Facultad de Ingeniería, Diseño y Ciencias Aplicadas
Ingeniería de Sistemas
Proyecto integrador I
Profesores: Milton Sarria Paja, Claudia Castiblanco, Ingri Lorena Jojoa

Taller de Git y GitHub Parte 1

Objetivo del Taller

El objetivo de este taller es que los estudiantes desarrollen competencias prácticas y colaborativas en el uso de Git y GitHub para el control de versiones en proyectos de software. A través de actividades concretas, aprenderán a manejar ramas, realizar merges, gestionar commits, y resolver conflictos en un entorno colaborativo. Se trabajará en los equipos ya creados para potenciar la experiencia de trabajo en equipo.

Información previa:

- **`README.md`:** Cada grupo debe actualizar el `README.md` al inicio del proyecto, incluyendo una descripción clara del mismo, instrucciones para configurarlo y cualquier otra información relevante. Esto debe realizarse cada vez que se complete una historia de usuario o una nueva funcionalidad.
- **`CODESTYLE.md`:** Antes de comenzar con la codificación, los equipos deben acordar un estándar de codificación y documentarlo en `CODESTYLE.md`. Todos los miembros del equipo deben seguir este estándar al escribir código, y cualquier modificación en las reglas debe ser consensuada y actualizada en este archivo.

1. Configuración Inicial y Creación de Repositorio *(10%)*

- ✓ **Objetivo:** Configurar Git en los equipos locales y crear un repositorio en GitHub para el proyecto.
- ✓ **Actividades:**
  - Todos los usuarios deben haber creado
  - Configurar el nombre de usuario y correo electrónico en Git utilizando:

```
git config --global user.name "Tu Nombre"
git config --global user.email "tuemail@ejemplo.com"
```

  - Los estudiantes deben usar el enlace proporcionado por el instructor para aceptar la asignación en GitHub Classroom.
  - Se les pedirá que se unan a un grupo existente o formen uno nuevo.
  - Una vez que acepten la asignación, GitHub Classroom generará un repositorio para su equipo.
  - Los estudiantes deben clonar el repositorio en sus computadoras locales

```
git clone <URL-del-repositorio>
```

- ✓ **Entrega:** Capturas de pantalla mostrando la configuración y clonación del repositorio.
- ✓ **Pregunta para reflexión:**

¿Cuál es la diferencia entre clonar un repositorio y hacer un fork? ¿En qué situaciones utilizarías cada uno?

2. **Colaboración en Equipo usando Ramas** *(20%)*

- ✓ **Objetivo:** Colaborar en equipo utilizando ramas para desarrollar diferentes funcionalidades de forma paralela.
- ✓ **Actividades:**
  - Crear ramas específicas para cada funcionalidad o tarea asignada (Se encuentran al final de este documento) usando:

    ```
    git checkout
    Ejemplo: git checkout -b feat/crear-usuario
    ```

  - Modificar el archivo `README.md` y añadir la información relacionada con la historia de usuario.
- ✓ **Entrega:** Capturas de pantalla de las ramas creadas y el contenido actualizado del archivo `README.md`.
- ✓ **Tips de comandos:**
  - Para listar todas las ramas locales y remotas:
    ```
    git branch -a
    ```
  - Para cambiar de rama:
    ```
    git checkout <nombre-de-la-rama>
    ```

- ✓ **Pregunta para reflexión:**

¿Por qué es importante seguir una convención para nombrar las ramas? ¿Qué beneficios tiene en un equipo grande?

3. **Gestión de Commits y Estándares de Codificación** *(20%)*

- ✓ **Objetivo:** Realizar commits significativos y coherentes, siguiendo un estándar acordado por el equipo.
- ✓ **Actividades:**
  - Hacer commits con mensajes descriptivos que reflejen los cambios realizados. Ejemplo:

    ```
    git commit -m "feat: añadir validación al formulario de registro"
    ```

  - Documentar el estilo de codificación utilizado en el proyecto en un archivo `CODESTYLE.md`.

Paso 1: Definir el Estándar de Codificación: Todos los usuarios (A, B, C, D, y E) deben reunirse para discutir y acordar un estándar de codificación que todos seguirán en el proyecto (**`CODESTYLE.md`**. Decidan aspectos como:

- Nombres de variables y funciones (camelCase, PascalCase, etc.).
- Reglas de indentación (espacios vs. tabuladores).
- Longitud máxima de líneas de código.
- Formato de comentarios y documentación del código.

Crear el Archivo `CODESTYLE.md`:

- Un usuario (por ejemplo, el usuario A) crea el archivo `CODESTYLE.md` en la rama `main` o `develop`.
- Este archivo debe incluir todas las reglas acordadas en la reunión.
- **Commit:** El usuario A realiza un commit con un mensaje claro, por ejemplo:
- `git add CODESTYLE.md`
- `git commit -m "docs: Added coding style guidelines to CODESTYLE.md"`
- `git push origin main`

Paso 2: Realizar Cambios y Gestionar Commits: Asignar a cada usuario una tarea que implique hacer cambios en el código. Cada tarea debe realizarse en una rama separada. **Ejemplo de tareas:**

- **Usuario A:** Implementar la funcionalidad de registro de usuarios.
- **Usuario B:** Añadir validación al formulario de inicio de sesión.
- **Usuario C:** Crear la página de perfil del usuario.
- **Usuario D:** Mejorar el sistema de autenticación.
- **Usuario E:** Refactorizar el código del controlador de usuarios.

Crear una Rama para Cada Tarea:

- Cada usuario crea una nueva rama desde `main` o `develop` para trabajar en su tarea asignada.
  Ejemplo:
  `git checkout -b feat/registro-usuario`

Paso 3: Realizar Cambios y Hacer Commits:

- Cada usuario realiza los cambios necesarios en su rama. Siguen el estándar de codificación definido en el archivo `CODESTYLE.md`.
- Los usuarios deben hacer commits pequeños y frecuentes, con mensajes descriptivos que expliquen los cambios realizados.
- Ejemplo de commits:
- `git add registro.js`
- `git commit -m "feat: Implement user registration functionality"`
- `git add auth.js`
- `git commit -m "fix: Correct validation error in login form"`

- ✓ **Entrega:** Capturas de pantalla de los commits realizados y el contenido del archivo `CODESTYLE.md`.
- ✓ **Tips de comandos:**
  - Para añadir cambios al área de preparación (staging area):
  - `git add <nombre-del-archivo>`
  - Para ver el historial de commits:
  - `git log --oneline`
- ✓ **Pregunta para reflexión:**

¿Qué diferencia hay entre un commit estándar y uno amend? ¿Cuándo usarías cada uno?

4. **Merge y Resolución de Conflictos** *(30%)*

- ✓ **Objetivo:** Realizar merges efectivos y resolver conflictos que puedan surgir al integrar cambios.
- ✓ **Actividades:**
  - Hacer merge de las ramas en la rama `develop` o `main`, resolviendo conflictos si es necesario:
  - `git merge <nombre-de-la-rama>`
  - Utilizar un editor de texto o IDE como Visual Studio Code para resolver los conflictos manualmente.

Revisión del Código:

- Antes de realizar el merge, cada usuario revisa su código para asegurarse de que cumple con el estándar de codificación.
- Si es necesario, ajustan el código para alinearse con las pautas en `CODESTYLE.md`.

Realizar el Merge y Resolver Conflictos: Sincronización con `main` o `develop`:

- Antes de hacer un pull request (PR), cada usuario debe sincronizar su rama con `main` o `develop` para asegurarse de que están trabajando con la versión más reciente del código. Esto se hace con:

```
git fetch origin
git merge origin/main
```

Resolver Conflictos (si los hay): Si surgen conflictos durante el merge, los usuarios deben resolverlos de acuerdo con el estándar de codificación y hacer un commit de la resolución de conflictos.

Crear un Pull Request (PR): Cada usuario crea un pull request desde su rama hacia `main` o `develop`.

- El PR debe incluir una descripción clara de los cambios realizados y una referencia al estándar de codificación utilizado.
- Solicitan una revisión por parte de otro miembro del equipo.

Revisión por Pares y Aprobación:

- Otro usuario revisa el PR para asegurarse de que el código sigue el estándar de codificación y cumple con los requisitos funcionales.
- Si todo está en orden, aprueban el PR y realizan el merge.

Tips de Comandos a Utilizar:

- **Para cambiar de rama:**
  `git checkout <nombre-de-la-rama>`
- **Para listar commits recientes:**
  `git log --oneline`
- **Para ver las diferencias antes de un commit:**
  `git diff`
- **Para visualizar y resolver conflictos:**
  `git status`
- **Para fusionar una rama con `main` o `develop`:**
  `git merge <nombre-de-la-rama>`

  - ✓ **Entrega:** Capturas de pantalla del proceso de merge y la resolución de conflictos, junto con una explicación de los pasos seguidos.
  - ✓ **Tips de comandos:**
    - Para ver los archivos en conflicto después de un merge:
    - `git status`
    - Para abortar un merge en caso de errores:
    - `git merge --abort`
  - ✓ **Pregunta para reflexión:**

¿Qué estrategias de resolución de conflictos podrías aplicar en un proyecto con múltiples colaboradores?

5. Trabajo Final y Documentación del Proyecto *(20%)*

- ✓ **Objetivo:** Consolidar el trabajo realizado y documentar el proceso del proyecto.
- ✓ **Actividades:**
  - Completar el proyecto y preparar un informe en formato markdown (`informe.md`) que contenga: Nombres de los participantes, Evidencias de los commits, merges y conflictos resueltos, Descripción de los estándares de codificación, Capturas de pantalla del proceso completo.

Documentar el Proceso en `informe.md`:

- Cada usuario documenta en un archivo `informe.md` cómo realizó su tarea, cómo gestionó los commits, y cómo resolvió cualquier conflicto.
- El archivo debe incluir capturas de pantalla de los commits realizados, los conflictos resueltos, y la revisión del código.

- ✓ **Entrega:** Subir el informe al repositorio en GitHub
- ✓ **Pregunta para reflexión:**

¿Cómo puedes asegurarte de que la documentación de tu proyecto sea útil para futuros colaboradores?

Recomendaciones para el Taller

- **Trabajo en Equipo:** Trabajar en grupos personas para mejorar la colaboración y la gestión de proyectos.
- **Revisión por Pares:** Configurar reglas de branches para requerir revisiones antes de hacer merge a `develop` o `main`.

- **Comunicación:** En el contexto no es necesario usar herramientas porque trabajarán en clase y podrán comunicarse directamente con sus compañeros. En el desarrollo de otros proyectos pueden usar herramientas que les permitan chatear o hacer videollamadas (Teams, Slack, Discord, etc.) para mantener una comunicación constante.

Contexto y Tareas Por Realizar

Cada miembro del equipo trabajará en una parte de un sistema de gestión de vehículos, desarrollando clases que representen distintos aspectos como información técnica, historial de mantenimiento y gestión. A su vez, se incluirá un archivo `README.md` para documentar el uso del sistema.

Deben crear una carpeta `code` donde estarán todos los archivos que se especificarán durante la realización de los pasos y el archivo de `CODINGSTYLE.md`. Por fuera de esta carpeta debe estar el archivo `README.md` y el archivo `informe.md`.

Trabajen en la rama `develop` para crear la estructura básica del proyecto. Es importante que todos terminen la primera parte de las tareas antes de seguir con la segunda. Por cada tarea realizada actualicen el `README.md` con la estructura y funcionalidades nuevas del proyecto.

Nota: No eliminen las ramas creadas una vez finalicen su propósito.

1. Creación de estructura inicial

Usuario A:

- Crea una clase "Vehiculo" que debe incluir los atributos "marca", "modelo", "año", "kilometraje", "estado_actual" y "tipo_combustible", junto con sus respectivos métodos getter y setter.

Usuario B:

- Crea una clase "HistorialMantenimiento" que almacene información sobre las reparaciones y mantenimientos realizados a un vehículo. Debe incluir los atributos "fecha", "descripcion_servicio", "kilometraje_en_servicio", "costo", y "nombre_mecanico", junto con sus respectivos métodos getter y setter.

Usuario C:

- Implementa la clase "Main" que posea una lista de vehículos y permita agregarlos y buscarlos por año.

Usuario D:

- Implementa validaciones adicionales en la clase "Vehiculo", asegurando que el tipo de combustible solo pueda ser de una lista predefinida (ej. "Gasolina", "Diesel", "Eléctrico").

Usuario E:

- Implementa un método en la clase "Main" que permita imprimir todos los vehículos de la flota con las características de cada uno.

2. Nuevas Funcionalidades

Usuario A:

- Modifica la clase "Main", para que ahora el filtro por año permita buscar vehículos que se encuentren en un rango de años. Actualiza el `README.md` con ejemplos de cómo calcular la antigüedad de un vehículo.

Usuario B:

- Modifica la clase "Main", para que ahora el filtro por año reciba un parámetro que especifique si es mayor o menor, y que, dependiendo de este, liste los vehículos que son mayores o menores al año especificado. Actualiza el `README.md` con instrucciones sobre cómo utilizar esta funcionalidad.

Usuario C:

- Modifica la clase “Vehiculo”, para agregar un nuevo atributo “color”. Agrega los getter y setter pertinentes y actualiza el `README.md` con el nuevo atributo.

Usuario D:

- Modifica la clase “Vehiculo”, para agregar un nuevo atributo “potencia” (número que indique caballos de fuerza). Agrega los getter y setter pertinentes y actualiza el `README.md` con el nuevo atributo.

Usuario E:

- Modifica el método de impresión de datos de todos los vehículos para agregar los nuevos atributos definidos por el usuario D y E.

Cuando el equipo termine las modificaciones y la rama `develop` esté actualizada con los últimos cambios, uno de los integrantes deberá hacer un pull request a `main` y integrantes diferentes deberán aceptarlo, de forma que la rama `main` quede actualizada también.

En caso de presentar conflictos, documéntenlos en el archivo `informe.md` con el lugar del conflicto y cómo se solucionaron.