Universidad José Antonio Paez.
Facultad de Ingeniería.
Escuela de Ingeniería en Computación.

Parcial IV (15%). TDA Tipos de Datos Abstractos.

Enunciado del Proyecto

En este proyecto, se desarrollará una **simulación de un sistema Git** utilizando estructuras de datos avanzadas como **pilas**, **colas**, **listas enlazadas**. El equipo de desarrollo deberá modelar componentes clave de un sistema de control de versiones, implementar sus operaciones, y comprender cómo estas estructuras de datos facilitan la gestión eficiente de versiones, historial y ramas.

El sistema debe incluir una interfaz basada en **comandos básicos** de Git (simulados), tales como `add`, `commit`, `branch`, `checkout`, y `merge`.

Para este proyecto, desarrollar los siguientes módulos:

1.-Módulo Principal para el Control de Versiones (4 ptos)

Este módulo actúa como la interfaz principal para gestionar múltiples **repositorios** simulados de control de versiones. En él, se coordinarán las interacciones entre los diferentes módulos de gestión de commits, área de staging, proporcionando a los usuarios una forma sencilla de realizar operaciones como agregar archivos, crear commits y más.

Este modelo permitirá a los usuarios interactuar con el programa a través de comandos. Todos los comandos deben ser programado utilizando el patrón de diseño comandos de forma que un comando pueda ser habilitado o deshabilitado a través de un archivo de texto de configuración

El programa deberá permitir gestionar de forma lógica múltiples repositorios(crear, modificar, listar, abrir) en una lista enlazada y a su vez cada repositorio contará con su estructura de archivos y commit.

Un commit alacena Identificación (hash SHA-1)), Fecha y hora del commi, Correo electrónico del autor, Mensaje del Commit, Referencia al Commit Padre, Archivos Modificados, nombre de la rama

`git init`

- Descripción: Crea un nuevo repositorio vacío en el directorio actual. Inicia un repositorio de Git, configurando la estructura básica de control de versiones.
- Cuando creas un repositorio con `git init`, el nombre del repositorio se asigna automáticamente al nombre del directorio en el que lo estás inicializando
- Salida: Un nuevo repositorio Git es creado en el directorio, con un `.git` que contiene toda la información y la configuración.

`git status`

- Descripción: Muestra el estado actual del repositorio local, indicando si hay cambios pendientes, archivos no rastreados, archivos modificados, etc.

`git status`

-
- Salida: Muestra el estado de los archivos en el directorio de trabajo y el área de staging.

`git log`

- Descripción: Muestra el historial de commits del repositorio. Puedes ver los detalles de cada commit (autor, fecha, mensaje).

2.-Módulo para Gestión de Commits (lista enlazada)(4 ptos)

Este módulo maneja los commits, que representan los cambios registrados en el sistema de control de versiones. Cada commit almacena una referencia a su commit anterior, formando una lista enlazada que permite navegar por el historial de versiones.

Comandos:

`git add <archivo>`

- Descripción: Agrega un archivo o conjunto de archivos al área de preparación para ser comprometidos en el siguiente commit.
- Parámetros:
  - `<archivo>`: El archivo o archivos a agregar.
- Salida: Los archivos seleccionados se marcan para su inclusión en el siguiente commit.

`git commit -m "<mensaje>"`

- Descripción: Crea un nuevo commit con los archivos previamente agregados, incluyendo un mensaje descriptivo del cambio.
- Parámetros:
  - `<mensaje>`: Descripción del commit, que explica los cambios realizados.
- Salida: Un commit que incluye los archivos añadidos y el mensaje proporcionado. Este commit se enlaza con el commit anterior.

`git status`

- Descripción: Muestra el estado actual del repositorio, indicando qué archivos están modificados, cuáles están listos para ser comprometidos y si hay algún commit pendiente.
- Salida: Información sobre los cambios en los archivos, si están listos para ser añadidos o si hay commits pendientes.

`git log`

- Descripción: Muestra el historial completo de commits realizados en el repositorio.

- Salida: Lista de todos los commits, con detalles como el identificador único, mensaje y fecha de cada commit.

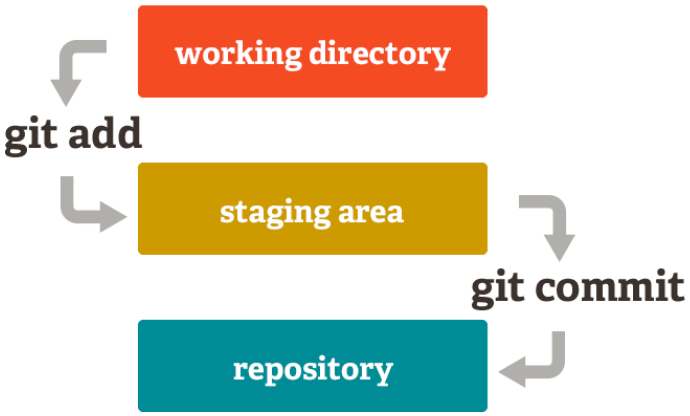
```
git checkout <commit_id>
```

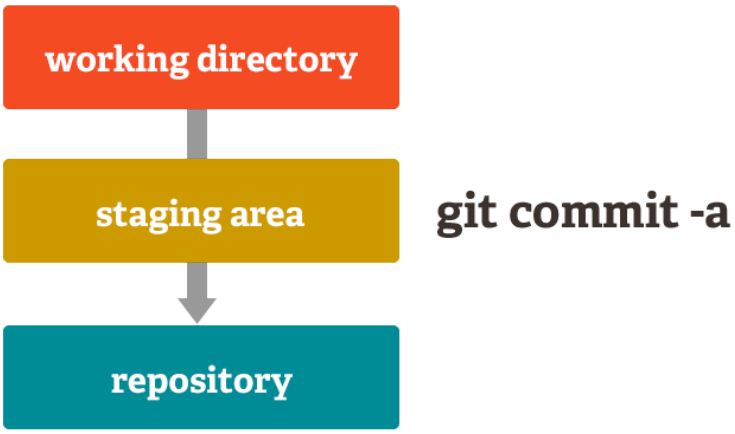
- Descripción: Permite navegar a un commit específico en el historial, restableciendo el estado del repositorio a ese commit.
- Parámetros:
  - `<commit_id>`: El identificador único del commit al que se desea volver.
- Salida: El repositorio se actualiza para reflejar el estado del commit especificado.

3.-Módulo para el Área de Staging(Pila)(4 ptos)

Este módulo gestiona el área de staging, un espacio temporal donde se almacenan los archivos modificados antes de ser confirmados en un commit. El proceso de *staging* permite seleccionar de manera precisa qué cambios se desean incluir en el siguiente commit, asegurando que solo los archivos que se elijan sean registrados en el repositorio. Los archivos se agregan al área de staging (mediante un sistema de pila) y permanecen allí hasta que el usuario decida confirmarlos. Esta etapa proporciona control total sobre los cambios que se incluyen, permitiendo al usuario revisar y preparar los archivos antes de su confirmación final. De esta forma, se evita que todos los archivos modificados se comprometan de forma automática, dando la opción de elegir cuáles se incluirán en el historial de versiones.

El área de staging debe guardar Lista de archivos preparados para el commit, Estado de los archivos(Añadido A, Modificado M, Eliminado D), Checksum (SHA-1) de cada archivo, Metadatos de los archivos(nombre del archivo, ubicación del archivo), referencia con el último commit





4.- Módulo de Pull Request (Cola)(4 ptos)

Este módulo gestiona los pull requests mediante una estructura de cola, que permite administrar de manera ordenada y secuencial las solicitudes de integración de cambios en el repositorio.

En este proceso, cada pull request se coloca en la cola, lo que asegura que las solicitudes se procesan en el orden en que fueron recibidas, respetando la prioridad y la secuencia de revisión.

Los cambios propuestos por los colaboradores se almacenan en la cola hasta que son revisados, aprobados y fusionados con el código principal. Este enfoque garantiza que los cambios se gestionan de manera organizada, facilitando la revisión y la integración de manera controlada y estructurada, al tiempo que permite a los desarrolladores gestionar las solicitudes pendientes de forma eficiente.

Cada pull request debe guardar : Identificador único (ID), Título, Descripción, Autor, Fecha de creación, Rama de origen (source branch), Rama de destino (target branch), Commits asociados(Referencia a una lista de commit), Archivos modificados, Revisores asignados, Fecha de cierre o fusión

1. `git pr create <rama_origen> <rama_destino>`
    - Crea un nuevo pull request para fusionar los cambios de la rama de origen en la rama de destino.
    - Parámetros:
        - `<rama_origen>`: La rama que contiene los cambios a fusionar.
        - `<rama_destino>`: La rama de destino en la que se desean integrar los cambios.
    - Salida: Un pull request es creado y colocado en la cola de solicitudes pendientes.
2. `git pr status`
    - Muestra el estado actual de los pull requests, indicando cuáles están pendientes, en revisión, aprobados, fusionados o rechazados.
    - Salida: Una lista de los pull requests en la cola con su estado actual.

3. `git pr review <id_pr>`
    - Revisa un pull request específico, proporcionando comentarios y sugerencias sobre los cambios propuestos.
    - Parámetros:
        - `<id_pr>`: El identificador único del pull request que se desea revisar.
    - Salida: El pull request es marcado como "en revisión" y se agregan los comentarios.
4. `git pr approve <id_pr>`
    - Aprueba un pull request, indicando que los cambios propuestos son aptos para ser fusionados.
    - Parámetros:
        - `<id_pr>`: El identificador del pull request que se aprueba.
    - Salida: El pull request es aprobado y marcado como listo para fusionar.
5. `git pr reject <id_pr>`
    - Rechaza un pull request, indicando al autor que no se pueden integrar los cambios propuestos.
    - Parámetros:
        - `<id_pr>`: El identificador del pull request que se rechaza.
    - Salida: El pull request es retirado de la cola y marcado como rechazado.
6. `git pr cancel <id_pr>`
    - Cancela un pull request en la cola, eliminándolo sin que se procese ni revise.
    - Parámetros:
        - `<id_pr>`: El identificador único del pull request que se desea cancelar.
    - Salida: El pull request es eliminado de la cola sin ser revisado ni fusionado.
7. `git pr list`
    - Muestra todos los pull requests actuales en la cola, incluyendo el estado de cada uno.
    - Salida: Una lista detallada de todos los pull requests pendientes, en revisión, aprobados, fusionados, o rechazados.
8. `git pr next`
    - Procesa el siguiente pull request en la cola (el primero que está esperando revisión).
    - Salida: El pull request más antiguo en la cola se mueve a revisión y se marca como "en proceso".
9. `git pr tag <id_pr> <etiqueta>`
    - Asigna una etiqueta a un pull request para identificarlo fácilmente, como "urgente", "en revisión", "mejoras", etc.
    - Parámetros:
        - `<id_pr>`: El identificador único del pull request.
        - `<etiqueta>`: La etiqueta que se desea asignar.
    - Salida: Se asigna la etiqueta al pull request para su fácil identificación y organización.
10. `git pr clear`
    - Elimina todos los pull requests pendientes de la cola sin procesarlos.

- Salida: Todos los pull requests pendientes son eliminados de la cola sin ser revisados ni fusionados.

Pautas de Evaluación.

1. La evaluación es individual o en equipo de un máximo de tres personas.
2. Usar paradigma de programación orientada a objetos
3. La defensa del proyecto tiene un valor de 3 puntos.
4. Utilizar repositorios github tiene un valor de 1 punto.
5. Los códigos iguales tendrán una penalización de puntos menos.
6. La entrega y defensa se realizará de forma presencial en hora de clases.
7. Realizar validaciones de datos introducidos por el usuario en los comandos y datos cargados.
8. Los datos deben ser guardados como archivos.json y cargados al momento de iniciar el programa.
9. El código deberá estar comentado.
10. Tener datos por defectos para tomarlos como prueba.
11. En cada módulo se evaluará los siguiente:
    - Funcionamiento del módulo(errores, resultados correctos, independencia).
    - Legibilidad del código(nombres de variables, código comentado correctamente)
    - El alumno aplicó elementos conceptuales en la programación del módulo
    - Módulo entregado puntualmente.