Análisis y Diseño de Software

Curso 2024-2025

Práctica 3

Introducción a la Programación Orientada a Objetos con Java

Inicio: A partir del 24 de Febrero.
Duración: 3 semanas.
Entrega: En Moodle, una hora antes del comienzo de la siguiente práctica según grupos (semana del 17 de Marzo)
Peso de la práctica: 20%

El objetivo de la práctica es introducir al alumno en la programación orientada a objetos con el lenguaje Java. Se pide desarrollar de forma incremental varias clases en Java, incluyendo sus pruebas y documentación. Las clases implementarán funcionalidades relacionadas con un gestor de versiones distribuido.

En el desarrollo de la práctica se utilizarán principalmente los siguientes conceptos de Java:

- *tipos de datos primitivos, String, arrays, listas, mapas*, *tipos referencia* (objetos) definidos por el programador
- *clases* sencillas definidas por el programador para implementar objetos con *atributos (o variables) de instancia y de clase, métodos de instancia, métodos de clase,* y *constructores*
- *herencia, sobrescritura de métodos*
- *comentarios para documentación automática mediante javadoc*

Introducción

En esta práctica desarrollaremos – de manera extremadamente simplificada – parte de un sistema para el control de versiones tipo git (https://en.wikipedia.org/wiki/Git). Estos sistemas son muy populares para permitir a equipos de programadores trabajar en el mismo proyecto de manera concurrente. En particular, permiten identificar los cambios realizados a los distintos ficheros almacenados, incluyendo la creación de ramas (*branches*), y su mezcla (*merge*).

Apartado 1. Cambios (2 puntos)

Comenzaremos creando un diseño para representar cambios en un fichero de texto. Consideraremos tres tipos de cambios: agregar líneas, modificar líneas o eliminar líneas. Todos los cambios han de guardar la ruta al fichero, y un número de línea inicial desde la que agregar, eliminar o modificar. Además, en el caso de agregar o modificar, el cambio contendrá el contenido a incluir. En el caso de modificar y eliminar, el cambio debe almacenar un número de línea de fin. Esto indica que se han eliminado todas las líneas incluidas entre el inicio y el fin, y en el caso de modificar se intercambiaron por el nuevo contenido.

Para el tipo de cambio se utilizará la siguiente nomenclatura:

- "+" si es un cambio de tipo agregar
- "/" si es un cambio de tipo modificar
- "-" si es un cambio de tipo eliminar

A modo de ejemplo, el siguiente programa crea varios cambios y los muestra por pantalla.

```
public class ChangeTester {
    public static void main (String[] args) {
        for (Change change : createChanges())
            System.out.println(change);
    }

    public static List<Change> createChanges () {
        Change c1 = new AddChange   (0, "/src/main/NuevaClase.java",          // adds in line 0 (i.e.,before line 1)
                                     "import java.util.*;\nimport java.io.*;");
        Change c2 = new ModifyChange(10, 10, "/src/main/ClaseExistente.java",   // replaces line 10
                                     "// Modificación en la clase existente");
        Change c3 = new RemoveChange(1, 2, "/src/main/ClaseObsoleta.java");     // removes lines 1 and 2
        return List.of(c1,c2,c3);
    }
}
```

Salida esperada:

```
{
  type=+,
  start line=0,
  file path='/src/main/NuevaClase.java',
  content='import java.util.*;
import java.io.*;',
  number of lines=2
}
{
  type=/,
  start line=10,
  file path='/src/main/ClaseExistente.java',
  content='// Modificación en la clase existente',
  number of lines=1,
  end line=10
}
{
  type=-,
  start line=1,
  file path='/src/main/ClaseObsoleta.java',
  end line=2
}
```

Como puedes observar, los cambios que incluyen contenido (añadir y modificar) calculan el número de líneas del nuevo contenido (*number of lines* en la salida).

Se pide: Usando principios de orientación a objetos, crea todo el código necesario para la representación de cambios. Crea testers adicionales que prueben la funcionalidad que has implementado.

Apartado 2. Commits (2.5 puntos)

Un *commit* de cambio (`ChangeCommit`) corresponde a una serie de cambios que son confirmados en un repositorio. Además de los cambios, cada *commit* tiene un identificador único que se genera automáticamente, el nombre del autor del *commit*, una fecha (no se recibe por parámetro, se obtiene en el momento de crear el *commit*), y una descripción de los cambios incluidos en el *commit*. El identificador se compone de dos partes: un entero único incremental de 5 cifras (es decir, toma valores desde 00001 hasta 99999, y debe ser único para cada *commit*), seguido de un código alfanumérico aleatorio de 15 cifras (para generarlo, puedes utilizar `java.util.UUID`).

Opcionalmente, los *commits* de cambio pueden omitir la descripción en el constructor, en cuyo caso se utiliza una descripción por defecto, pero que debe ser configurable (de manera global, para todos los *commits*). Igualmente, puede no especificarse descripción ni usuario, y entonces se utiliza un usuario por defecto, que también debe ser configurable.

Al imprimir los *commits* por pantalla, se debe mostrar el identificador del *commit*, el autor, la fecha y la descripción en distintas líneas. Además, para cada cambio contenido en el *commit*, debe imprimirse el tipo de cambio, la ruta del fichero y la diferencia en número de líneas del fichero (por ejemplo, +1 si tras el cambio hay una línea más, -2 si hay 2 líneas menos). Finalmente, dado un *commit*, debe haber métodos para obtener el número total de líneas incrementadas o decrementadas en todos los ficheros, y para obtener todos los cambios que contiene el *commit*.

Consideraremos otro tipo de *commit*, llamado `MergeCommit`, que agrupa *commits* de cambio (`ChangeCommit`) y otros *commits* de tipo `MergeCommit`. Para su construcción necesita los mismos parámetros que un `ChangeCommit`, pero en vez de cambios, recibe los *commits* que agrupa. Al imprimirlo por pantalla, muestra la misma información básica que un `ChangeCommit`, pero en vez de mostrar los cambios, muestra el identificador y la fecha de cada *commit* que contiene. Al igual que en un `ChangeCommit`, debe haber métodos para obtener el número total de líneas modificadas (que será la suma de las líneas modificadas por los *commits* que agrupa) y los cambios que contiene (que serán todos los cambios de los *commits* que agrupa).

Como ejemplo, a continuación tienes la impresión de dos `ChangeCommit`, y un `MergeCommit` que contiene estos dos *commits*. El resultado de obtener las líneas totales en esos *commits* sería 0, 4 y 4, respectivamente.

```
commit 00001d9309842194543b
Author: John Doe
Date: 2025-02-21
Description: no comment
+ : /src/main/NuevaClase.java (+2)
/ : /src/main/ClaseExistente.java (0)
- : /src/main/ClaseObsoleta.java (-2)

commit 00002d4ac591839324d5
Author: John Doe
Date: 2025-02-21
Description: Decorator interface
+ : /src/pkg1/Decorator.java (+3)
+ : /src/pkg1/Decorator.java (+1)
/ : /src/pkg1/Decorator.java (0)

commit 000034a360c855a204ca
Author: John Doe
Date: 2025-02-21
Description: Merging previous commits
Merged commits:
00001d9309842194543b on 2025-02-21
00002d4ac591839324d5 on 2025-02-21
```

Se pide: Usando principios de orientación a objetos, crea el código necesario para el manejo de *commits*. Crea un tester que produzca la salida de más arriba (por supuesto, pueden variar las fechas y la segunda parte de los identificadores), así como testers adicionales que comprueben la funcionalidad implementada de manera exhaustiva, con otros escenarios.

Apartado 3. Ramas (1.5 puntos)

En este apartado se tratará la definición de ramas (*branches*). Las ramas tienen un nombre y contienen *commits*. Deben soportar añadir *commits* en cualquier momento mediante un método `commit`. Una rama puede crearse a partir de otra, en cuyo caso recibe la rama origen como parámetro en el constructor, y copia los *commits* de esa rama. Al imprimir una rama se debe incluir un resumen de cada uno de sus *commits*, de tal modo que cada línea muestre los 5 primeros caracteres del identificador del *commit*, los primeros 30 caracteres de la descripción, y la fecha. Dada una rama, debe ser posible obtener sus *commits*.

A modo de ejemplo, la siguiente salida muestra una rama "main", y otra "Solving issue #1" creada a partir de la primera rama.

```
Branch: main
3 commits:
00001 - no comment at 2025-02-21
00002 - Decorator interface at 2025-02-21
00003 - Merging previous commits at 2025-02-21

Branch: Solving issue #1 (from main)
3 commits:
00001 - no comment at 2025-02-21
00002 - Decorator interface at 2025-02-21
00003 - Merging previous commits at 2025-02-21
```

Se pide: Usando principios de orientación a objetos, crea todo el código necesario para el manejo de ramas. Crea un tester que produzca la salida de más arriba (las fechas pueden variar), así como testers adicionales que comprueben la funcionalidad implementada de manera exhaustiva, con otros escenarios.

Apartado 4. Repositorios (1.5 puntos)

En este apartado darás soporte a repositorios. Un repositorio tiene un nombre y contiene ramas, una de las cuales está activa. Al crear un repositorio se creará automáticamente una rama con nombre "*main*", que será la rama activa. Al imprimir un repositorio, se debe imprimir el nombre de las ramas existentes en el repositorio, identificando la activa. A continuación, se imprimirá la rama activa del repositorio con el mismo formato descrito en el apartado 3.

Adicionalmente, un repositorio tiene usuarios con permiso para realizar *commits* en la rama activa. Esto es, si al realizar un *commit* (en la rama activa del repositorio), el usuario del *commit* no está en el repositorio, el *commit* no debe añadirse a la rama. El repositorio debe contar con la posibilidad de crear ramas nuevas a partir de otras, y cambiar la rama activa del repositorio. Como un repositorio puede contener cientos de ramas, obtener una rama a partir de su nombre debe ser lo más eficiente posible.

A modo de ejemplo, la siguiente salida muestra la impresión de un repositorio que tiene las dos ramas del apartado anterior.

```
Repository: ADSOF p3
Branches:
- main (active)
- Solving issue #1
Branch: main
3 commits:
00001 - no comment at 2025-02-21
00002 - Decorator interface at 2025-02-21
00003 - Merging previous commits at 2025-02-21
```

Se pide: Usando principios de orientación a objetos, crea todo el código necesario para el soporte a repositorios. Crea un tester que produzca la salida de más arriba (las fechas pueden variar), así como testers adicionales que comprueben la funcionalidad implementada de manera exhaustiva, con otros escenarios.

Apartado 5. Fusión de ramas (2.5 puntos)

Por último, deberás añadir al repositorio la posibilidad de fusionar (*merge*) una rama origen en otra rama destino, dados sus nombres. La fusión de ramas implica buscar el último *commit* en común (con el mismo identificador) de las dos ramas, y copiar los *commits* posteriores en la rama origen, después del último *commit* de la rama destino. Si las ramas no tienen *commits* en común, no pueden fusionarse. Al añadir los *commits* en la rama destino, se agruparán en un `MergeCommit` con la descripción "*Merge branches*" seguida del nombre de las ramas.

La operación de fusión también debe resolver posibles conflictos. Cuando un *commit* de la rama origen modifica el mismo fichero que un *commit* de la rama destino, se produce un conflicto entre el *commit* de origen y el *commit* de destino. Lógicamente, sólo se deben resolver conflictos de *commits* efectuados a partir del último *commit* en común. La resolución del conflicto podrá hacerse según tres estrategias: origen, destino, o nula. La operación de fusión recibirá una estrategia, o si no recibe ninguna, usará la que el repositorio tenga por defecto. Estas estrategias resolverán los conflictos entre *commits* del siguiente modo:

- Si la estrategia es origen, el *commit* de origen se mantiene, y el *commit* de destino se descarta (se elimina de la rama destino).
- Si la estrategia es destino, el *commit* de destino se mantiene, y el *commit* de origen se descarta (no se añade al `MergeCommit`).
- Si la estrategia es nula, no se podrá decidir el *commit* a mantener, por lo que la fusión no se realizará.

El diseño de la solución debe poder extenderse fácilmente con nuevas estrategias de resolución de conflictos.

Si no hay conflictos sin resolver, la fusión se realiza, y la rama destino pasa a incluir los commits fusionados en forma de `MergeCommit`. Si hay conflictos no resueltos, la fusión no se realiza, y se debe devolver la lista de conflictos no resueltos. Al imprimir los conflictos no resueltos, se debe mostrar al menos el fichero en conflicto (por ejemplo: `[Conflict on '/src/main/Main.java', Conflict on '/src/main/Decorator.java']).`

La siguiente salida muestra un repositorio que inicialmente tenía una rama "main" con tres *commits*. A continuación, se crea una rama a partir de "main", poniéndola activa, y se le añade un *commit*. Finalmente, se fusiona esta rama sobre la rama "main". La rama "main" sigue teniendo sus commits originales, más un `MergeCommit` que contiene el *commit* adicional de la segunda rama.

```
Repository: ADSOF p3
Branches:
- main (active)
Branch: main
3 commits:
00001 - no comment at 2025-02-21
00002 - Decorator interface at 2025-02-21
00003 - Merging previous commits at 2025-02-21

Repository: ADSOF p3
Branches:
- main
- Solving issue #1 (active)
Branch: Solving issue #1 (from main)
4 commits:
00001 - no comment at 2025-02-21
00002 - Decorator interface at 2025-02-21
00003 - Merging previous commits at 2025-02-21
00004 - Solving the issue at 2025-02-21

Repository: ADSOF p3
Branches:
- main (active)
- Solving issue #1
Branch: main
4 commits:
00001 - no comment at 2025-02-21
00002 - Decorator interface at 2025-02-21
00003 - Merging previous commits at 2025-02-21
00005 - Merge branches main and Solvin at 2025-02-21
```

Se pide: Usando principios de orientación a objetos, crea el código necesario para la fusión de ramas. Crea un tester que produzca la salida de más arriba (las fechas pueden variar), así como testers adicionales que comprueben la funcionalidad implementada, incluida la resolución de conflictos. Para demostrar la extensibilidad de tu diseño, crea una estrategia adicional de resolución de conflictos. Por ejemplo, puedes crear una estrategia que cree un `MergeCommit` con ambos *commits* si todos los conflictos son por cambios de tipo "agregar". Este `MergeCommit` se añadirá a la rama target y consecuentemente, se ha de borrar el commit original de la rama target.

Comentarios adicionales

- No olvides que, además del correcto funcionamiento de la práctica, un aspecto fundamental en la evaluación será la **calidad del diseño**. Tu diseño debe utilizar los principios de orientación a objetos, además de ser claro, fácil de entender, extensible y flexible. Presta atención a la calidad del código, evitando redundancias.
- Organiza el código en **paquetes**.
- Crea **programas de prueba** para ejercitar el código de cada apartado, y entrégalos con tu código.

Normas de entrega

- Se debe entregar el **código Java** de los apartados, la **documentación** generada con *javadoc*, los **programas de prueba** creados, y un **diagrama de diseño** de toda la práctica (en PDF) junto con una breve explicación.
- El nombre de los alumnos debe ir en la cabecera *javadoc* de todas las clases entregadas.
- La entrega la realizará uno de los alumnos de la pareja a través de Moodle.
- Se debe entregar un único fichero ZIP / RAR con todo lo solicitado, que debe llamarse de la siguiente manera: `GR<numero_grupo>_<nombre_estudiantes>.zip`. Por ejemplo, Marisa y Pedro, del grupo 2261, entregarían el fichero: `GR2261_MarisaPedro.zip`.