

Paradigmas de programación.  
**Laboratorio 4: Multiparadigma**

Nombre: Jordan Nicolas Godoy Rojas

Rut: 18.633.313-6

Profesor: Daniel Gacitúa

Sección: C-3

# TABLA DE CONTENIDOS

Introducción.....	4
Descripción del problema .....	4
Metodologías y herramientas utilizadas.....	5
Análisis del problema .....	6
Diseño de la solución.....	8
Aspectos de la implementación.....	10
Instrucciones de uso.....	11
Resultados y autoevaluación .....	15
Conclusiones del trabajo .....	17
Referencias.....	17

# TABLA DE ILUSTRACIONES

Ilustración 1, Diagrama de clase UML inicial.....	7
Ilustración 2, Método gitInit .....	9
Ilustración 3, Método gitAdd.....	9
Ilustración 4, Método gitAddAll.....	9
Ilustración 5, Método gitCommit.....	9
Ilustración 6, Método gitPush .....	9
Ilustración 7, Método gitPull .....	10
Ilustración 8, Método crearArchivo.....	10
Ilustración 9, Método gitStatusWorkspace .....	10
Ilustración 10, Método gitLog .....	10
Ilustración 11, Como compilar y ejecutar el programa .....	11
Ilustración 12, Interfaz gráfica del programa .....	11
Ilustración 13, Uso del método gitInit .....	12
Ilustración 14, Uso del método crearArchivo.....	13
Ilustración 15, Uso del método gitAdd.....	13
Ilustración 16, Uso del método gitCommit.....	13
Ilustración 17, Uso del método gitPush.....	13
Ilustración 18, Uso del método gitPull .....	14
Ilustración 19, Uso del método gitStatusWorkspace .....	14
Ilustración 20, Uso del método gitLog .....	14
Ilustración 21, Diagrama de clase UML final .....	16

## **INTRODUCCIÓN**

En este informe se describirá el problema planteado en el laboratorio 4 de la asignatura Paradigmas de programación, el cual corresponde a la realización de una simulación de un software de sistema de control de versiones de código fuente, en este caso Git, el cual será implementado en el lenguaje de programación Java. Además, en este informe también se mostrará una solución a dicho problema, mencionando los métodos de resolución utilizados y las limitaciones de esta implementación.

## **DESCRIPCIÓN DEL PROBLEMA**

Como se mencionó anteriormente, la implementación de la solución al problema fue desarrollada en el lenguaje de programación Java, el cual corresponde a un lenguaje del paradigma orientado a objetos, paradigma que ha sido estudiado en clases. La problemática gira en torno a la implementación de un programa que simula las funcionalidades y comportamientos de un sistema de versionamiento como lo es Git, el cual es un software donde se puede llevar un control y rastreo de los cambios de código fuente en proyectos de software. Además, este simulador debe ser implementado en un entorno gráfico para lo cual se utilizará la biblioteca Swing de Java.

Git tiene tres elementos básicos: commit, zonas de trabajo y comandos a ejecutar, siendo el commit la unidad de trabajo básica de Git, ya que constituyen a una copia del estado actual de los archivos de código, y únicamente almacenando los cambios entre el commit actual y anterior. Las zonas de trabajo son las instancias encargadas de almacenar los commits de forma local o remota, existiendo cuatro de estas:

Workspace, Index, Local Repository y Remote Repository. Y finalmente los comandos son acciones que crean, modifican y transfieren los commits entre las zonas de trabajo. Los comandos principales de Git son 6: Pull, Add, Commit, Push, Status, Log.

# METODOLOGÍAS Y HERRAMIENTAS UTILIZADAS

**Paradigma:** Son ejemplos, modelos o patrón que establecen formas o estilos de hacer las cosas.

**Paradigma Orientado a Objetos:** Es un paradigma de programación basado en el concepto de objetos, los cuales son entidades que combinan un estado (datos) y un comportamiento (procedimientos o métodos). Estos objetos se comunican entre ellos para realizar tareas.

**Clase:** Es una especie de plantilla en la que se definen los atributos y los métodos predeterminados de un tipo de objeto. Siendo los atributos las características de la clase y los métodos el comportamiento que puede tener un objeto (son equivalente a las funciones y procedimientos del paradigma imperativo-procedural).

**Objetos:** Es una instancia de clase. Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos), los mismos que consecuentemente reaccionan a eventos.

**Herencia:** Se refiere a que una subclase es capaz de obtener todos los métodos y atributos marcados como public y protected de la superclase.

**UML (Lenguaje unificado de modelado):** Es un estándar que combina varias notaciones para el paradigma orientado a objetos. Permite establecer relaciones entre los objetos y componentes que forman una aplicación del paradigma. También, permite comunicar efectivamente un diseño del paradigma entre un grupo de desarrollo.

**Java:** Lenguaje basado en el paradigma orientado a objetos y basado en clases, el cual está diseñado para tener la menor cantidad de dependencias posibles en su implementación. Es un lenguaje de programación de propósito general que busca permitir que el código de Java compilado pueda ejecutarse en todas las plataformas que admiten Java sin tener la necesidad de volver a compilarlo.

**Paradigma dirigido por eventos:** En este paradigma tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen.

## ANÁLISIS DEL PROBLEMA

Para realizar la implementación de este programa se trabajará en el lenguaje de programación Java y por tanto solo se utilizará la librería estándar de este, incluyendo la biblioteca Swing para la realización del entorno gráfico.

Los requerimientos solicitados para la realización de este son las bases de la Programación Orientada a Objetos como lo son las Clases, Atributos y los Métodos que se crean necesarios para el correcto funcionamiento del programa.

Como parte del diseño y para que la solución cumpla con el Paradigma Orientado a Objetos, se debe modelar como mínimo las siguientes entidades dentro del programa: Archivo de Texto Plano, Commit, Zonas de Trabajo y Repositorio. Y para que la solución cumpla con el Paradigma Dirigido por Eventos, se debe realizar un entorno gráfico que se pueda observar como el flujo del programa se va definiendo debido a las acciones que realiza el sistema o el usuario.

El método gitInit será la funcionalidad que nos permitirá asignarle un nombre y un autor al repositorio. Esta funcionalidad debe ser la primera en ejecutarse, debido a que el resto de funcionalidades no podrán ser utilizadas sin inicializar el repositorio.

Luego, el método gitAdd tiene como funcionalidad agregar al Index uno o más archivos de texto plano, siempre y cuando se encuentren en el Workspace.

Por otra parte, el método gitCommit tiene como funcionalidad crear un nuevo commit en el Local Repository con los contenidos del Index, y debe solicitar un nombre del autor del commit, junto a un mensaje que describa de dicho commit.

El método gitPush se encarga de tomar todos los commits del Local Repository y los envía al Remote Repository.

El método gitPull se encarga de tomar todos los archivos que se encuentren en el Remote Repository y los copia en el Workspace.

Y finalmente se pide implementar una funcionalidad que permita obtener el estado de cada Zona de trabajo, mostrando una lista con los archivos que se encuentren en el Workspace e Index, o los tres últimos commits que se encuentren en el Local Repository y Remote Repository.

Se debe tener en cuenta que el programa debe ser realizado en un entorno gráfico, que permita al usuario interactuar con el simulador, y ver como se va comportando a medida que se van utilizando las funcionalidades de git.

Luego todo esto se procede a realizar el diagrama de clases UML inicial, que describirá las entidades y relaciones del problema abordado.

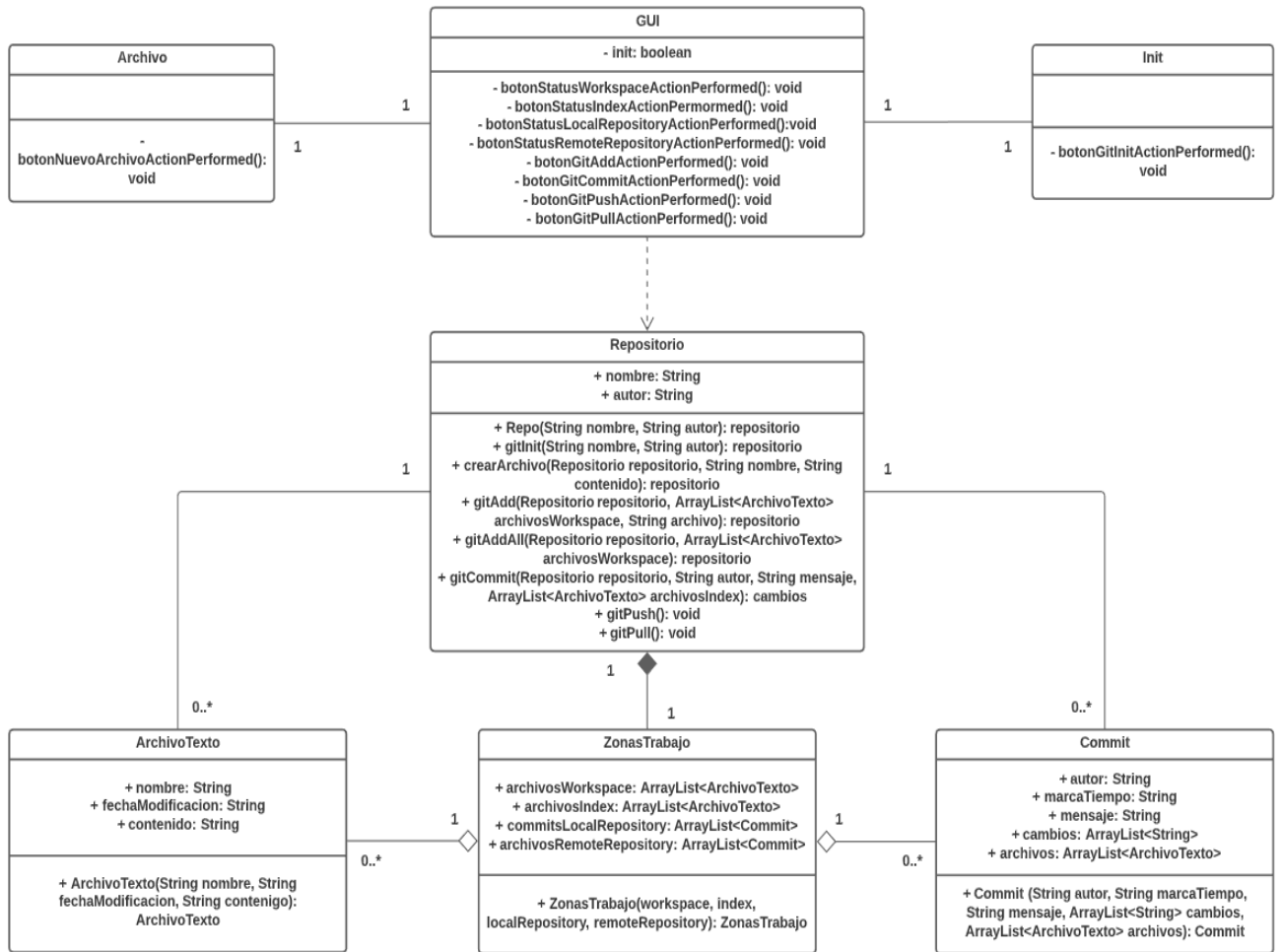


Ilustración 1: Diagrama de clases UML inicial.

## DISEÑO DE LA SOLUCIÓN

La implementación de la solución para el problema planteado se realizó mediante el uso del lenguaje de programación Java y de los conocimientos sobre este en el curso de Paradigmas de programación, además de lo investigado de forma individual.

Para resolver el problema inicial y lograr implementar la simulación del software de control de versiones Git, se modeló una solución desde lo teórico (abstracción) definiendo las clases y sus componentes, ya que de esta manera fue más sencillo implementar las funcionalidades requeridas para el correcto funcionamiento del programa.

El programa consta de 5 archivos con extensión “.java” y están separados en 2 paquetes. El primer paquete llamado modelo posee 4 archivos, los cuales son el código fuente del programa y el otro paquete llamado vista posee 1 archivo, el cual sirve para el entorno gráfico.

Las Clases implementadas en este programa son las siguientes:

**ArchivoTexto:** Un ArchivoTexto corresponde a un archivo de texto que posee como atributos nombre (nombre del archivo de texto), fechaModificacion (fecha de modificación del archivo) y contenido (contenido del archivo), todos los atributos son Strings. Posee métodos acordes a su comportamiento para este problema.

**Commit:** Un Commit está compuesto por: autor (string que representa el autor que realizó el commit), marcaTiempo (string que representa la marca de tiempo que registra la fecha cuando se realizó el commit), mensaje (string que representa el mensaje que describe el commit), cambios (ArrayList que contiene los cambios a los archivos) y archivos (ArrayList que contiene los archivos). Posee métodos acordes a su comportamiento para este problema.

**Repositorio:** Un Repositorio está compuesto por un nombre (string que representa el nombre del repositorio) y un autor (string que representa al autor del repositorio). Posee todos los métodos que modelan el funcionamiento de git.

**ZonasTrabajo:** Clase que modela el comportamiento de una zona de trabajo, será representada con 4 ArrayList. Posee como atributo: archivosWorkspace (ArrayList que representa los archivos dentro del Workspace), archivosIndex (ArrayList que representa los archivos dentro del Index), commitsLocalRepository (ArrayList que representa los commits dentro del Local Repository), y archivosRemoteRepository (ArrayList que representa los archivos dentro del Remote Repository). Posee métodos acordes a su comportamiento para este problema.

**GUI:** Clase que se encarga del entorno gráfico, esta compuesta por varios métodos, los cuales permiten al usuario o sistema interactuar con el simulador. Posee métodos acordes a su comportamiento para este problema, y además los métodos pueden ejecutar funciones de la clase Repositorio, esto es debido a que tienen una relación de dependencia. Esta clase será la encargada de pedirle al usuario que ingrese datos con los que operarán las demás clases.



Luego de implementar las clases, están los métodos apropiados para facilitar la resolución del problema:

**gitInit ()**: Esta funcionalidad se encarga de obtener los atributos nombre y autor, y los pasa como atributos de entrada al método Repo para que le asigne los atributos al Repositorio. Este método retorna el repositorio.

```
public Repositorio gitInit(String nombre, String autor)
```

*Ilustración 2, método gitInit.*

**gitAdd ()**: Esta funcionalidad se encarga de agregar al Index uno o más archivos desde el Workspace. Esta funcionalidad ingresa como entrada el repositorio, el Workspace, y un nombre de archivo que será entregado por el usuario. Luego de tener todos los atributos el método procede a buscar el archivo en el Workspace, en caso de encontrarlo lo agrega al Index, en caso contrario le indicará al usuario que el nombre ingresado no es válido o no se encuentra en el Workspace.

```
public Repositorio gitAdd(Repositorio repositorio, ArrayList<ArchivoTexto> archivosWorkspace, String archivo)
```

*Ilustración 3, método gitAdd.*

**gitAddAll ()**: Este método se encarga de agregar todos los archivos del Workspace al Index (simula un add –all de git). Este método recibe como atributos el repositorio y el Workspace.

```
public Repositorio gitAddAll(Repositorio repositorio, ArrayList<ArchivoTexto> archivosWorkspace)
```

*Ilustración 4, método gitAddAll.*

**gitCommit ()**: Esta funcionalidad se encarga de crear un nuevo commit en el Local Repository con los contenidos del Index, solicitando al usuario un autor del commit, como también un mensaje que describa el commit. Este método primero verificará si es que existen archivos en el Index, debido a que sin archivos no se puede realizar el commit. Este método recibe como atributos el repositorio, autor, mensaje y el Index, y retorna un arreglo de cambios.

```
public ArrayList<String> gitCommit(Repositorio repositorio, String autor, String mensaje, ArrayList<ArchivoTexto> archivosIndex)
```

*Ilustración 5, método gitCommit.*

**gitPush ()**: Esta funcionalidad se encarga de tomar todos los commits del Local Repository y los envía al Remote Repository. Este método al iniciarlo verificará si es que hay commits en el Local Repository, debido a que no puede enviar commits al Remote Repository si no existen. Este método es un void, por tanto, no tiene atributos de entrada ni retorno.

```
public void gitPush()
```

*Ilustración 6, método gitPush.*

**gitPull ()**: Esta funcionalidad se encarga de tomar todos los archivos del Remote Repository y los copia en el Workspace. Este método al iniciarlo comprobará si existen archivos dentro del Remote Repository, debido a que en caso de que no existan, no podrá copiar nada al Workspace. Este método es un void, por tanto, no tiene atributos de entrada ni retorno.

```
public void gitPull()
```

*Ilustración 7, método gitPull.*

**crearArchivo ():** Esta funcionalidad se encarga de crear un nuevo archivo de texto, recibiendo como parámetros de entrada el repositorio, un nombre y un contenido para el archivo de texto, los cuales deben ser ingresados por el usuario. Este método agrega el archivo de texto al Workspace y retorna el repositorio.

```
public Repositorio crearArchivo(Repositorio repositorio, String nombre, String contenido)
```

*Ilustración 8, método crearArchivo.*

**gitStatus ():** Esta funcionalidad se encarga de mostrar el contenido de cada zona de trabajo, en caso de el Workspace e Index, se encarga de mostrar todos los archivos contenidos en estos. Por otra parte, en el caso del Local Repository y Remote Repository se encarga de mostrar los 3 últimos commits de estos. Esta funcionalidad se implementó dividiéndola en 4 métodos, un método para cada zona de trabajo.

```
public String gitStatusWorkspace(Repositorio repositorio, ArrayList<ArchivoTexto> archivosWorkspace) {
```

*Ilustración 9, método gitStatusWorkspace.*

**GitLog ():** Esta funcionalidad se encarga de mostrar los últimos 5 commits del Local Repository indicando fecha, mensaje y archivos que se encuentren en este. Este método recibe como parámetro de entrada el repositorio y la zona Local Repository, retornando un string que posee los datos necesarios para la implementación.

```
public String gitLog(Repositorio repositorio, ArrayList<Commit> commitsLocalRepository)
```

*Ilustración 10, método gitLog.*

## ASPECTOS DE LA IMPLEMENTACIÓN

Como se mencionó anteriormente, la solución fue programada en el lenguaje Java, el cual corresponde a un lenguaje del Paradigma Orientado a Objetos y el IDE Apache NetBeans versión 12.1. Para la implementación de este laboratorio se hizo uso de la librería estándar de Java incluida la biblioteca Swing.

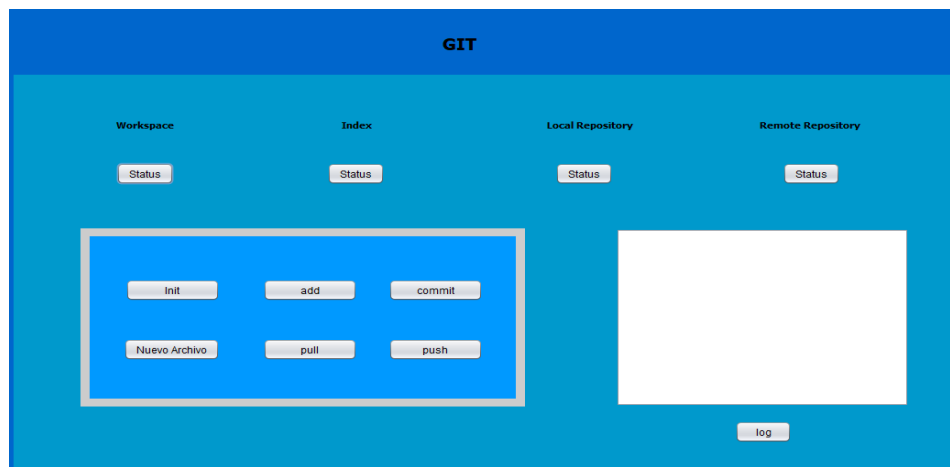
El programa consta de cinco archivos, de los cuatro cuales representan a las clases del programa y el último al GUI que representa el entorno gráfico. Las clases se llaman ArchivoTexto.java, Commit.java, Repositorio.java y ZonasTrabajo.java, y el GUI se llama GUI.java. Los requerimientos funcionales obligatorios se encuentran en su mayoría en la clase repositorio, en este caso en el archivo Repositorio.java. Todos estos archivos deben estar en el mismo directorio y deben ser compilados en conjunto, o si no, el programa no compilará ni tampoco funcionará.

## INSTRUCCIONES DE USO

Para comenzar, es necesario descargar e instalar la versión más reciente de JDK, la cual se puede descargar desde el sitio web oficial de Oracle (revisar referencias del informe). Una vez descargado e instalado, se debe buscar la ruta de instalación del JDK (por ejemplo C:\Program Files\Java\jdk1.8.0\_111\bin), luego esta dirección debe agregarse a la variable de entorno Path para poder usar la consola de comandos de Windows (cmd), y para lograr esto se deben realizar los siguientes pasos (esto en Windows 10): apretar tecla de inicio de Windows -> escribir ver la configuración avanzada del sistema -> en la pestaña opciones avanzadas hacer click en la parte inferior derecha, la cual dice Variables de entorno -> buscar en la parte de abajo en Variables del sistema la variable Path y dar click en editar -> click derecho arriba a la derecha en nuevo -> agregar la ruta de instalación JDK -> aceptar y estará listo para usar la consola de Windows. Una vez agregado la ruta al Path lo que se debe hacer es presionar la tecla de Windows + la letra R, escribir “cmd y presionar la tecla enter. Una vez realizado esto, se debe ir a la ruta en donde se encuentran los archivos a compilar (ruta de la carpeta contenedora de los archivos .java), luego se debe ir a la carpeta llamada “src”. Para esto, se escribe en la consola de Windows cd y se pega la ruta, realizado esto, necesita escribir el comando “javac modelo/\*.java” y presionar la tecla enter, luego hace lo mismo, pero con el paquete vista, para esto tiene que escribir “javac vista/\*.java”. Luego de un momento, los archivos se compilarán y estarán listos para ejecutarse. Para esto, se debe escribir el comando “java vista/GUI” y presionar la tecla enter. A partir de aquí el programa se ejecutará a través de la interfaz gráfica, permitiéndole al usuario solo usar las funcionalidades después de que aprete el botón Init para asignarle un nombre y autor al Repositorio, una vez realizado esto, el usuario podrá hacer uso de todas las funcionalidades que ofrece el programa.

```
C:\Users\Nico\Desktop\git>cd src
C:\Users\Nico\Desktop\git\src>javac modelo/*.java
C:\Users\Nico\Desktop\git\src>javac vista/*.java
C:\Users\Nico\Desktop\git\src>java vista/GUI
```

*Ilustración 11: Como compilar y ejecutar el programa.*



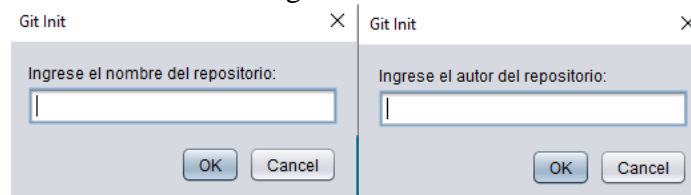
*Ilustración 12: Interfaz gráfica del programa.*

El simulador de git funciona en base a un entorno gráfico, el cual permite una mejor experiencia para el usuario. Las principales funcionalidades del entorno gráfico son las siguientes:

- Init: Le pide al usuario un nombre y autor para el repositorio. **Esta funcionalidad obligatoriamente debe ser la primera en utilizarse, debido que esta es la que inicia el repositorio. Si esta funcionalidad no se usa al inicio, no se podrá usar ninguna otra funcionalidad.**
- Nuevo Archivo: Le pide al usuario un nombre y contenido para el nuevo archivo de texto. Este archivo se crea en el Workspace.
- Add: Esta funcionalidad le hace pregunta al usuario si quiere agregar uno o todos los archivos desde el Workspace al Index.
- Commit: Esta opción le pide al usuario un nombre de autor y mensaje descriptivo para el commit. Luego crea un nuevo commit en el Local Repository con los archivos contenido en el Index.
- Push: Esta opción toma todos los archivos que se encuentran en el Local Repository y los envía al Remote Repository.
- Pull: Esta opción toma todos los archivos que se encuentran en el Remote Repository y los copia en el Workspace.
- Status: Esta opción se encuentra habilitada para las 4 zonas de trabajo (Workspace, Index, Local Repository y Remote Repository). Esta funcionalidad muestra todos los archivos incluidos en el Workspace e Index, o los 3 últimos commits en el Local Repository y Remote Repository.
- Log: opción que muestra los últimos 5 commits del Local Repository, indicando fecha, mensaje y archivos que se encuentren dentro de este.

El programa funciona en base a métodos específicos y otros que permiten una mejor simulación de git. Los métodos principales de git son los siguientes:

- gitInit: Esta funcionalidad debe ser la primera en usarle luego de ejecutar el programa, debido a que es la que le da los atributos al repositorio. Luego de usarla y rellenar los campos requeridos podrán utilizarse las demás funcionalidades. Un ejemplo de esta funcionalidad es el siguiente:



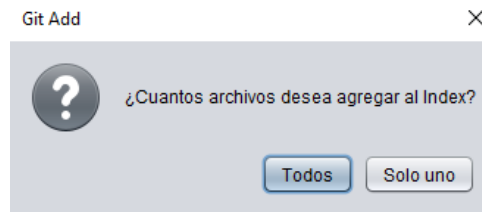
*Ilustración 13, uso del método gitInit.*

- crearArchivo: Esta funcionalidad es la encargada de crear los archivos de texto, pidiéndole al usuario un nombre y contenido para estos. Se validarán los datos ingresados, y en caso de que sean válidos se agregarán al Workspace.



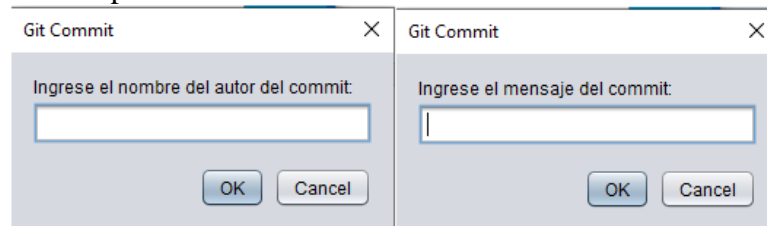
*Ilustración 14, uso del método CrearArchivo.*

- gitAdd: Esta funcionalidad tiene 2 formas de uso, dependiendo de la opción del usuario, siendo las opciones las siguientes:
  1. Si escogió la opción Solo uno, se le pedirá que ingrese el nombre del archivo que desea agregar al Index.
  2. Si escogió la opción Todos, agregará todos los archivos del Workspace al Index.



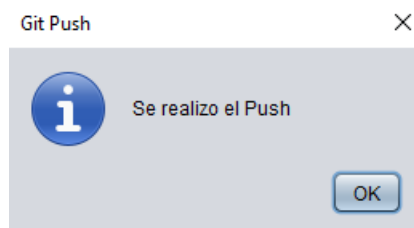
*Ilustración 15, uso del método gitAdd.*

- gitCommit: Esta funcionalidad tiene una única forma de uso y es luego de que usuario la seleccione en el entorno gráfico. Este método funcionará siempre y cuando haya uno o más archivos en el Index, en caso contrario le dirá por pantalla al usuario que no hay archivos para realizar un commit.



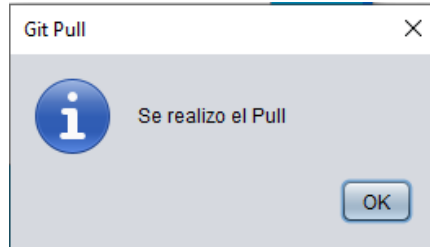
*Ilustración 16, uso del método gitCommit.*

- gitPush: Esta funcionalidad tiene una única forma de uso y es luego de que el usuario la seleccione en el entorno gráfico. Este método funcionará siempre y cuando exista uno o más commits en el Local Repository, en caso contrario le dirá por pantalla al usuario que no hay commits para enviarlos al Remote Repository.



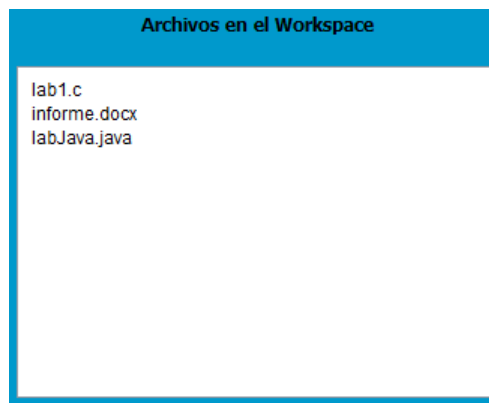
*Ilustración 17, uso del método gitPush.*

- gitPull: Esta funcionalidad tiene una única forma de uso y es luego de que el usuario la seleccione en el entorno gráfico. Este método funcionará siempre y cuando exista uno o más archivos en el Remote Repository, en caso contrario le dirá al usuario por pantalla que no hay archivos para copiar al Workspace.



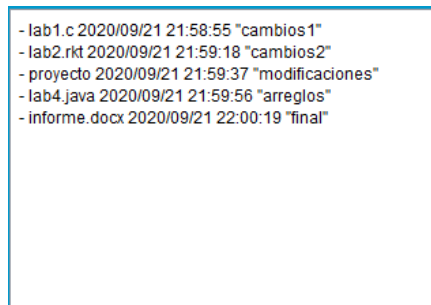
*Ilustración 18, uso del método gitPull.*

- gitStatus: Esta funcionalidad tiene como fin mostrar el contenido dentro de cada zona de trabajo. Se realizaron 4 métodos para realizar esta funcionalidad, un método para cada zona de trabajo. Este método mostrará todos los archivos que se encuentren en el Workspace e Index, y los 3 últimos commits del Local Repository y Remote Repository.



*Ilustración 19, uso del método gitStatusWorkspace.*

- gitLog: Esta funcionalidad tiene como fin mostrar los últimos 5 commits que se encuentren en el Local Repository, indicando fecha, mensaje y archivos en este.



*Ilustración 20, uso del método gitLog.*

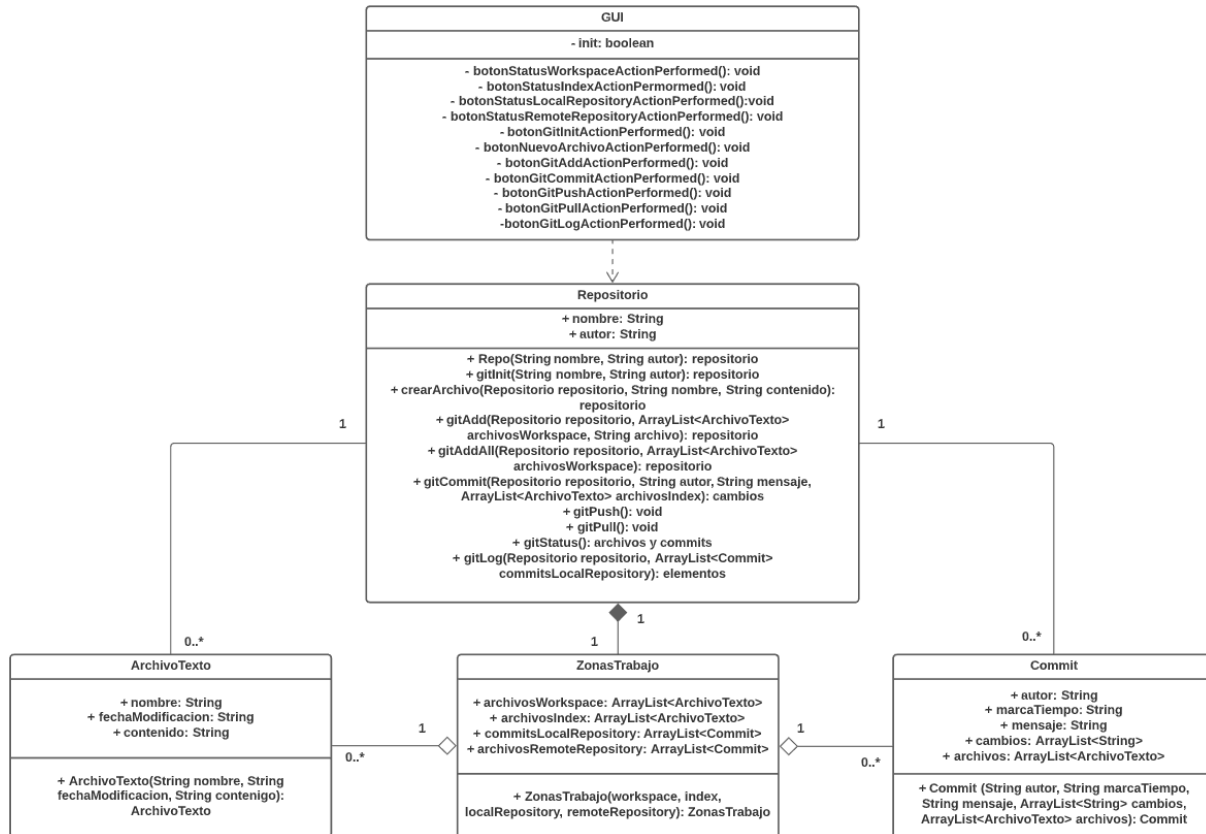
## RESULTADOS Y AUTOEVALUACIÓN

Este laboratorio tenía los siguientes requerimientos funcionales:

- **Clases y estructuras que forman el programa:** Se realizaron las clases pedidas para la implementación de git, que en este caso fueron cuatro: ArchivoTexto, Commit, Repositorio y ZonasTrabajo.
- **gitInit:** Se realizó completamente el requerimiento de gitInit, el cual debe ejecutarse al inicio del programa. Esta funcionalidad le pide al usuario un nombre y autor del repositorio para asignarle los atributos a este. Se realizó una validación que impide usar las otras funcionalidades antes que el Init.
- **gitAdd:** Se realizó completamente el requerimiento gitAdd, el cual le pregunta al usuario si quiere agregar uno o todos los archivos del Workspace al Index. Si se escoge la opción de un archivo, este método le pedirá al usuario que ingrese por el nombre del archivo, y en caso de encontrar el archivo en el Workspace se agregará el archivo al Index. En caso de no encontrar el archivo en el Workspace, le informará al usuario.
- **gitCommit:** Se realizó completamente el requerimiento gitCommit, en el cual primero se comprueba que existan archivos en el Index antes de realizar un commit y que el tamaño del Local Repository no sea mayor al Index, esto es debido a que no se pueden realizar commits sin archivos. Luego se crea el commit con sus respectivos atributos.
- **gitPush:** Se realizó completamente el requerimiento gitPush, en el cual se hace una validación al inicio del método, en el cual comprueba que existan commits en el Local Repository para que sea capaz de enviarlo al Remote Repository. Luego de esto se copian los archivos al Remote Repository.
- **gitPull:** Se realizó completamente este requerimiento, en el cual primero se comprueba que existan elementos en el Remote Repository, en caso de existir se realiza una comparación de elementos, en caso de encontrar elementos que sean iguales a los que hay en el Workspace, se elimina el archivo del Workspace y se reemplaza con el elemento del Remote Repository.
- **crearArchivo:** Se realizó completamente este requerimiento, en el cual primero le pide al usuario que ingrese un nombre y contenido para el archivo, y se hace una validación de que los datos tengan contenido antes de aceptarlos. Luego se hace otra validación que comprueba que en el Workspace no existan archivos con el mismo nombre del nuevo archivo.
- **gitStatus:** Esta funcionalidad se realizó completamente, mostrando todo lo pedido, el único problema fue que se tuvo que realizar un método para que muestre el status de cada zona de trabajo. El único inconveniente con esta funcionalidad es que no muestra todos los archivos que se encuentran en los commits.

- **gitLog:** Esta función realiza lo pedido mostrando los 5 últimos commits en el Local Repository. El único inconveniente de esta funcionalidad es que no muestra todos los archivos del commit, solo muestra el primero.
- **Interacción con el usuario:** Por último, este requerimiento se cumplió en su totalidad, se realizaron validaciones de entradas para que no se caiga el programa.

Para finalizar se concluye que los resultados fueron buenos en todas las funcionalidades.



*Ilustración 21: Diagrama de clases UML final.*

Conclusiones respecto a los cambios entre los diagramas de clases:

- Se eliminaron 2 clases del paquete vista: Init y Archivo, y sus métodos se agregaron a la clase GUI.
- A pesar de las 2 clases eliminadas las relaciones entre las clases siguieron de la misma manera, por lo que se puede considerar que el diagrama inicial no estaba mal, si no que solamente faltó simplificarlo.
- Las clases ArchivoTexto, Repositorio, Commit y ZonasTrabajo no sufrieron cambios.

Se puede ver la razón de porque pensar bien y modelar un programa puede resultar en un programa mejor conformado que requiera menos tiempo de programación.



# CONCLUSIÓN

Para el desarrollo de la simulación del sistema de control de versiones git, fue importante incorporar los conceptos aprendidos sobre el Paradigma Orientado a Objetos y en el Paradigma Dirigido por Eventos, ya que de esta manera se logró dividir mejor los problemas en otros más pequeños, y por tanto más abordables. El diseño de las Clases, sus atributos y métodos, fueron una dificultad al momento de pensar en la solución del problema, ya que como se mostró en el primer diagrama UML, la solución inicial planteada es muy distinta a la solución final. Esto hace querer realizar el modelado de la mejor forma posible, debido a que esto provoca que la programación sea más directa que pensar mientras se está programando.

El manejo de los objetos, la necesidad de instanciar o inicializar las Clases para poder hacer uso de estas conllevó una dificultad abordable, ya que a medida que se ahondaba en el aprendizaje del paradigma, este se volvía más amigable de forma que resultaba más entendible.

El paradigma orientado a objetos junto con el dirigido por eventos, fueron de gran utilidad, debido a que generan la obligación de pensar mejor como modelar la solución antes de programarla, debido a que esto puede dificultar la solución al momento de aplicarla en un entorno gráfico. Fue gracias a esto que se logró completar el proyecto satisfactoriamente, y a través de este laboratorio comprender algunos de los conceptos de la programación orientada a objetos como lo es el proceso de instanciación y el uso del entorno gráfico. Se consiguió implementar todas las funciones obligatorias y una extra.

# REFERENCIAS

Página web de descarga JDK8:

- <https://www.oracle.com/cl/java/technologies/javase/javase-jdk8-downloads.html>

Página web de descarga Apache NetBeans 12.1:

- <http://netbeans.apache.org/>

Enunciado para el proyecto de paradigmas de programación:

- [https://docs.google.com/document/d/1WF8EDXxDPcdJhV0UrQY94HV9\\_NLKCsGMHHgu\\_NSApGk/edit](https://docs.google.com/document/d/1WF8EDXxDPcdJhV0UrQY94HV9_NLKCsGMHHgu_NSApGk/edit)

Java plataforma, Standard Edition 7:

- <https://docs.oracle.com/javase/7/docs/index.html>