DOCUMENTO DE ARQUITECTURA DE SOFTWARE

Equipo 2

10 de mayo de 2025

Realizó Mantenimiento:

Canul Ordoñez, Josué Israel

Garcilazo Cuevas, Mónica

Leo Fernández, José Carlos

Pool Flores, Endrick Alfredo

Rodríguez Coral, Samuel David

1 CONTROL DE DOCUMENTACIÓN

1.1 Control de configuración

Título:	Documento de arquitectura de software
Referencia:	N/A
Autor:	Cristian David Pan Zaldivar

Fecha:	27 de febrero de 2025

1.2 Histórico de versiones

Versión	Fecha	Estado	Responsable	Nombre del archivo
1.0.1	19 – 02 – 2025	B	Cristian Pan	Documento_arquitectura _de_software_1.0.1
1.0.2	25 – 02 – 2025	B	Cristian Pan	Documento_arquitectura _de_software_1.0.2
1.0.3	27 – 02 – 2025	B	Cristian Pan	Documento_arquitectura _de_software_1.0.3
1.0.4	4 – 03 – 2025	B	Cristian Pan	Documento_arquitectura _de_software_1.0.4
2.0.0	03 – 04 – 2025	B	Daniel Rosado Arturo Quezada	Documento_arquitectura_de_software_2.0.0
3.0.0	10 – 05 – 2025	A	Endrick Pool	Documento_arquitectura_de_software_3.0.0

Estado: (B)orador, (R)evisión, (A)probado

1.3 Histórico de cambios

Versión	Fecha	Cambios
1.0.1	19 – 02 – 2025	Se estableció la estructura y se empezó con la redacción de los apartados
1.0.2	25 – 02 – 2025	Se finalizo el detalle de cada uno de los elementos
1.0.3	27 – 02 – 2025	Se incorporaron los diagramas representativos de la arquitectura

1.0.4	04 – 03 – 2025	Se actualizo la arquitectura con respecto a la nueva lógica de conteo para líneas lógicas
2.0.0	03 – 04 – 2025	Se actualizó la arquitectura para eliminar lo referente a líneas lógicas y agregar el conteo de métodos. Se actualizaron los diagramas de casos de uso, de paquetes, de clases y de secuencia.
3.0.0	10 – 05 – 2025	Se actualizó la arquitectura para agregar la comparación de versiones de un proyecto de software y, por ende, se actualizaron los diagramas de casos de uso, de paquetes, de clases y de secuencia.

CONTENIDO

2 INTRODUCCIÓN

2.1 Objetivo

El presente documento de arquitectura de software (DAS) proporciona una descripción general de la arquitectura completa del sistema, utilizando las principales vistas arquitectónicas para su representación. Por ende, su objetivo es captar y transmitir las decisiones que se han tomado para la implementación del sistema.

Este documento se dirige a los miembros del equipo de desarrollo, ya que será de utilidad para tener una guía clara sobre la estructura del sistema, facilitando su implementación.

2.2 Ámbito de la aplicación

El sistema permitirá contar las líneas lógicas de código y los métodos por clase en programas desarrollados en Java bajo el paradigma orientado a objetos. El usuario podrá ingresar la ubicación del proyecto (ruta de la carpeta) para obtener estos conteos, conforme a lo establecido en el documento Estándar de conteo de líneas de código.

Además, el sistema ofrecerá la funcionalidad de comparar dos versiones distintas de un proyecto, identificando y clasificando los cambios entre ellas. Esta comparación facilitará el análisis de la evolución del software y apoyará las tareas de mantenimiento y control de versiones.

2.3 Definiciones y acrónimos

- DAS: Documento de arquitectura de software
- UML: Lenguaje Unificado de Modelado

2.4 Referencias

3 REPRESENTACIÓN ARQUITECTÓNICA

El presente documento presenta la arquitectura como una serie de vistas: vistas de caso de uso, vista lógica y vista de proceso a través lenguaje unificado de modelado (UML). Por cada una de las vistas se proporcionan los diagramas, así como una breve descripción para una mejor interpretación.

3.1 Objetivos y limitaciones arquitectónicas

1. El sistema no contará con una interfaz de usuario, por lo que deberá de poder ser accesible desde la terminal o consola de comandos para introducir la ruta del proyecto.

2. Se aplicará un diseño modular del software, permitiendo que las actualizaciones y correcciones se realicen en módulos específicos sin afectar al sistema completo.

4 VISTA DE CASOS DE USO

Se presenta el diagrama con el caso de uso central, así como con la descripción breve de las funcionalidades que componen al sistema y que deberán de ser abordadas para la definición de la arquitectura del sistema de software.

4.1 Caso de uso central

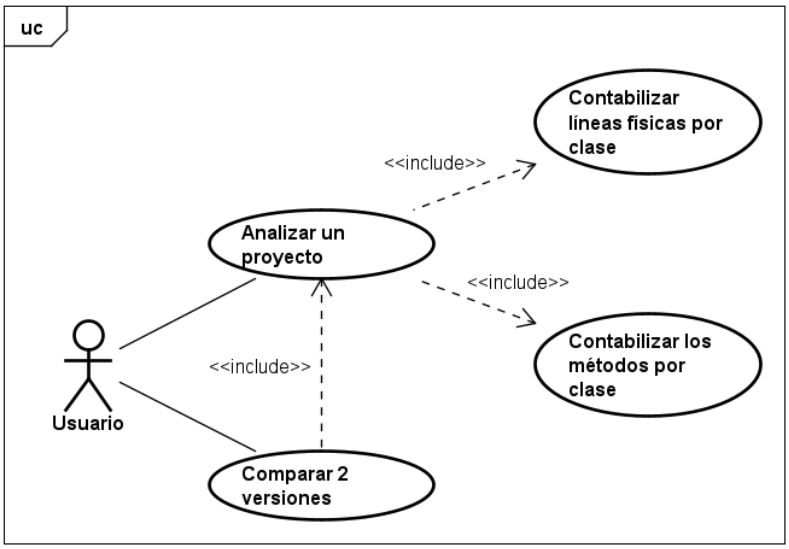

Imagen 1. Diagrama de casos de uso

4.2 Listado de los casos de uso del sistema

A continuación, se presenta la breve descripción de la funcionalidad que el sistema debe de satisfacer.

1. **Contabilizar líneas físicas de código**

   Este caso toma lugar cuando el usuario introduce una ruta válida de una carpeta que contiene un proyecto basado en java. En este se realiza validación de formato y se contabilizan las líneas físicas por cada clase con base en lo definido por el estándar de conteo.

2. **Contabilizar métodos por clase**

Este caso toma lugar a la par del caso anterior. En este se contabiliza los métodos por cada clase con base en lo definido por el estándar de conteo.

3. **Comparar dos versiones de un proyecto de software**

   Este caso toma lugar cuando el usuario introduce dos rutas válidas de carpetas que contengan dos versiones de un proyecto basado en java. En este se realizan los casos anteriores para cada proyecto y posteriormente se realiza una comparación de estas versiones para detectar los cambios entre ellas.

5 VISTA LÓGICA

5.1 Descripción general

Se presenta la vista lógica del sistema mediante un diagrama de paquetes y un diagrama de clases, con el objetivo de mostrar su organización y los elementos que lo conforman. La arquitectura está basada en capas para garantizar que cada entidad tenga una única responsabilidad, lo que mejora la mantenibilidad y flexibilidad del sistema a través de un enfoque modular.

5.2 Paquetes de diseño arquitectónicamente significativos

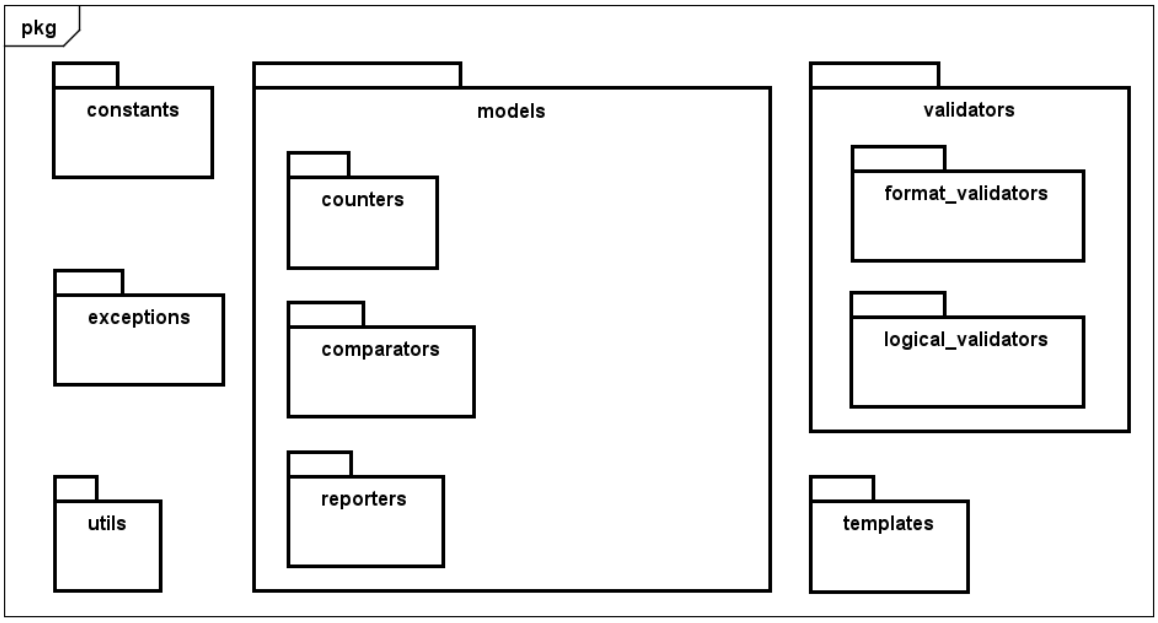

Imagen 2. Diagrama de paquetes

5.2.1 Paquetes principales

1. **Models:** Representa la abstracción de las entidades utilizadas en el sistema. A su vez contiene dos paquetes: reporters y counters. Counters contiene elementos esenciales como el contador de líneas físicas y de métodos. Por su parte, reporters contiene el generador de reportes. El paquete también contiene elementos esenciales como la abstracción de una clase Java, un archivo Java, el analizador de código y el constructor de programa.

2. **Utils:** Agrupa clases que proporcionan utilidades generales y reutilizables en diferentes partes del sistema.

3. **Validators:** Contiene un conjunto de validaciones que son necesarias para la funcionalidad del programa dada la lógica a realizar. Para este caso particular contiene validaciones agrupadas por validaciones de formato y validaciones de declaración de métodos y estructuras como clases o interfaces.

4. **Exceptions:** Almacena excepciones personalizadas dentro del sistema para mejorar el control sobre los errores que pueden generarse debido a fallos humanos.

5. **Constants:** Contiene valores constantes que se utilizan en toda la aplicación, como mensajes de error y expresiones regulares asociadas al lenguaje Java.

6. **Templates**: Contiene las clases bases para la definición de comportamientos de las clases de validadores.

5.2.2 Diagrama de clases significativas del sistema

Dado que el diagrama es lo suficientemente grande como para no poder apreciarse de forma adecuada en el documento, se adjunta el link para su visualización:

Diagrama de clases

Models

- **CodeAnalyzer:** Examina línea por línea archivos Java para identificar estructuras clave y calcular métricas relevantes, de tal manera que gestiona a los elementos necesarios para el conteo y obtención de resultados.
- **JavaClass:** Esta clase es un modelo que representa una clase Java con funcionalidad para contar sus elementos principales.
- **JavaFile:** Modela un archivo de Java, de tal manera que, dada una ruta específica, accede a su contenido y lo almacena en memoria, permitiendo el acceso al

contenido y a realizar operaciones específicas como remover comentarios, ya sea en línea o en bloque, así como eliminar los espacios en blanco, para facilitar el análisis del archivo.

- **ProgramBuilder:** Abstrae la lógica para contar las líneas de código del archivo por analizar, permitiendo contabilizar los metodos y líneas lógicas, así como el acceso a los resultados obtenidos.
- **Proyect:** Esta clase se encarga de representar la abstracción de un proyecto de software java con la ruta y los archivos que contiene. La cual, contiene los archivos JavaFile del proyecto.
- **JavaFileComparator:** Esta clase se encarga de comparar dos archivos Java, toma dos listas de cadenas, cada una representando el contenido de un archivo Java, y las compara línea por línea. A su vez, guarda la información de las líneas indicando si son originales, modificadas, nuevas, eliminadas o cortadas.
- **ProyectComparator:** This class is responsible for comparing two projects and generating a report of the differences between them.
- **STATUS:** Esta enumeración representa el estado de una línea en el proceso de comparación (NEW, MODIFIED, DELETED, ORIGINAL, SPLITED).
- **StructCounter:** Gestiona y acumula métricas de código Java, almacenando una lista de clases, y actuando como puente entre el análisis de código y la generación de reportes, manteniendo un registro estructurado de todas las métricas recolectadas durante el proceso de escane
- **Reporter:** Abstrae la representación de un generador de reportes, de tal manera que define el comportamiento de los posibles tipos de generadores de reportes que puedan existir a futuro.
    - **Terminal Reporter:** Abstrae lógica particular para generar reportes desde la terminal, de tal manera que genera el reporte final del análisis del programa, proporcionando los resultados obtenidos del conteo de líneas lógicas y físicas.
- **ComparationReport:** Esta clase se utiliza para generar un informe de las diferencias entre dos contenidos. Almacena las líneas originales, modificadas, nuevas o eliminadas, y proporciona métodos para actualizar el informe.
- **TxtReporter:** Genera reportes en formato TXT con los resultados de la comparación de archivos Java. Además, crea un archivo por cada clase analizada y proporciona estadísticas globales de cambios.

Utils

- **JavaFilesScanner:** Modela un buscador de rutas de archivos java dentro de un directorio específico, de tal manera que se encarga de obtener las rutas de archivos java de forma recursiva en todo el proyecto y proporciona el acceso a dichas rutas.
- **LineSplitter:** Su propósito es dividir líneas largas de código en segmentos más pequeños para garantizar que ninguna exceda la longitud máxima especificada. La división se realiza en los espacios en blanco para evitar la división de palabras, además, la clase también gestiona el estado de las líneas, indicando si se han dividido.

Validators

- **FormatValidators**
    - **FormatValidator**.
        - **ImportValidator:** Define una validación concreta para determinar si una línea que representa la sentencia de un archivo de java corresponde a la declaración de un import y en caso de ser así, determina si está correctamente declarado o no, es decir, que no esté haciendo uso de imports con comodín
        - **SingleAnnotarionValidator:** Define una validación concreta para determinar si una línea que representa la sentencia de un archivo de java corresponde a la declaración anotaciones del lenguaje y en caso de ser así, determina si la declaración es correcta o no, es decir, que la declaración sea por línea.
        - **SingleDeclarationValidator:** Define una validación concreta para determinar si una línea que representa la sentencia de un archivo de java corresponde a la declaración de tipos de datos o asignaciones, verificando que no exista la declaración de sentencias múltiples.
        - **StyleKAndRValidator:** Define una validación concreta para determinar que se está haciendo uso el formato K&R para la declaración de bloques de código.
- **LogicalValidators**
    - **LogicalValidator.**
        - **MethodDeclarationValidator:** Implementa la lógica de validación para determinar si la sentencia por analizar corresponde a la declaración de un método, incluyendo la declaración del constructor

de una clase. Este validador omite casos particulares como la declaración de métodos abstractos.

        - **TypeDeclarationValidator:** Implementa la lógica de validación para determinar si la sentencia por analizar corresponde a la declaración de un tipo de dato, es decir, declaración de una clase, enumeral o interfaz

- **ValidatorManager:** Actúa como un gestor de acceso a los validadores de formato y de líneas lógicas. Además, define el flujo de validación y permite la adición de nuevas validaciones

Exceptions

- **FileNotFoundException:** Excepción que se lanza en caso de que la ruta del archivo por analizar no exista o no corresponda a un archivo
- **FolderNotFoundException:** Excepción que se lanza en caso de que la ruta del proyecto por analizar no exista o no corresponda a una carpeta
- **InvalidFormatException:** Excepción que se lanza en caso de que el archivo javo no cuente con aspectos de formato especificados
- **JavaFilesNotFoundException:** Excepción que se lanza en caso de que dentro de la carpeta del proyecto no haya archivos java por analizar.

Constants

- **JavaRegexConstanst:** Contiene la declaración de expresiones regulares asociadas a algunas palabras reservadas del lenguaje o a patrones que swon requeridos en varias partes del sistema.
- **InvalidFormatReason:** Define un tipo de dato a utilizar en la excepción InvalidFormatException, de tal manera que se garantice que el mensaje siempre sea uno de los establecidos, por ende, proporciona constantes con la declaración de objetos y mensajes preestablecidos

Templates

- **LogicalValidator:** Representa la abstracción de los validadores para el conteo de las líneas de codigo, definiendo el comportamiento principal para asignar una nueva validación y verificar si existe un validador en la cadena de responsabilidad.

- **FormatValidtor:** Representa la abstracción de los validadores de formato, definiendo el comportamiento principal para asignar una nueva validación y verificar si existe un validador en la cadena de responsabilidad.

6 VISTA DE PROCESO

6.1 Análisis de las líneas de código

Esta sección describe el flujo de los procesos que se llevan a cabo para el análisis de líneas de código de un proyecto, a través de la representación de un diagrama de secuencia que permita identificar la interacción entre los elementos. Sin embargo, el diagrama solo representa el flujo a grandes rasgos para poder comprender la implementación.

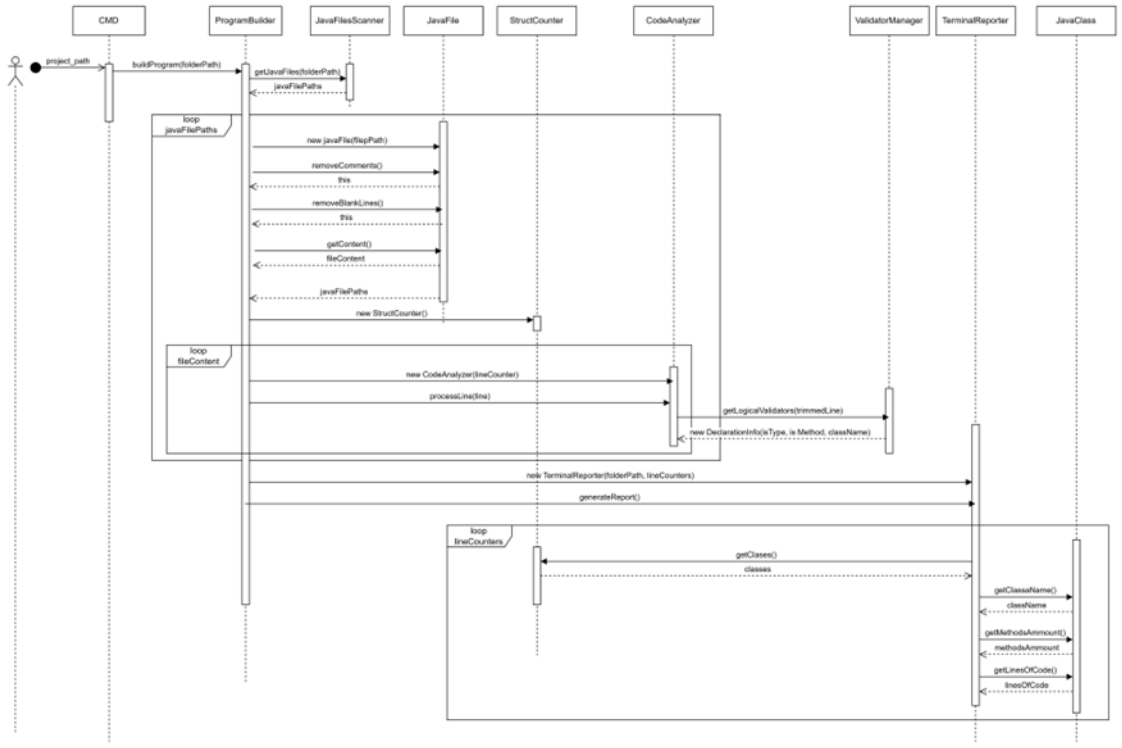

Imagen 3. Diagrama de secuencia del happy path del programa

La descripción de la secuencia del siguiente diagrama es el siguiente:

- El usuario introduce en la línea de comandos (CMD) la ruta de la carpeta del programa por analizar a través de la terminal o línea de comandos
- El ProgramBuilder recibe esta ruta y, con ayuda de JavaFilesScanner, obtiene las rutas de todos los archivos Java del proyecto.

- Con las rutas, iterando ruta-por-ruta, ProgramBuilder arma los objetos JavaFile.
    - ProgramBuilder utiliza los métodos removeComments(), removeBlankLines() y getContent() de la clase JavaFile, para obtener el contenido limpio de cada archivo Java y poder proceder con el conteo de líneas físicas y métodos por clase.
- Una vez con el contenido, ProgramBuilder crea un StructCounter.
- Con el StructCounter, ProgramBuilder crea una instancia de CodeAnalyzer pasando como parámetro en constructor de CodeAnalyzer el StructCounter que se creó en el paso anterior.
    - Este StructCounter nos servirá para procesar el contenido limpio que se extrajo de cada archivo java.
    - Para analizar el contenido, dentro de la clase ProgramBuilder se itera línea por línea y se utiliza el método “processLine” dentro del CodeAnalyzer que instanciamos en el paso anterior.
- El CodeAnalyzer recibe las líneas que iteramos una por una y utiliza un ValidatorManager para determinar si la línea es un método.
- Con esta información, ProgramBuilder crea un TerminalReporter el cual servirá para reportar los resultados del análisis.
- Una vez con la instancia de TerminalReporter, se llama al método generateReport() el cual generará el reporte que verá el usuario en la terminal.
    - Para esto, la clase TerminalCounter itera sobre sus lineCounters para obtener las clases que conforman el proyecto y con las que se realizará el conteo de líneas físicas y métodos. Por cada lineCounter, se obtienen sus clases.
    - Con estas clases, las cuales pertenecen al tipo JavaClass, se utiizan sus métodos de acceso a datos: getClassName(), getMethodsAmmount() y getLinesOfCode() para obtener los resultados que se mostrarán por clase al usuario.
    - Finalmente, TerminalReporter imprime los resultados en la terminal con un formato legible para que el usuario los pueda observar.

6.2 Comparación de dos versiones de un proyecto

Esta sección describe el flujo de los procesos que se llevan a cabo para a comparación de dos versiones de un proyecto, a través de la representación de un diagrama de secuencia

que permita identificar la interacción entre los elementos. Sin embargo, el diagrama solo representa el flujo a grandes rasgos para poder comprender la implementación.

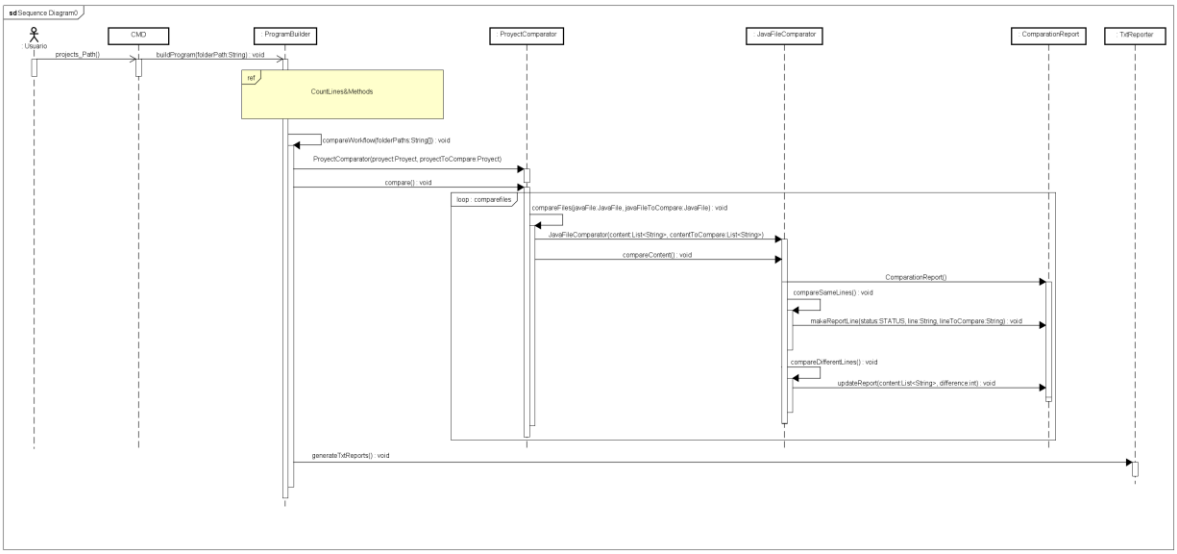

Imagen 4. Diagrama de secuencia de la comparación de dos versiones de un proyecto

La descripción de la secuencia del siguiente diagrama es el siguiente:

- El usuario introduce en la línea de comandos (CMD) las rutas de las carpetas de ambas versiones del programa por analizar a través de la terminal o línea de comandos
- El ProgramBuilder recibe esta ruta y realiza el conteo de líneas físicas y métodos por cada versión (Tal cual como se explica en el diagrama anterior).
- Una vez que termine el proceso anterior, ProgramBuilder crea genera dos instancias de Proyect a través del método compareWorkflow.
- Una vez construidos ambos proyectos, se instancia ProyectComparator(proyect, proyectToCompare) pasando ambos proyectos como argumentos.
- Con la creación de ProyectComparator, se invoca el método compare() para iniciar la comparación entre ambos proyectos.
- Para cada archivo .java equivalente entre ambos proyectos, se llama compareFiles(javaFile, javaFileToCompare).
    - Donde se construye el objeto JavaFileComparator().
    - Se invoca el método compareContent() que inicia la comparación línea por línea entre ambos archivos.
    - Se construye el objeto ComparationReport(),

- En JavaFileComparator se comparan las líneas iguales mediante compareSameLines(),
    - Donde se genera reportes con makeReportLine().
- Luego se comparan las líneas diferentes con compareDifferentLines().
    - Así mismo, se actualiza el reporte usando updateReport().
- Finalmente, con los reportes generados, se llama a generateTxReports() para escribir los resultados en archivos .txt dentro del directorio de salida proporcionado por el usuario.

7 CALIDAD

La arquitectura basada en capas contribuye significativamente a la calidad del sistema al fomentar la separación de responsabilidades entre las entidades que la conforman, contribuyendo a un diseño modular, de tal manera que permita la adición de nuevas funcionalidades sin impactar significativamente a la implementación realizada, así como la detección y corrección de fallos.