Optimización y Documentación (UD3)

1. Refactorización de Código

La refactorización es una técnica disciplinada para reestructurar un cuerpo de código existente, alterando su estructura interna sin cambiar su comportamiento externo. Su objetivo principal es limpiar el código para minimizar la introducción de errores y mejorar su mantenibilidad.

1.1. Principios y Ciclo de Vida

- **Objetivo:** Hacer el código más entendible y barato de modificar.
- **Cuándo aplicar:** No es una fase aislada al final del desarrollo. Debe realizarse de forma continua: al añadir una funcionalidad, al corregir un error o al revisar código.
- **Relación con las pruebas:** Es imprescindible contar con pruebas automáticas (Tests) antes de refactorizar. Esto garantiza que los cambios estructurales no rompan la funcionalidad (enfoque TDD - Test Driven Development).

1.2. "Malos Olores" (Bad Smells)

Son indicadores o síntomas en el código que sugieren la necesidad de una refactorización:

- **Código duplicado:** El problema más común. Si ves la misma estructura en más de un lugar, unifícala.
- **Método largo:** Cuanto más largo es un método, más difícil es de entender.
- **Clase grande:** Una clase que intenta hacer demasiadas cosas (baja cohesión).
- **Lista de parámetros larga:** Difícil de entender y propensa a errores.

1.3. Patrones de Refactorización Comunes

- **Extraer Método (Extract Method):** Convertir un fragmento de código dentro de un método largo en un nuevo método con un nombre descriptivo.
- **Encapsular Campo:** Hacer privados los campos públicos y proporcionar métodos de acceso (getters/setters).
- **Reemplazar número mágico:** Sustituir literales numéricos en el código por constantes con nombres significativos.
- **Consolidar expresiones condicionales:** Unificar una secuencia de comprobaciones condicionales con el mismo resultado en una sola comprobación.

1.4. Limitaciones

La refactorización puede ser costosa o inviable cuando:

- Las bases de datos están fuertemente acopladas al código y requieren migración de datos.
- Se modifican interfaces públicas que ya están siendo utilizadas por sistemas externos (rompería la compatibilidad).

2. Control de Versiones

El control de versiones es la gestión de los cambios que se realizan sobre los elementos de un producto (código, documentación, imágenes). Permite volver a estados anteriores y facilita el trabajo colaborativo.

2.1. Tipos de Sistemas

- **Centralizados (ej. Subversion/SVN):** Un único servidor contiene el historial. Los desarrolladores trabajan sobre una copia local de la última versión. Requiere conexión para la mayoría de acciones.
- **Distribuidos (ej. Git):** Cada desarrollador tiene una copia completa del repositorio (historial incluido) en su máquina. Permite trabajar desconectado y realizar operaciones complejas de manera local.

2.2. Conceptos Fundamentales

- **Repositorio:** Base de datos donde se almacenan los archivos y el historial de cambios.
- **Working Copy (Copia de trabajo):** Directorio local donde se editan los archivos.
- **Commit:** Acción de confirmar y guardar los cambios de la copia de trabajo en el repositorio (crea una nueva versión o "foto" del estado actual).
- **Update / Pull:** Traer cambios del repositorio remoto a la copia local.
- **Push:** Enviar los cambios locales confirmados al repositorio remoto.

2.3. Git y GitHub en el Entorno de Desarrollo

- **Git:** Es la herramienta de línea de comandos o integrada en el IDE que gestiona las versiones.
- **GitHub:** Plataforma de alojamiento para repositorios Git que añade funciones sociales y de gestión de proyectos.
- **Integración en IDE (NetBeans/IntelliJ):** Los entornos modernos permiten realizar las operaciones de *init, commit, push, pull y clone* visualmente sin salir del editor.

3. Documentación

La documentación del software es crítica para asegurar que el sistema pueda ser entendido y mantenido por otros desarrolladores (o por uno mismo) en el futuro.

3.1. Tipos de Comentarios en el Código

1. **Comentarios de implementación:** Aclaraciones breves sobre el "cómo" o el "por qué" de una lógica compleja dentro de un método. Se debe evitar comentar lo obvio; el mejor comentario es un buen nombre de variable o método.

2. **Comentarios de documentación (Doc Comments):** Destinados a explicar la API pública (clases, interfaces, métodos). Describen "qué" hace el código, no "cómo" lo hace.

3.2. JavaDoc

Es el estándar industrial para documentar código Java. Utiliza comentarios especiales que el compilador o herramientas externas procesan para generar documentación navegable (HTML).

- **Sintaxis:** Comienza con /** y termina con */.
- **Etiquetas Principales:**
    - @author: Nombre del desarrollador.
    - @version: Versión de la clase o método.
    - @param: Describe un parámetro de entrada de un método.
    - @return: Describe qué devuelve el método.
    - @throws / @exception: Indica qué errores puede lanzar el método.
    - @see: Enlaces a otra parte de la documentación.

3.3. Generación de Documentación

Los IDEs permiten generar automáticamente la estructura web (HTML) de la documentación del proyecto basándose en estos comentarios, facilitando la creación de manuales técnicos actualizados sin esfuerzo extra de maquetación.

GUÍA DE TALLER: GIT Y GITHUB (UD3 - ENDES)

1. Introducción Teórica: ¿Qué es Git?

Antes de escribir comandos, es vital entender qué está ocurriendo "por debajo".

- **Control de Versiones:** Sistema que registra los cambios en un archivo o conjunto de archivos a lo largo del tiempo.
- **Git (Distribuido):** A diferencia de los sistemas centralizados, en Git cada cliente tiene una copia completa del historial (repositorio).
- **Las 3 Áreas de Git:**
    1. **Working Directory (Directorio de trabajo):** Los archivos tal cual los ves y editas en tu carpeta.
    2. **Staging Area (Área de preparación):** Un limbo donde "apuntas" qué cambios irán en la próxima "foto".
    3. **Repository (.git):** Donde Git almacena permanentemente los cambios confirmados (snapshots).

---

2. Configuración Inicial (Prerrequisitos)

Objetivo: Decirle a Git quiénes somos para que los cambios lleven nuestra firma.

Comandos a ejecutar en la terminal (Git Bash):

Bash

```
# Configurar nombre de usuario
git config --global user.name "Tu Nombre y Apellido"

# Configurar correo electrónico (debe coincidir con el de GitHub)
git config --global user.email "tu_email@ejemplo.com"

# Verificar la configuración
git config --list
```

---

3. Escenario 1: Flujo de Trabajo Local

Objetivo: Aprender a gestionar versiones en mi propio ordenador.

Paso 3.1: Inicializar y Estado

Creamos una carpeta nueva para el proyecto y arrancamos Git.

- **Teoría:** git init crea una carpeta oculta .git que contiene toda la base de datos de versiones.
- **Práctica:**

Bash

```
# Crear directorio (o hazlo desde Windows)
mkdir proyecto_endes
cd proyecto_endes

# Inicializar repositorio
git init
# Salida esperada: "Initialized empty Git repository in..."

# Ver estado actual (¡Úsalo siempre!)
git status
# Salida esperada: "No commits yet", "nothing to commit"
```

Paso 3.2: Crear contenido y Añadir (Staging)

Creamos un archivo readme.md con algún texto.

- **Teoría:** Git no rastrea archivos nuevos automáticamente. Debemos pasarlos al *Staging Area* con add.
- **Práctica:**

Bash

```
# (Crea el archivo manualmente o con comando)
echo "Hola Mundo Git" > readme.md

# Comprobar que Git detecta el archivo como "Untracked" (rojo)
git status

# Añadir el archivo al área de preparación
git add readme.md
# (O usa 'git add .' para añadir todo)

# Comprobar que ahora está listo para commit (verde)
git status
```

Paso 3.3: Confirmar cambios (Commit)

- **Teoría:** El commit toma todo lo que hay en el *Staging Area* y lo guarda permanentemente en el historial con un mensaje descriptivo.
- **Práctica:**

Bash

```
git commit -m "Primer commit: Creación del readme"
# Salida: [master (root-commit) ...] 1 file changed...
```

Paso 3.4: Historial y Diferencias

Modifica el archivo readme.md añadiendo una segunda línea de texto.

- **Comandos de inspección:**

Bash

```
# Ver qué ha cambiado exactamente antes de añadir
git diff

# Ver el historial de cambios resumido
git log --oneline
```

Paso 3.5: Gestión de Archivos (Borrar y Renombrar)

Según tu documento, trabajamos operaciones de sistema a través de Git.

- **Práctica:**

  Bash

```
# Renombrar un archivo (forma correcta en Git)
git mv readme.md leeme.md

# Confirmar el cambio de nombre
git commit -m "Refactor: Renombrado readme a leeme"

# Eliminar un archivo
git rm leeme.md
git commit -m "Delete: Eliminado archivo leeme"
```

---

4. Escenario 2: Conexión Remota y Autenticación (TOKEN)

Importante: GitHub eliminó la autenticación por contraseña básica en 2021. Ahora es obligatorio usar **Tokens de Acceso Personal (PAT)** o claves SSH. Integramos aquí el documento "TOKEN".

Paso 4.1: Generar el Token en GitHub

Antes de intentar subir código, necesitamos la "llave".

1. Entra en GitHub.com -> Foto de perfil -> **Settings**.
2. Menú lateral izquierdo (abajo del todo) -> **Developer Settings**.
3. Menú lateral -> **Personal access tokens** -> **Tokens (classic)**.
4. Botón: **Generate new token (classic)**.
5. **Note:** Ponle un nombre (ej: "Clase ENDES").
6. **Expiration:** Ponle fecha de caducidad (ej: 30 días).
7. **Scopes (Alcance):** Marca la casilla **repo** (Full control of private repositories).
8. **IMPORTANTE:** Copia el código del token ahora (ghp_...). **No podrás verlo nunca más.**

Paso 4.2: Vincular con Repositorio Remoto

1. Crea un repositorio vacío en GitHub (botón "+"). No lo inicialices con README.
2. Copia la URL del repositorio (HTTPS).
3. En tu terminal local (dentro de tu proyecto):

Bash

```
# Vincular tu carpeta local con la nube
# Sustituye la URL por la tuya
git remote add origin https://github.com/tu_usuario/tu_proyecto.git

# Verificar la conexión
git remote -v
```

Paso 4.3: Subir cambios (Push) con Token

Aquí es donde usaremos el token generado.

- **Práctica:**

  Bash

  ```
  # Enviar la rama 'master' al remoto 'origin'
  git push -u origin master
  ```

- **Autenticación:**
  - Te pedirá usuario: Escribe tu nombre de usuario de GitHub.
  - Te pedirá contraseña: **PEGA EL TOKEN** (no tu contraseña de email).
  - *Nota: Al pegar en la terminal, a veces no se mueven el cursor ni aparecen asteriscos. Pega y pulsa Enter.*

---

5. Escenario 3: Trabajo Colaborativo (Clonar)

Objetivo: Descargar un proyecto existente para trabajar en él (simulando que te unes a un equipo o descargas el proyecto en otro PC).

- **Práctica:**

  Bash

  ```
  # Salir de la carpeta actual
  cd ..

  # Clonar el repositorio (descargar copia completa)
  git clone https://github.com/tu_usuario/tu_proyecto.git proyecto_descargado

  # Entrar en la nueva carpeta
  cd proyecto_descargado

  # Comprobar que está todo el historial
  git log --oneline
  ```

Flujo típico de actualización (Pull)

Si trabajas en equipo, antes de empezar a trabajar cada día:

Bash

```
# Bajar cambios que otros hayan subido
```

git pull origin master

Resumen de Comandos Clave para el Alumno

Comando	Descripción
git init	Crea un repositorio nuevo.
git status	Muestra el estado de los archivos (modificados, nuevos...).
git add <archivo>	Prepara el archivo para el commit.
git commit -m "msg"	Guarda la versión con un mensaje.
git log --oneline	Muestra el historial resumido.
git remote add origin	Conecta con GitHub.
git push origin master	Sube los cambios a la nube.
git pull origin master	Baja los cambios de la nube.