archivo **plane.c**, como es posible apreciar, el sistema de revisiones ha dispuesto dos grupos de líneas de código, delimitadas por texto similar al presentado a continuación:

```
/** plane.c **/
.
.
.
<<<<<<< HEAD //Corresponde al contenido del archivo existente en la rama actual

//Aquí se encuentran algunas líneas de código...

======= //Corresponde al contenido del archivo existente en la rama sujeta al
↪  proceso de mezcla

//Aquí continua el documento...

>>>>>>> adee62c6035110b5ed5eb64109ae7fa9af71fb7f
```

Si bien, las marcas pueden resultar confusas, es necesario comprender que éstas contienen las líneas del archivo presentes en ambas ramas, es necesario notar que, entre la marca **HEAD** y el delimitador **=======,** comprende las líneas existentes en el archivo en la rama actual, mientras que el contenido encapsulado entre el delimitador previamente mencionado, y la marca final **adee62c6...,** corresponde a las líneas contenidas en el archivo, bajo la revisión de la rama sobre la cual se desea realizar la mezcla. Si se eliminan los delimitadores y las líneas no deseadas, es posible completar el proceso de mezcla de forma satisfactoria. No olvide realizar este proceso sobre todos los archivos involucrados, debido a que las marcas introducidas por parte de git, no serán removidas al momento de actualizar cambios en cada uno de los repositorios remotos. Con el objetivo de finalizar el proceso de mezcla, guarde todos los cambios sobre el archivo, y actualice el estado del repositorio remoto, finalmente, es necesario reintentar el proceso de Pull Request.

Mayor información sobre funcionamiento de GIT

- Git Tutorials and Training, Atlassian, recuperado desde https://www.atlassian.com/git/tutorials/
- Github Tutorial, Github Labs, recuperado desde https://try.github.io
- Learn Git Branching, recuperado desde: http://pcottle.github.io/learnGitBranching/

Parte 2 – Diseño de aplicaciones por línea de comandos (CLI) (trabajo en clase)

Java y ejecución del entorno

De acuerdo a la documentación y filosofía de Java, el lenguaje se encuentra construido bajo la premisa "Write once, run anywhere", es decir, las instrucciones dispuestas en un archivo de código fuente pueden ser ejecutadas de forma equivalente en diversos sistemas operativos o arquitecturas de procesador.

Con el fin de abstraer estas características, Java se basa en una máquina virtual, la cual se encarga de procesar el bytecode, primero interpretándolo y luego traduciéndolo a instrucciones nativas de la máquina, de tal forma que el dispositivo huésped pueda ejecutar el programa final. Conforme a la anterior descripción, es necesario definir un compilador de java (javac) y un entorno de ejecución de Java (concretamente, el JRE se invoca con el comando java, y el compilador de java usando el comando javac).

Adicionalmente, si desea conocer acerca de las opciones de los comandos de compilación y ejecución de java en consola, es posible escribir java -help, de igual manera para el comando javac.

Java y las entradas estándar

Actualmente, se definieron tres tipos de flujo: Un flujo de entrada (stdin), un flujo de salida (stdout), y finalmente, un flujo de diagnósticos y errores (stderr). Con el fin de interactuar directamente con el sistema operativo, la JVM define a través de la clase System, tres atributos relacionados con las flujos de información estándar. El siguiente código muestra un fragmento de esta clase:

```
package java.lang;

import java.io.*;

public class System{
    ...

    public final static PrintStream out = ...;   //flujo de salida
    public final static InputStream in  = ...;   //flujo de entrada
    public final static PrintStream err = ...;   //flujo de diagnóstico de errores

    ...
}
```

Lectura en consola

Existen varias formas de leer desde la consola: utilizando un BufferedReader (la misma clase usada para leer archivos), leyendo directamente carácter por carácter o usando la clase Scanner, entre otras opciones. La recomendación en este curso es utilizar la clase Scanner de java que permite leer fácilmente desde un InputStream, que en este caso es el flujo de entrada de la consola (System.in). Esta clase ofrece funcionalidades para analizar la entrada y convertirla en tipos primitivos o strings.

Mayor información: http://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html

El siguiente ejemplo muestra como utilizar la clase Scanner para leer dos enteros de la consola e imprime el resultado de su suma. Se recomienda crear una aplicación para ejecutar y probar este código.

```
    Scanner sc = new Scanner(System.in);
    while(sc.hasNextLine()){
        int a = sc.nextInt();
        int b = sc.nextInt();
        int c = a+b;
        System.out.println(c);
    }
    sc.close();
```

Escritura en consola

Para imprimir en la consola solo debe invocar el método print o println sobre el atributo out de la clase System.

print *Imprime el string que llega como parámetro*

println *Imprime el string que llega como parámetro y un salto de línea("\n")*

```
System.out.println(<valor a imprimir>);
```

Además de los métodos **print** y **println**, la clase **PrintStream** ofrece el método format que permite imprimir un string con un formato específico. Esto es particularmente útil cuando se quieren imprimir números o una cadena de texto que debe ser compuesta utilizando múltiples variables, por ejemplo:

```
System.out.format("El estudiante %s"
                +" se encuentra en %d semestre"
                +" y su promedio es %.2f", "Pedro perez", 8, 3.9);
```

Imprimirá en la consola:

```
El estudiante Pedro perez se encuentra en 8 semestre y su promedio es 3,90
```

Un ejemplo completo

A continuación se define un ejemplo que involucra el uso de las tres entradas:

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.util.Scanner;

public class HelloStudent
{
     public static void main(String... args)
     {
        try
        {
            Scanner sc = new Scanner(System.in);
            System.out.println("Ingrese su nombre completo: ");
            String name = sc.nextLine();
            System.out.println("Ingrese su edad: ");
            int age = sc.nextInt();
            System.out.format("Bienvenido a ISIS1206 %s su edad es %d años",name, age);
        }
        catch (Exception e)
        {
            System.err.println("Error:" + e.getMessage());
                        e.printStackTrace();
```

```
        }
    }
}
```

Nota: Ejecutar la instrucción e.printStackTrace() tras capturar una excepción, es equivalente a escribir el Stack trace generado por el entorno de ejecución de Java en la entrada de error y diagnóstico, usando el bloque de ejecución:

```
try
{
   ...
}
catch(Exception e)
{
    for(StackTraceElement ste : e.getStackTrace())
    {
         System.err.println(ste);
    }
}
```

Lo que usted debe hacer

Durante esta sección, se desea diseñar e implementar una interfaz gráfica basada en el uso de elementos y flujos de entrada estándar característicos de toda aplicación basada en Consola. La aplicación que usted debe completar es un simple juego de Conecta Cuatro (Línea cuatro). Si no está familiarizado con el juego puede entenderlo un poco más por medio del siguiente link https://es.wikipedia.org/wiki/Conecta_4. En primer lugar, identifique los elementos de ejecución principales del programa basándose en los métodos existentes.

Complete la aplicación basado en la documentación en el código. Usted debe realizar cambios en las clases **LineaCuatro.java** e **Interfaz.java**. Es necesario registrar los cambios realizados en el repositorio, tanto a nivel local, como a nivel remoto. Extienda su aplicación para que:

- El usuario pueda jugar contra la maquina (no es necesario hacer nada de inteligencia artificial, asuma que la máquina pone fichas de manera aleatoria en el tablero).
- El usuario pueda ajustar el tablero de juego a algún tamaño deseado (X filas por Y columnas). Para desarrollar esta parte recuerde que mínimo debe tener 4 filas y 4 columnas
- Que haya un modo multijugador (2 o más) en el que cada jugador elija con qué símbolo (**Una** letra o **un** número) desea jugar en el tablero
- Los jugadores deben registrarse con su nombre
- Recuerde que se puede ganar bien sea verticalmente, horizontalmente o en diagonal
- La aplicación debe “dirigir” el juego (A quien le toca jugar), avisar cuando hay un ganador y debe habilitar una opción para jugar una nueva partida una vez terminada la actual.
- Extienda su aplicación para que el usuario pueda elegir el número de “fichas” juntas con las que se gana.

Entrega

1. Verifique que su proyecto cumple con los requisitos de la entrega de talleres.
2. Entregue su taller por medio de **BitBucket**. Recuerde, si su repositorio no tiene el taller o está vacío, su taller no será calificado por más de que lo haya desarrollado. No olvide crear el archivo **calificacion.txt**.