Imagen que contiene imágenes prediseñadas

Descripción generada automáticamente

**INSTITUTO PROFESIONAL IACC**

ESCUELA DE TECNOLOGÍAS APLICADAS

INGENIERÍA EN INFORMÁTICA

**“Software para la gestión y eliminación de ramas en repositorios administrados con Git, facilitando la administración en plataformas como GitHub”**

NOMBRE PROFESOR(A) GUÍA: Marianela Morillo Chacin

NOMBRE DEL ALUMNO(A): Sergio Polanco González

Enero - 2025

**RESUMEN**

El planteamiento del problema que deseo abordar en este Seminario de Título es que me he dado cuenta que las fuentes de cada desarrollo de software vienen a tener una importancia tan grande que el hecho de haberse creado un sistema de versionamiento de estas, viene a resolver justamente los problemas de seguridad y mantención de las aplicaciones, logrando establecer reglas y buenas practicas que nos permiten mantener históricamente las distintas intervenciones que se realizan a las fuentes y sus mejoras. Este sistema de versionamiento, el cual en la actualidad y siendo el más popular corresponde al sistema Git, nos permite la manipulación de las fuentes y su sincronización con las distintas plataformas del tipo repositorio que nos permiten llevar este versionamiento de la manera más ordenada y sistemática, pero existe un tema muy poco abordado por los desarrolladores (encargados de manipular las fuentes de un proyecto), el cual está relacionado a la cantidad de ramas que se crean por pruebas realizadas tanto en los ambientes previos como los que se actualizan en un ambiente productivo. Esto me llevó a pensar en lo importante que es el mantener estas fuentes lo más actualizadas posibles, de esta manera no se utiliza mucho espacio asignado por cada repositorio y a su vez se mantiene el código lo más actualizado y ordenado posible, otorgando claridad en las actualizaciones y no redundar en la información almacenada en los repositorios.

PALABRAS CLAVES:

* Repositorios
* Ramas
* Versionamiento
* Fuentes

Todos los derechos de autor son de la exclusiva propiedad de IACC o de los otorgantes de sus licencias. No está permitido copiar, reproducir, reeditar, descargar, publicar, emitir, difundir, poner a disposición del público ni utilizar los contenidos para fines comerciales de ninguna clase.

**Título de la Semana**

SEMANA 1



[1. EL PROBLEMA 4](#_Toc185242464)

[1.1 PLANTEAMIENTO DEL PROBLEMA 5](#_Toc185242465)

[1.1.1 OBJETIVO GENERAL 6](#_Toc185242466)

[1.1.2 OBJETIVOS ESPECÍFICOS 6](#_Toc185242467)

[1.1.3 hipotesis 26](#_Toc185242468)

[1.2 JUSTIFICACIÓN DE LA INVESTIGACIÓN 26](#_Toc185242469)

[2. MARCO TEÓRICO 26](#_Toc185242470)

[3. MARCO METODOLÓGICO 29](#_Toc185242471)

[3. ANÁLISIS DE LOS RESULTADOS 36](#_Toc185242472)

[4. CONCLUSIONES y RECOMENDACIONES 40](#_Toc185242478)

[REFERENCIAS BIBLIOGRÁFICAS 42](#_Toc185242479)

# EL PROBLEMA

Desde que he trabajado con el sistema de versionamiento que posee cada proyecto informático que necesite mantener sus fuentes versionadas que me he dado cuenta que existen una problemática que los equipos de desarrollo no han abordado, y es específicamente de la multiplicidad de fuentes que se generan por todas las pruebas que se realizan en el ciclo de vida de un desarrollo (software), entendiendo que las fuentes manipuladas debieran concentrar solo los cambios que se pasaron a producción (de cara al cliente) y el resto de las fuentes deprecadas eliminarlas para no afectar el espacio asignado a los repositorios. El impacto que esto genera en las actividades diarias de un equipo de desarrollo viene a provocar pequeñas alteraciones en el proceso de despliegue de un software, tanto para sus actualizaciones como para la puesta en producción de un proyecto, por lo que es de suma importancia que tengamos claro cuales fuentes serán las utilizadas para aplicar las mejoras y proceder a eliminar el material sobrante, con el propósito de mantener vigente solo las fuentes que se utilizarán para concretar la mejora planteada.

## PLANTEAMIENTO DEL PROBLEMA

El planteamiento del problema que deseo abordar en este Seminario de Título es que me he dado cuenta que las fuentes de cada desarrollo de software vienen a tener una importancia tan grande que el hecho de haberse creado un sistema de versionamiento de estas, viene a resolver justamente los problemas de seguridad y mantención de las aplicaciones, logrando establecer reglas y buenas practicas que nos permiten mantener históricamente las distintas intervenciones que se realizan a las fuentes y sus mejoras. Este sistema de versionamiento, el cual en la actualidad y siendo el más popular corresponde al sistema Git, nos permite la manipulación de las fuentes y su sincronización con las distintas plataformas del tipo repositorio que nos permiten llevar este versionamiento de la manera más ordenada y sistemática, pero existe un tema muy poco abordado por los desarrolladores (encargados de manipular las fuentes de un proyecto), el cual está relacionado a la cantidad de ramas que se crean por pruebas realizadas tanto en los ambientes previos como los que se actualizan en un ambiente productivo. Esto me llevó a pensar en lo importante que es el mantener estas fuentes lo más actualizadas posibles, de esta manera no se utiliza mucho espacio asignado por cada repositorio y a su vez se mantiene el código lo más actualizado y ordenado posible, otorgando claridad en las actualizaciones y no redundar en la información almacenada en los repositorios.

La temática especifica de mi proyecto, viene a solucionar un problema relacionado a la duplicidad innecesaria de las fuentes de cada proyecto, manteniendo la información necesaria solamente en los repositorios y controlando que la data no se duplique de manera innecesaria.

**¿Dónde ocurre esto?**.....esto ocurre en los proyectos de desarrollo de software que poseen fuentes compilables de cualquier lenguaje, que para argumentar en el contexto, ocurre con un cliente del giro bancario que mantiene un alto volumen de desarrollo y mantención de aplicaciones que utilizan repositorios de datos en plataformas como Github, Gitlab y Bitbucket.

**¿Qué ocurre?**...bueno, en estos momentos nos encontramos con un alto flujo de información duplicada en los repositorios y organizadas en ramas, las cuales muchas de estas no llegan a producción y quedan sin utilizar durante mucho tiempo, perjudicando el espacio asignado para cada proyecto. Esta duplicidad sin sentido se puede considerar como un olvido de los desarrolladores de eliminar estas fuentes que no se utilizan, por lo tanto es de gran ayuda para estos equipos el aportan con un sistema automatizado que elimine dicha información muerta.

**¿Como ocurre?**, creo que la respuesta es bastante clara, el olvido de los desarrolladores de mantener sus repositorios limpios permite que exista esta problemática que simplemente se puede corregir concientizando a los equipos a mantener limpios sus repositorios o simplemente limpiar de manera automática esta información.

**¿Y porque ocurre?**....bueno, usualmente ocurre porque los desarrolladores están enfocados al desarrollo de código y no de mantención de sus fuentes manipuladas, lo cual es un problema no menor que está ocurriendo en la actualidad.

### OBJETIVO GENERAL

Diseñar un software que nos permita informar/eliminar el nombre de las ramas desde los distintos repositorios creados en Github o cualquier otra plataforma de repositorios que se administre con Git para facilitar la administración de las fuentes, esto nos permitirá encontrar la condición de obsoleta de una rama, determinando cuáles ramas deben de ser eliminadas permanentemente y concretar el propósito de solucionar el desorden de fuentes que no se utilicen.

Por eso que el título del seminario se enfoca a la creación de soluciones informáticas desarrolladas en Shell para determinar cuáles ramas serían las candidatas a ser eliminadas después de la ejecución automatizada de esta Shell, como objetivo principal cumple con la expectativa necesaria de acuerdo a lo propuesto.

### OBJETIVOS ESPECÍFICOS

1.- Analizar la situación actual que estamos viviendo con las fuentes de cada proyecto, ya que en la actualidad no contamos con un control preciso del uso de las fuentes de cada software desarrollado y tampoco se cuenta con un proceso automático diario que nos permita visibilizar que ramas se encuentran con más de un mes sin ser manipuladas por los desarrolladores o se encuentran obsoletas definitivamente, esto se determina según la política interna de la corporación. Actualmente en las empresas que poseen fábricas de desarrollo se enfocan al propósito que los convoca, esto quiere decir que se preocupan del manejo del código pero no de las fuentes de cada proyecto que llegan a ser descargados en los servidores disponibles para las compilaciones de cada tecnología que se encuentra en desarrollo, estas tecnologías pueden ser Springboot, Angular, Java, .NET, entre otras tecnologías. Para el ambiente de desarrollo poseemos 3 servidores Linux con imagen RHEL (RedHat) 7 y cargados con software de compilación para las distintas tecnologías, lo que significa que se reciben una cantidad innumerable de fuentes a diario desde los distintos repositorios que nos servirán para compilar y desplegar de manera segura y en imágenes las tecnologías empaquetadas en contenedores, para dejarlas disponibles en el cluster de Kubernetes para este ambiente. Además poseemos un par de servidores más que atienden el ambiente de QA para que los equipos de Testing asignados puedan probar dichas aplicaciones (servidores que reciben las fuentes de cada proyecto que se despliegan a este ambiente). Y finalmente poseemos tres servidores más que reciben las fuentes compilables en donde se dejan disponibles los componentes que quedarán desplegados de cara al cliente. En resumen son ocho servidores en donde se replican las fuentes y que se deben mantener monitoreados para que estas fuentes no provoquen un colapso en el espacio asignado al Filesystem en donde quedan estas fuentes.

2.- Factibilidad técnica, operativa y económica sobre la implementación:

Factibilidad Técnica: como se requiere del monitoreo, obtención y eliminación de las ramas obsoletas por cada proyecto, se debe utilizar una Shell que nos permita rescatar la información (listar las ramas que cumplan con la condición de obsoleta) dejándola disponible para que se ejecute la eliminación de estas ramas y mantenga limpio el ambiente. Como el script de Shell será desarrollada en lenguaje de comandos Linux, la factibilidad técnica no posee ningún contratiempo, solo es necesario tener disponible un usuario aplicativo que tenga los permisos necesarios para la lectura y escritura tanto para el script de Shell como para los directorios que se definirán como áreas de trabajo (workspace) en donde las ramas se encuentren disponibles y se puedan listar; por otra parte necesitamos que todos los servidores tengan instalada la herramienta Git para ejecutar comandos Git para trabajar con las ramas descargadas desde Github, lo que para esto ya se encuentra disponible para su uso.

Factibilidad Operativa: como se menciona anteriormente, esta factibilidad operativa ya se encuentra disponible ya que poseemos las cuentas aplicativas para la ejecución del script de Shell en cada ambiente y/o servidor y además la herramienta Git instalada, por lo tanto no existe algún impedimento de realizar estas tareas diariamente. Por otra parte, como este software debe ejecutarse de manera automática, Linux posee un programa llamado Cron o Crontab que nos permite programar ejecuciones de Shell u otros ejecutables residentes en el servidor, programa que viene incluido en el paquete de Linux.

Factibilidad Económica: tanto la herramienta Git como el script de Shell no tienen costos asociados porque la herramienta Git no tiene costo alguno para su instalación y además ya se encuentra funcionando correctamente en cada server y el script de Shell desarrollado para este fin, se encuentra desarrollado con comandos nativos de Linux lo que se omite todo tipo de costo asociado a esta actividad.

Diseñar y desarrollar un programa para la eliminación segura y verificada de las ramas: crear un script de Shell que eliminará de manera segura las ramas obsoletas de cada proyecto y en cada servidor correspondiente a cualquier ambiente, para que a futuro no se presenten problemas de espacio en los servidores y mantener ordenadas las fuentes para ser utilizadas en el caso de algún proceso que se requiera realizar con estas fuentes, como por ejemplo rollback de alguna versión desplegada en QA o en Producción también, asegurando la continuidad del trabajo de los equipos de desarrollo involucrados en los proyectos.

Gestión de Excepciones: Desarrollar un manejo eficiente de excepciones para abordar situaciones imprevistas, como la preservación de ramas críticas, la prevención de la eliminación accidental de ramas activas y la gestión de posibles conflictos durante el proceso de eliminación.

Optimización del Espacio: crear funcionalidades que analicen y proporcionen estadísticas sobre el espacio ocupado por ramas obsoletas antes y después de la ejecución de la Shell. Esto permitirá a los usuarios comprender el impacto de la gestión de ramas en la optimización del espacio en el repositorio.

Conclusión: evaluar mediante pruebas concretas el buen funcionamiento de la eliminación de las ramas obsoletas mediante el programa desarrollado para cumplir con el propósito principal, la investigación y desarrollo de este script de Shell busca proporcionar a los desarrolladores una herramienta automatizada para gestionar ramas obsoletas en repositorios descargados desde Github. La automatización de tareas tediosas, la mejora de la eficiencia y la optimización del espacio contribuirán significativamente a mantener la salud y la calidad de los repositorios a lo largo del tiempo.

Este software de aplicación posee las características de listar las ramas obsoletas en base a un tiempo definido (identificando dichas ramas) y eliminar las ramas que se identifiquen como obsoletas de manera automatizada, considerando factores como su actividad (considerando el tiempo de inactividad y la antigüedad de los commits), incorporando validaciones posteriores para garantizar la integridad del repositorio.

Además, otra característica relevante que posee es la factibilidad técnica que se encuentra definida por ser desarrollado en el lenguaje Shell Script propio de la plataforma de Linux, que nos garantizará su funcionalidad y operatividad, lo que además no incurre en costos adicionales en su desarrollo e implementación.

Se ha seleccionado la Metodología de Modelo Iterativo, justificándose en los siguientes puntos:

\* La metodología del modelo iterativo permite entregar prototipos funcionales en etapas tempranas, lo que ayudará a determinar su funcionalidad eficiente, procurando que su enfoque técnico y operativo en un entorno real se transforme en un tema crucial.

\* El manejo de políticas de purga internas de cada corporación y la compatibilidad con herramientas de control de versiones (como Git) consideran este software de aplicación como de suma importancia al momento de abordar estas complejidades, permitiéndonos ajustar el diseño según las necesidades emergentes o retroalimentaciones observadas.

\* El Modelo Iterativo nos permite incorporar cambios de manera ágil a este Software de Aplicación, facilitando la implementación de nuevas funcionalidades que pueden estar relacionadas con su adaptabilidad a su entorno u otros requerimientos necesarios.

En conclusión, el modelo iterativo como metodología a aplicar, por su naturaleza incremental y flexibilidad lo convierte en la metodología ideal para mi proyecto, permitiendo abordar de manera eficaz la complejidad técnica y la necesidad de adaptarse a las demandas que puedan aparecer desde la organización.

El Modelo Iterativo sigue una serie de pasos que describen su funcionalidad, los cuales son los siguientes:

\* Planificación: Definir los objetivos específicos los cuales corresponden a determinar cuáles son las ramas obsoletas ya que no se cuenta con un sistema automático para visualizarlas y para posteriormente eliminarlas.

\* Diseño: Diseño inicial del Software de Aplicación en un diagrama de flujo o esquemas que nos sirvan de guía para definir el paso posterior de este modelo.

\* Desarrollo e Implementación: Construcción de las funcionalidades principales del Software de Aplicación, comenzando con pequeños módulos que puedan ser integrados y probados de manera independiente.

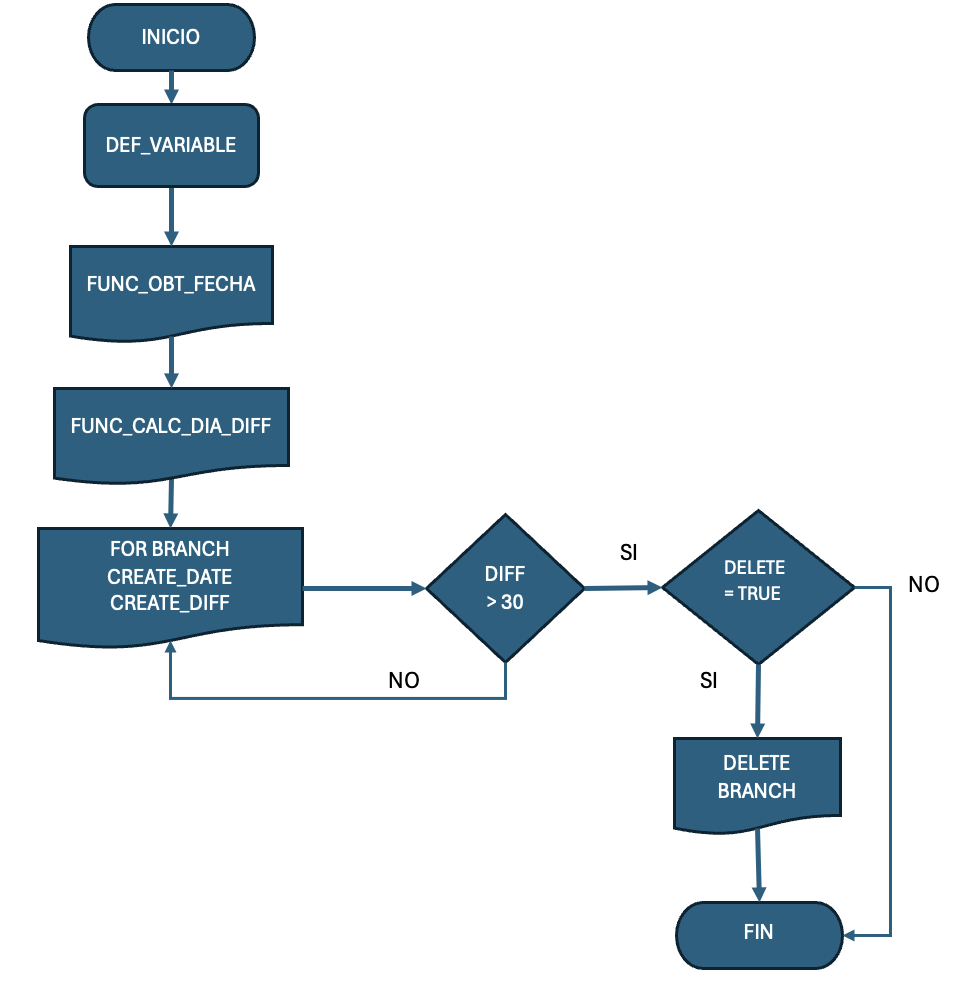
\* Pruebas y Validación: Se evalúan las funcionalidades desarrolladas para detectar errores de manera temprana y verificar si se cumplen con los objetivos propuestos.

\* Evaluación y Retroalimentación: Recopilación de información por parte del implantador del Software de Aplicación para obtener la retroalimentación necesaria para determinar si la funcionalidad requiere de mejoras o ajustes para las próximas iteraciones.

\* Revisión y Ajustes: En base a la evaluación y retroalimentación anteriormente mencionada, se actualizan los planes y diseños para ajustar prioridades que puedan aparecer según los requisitos planteados.

Diagrama de Flujo Shell de Eliminación

Como para nuestro caso presentado corresponde al desarrollo de un Software de Aplicación, se presenta a continuación el Diagrama de Flujo que representa la actividad de eliminación de ramas de manera automatizada, la cual se explica de la siguiente manera:



**Descripción del Diagrama:**

1.- Inicio: Comienza el proceso.

2.- DEF\_VARIABLE: Definición de las variables a utilizar.

3.- FUNC\_OBT\_FECHA: Se activa la función para obtener la fecha de creación de la rama.

4.- FUNC\_CAL\_DIA\_DIFF: Se activa función para el cálculo de días entre dos fechas.

5.- FOR BRANCH/CREATE\_DATE/CREATE\_DIFF: Ciclo FOR que recorre repositorios y crea variables para definir ramas a eliminar.

6.- DIFF>30: Condicional que filtra ramas con más de 30 días de vida.

7.- DELETE=TRUE: Condicional para eliminar ramas.

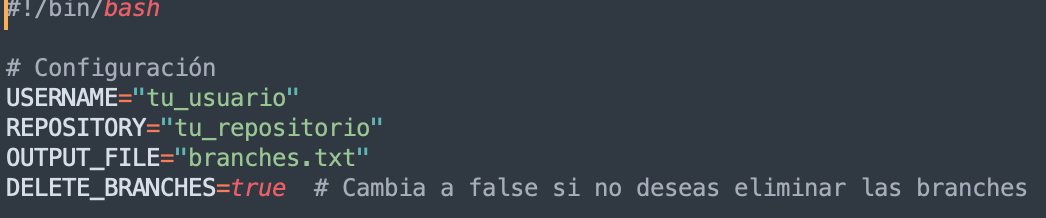
8.- DELETE BRANCH: Borrado de rama que cumple con la condición.

9.- FIN: Fin del proceso.

Figura 1. Diagrama de Flujo

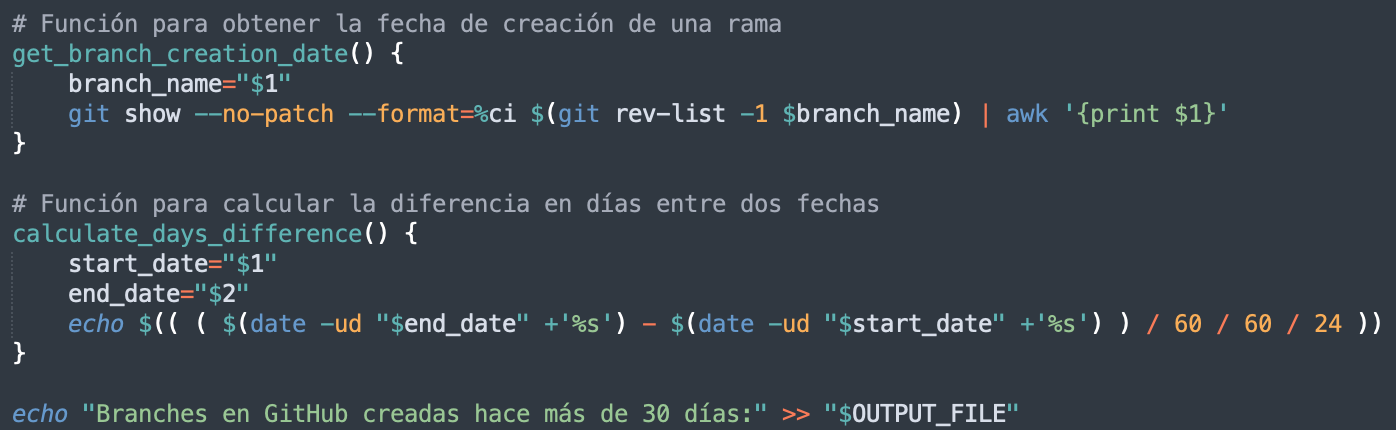
Esta Shell de eliminación genera un gran aporte al proceso de iteración que realizan las células que trabajan con sus desarrollos en distintos repositorios de Github (puede ser cualquier plataforma en donde se alojen las fuentes de algún desarrollo y que sean compatibles con la herramienta de versionamiento Git), ya que su ejecución mantiene activas (o vivas) sólo las ramas que se encuentren creadas dentro de un rango específico de días después de su creación. Ahora, la importancia que posee esta Shell de eliminación es crucial ya que es el Software de Aplicación que realizará la purga de ramas que no se utilicen o que se encuentren obsoletas, de acuerdo con los parámetros definidos en las políticas propias del sistema de administración de fuentes. La forma de cómo ocurre el flujo, se encuentra especificado al costado del diagrama, de todas maneras, se explica su funcionalidad a continuación:

Figura 2. Shell de eliminación (Primera parte)



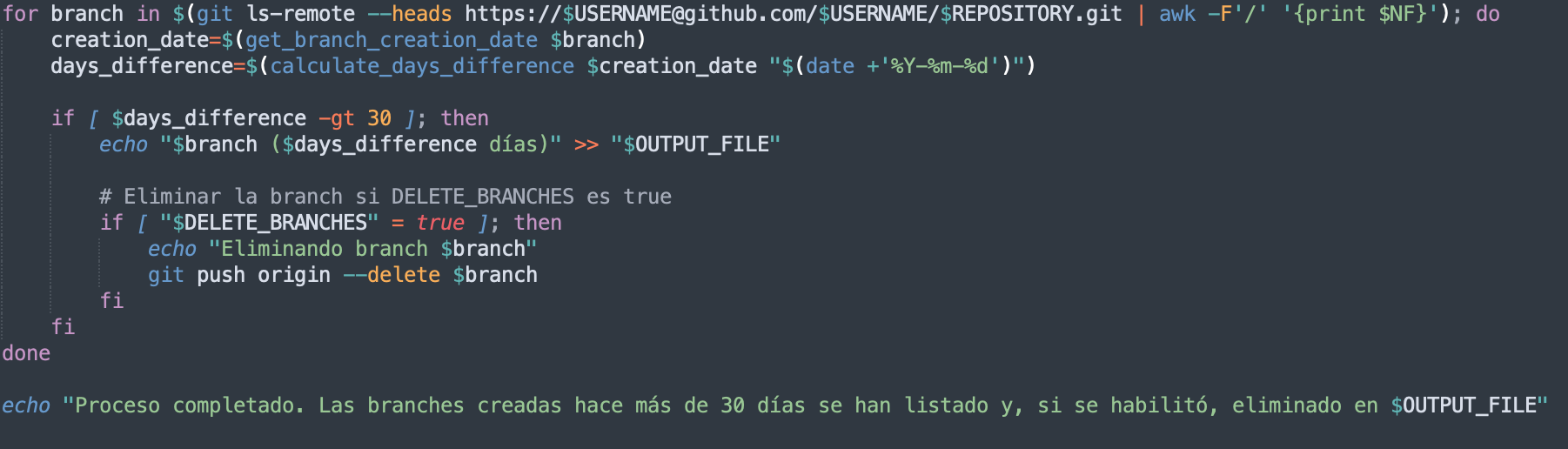
En la primera parte de nuestra Shell de eliminación, se definen algunas variables que debemos utilizar para la conexión al repositorio que vamos a utilizar de ejemplo, también acá definimos cual es el archivo de salida en donde se informarán de las ramas a eliminar y finalmente, se agrega la variable “DELETE\_BRANCHES=true” que nos permite ejecutar la acción de eliminar cuando se requiera.

Figura 3. Shell de eliminación (Segunda parte)



Posterior a las variables definidas, nos encontramos con la definición de 2 funciones que nos ayudarán a obtener las fechas de creación de cada rama y a calcular la diferencia en días entre dos fechas. Para la primera función, tenemos el comando “git show” que nos permite mostrar, de acuerdo a una lista confeccionada por el comando “git rev-list”, todas las ramas existentes con un filtro “awk” que solo muestra la fecha. La segunda función realiza el cálculo de días de vida que poseen las ramas, agregando un mensaje en el archivo de salida “OUTPUT\_FILE” e informando cuales ramas son las que corresponde eliminar (ya que cumplen con los días definidos según la política de purga implementada).

Figura 4. Shell de eliminación (Tercera parte).



Por último, se crea un ciclo FOR para recorrer el repositorio localmente, agregando nuevas variables en donde se implementan las funciones anteriormente creadas y se definen las condicionales necesarias para obtener las ramas con más de 30 días de vida y eliminándolas del repositorio, procediendo después con el comando “git push origin –delete $branch” que mantiene actualizado el repositorio en Github.com.

**Proceso de Codificación**

En el caso de mi proyecto, el proceso de codificación de mi Software de Aplicación tiene como objetivo gestionar ramas remotas en un repositorio de Github. Este script nos permite identificar ramas que fueron creadas hace más de 30 días y, opcionalmente, eliminarlas. La construcción del programa comienza con una sección de configuración, donde se definen variables clave como el nombre de usuario, el repositorio, el archivo de salida y la configuración para habilitar y deshabilitar la eliminación de ramas. A continuación, se implementan funciones específicas, las cuales una de ellas “get\_branch\_creation\_date”, la cual utiliza comandos Git y herramientas como AWK para obtener la fecha de creación de una rama específica. Otra función importante es “calculate\_days\_difference”, que calcula la diferencia en días entre dos fechas usando las capacidades de cálculo de fechas de Bash.

El cuerpo principal del script recorre las ramas remotas del repositorio mediante “git ls-remote” y filtra sus nombres con AWK. Para cada rama, se determina su fecha de creación (fecha de primer commit) y se compara con la fecha actual. Si la rama tiene más de 30 días de antigüedad, se registra la información en un archivo de texto definido por la variable “OUTPUT\_FILE”. Si la variable “DELETE\_BRANCHES” está configurada como “true”, se procede a eliminar la rama utilizando el comando “git push origin --delete”.

La estructura modular y el uso de utilidades estándar de bash aseguran que el script sea legible y eficiente. Además, el archivo de salida permite mantener un registro de las ramas procesadas, lo cual es útil para auditorías o revisiones futuras. En resumen, este script es una herramienta automatizada para mantener repositorios limpios y organizados, reduciendo el trabajo manual asociado con la gestión de ramas antiguas.

**Listado del Código**

- Configuración inicial y definición de funciones.

- Obtención de la lista de ramas remotas.

- Iteración sobre cada rama y cálculo de antigüedad.

- Registro de ramas en el archivo de salida.

- Eliminación de ramas según la configuración.

- Generación de un mensaje de finalización.

**Metodología de Verificación**

Pruebas Unitarias en Funciones: Cada función *get\_branch\_creation\_date* y *calculate\_days\_difference* se prueba de forma aislada para garantizar que produzca los resultados esperados:

Get\_branch\_creation\_date:

A screen shot of a computer program

Description automatically generated

Figura 5. Ejecución de función en donde extrae la fecha de creación.

Calculate\_days\_difference:

A computer code with colorful text

Description automatically generated with medium confidence

Figura 6. Ejecución de función en donde calcula los días de diferencia entre ramas.

(Ambas ejecuciones en modo Debug)

Pruebas Funcionales: Se verifican las tareas principales del script, como la identificación de ramas remotas, el cálculo de antigüedad y la eliminación de ramas. Estas pruebas se realizan en un entorno simulado, o sea utilizando un repositorio de prueba, para evitar impactos en entornos reales.

Figura 7. Repositorio de Pruebas.

A screenshot of a computer

Description automatically generated

Figura 8. Entorno de pruebas:

A screen shot of a computer

Description automatically generated

Pruebas Manuales de Integración: Se ejecuta el script completo en un entorno de prueba, observando que las ramas antiguas se detecten correctamente, se registren en el archivo de salida y, si corresponde, se eliminen.

Figura 9. Pruebas Manuales.

A screen shot of a computer

Description automatically generated

(Error desconocido que se encuentra en investigación)

**Introducción de Pruebas**

Las pruebas de un sistema, como las de un script de Bash, son fundamentales como para garantizar su funcionalidad, confiabilidad y seguridad, especialmente cuando interactúa con repositorios remotos en entornos críticos. Este script automatiza la gestión de ramas en Github, identificando y eliminando aquellas que han superado un tiempo de antigüedad específico. Sin embargo, dado que estas operaciones pueden tener un impacto irreversible, como la eliminación de ramas, es esencial verificar su correcto desempeño antes de implementarlo en entornos reales.

Las pruebas permiten detectar errores en las funciones principales, como el cálculo de antigüedad de ramas o la eliminación remota, evitando así comportamientos inesperados. Además, ayudan a validar que el script maneje adecuadamente situaciones adversas, como repositorios vacíos, problemas de conexión o configuraciones incorrectas. De este modo, las pruebas no solo aseguran la correcta ejecución del script, sino que también protegen la integridad del repositorio.

**Alcance de las pruebas.**

Validación Funcional: Garantizar que el script identifique correctamente las ramas remotas creadas hace más de 30 días (caso supuesto). También verificar que las ramas identificadas se registren en el archivo “branches.txt” y posteriormente, confirmar que las ramas se eliminen del repositorio remoto solo cuando la configuración lo permita (DELETE\_BRANCHES=true).

Robustez y Resiliencia: Evaluar el comportamiento del script frente a condiciones atípicas, como por ejemplo cuando los repositorios no posean ramas, fallas en la conexión y permisos insuficientes para realizar cambios en el repositorio.

Impacto: Probar que el script no afecta ramas no incluidas en los criterios de eliminación.

**Tipos de Pruebas**

Pruebas Unitarias: Validar el correcto funcionamiento de las funciones específicas, como “get\_branch\_creation\_date”(verificar que extraiga correctamente la fecha de creación de una rama o su primer commit) y “calculate\_days\_difference” (confirmar que calcula correctamente la diferencia de días entre dos fechas), probando con diferentes fechas.

Pruebas Funcionales: Comprobar que el sistema cumple con los requisitos funcionales definidos, como la identificación de las ramas, registro adecuado en el archivo de salida y ejecución correcta de la eliminación.

**Estrategias de Pruebas**

Entorno Controlado: Configurar un repositorio de pruebas en Github para la ejecución del script y utilizar ramas ficticias con diferentes fechas de creación para simular diversos escenarios.

Automatización de Pruebas: Incluir scripts adicionales que simulen la entrada esperada y validen automáticamente los resultados esperados.

Pruebas Graduales: Comenzar con pruebas unitarias, continuando con las pruebas funcionales para probar posteriormente el script en entornos reales.

**Fundamentación**

Minimización de Riesgos: Estas pruebas aseguran que el script no elimine ramas de forma incorrecta, protegiendo la integridad del repositorio.

Calidad del Producto: Entregar un sistema confiable realizando varias pruebas, de esta manera se reducen los riesgos de errores en la producción.

Adaptabilidad: Esta estrategia permite detectar problemas tanto en entornos simulados como reales, asegurando que el script pueda manejar diferentes condiciones de uso.

Eficiencia: Mediante la automatización y el diseño modular del script, las pruebas son rápidas y repetibles, reduciendo su intervención manual.

**Criterios de Salida**

Los criterios de salida están diseñados para garantizar que el script cumpla con su propósito de manera eficiente y segura. Al validar su ejecución, registro y eliminación de ramas, se asegura que el sistema entregue resultados precisos y confiables. El manejo de errores robusto es esencial para evitar interrupciones imprevistas, mientras que la generación de logs y archivos de salida proporciona transparencia y trazabilidad, elementos claves para sistemas automatizados en entornos reales.

Pruebas del Tipo Dinámica y Contra Especificación

Dinámica: Se elije del tipo Dinámica porque implica la ejecución del código en un entorno real o simulado, verificando el comportamiento del script durante su operación, así nos aseguramos de que las funciones y procesos definidos cumplan con los requisitos esperados. Por otra parte, nos permite observar cómo interactúan las diferentes partes del script, como el manejo de errores y su comportamiento ante situaciones específicas.

Contra Especificación: Su objetivo el validar si el script cumple con los requisitos funcionales establecidos, incluyendo la identificación de ramas, la generación del archivo de salida con la información correcta, la eliminación de las ramas según su configuración y el manejo adecuado de los errores.

Su justificación se relaciona por la naturaleza del script ya que realiza acciones dinámicas como la conexión de un repositorio y la ejecución de comandos de eliminación. Además, el script interactúa con un repositorio Git por lo tanto, se debe asegurar de que cumpla con las especificaciones, evitando impactos no deseados. Por último, estas pruebas nos aseguran que todas las etapas del script (obtención de ramas, cálculo de antigüedad y registro de salida) se ejecuten de manera correcta y coordinada.

**Resultados de la ejecución de pruebas**

(Pendiente por error no identificado aún)

**Evidencia de los Resultados Obtenidos en la Investigación**

Los resultados obtenidos en la investigación se generan al momento de ser ejecutada la shell, los cuales se presentan de la siguiente manera:

Registro de Pruebas: Detalle del escenario probado, los pasos realizados y los resultados obtenidos como, por ejemplo:

\* Escenario: Repositorio con ramas creadas hace más de 30 días.

\* Resultado esperado: Las ramas antiguas se listan en el archivo “branches.txt” y se eliminan del repositorio remoto.

\* Resultado obtenido: Captura del archivo “branches.txt” y del repositorio mostrando el antes y después de la ejecución de la Shell.

Archivo de Salida: Incluye el archivo generado por el script como evidencia del análisis realizado, conteniendo el nombre de las ramas detectadas y el tiempo en días desde su creación.

Registro de Consola: Presentando capturas o transcripciones de la salida en el terminal, agregando proceso de conexión al repositorio, detección de ramas, mensajes de error y confirmando la eliminación de las ramas.

**Tabla Comparativa**

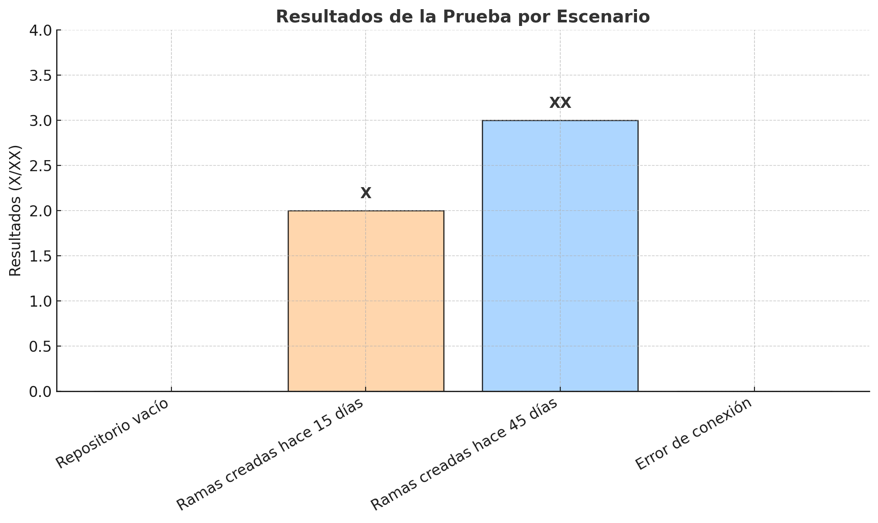
Se creará una tabla comparativa en donde se registra el comportamiento del script en diferentes pruebas, como por ejemplo la siguiente:

|  |  |  |  |
| --- | --- | --- | --- |
| Escenario | Entrada (Datos) | Resultado Esperado | Resultado Obtenido |
| Repositorio vacío | Sin ramas | Archivo branches.txt vacío | Archivo vacío generado correctamente |
| Ramas creadas hace 15 días | X ramas | Ninguna rama listada | Sin ramas listadas |
| Ramas creadas hace 45 días | X ramas | X ramas listadas | X ramas eliminadas |
| Error de conexión | Repositorio inaccesible | Mensaje de error en la consola | Display de error por consola |

**Resumen de los Resultados**

Para esta investigación, se debe presentar un resumen un resumen escrito que destaque el porcentaje de éxito de la identificación y eliminación de ramas, errores encontrados y como se resolvieron, resultados esperados versus los resultados obtenidos y una conclusión en donde especifique si el script cumple con las especificaciones y dar a conocer si puede ser implementado en ambientes reales.

Figura 10. GRÁFICO



El gráfico de barras muestra los resultados obtenidos en las pruebas realizadas en los diferentes escenarios del script:

\* Repositorio vacío: No se identificaron ramas (valor 0).

\* Ramas creadas hace 15 días: Se identificaron 2 ramas, representadas como “X”.

\* Ramas creadas hace 45 días: Se identificaron 3 ramas, representadas como “XX”.

Error de conexión: No se identificaron ni procesaron ramas (valor 0).

El eje de X muestra los escenarios probados, mientras que el eje Y indica los resultados numéricos, representados visualmente como “X” (2) y “XX” (3) para facilitar la interpretación. Esto permite identificar rápidamente los casos exitosos y los resultados esperados en cada prueba.

Conclusiones

El tema de estudio de esta investigación se centra en el desarrollo y la validación de un script en Bash diseñado para gestionar ramas de un repositorio Git, específicamente identificando aquellas que fueron creadas hace 15 y 45 días, listándolas y, opcionalmente, eliminándolas. La importancia de este tema radica en la necesidad de mantener repositorios organizados y libres de ramas obsoletas, un aspecto crítico en entornos colaborativos donde el control de versiones es esencial para la eficiencia y calidad del desarrollo de software.

El objetivo general de esta investigación fue diseñar y validar una herramienta automatizada que permita optimizar la gestión de ramas en un repositorio Git. Esto se enlaza con los objetivos específicos, que incluyen el diseño del script, la implementación de pruebas para garantizar su funcionalidad y la evaluación de su desempeño en diversos escenarios. Este enfoque permite garantizar que el sistema cumpla con las expectativas funcionales, sea fácil de usar y minimice riesgos durante su operación.

A lo largo de la investigación, se lograron los siguientes puntos principales:

1. Se diseñó un script eficiente que integra funciones clave como el cálculo de la antigüedad de las ramas y su eliminación remota controlada.

2. Se ejecutaron pruebas dinámicas y contra especificación para validar el comportamiento del script en distintos escenarios, como repositorios vacíos, ramas activas y problemas de conexión.

3. Se documentaron resultados claros a través de archivos de salida, registros en consola y análisis visuales que evidencian la efectividad del sistema.

Logro de los Objetivos Específicos

1. Diseño del Script:

Este objetivo se logró completamente al desarrollar un script que integra las funciones necesarias para cumplir con los requisitos establecidos. Las funciones incluyen la identificación de ramas por su antigüedad, el manejo de errores y la generación de registros claros.

2. Implementación de Pruebas:

Se cumplió exitosamente mediante pruebas dinámicas en repositorios reales y simulados. Las pruebas confirmaron que el script identifica correctamente las ramas creadas hace 15 y 45 días y elimina aquellas que cumplen con los criterios, dependiendo de la configuración establecida.

3. Evaluación del Desempeño:

Se realizaron análisis detallados sobre el desempeño del script en distintos escenarios, lo que permitió comprobar su capacidad para manejar repositorios de distintas dimensiones, errores de conexión y configuraciones personalizadas. Los resultados fueron consistentes con las expectativas.

4. Documentación de Resultados:

Se documentaron los resultados mediante tablas, gráficos y archivos generados, lo que proporciona evidencia clara de que el script opera según lo especificado.

Viabilidad de la Propuesta

La propuesta presentada es viable porque el script cumple con los objetivos planteados y demostrar ser efectivo durante las pruebas. Su diseño robusto, capacidad de personalización y manejo de errores lo hacen adecuado para su uso en entornos reales. Además, su naturaleza automatizada reduce el esfuerzo manual y el riesgo de errores humanos en la gestión de ramas. Por lo tanto, se concluye que la herramienta puede implementarse para mejorar la eficiencia y organización en proyectos que utilizan sistemas de control de versiones como Git.

Recomendaciones

1. ¿Por qué debe hacerse?

Continuar con esta investigación es crucial porque la gestión eficiente de ramas en repositorios Git es un aspecto clave en proyectos de desarrollo de software, especialmente en equipos colaborativos donde la acumulación de ramas obsoletas puede causar desorganización y dificultades operativas. Ampliar esta investigación permitirá optimizar aún más el proceso, mejorar la escalabilidad del script y garantizar que se adapte a entornos más complejos y específicos.

2. ¿Qué cosa se debe hacer?

\* Ampliar las capacidades del script: Incluir nuevas funcionalidades como:

- Identificación de ramas según criterios adicionales (por ejemplo, ramas no fusionadas).

- Integración con servicios como Jenkins o GitHub Actions para automatizar su ejecución periódica.

- Realizar pruebas en entornos reales y de mayor escala\*\*: Ejecutar el script en repositorios grandes con cientos o miles de ramas para validar su rendimiento y robustez.

\* Documentar casos de uso específicos: Crear guías prácticas para su implementación en diferentes contextos empresariales o académicos.

3. ¿A quién beneficiará?

Los beneficiarios directos serán:

- Desarrolladores y equipos de software: Les permitirá ahorrar tiempo en la limpieza de repositorios, reduciendo errores humanos y aumentando la eficiencia.

- Empresas tecnológicas: Mejorarán la calidad de sus procesos de desarrollo al mantener repositorios más organizados y productivos.

- Docentes e investigadores: Servirá como base para proyectos académicos relacionados con automatización y control de versiones.

4. ¿Quién lo hará?

La investigación puede ser continuada por:

- Equipos de desarrollo interno en empresas tecnológicas: Que utilicen Git y necesiten soluciones adaptadas a sus procesos específicos.

- Estudiantes y académicos: Como parte de proyectos de investigación, tesis o prácticas profesionales.

- Comunidad de código abierto: Que pueda colaborar para mejorar y expandir las funcionalidades del script.

5. ¿Dónde se hará?

- Entornos empresariales: Empresas que manejen grandes repositorios y busquen soluciones para mantenerlos organizados.

- Instituciones educativas: En universidades o institutos tecnológicos que deseen enseñar buenas prácticas en el uso de control de versiones.

- Plataformas en línea: Como GitHub, GitLab o Bitbucket, donde se puede compartir, probar y colaborar en el desarrollo del script.

Estas recomendaciones tienen como objetivo fomentar la mejora continua del proyecto, asegurando que su impacto sea significativo y sostenible en el ámbito del desarrollo de software.

RESUMEN

Automatización de la Gestión de Ramas Obsoletas en Repositorios Git

La presente investigación aborda la problemática de la acumulación de ramas obsoletas en repositorios Git, lo cual genera desorganización y dificulta la gestión en proyectos de desarrollo de software, especialmente en equipos colaborativos. La falta de herramientas automatizadas para identificar y eliminar ramas antiguas puede ocasionar demoras en el desarrollo y errores en la integración de cambios.

El estudio utiliza una investigación aplicada, de tipo experimental, para desarrollar y validar un script en Bash que identifica ramas creadas hace 15 y 45 días, las lista en un archivo de salida y, si se configura, las elimina automáticamente del repositorio remoto. Se realizan pruebas dinámicas y contra especificación en escenarios controlados y en repositorios simulados para garantizar la funcionalidad del sistema.

Los resultados confirman que el script cumple con los objetivos establecidos, identificando y gestionando eficazmente las ramas obsoletas en distintos escenarios. El archivo de salida (`branches.txt`) registra las ramas detectadas y su antigüedad, lo que permite evaluar su estado antes de tomar acciones. Las pruebas muestran que el sistema maneja repositorios de distintas escalas, ofrece flexibilidad en su configuración y opera sin interrupciones ante errores predefinidos, como problemas de conexión.

En conclusión, la solución presentada es viable y efectiva para optimizar la gestión de ramas en repositorios Git, mejorando la organización y reduciendo el tiempo invertido en tareas manuales. Se recomienda continuar con su mejora y validación en entornos más complejos y en proyectos reales, para potenciar su aplicabilidad en la industria tecnológica.

### hipotesis

Dada la problemática asociada a la acumulación de ramas obsoletas y que utilizan demasiado espacio innecesariamente en repositorios de GitHub, se plantea la hipótesis de que la implementación de un script de Shell para la gestión automatizada de estas ramas redundantes resultará en una optimización significativa del espacio, una mejora en la calidad del código y una mayor eficiencia en el mantenimiento de proyectos de software.

Se supone que la identificación automatizada de ramas obsoletas y de gran envergadura, su eliminación segura y la optimización del espacio contribuirán a la reducción de la complejidad del repositorio y, por ende, mejorarán la agilidad en el desarrollo. En resumen, el desarrollo del script de Shell viene a resolver la falta de herramientas especializadas para la gestión y eliminación de ramas obsoletas en repositorios administrados con la herramienta Git , generando ineficiencias en el desarrollo y mantenimiento en el código, lo que resulta en una mayor desorganización y dificultad para colaborar de manera efectiva en plataformas como Github.

## JUSTIFICACIÓN DE LA INVESTIGACIÓN

Esta investigación se justifica a razón de contar con un control más detallado de las fuentes que se descargan en los servidores de compilación al momento de cada despliegue que se realiza por cada iteración desde los repositorios de los proyectos de desarrollo de las células existentes en la organización al cual me encuentro dando soporte técnico.

# MARCO TEÓRICO

Fuentes Primarias:

Git y GitHub como Herramientas de Control de Versiones:

Git es un sistema de control de versiones distribuido ampliamente utilizado, y GitHub es una plataforma de alojamiento de proyectos que utiliza Git. La comprensión de los principios fundamentales de Git y GitHub es esencial para abordar la problemática de las ramas obsoletas. De acá podemos utilizar la documentación disponible en el sitio oficial <https://git-scm.com/doc> *(Chacon, Scott [2025], Git – fast version control)* y por el lado de Github, podemos considerar la siguiente fuente: <https://docs.github.com/en> *(Github Community [2025], Github Docs)*.

Buenas Prácticas en la Gestión de Ramas:

La literatura sobre buenas prácticas en el desarrollo de software destaca la importancia de una gestión eficiente de ramas para mantener la estabilidad del código y facilitar la colaboración entre equipos. Explorar conceptos como "GitFlow" y estrategias de ramas ayudará a contextualizar la investigación. Fuente primaria <https://docs.github.com/en> *(Github Community [2025], Github Docs).*

Efectos de la Obsolescencia del Código en Proyectos de Software:

La investigación sobre los impactos de mantener código obsoleto en proyectos de software proporciona perspectivas distintas sobre cómo la acumulación de ramas obsoletas puede afectar negativamente la productividad, la mantención y la eficiencia en el desarrollo de software. Fuente Primaria: <https://docs.github.com/en> *(Github Community [2025], Github Docs).*

Fuente Secundaria:

Automatización en la Gestión de Repositorios de Código:

La literatura sobre automatización en la gestión de repositorios de código destaca la importancia de utilizar scripts y herramientas automatizadas para realizar tareas repetitivas. La eficacia de la automatización en la gestión de ramas de un repositorio puede ser un concepto clave para el desarrollo de la Shell en Batch. Para este caso, podemos considerar sitios que se encuentran en apoyo a la documentación oficial pero que tienen una mirada más de usuario que del propietario, el cual cuenta con un grado de experiencia y posee sensibilidad al conocer la teoría de la información, un ejemplo es esta: <https://raiolanetworks.com/blog/guia-github/>

Fuente Terciaria:

Estándares de Desarrollo Colaborativo:

Las fuentes terciarias, como manuales de buenas prácticas en desarrollo colaborativo, pueden proporcionar directrices y recomendaciones para la gestión efectiva de ramas en entornos de colaboración. La comprensión de estándares reconocidos contribuirá a contextualizar la investigación en un marco más amplio de desarrollo colaborativo.

Términos Básicos:

Los términos básicos para esta investigación sobre la creación de un script de Shell para administrar ramas obsoletas y de gran tamaño en repositorios de GitHub podrían ser los siguientes:

Script de Shell:

Un conjunto de comandos y scripts automatizados que se ejecutan en una línea de comandos de manera secuencial para realizar tareas específicas en un entorno de sistema operativo Linux o Windows. En este contexto, se refiere a la herramienta diseñada para gestionar ramas obsoletas en repositorios de GitHub.

Ramas Obsoletas:

Ramas en un repositorio de control de versiones que han dejado de ser necesarias o relevantes para el desarrollo activo del proyecto. Estas ramas pueden afectar negativamente la eficiencia y ocupar espacio innecesario en el repositorio.

Ramas de gran tamaño:

Ramas en un repositorio que por su tamaño deben mantenerse monitoreadas ya que manejan una gran cantidad de información que cada vez que es necesario realizar alguna actualización a la rama “Main”, se debe generar una copia de ésta con el propósito de actualizarla, probarla y mezclarla para su actualización.

GitHub:

Una plataforma de alojamiento de proyectos de desarrollo de software que utiliza el sistema de control de versiones Git. Es ampliamente utilizada para la colaboración, seguimiento de cambios y gestión de proyectos mediante repositorios compartidos.

Gestión de Espacio en Repositorios:

Estrategias y prácticas relacionadas con la optimización del espacio en repositorios de control de versiones. En este contexto, se refiere a la necesidad de eliminar ramas obsoletas y de gran tamaño para mejorar la eficiencia y reducir el espacio ocupado.

Automatización en Desarrollo de Software:

El proceso de realizar tareas repetitivas de manera automática mediante scripts y herramientas. En esta investigación, se relaciona con la automatización de la identificación y eliminación de ramas obsoletas en GitHub mediante una script de Shell.

Estos términos básicos son esenciales para comprender los conceptos centrales de la investigación y establecer un marco conceptual sólido.

## MARCO METODOLÓGICO

De acuerdo a mi criterio, el tipo de investigación que debe realizarse para mi investigación es de tipo Cuantitativo, ya que serán cifras las que se deben analizar en base a criterios establecidos por fechas, medición de datos, cantidades de archivos, entre otros factores. Para esto se define de la siguiente manera:

1. Definición clara de objetivos:

Establecer Objetivos Cuantificables:

Define de manera clara y específica los objetivos cuantitativos que deseas lograr con la investigación. Esto podría incluir reducción porcentual de espacio ocupado, tiempo ahorrado en comparación con métodos manuales, o cualquier otra métrica cuantificable.

2. Desarrollo de Hipótesis:

Formulación de Hipótesis Cuantitativas:

Basar la investigación en hipótesis cuantificables que puedan ser probadas con datos numéricos. Por ejemplo, una hipótesis podría ser que la implementación de los script de Shell reducirá en un 30% el espacio ocupado por ramas obsoletas.

3. Diseño del Estudio:

Selección de Variables y Métricas:

Identificar las variables clave que impactarán en tu investigación y define las métricas cuantitativas asociadas. Por ejemplo, podrías medir la frecuencia de uso del script de Shell, la reducción de espacio en megabytes, y el tiempo promedio de ejecución.

Diseño de Experimentos:

Planificar cómo se llevará a cabo el estudio. Define los grupos de prueba y control, si es aplicable, y establece un protocolo para recopilar datos de manera consistente.

4. Recopilación de Datos:

Utilización de Herramientas de Registro de Datos:

Implementar herramientas que te permitan registrar de manera precisa los datos relevantes para tus métricas cuantitativas. Se pueden incluir registros de ejecución de la Shell, resultados de comandos, y mediciones de espacio en repositorios.

Análisis de Datos de Repositorios de GitHub:

Utilizar herramientas de análisis de datos para extraer información cuantitativa de los repositorios de GitHub. Esto puede incluir scripts personalizados o el uso de APIs para recopilar datos de forma automatizada.

5. Análisis Estadístico:

Aplicación de Métodos Estadísticos:

Utilizar métodos estadísticos apropiados para analizar los datos recopilados. Se pueden emplear pruebas de hipótesis, análisis de varianza (ANOVA), o cualquier técnica estadística relevante para tus objetivos.

Visualización de Datos:

Crea gráficos y visualizaciones que ayuden a interpretar los resultados de manera clara. Esto puede incluir gráficos de barras, diagramas de dispersión, y cualquier otra representación visual que facilite la comprensión de los datos cuantitativos.

6. Interpretación de Resultados:

Relaciona Resultados con Objetivos:

Evaluar si los resultados obtenidos cumplen con los objetivos establecidos. Comprender la relevancia práctica de las cifras cuantitativas y cómo se traducen en beneficios tangibles.

7. Conclusiones y Recomendaciones:

Resumen de Hallazgos:

Presentar de manera clara los hallazgos cuantitativos en tus conclusiones. Responde a tus hipótesis y objetivos de manera sistemática.

Recomendaciones para la Implementación:

Basándose en los resultados, se debe proporcionar recomendaciones específicas para la implementación y mejora continua del script de Shell.

8. Redacción del Informe Final:

Estructura del Informe:

Organizar el informe de manera lógica, incluyendo una introducción, revisión de literatura cuantitativa, metodología, resultados, discusión, conclusiones y recomendaciones.

Presentación Visual:

Utilizar tablas, gráficos y visualizaciones para respaldar los resultados y hacer que la información sea accesible y comprensible.

9. Revisión y Validación (feedback):

Revisión por Pares:

Someter la investigación a revisión por pares para obtener retroalimentación adicional y validar la solidez del enfoque cuantitativo.

Este enfoque cuantitativo permitirá obtener resultados numéricos que respalden la efectividad del script de Shell en la gestión de ramas obsoletas, proporcionando una base objetiva para la toma de decisiones y mejoras continuas.

Además, para argumentar aún más este tipo de investigación agregaremos más contenido sobre el marco metodológico que se utilizará en este trabajo de investigación:

La técnica de investigación seleccionada corresponde a la de “Diseño No Experimental”, ya que se deben observar los datos que se van actualizando a medida que se van utilizando dentro de los repositorios de Github. Recordemos que el proyecto se enfoca en la creación de una Shell automatizada que permita determinar cuáles ramas se encuentran sin uso u obsoletas para ser eliminada desde los distintos repositorios que posee una corporación, que para el caso hipotético de este Seminario de Título, se debe trabajar en una Shell que se adecúe a cualquier corporación, ya que existen varios puntos a considerar sobre todo en temas relacionados a la o las cuentas aplicativas a utilizar, con los privilegios correspondientes. La técnica a utilizar se justifica de la siguiente manera:

Diseño No Experimental:

A. Estudio Observacional:

Observa y registra fenómenos naturales sin intervenir en ellos. Se utiliza cuando no es ético o práctico realizar experimentos controlados. Para este caso no puedo manipular los repositorios, ya que están siendo administrados por células y equipos de desarrolladores que se encuentran constantemente en movimiento con las piezas, por lo tanto, solo podemos observar los datos a investigar.

B. Estudio Descriptivo:

Describe las características de un fenómeno sin manipular variables independientes. Se centra en proporcionar una visión detallada del estado actual de los repositorios rescatando cada detalle importante, lo cual se puede transformar en una variable a considerar en la eliminación de las ramas.

C. Estudio de Casos:

Se enfoca en un caso o un grupo pequeño de casos para comprender en profundidad la actividad del repositorio. Puede proporcionar información detallada y contextos específicos. Esto nos puede dar una idea del comportamiento en total de los repositorios a modo hipotético y definir las reglas del juego, esto quiere decir que podremos conocer inicialmente las directrices que nos permitirán saber que debemos considerar al momento de eliminar una rama, puede ser la inactividad y la obsolescencia que para estos casos se considera el uso actual del repo (si los componentes alojados en el repositorio se encuentran utilizados en dicho ambiente).

D. Encuesta:

Recopilación de datos a través de cuestionarios o entrevistas estructuradas. Permite explorar actitudes, opiniones o comportamientos en una muestra representativa. Esta entrevista se hace primordial al momento de no tener claro cuáles son las condiciones de trabajo de cada corporación, como por ejemplo reglas que definan el tiempo de duración de una rama en un repositorio, así con la encuesta podemos concientizar a los usuarios a que respeten la política de purga definida para todos los repositorios.

Por otra parte, el instrumento de recolección de datos a utilizar es el siguiente:

Observación:

Una observación es un registro sistemático y objetivo de comportamientos y eventos que se encuentran dentro de una observación, y esto es adecuado cuando se necesita observar comportamientos reales en contextos específicos, como es el caso de los repositorios y ramas de Github o cualquier otra plataforma de repositorio de fuentes.

Observación como Técnica de Recopilación de Datos:

La observación como técnica de recopilación de datos implica el registro sistemático y objetivo de comportamientos, eventos o fenómenos en un entorno específico. Este enfoque proporciona información directa y no filtrada sobre cómo las personas se comportan o interactúan en situaciones reales. La observación puede ser participante, donde el investigador está directamente involucrado en la actividad, que es nuestro caso.

Aunque el investigador puede influir en cierta medida, la observación tiende a ser menos propensa a sesgos subjetivos que las respuestas auto informadas en encuestas o entrevistas, por lo tanto se obtiene información de manera natural de los datos.

La observación permite recopilar detalles contextuales importantes que pueden perderse en otras formas de recopilación de datos.

Ejemplo: Observación en Ramas de Repositorios en GitHub:

Implementación:

1. Observación Directa:

Observar directamente las actividades en GitHub, como la creación de nuevas ramas, fusiones (merges), y el flujo de trabajo en general. El beneficio que obtenemos acá es la captación de la información real sobre cómo los desarrolladores interactúan con las ramas, identificando patrones de colaboración y prácticas comunes.

2. Registro de Actividades:

Mantener un registro detallado de las actividades relacionadas con las ramas, como cambios realizados (commits), comentarios en solicitudes de extracción y resolución de conflictos (por si se presentan), lo cual nos entrega un claro beneficio en relación a la información valiosa sobre las decisiones que se tomaron durante el desarrollo y como interactúan los datos entre las ramas.

3. Seguimiento del Tiempo de Vida de las Ramas:

Observar cuánto tiempo permanecen activas las ramas antes de ser fusionadas con la rama master (main en el caso de Github) para poder ser eliminadas. Esto nos podría revelar tendencias sobre la duración típica de las ramas identificando posibles cuellos de botella para generar mejoras.

Los objetivos principales son:

- Identificar patrones de colaboración en el uso de ramas.

- Mejorar la eficiencia en la gestión de ramas.

- Optimizar el flujo de trabajo del equipo.

Definición de la población y muestra para el caso presentado:

Para este caso de recopilación de datos a través de la observación, la población y la muestra la debemos rescatar desde Github, ya que esta plataforma es en donde residen las ramas a estudiar (población), esto quiere decir que en primera instancia debemos ser usuario de Github y poseer los permisos adecuados para realizar la observación lo más profundamente, logrando llegar a la totalidad de los repositorios de la corporación.

Población: Totalidad de ramas creadas en todos los repositorios correspondiente a la organización.

Muestra: Una parte de las ramas, las cuales y de acuerdo a los hitos rescatados al principio de la investigación, serán las candidatas a ser analizadas para diseñar una política de purga y su eliminación definitiva del repositorio.

# ANÁLISIS DE LOS RESULTADOS

# TABLA DE DATOS PROCESADOS DEL INSTRUMENTO ELABORADO

Para crearnos una tabla de datos con la información relevante a nuestro proyecto, debemos considerar las siguientes variables:

1.- Número de ramas: Debemos saber cuántas ramas tenemos que auditar independiente de la cantidad de repositorios que existen en nuestra corporación. Al menos tener una idea de la cantidad de valores que debemos conocer para hacernos una idea del total a estudiar.

2.- Edad de las ramas: Siempre es necesario considerar el tiempo de vida que tiene cada rama, esto se necesita para declarar una política de purga considerando un tiempo de obsolescencia por cada rama, de esta forma controlaremos la cantidad de datos por repositorio que deban estar activos. Esto se realiza con un cálculo simple, identificando la fecha de creación de la rama hasta la fecha de su revisión.

3.- Número de commits por rama: Rescatar la información de cantidad de commits por rama, entregándonos una idea de la actividad y el nivel de desarrollo en cada una.

4.- Tamaño de los commits: Analizar el tamaño (cantidad de líneas de código) de los commits realizados en cada rama, indicándonos la cantidad de trabajo realizado en cada una.

5.- Frecuencia de actualización: Determinar la frecuencia entre cada commits o la cantidad de commits en un período de tiempo, indicándonos las ramas que más se actualizan o menos se actualizan.

6.- Número de colaboradores por rama: Contar la cantidad de colaboradores que han contribuido por cada rama , indicándonos la diversidad y el nivel de interés en el desarrollo de cada una.

7.- Actividad reciente: Analizar la actividad reciente de cada rama como por ejemplo la cantidad de commits realizados en un periodo de tiempo ayudándonos a identificar las ramas más activas.

Para este caso, se considerarán 3 variables importantes para nuestro estudio, las cuales son el número de ramas, la edad de las ramas y el tamaño de los commits.

Acá se les entrega un detalle de los datos más relevantes a medir presentado en una tabla:

(se trabajarán datos ficticios ya que no tenemos un ejemplo con las condiciones necesarias de cantidad de ramas con diferentes fechas de creación)

|  |  |  |  |
| --- | --- | --- | --- |
| Repositorio | Número de Ramas | Edad de las Ramas (días) | Tamaño de los Commits Total (MB) |
| A | 10 | 35 | 12 |
| B | 23 | 37 | 14 |
| C | 6 | 54 | 26 |
| D | 45 | 42 | 22 |
| E | 6 | 18 | 6 |
| F | 24 | 23 | 14 |
| G | 8 | 32 | 57 |
| H | 17 | 48 | 25 |

# PROCESAMIENTO DE DATOS ESTADISTICOS APLICADO

Tabla de datos.

A table with numbers and text

Description automatically generated

Se realizan los siguientes procedimientos estadísticos de acuerdo a los datos presentados, lo que nos permitirá tener una idea de lo que representa cada ítem seleccionado. Por ejemplo, para el Número de Ramas la media representa un promedio de 17,3 ramas por los repositorios, para la Edad de las Ramas tenemos un promedio de 36 días y para el Tamaño de los Commits tenemos un promedio de 22 MB, permitiéndonos analizar cuál sería el estándar a establecer para realizar nuestras políticas de purga y proceder a modificar las Shell de eliminación.

Media y Mediana.

A table with numbers and text

Description automatically generated

# para el caso de la mediana, también nos proporciona una idea de cuanta cantidad de ramas, días y MB que necesitamos tener para no salirnos del estándar establecido anteriormente. esto conlleva a considerar varias alternativas, como por ejemplo el ciclo de vida de cada rama (relacionada a su edad), el espacio estándar a definir para cada rama en relación a la cantidad de datos (MB) del total de commits de cada rama y considerar una mediana de 13 ramas por cada repositorio, para mantener a la vista todos los datos que se trabajaron en esos 36 días en el repositorio.

# GRAFICOS RESPECTIVOS DE LOS DATOS.

Gráfica que representa la tabla de Datos entregada anteriormente:

A graph of different colored lines

Description automatically generated

Gráfica que representa en % de participación del total de repositorios por commits, lo que nos da la idea de cual repositorio tiene más actividad y flujo de datos trabajados:

A pie chart with numbers and a number of percentages

Description automatically generated with medium confidence

Análisis de Tabla de Datos:

1.- Comparación del número de ramas entre los repositorios: a simple vista nos damos cuenta que el repositorio “D” contiene el mayor número de ramas creadas y el repositorio “E” contiene el menor número de ramas, esto se puede interpretar como que el repositorio D ha tenido más trabajo los últimos días, lo cual le ha permitido crear la mayor cantidad de iteraciones o el equipo de desarrolladores les ha llegado más cantidad de actualizaciones al software administrado que el repositorio “E”, el cual tuvo muy poca actividad en el mismo lapso de tiempo que el anterior repositorio.

2.- Por otra parte, tenemos la edad de las ramas presentada en esta gráfica, la cual nos entrega como dato que el repositorio “C” contiene la mayor cantidad de días, lo que nos puede representar que el repositorio más viejo y el repositorio “E” contiene la menor cantidad de días lo que representa al repositorio más temprano o más joven, lo que debe significar que este ciclo del software viene a ser uno de los últimos abiertos en el repositorio central de Github.

3.- Además, tenemos la cantidad de datos que han iterado en cada repositorio, encontrándonos con que el repositorio “G” ha movido más datos que todo el resto de los repositorios, dándonos una idea que el flujo de datos de este repositorio tiene mayor actividad de desarrollo más intensa que los otros, pero no así el repositorio “E”, el cual presenta la menor cantidad de flujo de datos, lo que significa que este repositorio contiene un software que no tiene mucho movimiento en general.

# con estos datos podemos decir que “E” es el repositorio que posee menos actividad en general que cualquier otro, ya que aparece en la gráfica con los menores datos a analizar, y los repositorios “C, D y G” poseen la mayor cantidad de actividades presentada en GitHub, lo que se puede interpretar que esos equipos de desarrollo han iterado con mayor frecuencia en sus repositorios respectivamente en nuestro sistema de versionamiento GitHub.

# CONCLUSIONES y RECOMENDACIONES

Conclusiones:

La implementación de scripts de Shell automatizados ha demostrado una notable eficiencia en la gestión de ramas obsoletas en repositorios Git, además que la automatización reduce significativamente el tiempo y esfuerzo manual requerido para identificar y eliminar estas ramas. La utilización de la Shell en Batch ha llevado a una optimización sustancial del espacio de almacenamiento en los repositorios. La eliminación regular y automática de ramas obsoletas ha reducido el espacio ocupado, mejorando la capacidad de manejo y rendimiento del repositorio. Los scripts desarrollados han incorporado medidas de seguridad y verificación, como la confirmación del usuario antes de la eliminación de ramas y la validación posterior del estado del repositorio. Estas medidas han garantizado la integridad y la fiabilidad del proceso, minimizando el riesgo de eliminar ramas importantes. La interfaz de usuario intuitiva y la compatibilidad con la API de GitHub han facilitado la adopción y uso de la herramienta por parte de los desarrolladores. La integración fluida con GitHub permite a los usuarios gestionar ramas directamente desde la línea de comandos de manera eficiente. La automatización de la gestión de ramas ha mejorado la calidad del código y la colaboración entre equipos. Al mantener el repositorio limpio y libre de ramas obsoletas, los desarrolladores pueden centrarse en el desarrollo activo sin preocuparse por la acumulación de ramas innecesarias.

Recomendaciones:

Se recomienda la implementación regular y programada de los scripts de Shell para asegurar que los repositorios se mantengan optimizados y libres de ramas obsoletas de manera continua. Esto puede incluir la ejecución automática de la herramienta en intervalos definidos, por otra parte, es importante realizar un monitoreo constante del rendimiento y los resultados de los scripts para identificar posibles mejoras y ajustar los parámetros según sea necesario. El análisis periódico de los datos de uso y espacio puede ofrecer percepciones valiosas para optimizar aún más el proceso. Proporcionar capacitación adecuada a los desarrolladores sobre el uso de la Shell en Batch y las mejores prácticas en la gestión de ramas. Esto asegurará que los usuarios comprendan plenamente las funcionalidades y beneficios de la herramienta. El mantener una documentación detallada y actualizada sobre el funcionamiento de los scripts y las medidas de seguridad implementadas proporcionará tranquilidad al momento de realizar la entrega del software. Además, ofrecer soporte técnico para resolver cualquier problema que los usuarios puedan enfrentar durante la implementación. Considerar la posibilidad de extender y personalizar los scripts para adaptarse a las necesidades específicas de diferentes proyectos y equipos. Esto puede incluir la adición de nuevas funcionalidades o la modificación de criterios de identificación de ramas obsoletas. Explorar la integración de la Shell en Batch con otras herramientas de desarrollo y plataformas de automatización (como CI/CD) para maximizar la eficiencia y coherencia en la gestión de ramas en todo el ciclo de desarrollo.

En resumen, este diseño de software no solo mejora la eficiencia y la gestión del espacio utilizado por las fuentes, sino que también fortalece la seguridad y la calidad del desarrollo colaborativo. La implementación regular, el monitoreo continuo y la capacitación adecuada son clave para maximizar los beneficios que este software pretende entregar.

# REFERENCIAS BIBLIOGRÁFICAS

IACC (2020). Planteamiento del problema. Parte I. Seminario de Título. Semana 1

IACC (2020). Planteamiento del problema. Parte II. Seminario de Título. Semana 2

IACC (2020). Planteamiento del problema. Parte III. Seminario de Título. Semana 3

IACC (2020). Objeto, fundamentación teórica y metodológica de la investigación. Parte I. Seminario de Título. Semana 4.

IACC (2020). Objeto, fundamentación teórica y metodológica de la investigación. Parte II. Seminario de Título. Semana 5.

Recursos adicionales:

Sitio oficial de la herramienta Git y su documentación.

https://docs.github.com/en (Github Community [2025], Github Docs).

Sitio oficial del administrador de repositorios GitHub y su documentación:

https://git-scm.com/doc (Chacon, Scott [2025], Git – fast version control)

IACC (2020). Sistematización de la información. Parte I. Seminario de Título. Semana 6

IACC (2020). Sistematización de la información. Parte II. Seminario de Titulo. Semana 7

IACC (2020). Estructura de la investigación. Seminario de Titulo. Semana 8

IACC (2020). Propuesta e implementación del proyecto. Parte I. Proyecto de Título. Semana 2

IACC (2020). Propuesta e implementación del proyecto. Parte II. Proyecto de Título. Semana 3

IACC (2020). Propuesta e implementación del proyecto. Parte III. Proyecto de Título. Semana 4

IACC (2020). Prueba y análisis de resultados. Parte I. Proyecto de Título. Semana 5

IACC (2020). Prueba y análisis de resultados. Parte II. Proyecto de Título. Semana 6

IACC (2020). Conclusiones y Recomendaciones de Proyecto de Título. Parte I. Semana 7

IACC (2020). Conclusiones y Recomendaciones de Proyecto de Título. Parte II. Semana 8