Facultad de Ciencias

CONSTRUCCIÓN DE UN PLUG IN PARA ANALIZAR TAREAS DE PROGRAMACIÓN EN EL IDE ECLIPSE

CONSTRUCTION OF A PLUG IN FOR PROGRAMMING TASKS ANALYSIS IN ECLIPSE IDE

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Leopoldo Quintana Díaz

Director: Rafael Duque Medina

Co-Director: Santos Bringas Tejero

MARZO– 2021

RESUMEN

La programación es una de las tareas más importantes en el desarrollo tecnológico actual, muchas veces este trabajo debe ser desarrollado en grupos que algunas veces son difícil de coordinar y llevar una memoria del trabajo realizado por cada programador. Esto se ve resaltado en ámbitos académicos donde el profesor no puede saber de manera sencilla el tiempo que ha tenido que dedicar un alumno por ejemplo a la hora de desarrollar una clase de la aplicación o la tasa de errores de compilación del alumno.

Para intentar dar una solución a estos problemas se ha construido un plug-in para el IDE Eclipse que permita monitorizar el trabajo de los programadores y extraer datos que sean interesantes de sus diferentes sesiones de trabajo. Unos ejemplos de datos importantes son el tiemplo empleado, número de líneas de código fuente generado, tasa de errores de compilación, entre otros. Para permitir que el evaluador pueda decidir qué datos quiere y que sea independiente de un lenguaje de programación se usan expresiones regulares, guardadas en un archivo JSON. El evaluado debe tener acceso a los datos calculados con este objetivo se implementó en el plug-in una conexión SSH que envía un archivo con los resultados. Por último, se ha desarrollado una aplicación que puede leer los archivos de resultados y generar representaciones gráficas a partir de los datos.

ABSTRACT

Programing is one of the most important tasks in technological development nowadays, a lot of times this work must be done by workgroups which sometimes are difficult to coordinate and to do a report of the work done by each programmer. This is clearly seem in an academic context where for the teacher is not easy to know the time spend by the student while doing a class of the application for examples or the compilation error rate of the student.

In order to give a solution to these problems a plug-in for the Eclipse IDE have been made that allows to monitor the programmers work and extract relevant data in each of their work sessions. Some examples of relevant data are the time spent, number of lines used in the source code, compilation error rate, among other. To allow that the reviewer could choose what data they want and that it will be independent of a programming language regular expression are used and are stored in a JSON file. The reviewer must have access to the calculated data, with that in mind a SSH connection that send a file with results was implemented in the plug-in. Lastly, it has developed an application which can parse the file with the results and generate graphic representation from the data.

INDICE

RESUMEN 1
ABSTRACT 2
INDICE 3
INDICE DE FIGURAS 4
1. INTRODUCCIÓN 5
1.1 Motivación 5
1.2 OBJETIVOS 6
1.3 ESTRUCTURA DEL DOCUMENTO 6
2. MATERIAL Y MÉTODOS 7
2.1 HERRAMIENTAS Y TECNOLOGÍAS 7
2.1.1 ECLIPSE IDE 7
2.1.2 JAVA 8
2.1.3 JFREECHART 8
2.1.4 JSON 8
2.2 METODOLOGÍA 9
3. ANÁLISIS DE REQUISITOS 11
3.1 HISTORIAS DE USUARIO 11
3.2 ORGANIZACIÓN TEMPORAL DEL PROYECTO 13
4. DISEÑO E IMPLEMENTACIÓN 15
4.1 DISEÑO E IMPLEMENTACIÓN DEL REPOSITORIO DE MÉTRICAS JSON 15
4.2 DISEÑO E IMPLEMENTACIÓN DEL MOTOR PARA CALCULAR MÉTRICAS 17
4.3 DISEÑO E IMPLEMENTACIÓN DE LAS INTERFACES DE USUARIO 23
5. EVALUACIÓN Y PRUEBAS 30
5.1 PRUEBAS SOBRE EL PLUG-IN 30
5.2 PRUEBAS JUNIT 32
6. CONCLUSIONES 34
6.1 ANÁLISIS DEL CUMPLIMIENTO DE LOS OBJETIVOS 34
6.2 TRABAJOS FUTUROS 35
ANEXO I. ACRÓNIMOS 36
REFERENCIAS 37

INDICE DE FIGURAS

Figura 1. Ejemplo de sintaxis JSON 9
Figura 2. Formato JSON del fichero de entrada de las métricas 15
Figura 3. Formato JSON en el que son almacenadas las métricas 16
Figura 4. Conexión SSH 16
Figura 5. Diagrama UML de las clases Métricas 17
Figura 6. Diagrama de Interacciones UML 18
Figura 7. Código del constructor de LectorJSON 19
Figura 8. Método parseMetricaObject() de LectorJSON 19
Figura 9. Dos tipos de constructores dependiendo del tipo de métrica 20
Figura 10. Inicio del hilo 21
Figura 11. Calculo de métricas de código fuente 21
Figura 12. ConsoleListener 22
Figura 13. Fin del hilo y escritura de los resultados 23
Figura 14. Modelo de Navegación entre interfaces de usuario 24
Figura 15. Interfaz de inicio del plug-in 24
Figura 16. Botones del plug-in en la barra de herramientas 25
Figura 17. Interfaz conexión SSH 26
Figura 18. Clase Main de la aplicación 27
Figura 19. Constructor de la clase que permite leer el archivo JSON 27
Figura 20. Método que traducen la información del JSON 28
Figura 21. Interfaz de muestra de gráficas 29
Figura 22. Clase donde se escriben los algoritmos 32
Figura 23. Clase que prueba los algoritmos 33

1. INTRODUCCIÓN

En este capítulo se incluyen tres apartados: motivación, objetivos y estructura del documento. En el primer apartado titulado motivación se tratará la importancia de la programación de computadores en el mundo actual, qué es Eclipse y se justificará la creación de un plug-in para extraer las métricas que cuantifiquen la labor que realiza el programador. En el siguiente apartado se concretarán los objetivos de este trabajo. En el tercer apartado se explicará de forma resumida la estructura que seguirá este documento.

1.1 Motivación

La actividad conocida como "Programación de Computadores" consiste en escribir la lista de instrucciones que un computador debe ejecutar. Esta lista de instrucciones ordenada se denomina programa. En términos coloquiales, los programas suelen pedir algunos datos al usuario, procesar estos datos y devolver un resultado [1]. La programación de aplicaciones software ha permitido que muchos trabajos que antes se desarrollaban de forma manual ahora se puedan realizar de forma automatizada, por esto es uno de los pilares del desarrollo tecnológico actual. La mayoría de la tecnología que usamos hoy en día se basa en la programación. En este siglo el software forma parte de todos los dispositivos que requieren manipular información y que las personas utilizan en sus actividades cotidianas. La importancia de la industria del software a nivel económico radica en el respaldo de la operatividad y estabilidad que otorga a otros sectores industriales [3]. Con estos datos se puede concluir la importancia de la programación en el mundo laboral.

En el ámbito educativo aprender a programar es básico en muchas carreras relacionadas con las tecnologías, muchas son abandonadas por un alto número de alumnos en los primeros años. La labor de aplicar conceptos básicos o crear algoritmos sencillos se hace una tarea difícil para muchos alumnos [4]. Teniendo en mente estas dificultades se plantea este proyecto para poder facilitar la enseñanza de la programación. Para llevarlo a cabo se calcularán unas métricas cuyos valores permitan conocer las características de cómo el alumno se enfrenta a tareas de programación. Con este fin se buscarán dos tipos diferenciados de métricas, unas que nos permitan caracterizar el programa generado y otras que nos informan de la forma de trabajar del programador (por ejemplo, si el programador deja todas las pruebas del código para el final del trabajo, si documenta el código con comentarios en todo momento, etc.).

Uno de los IDE más utilizados para programar y que motiva la realización de este proyecto es Eclipse, el cual fue creado como una plataforma para herramientas plug-in, las cuales son pequeños programas que extienden las capacidades del IDE, además de añadirle funcionalidades y hacer que pueda funcionar con numerosos lenguajes de programación y aplicaciones. Cualquiera puede escribir estos programas que extienden Eclipse y hacer que trabajen directamente con cualquier otro plug-in, pues es de código abierto. Esto supone que es un tipo de software que gracias a su licencia cualquiera puede modificar y extender al tener completamente disponible su código fuente. Gracias a esto hay un gran número de plug-in de terceros que funcionan entre sí, lo que hace a Eclipse muy popular [6]. Algunos otros IDEs limitan los plug-ins a socios de la empresa. Este proyecto se plantea gracias a estas capacidades de Eclipse para crear plug-ins de modo que se pueda crear una herramienta que extraiga y calcule métricas sobre el trabajo que realiza el programador ya sea sobre el código fuente o sobre la salida de consola.

1.2 OBJETIVOS

El objetivo principal de este proyecto es desarrollar un plug-in para Eclipse IDE que permita monitorizar el trabajo de estudiantes en sus tareas de programación y calcular métricas que permitan al profesor cuantificar distintos aspectos de sus sesiones de trabajo. El plug-in mostrará una representación gráfica de los valores calculados para todas las métricas de forma que pueda ser de utilidad en ámbitos académicos para que los profesores tengan una idea de cómo los alumnos llevan a cabo las tareas de programación con el IDE. Los subobjetivos serán los siguientes:

- **Crear diferentes métricas.** Para crear métricas solo se deberá especificar una expresión regular y donde se deberá buscar esta, en código fuente o consola.
- **Que el profesor pueda crear sus propias métricas.** Se le dará la posibilidad al evaluador de generar sus propias métricas y de modificar las que ya existen para que se adecue al trabajo que deberán realizar sus alumnos.
- **Que las métricas puedan cuantificar el proceso de trabajo y el producto final del mismo.** Gracias a la flexibilidad de las expresiones regulares y de poder buscarlas tanto en consola como en el código fuente, se proporciona la capacidad del cálculo de un gran número de métricas que otorgan diferente información sobre el trabajo y el producto realizado por el programador.
- **Comunicación alumno y profesor.** El profesor necesitará conseguir acceso a los resultados del cálculo de métricas del alumno, por lo que se deberá de crear algún tipo de conexión entre los dos.
- **Representación gráfica de los valores de las métricas.** Los valores de las métricas se podrán visualizar a través de gráficas dependientes del tiempo.
- **Guardar un repositorio con los valores de las métricas para cada uno de los usuarios.** Cuando se terminen de calcular los valores de las métricas se podrán subir a un servidor donde se almacenarán los resultados para que el evaluador los pueda visualizar más adelante.

1.3 ESTRUCTURA DEL DOCUMENTO

Esta memoria incluye cinco capítulos adicionales, cuya estructura y contenidos será:

- **Capítulo 2. Material y método:** En este capítulo se tratarán las diferentes tecnologías y herramientas que se utilizarán para realizar este proyecto.
- **Capítulo 3. Análisis de requisitos:** En este capítulo se expondrán los requisitos funcionales y no funcionales.
- **Capítulo 4. Diseño e implementación:** En este capítulo se muestra como se ha diseñado e implementado el plug-in.
- **Capítulo 5. Evaluación y pruebas:** En este apartado serán descritas las diferentes pruebas realizadas para evaluar el plug-in.
- **Capítulo 6. Conclusiones y trabajos futuros:** Por último, se establecerán las conclusiones extraídas del desarrollo del plug-in y futuras líneas de trabajo a partir de este proyecto.

2. MATERIAL Y MÉTODOS

En este capítulo se describen las diferentes herramientas, tecnologías y metodologías utilizadas en la realización de proyecto. El primer apartado tratará de las herramientas y tecnologías y el segundo de la metodología seguida en el proyecto.

2.1 HERRAMIENTAS Y TECNOLOGÍAS

En este punto se tratarán las herramientas y tecnologías usadas y se explicará en qué consisten, algunas de estas son el lenguaje de programación Java y el IDE Eclipse.

2.1.1 ECLIPSE IDE

Una de las herramientas básicas de la programación son los IDE, los cuales son paquetes de software que proporcionan soporte a escribir código, testear y debuggear, y usualmente algunas características más enfocadas al desarrollo de software [17]. Un IDE consiste en una metodología de desarrollo completa y unificada y un conjunto de ayudas informáticas que respaldan el uso de la metodología [2].

El Proyecto Eclipse se compone de cinco subproyectos: platform, Java development tools (JDT), Plug-in Development Environment (PDE), e4 y Orion [8]. En este proyecto se usará Platform, JDT y PDE. La plataforma Eclipse está estructurada en subsistemas los cuales son uno o más plug-ins. El subsistema más importante del IDE Eclipse es el Workbench, el entorno de desarrollo de escritorio. El Workbench tiene como objetivo lograr una integración perfecta de las herramientas y un paradigma común para la creación, gestión y navegación de los recursos del espacio de trabajo. Cada ventana del Workbench contiene una o más perspectivas. Las perspectivas contienen vistas y editores [5]. Algunas de las ventajas que proporciona el uso de Eclipse sobre otros IDEs son bajos costes, rápido desarrollo e innovación y arquitectura elegante y amigable. La plataforma de desarrollo Eclipse es completamente gratuita y es código abierto, por lo que los desarrolladores tienen acceso al código fuente y pueden modificar o ampliarlo rápidamente para satisfacer las necesidades del usuario.

Platform define al conjunto de frameworks y servicios comunes que forman el Eclipse IDE. La JDT provee al Platform de plug-ins para implementar un Java IDE que permite el desarrollo de cualquier aplicación Java incluidos los plug-ins para Eclipse. Por último, el PDE provee vistas y editores que hacen más sencillo el desarrollo de plug-ins para eclipse [9].

El Plug-in Development Environment provee herramientas para crear, desarrollar, testear, debuggear y probar plug-ins. El PDE a su vez tiene tres componentes UI, Build y API Tools. El PDE UI tiene las herramientas que proveen las funcionalidades enunciadas anteriormente además de editores para crear archivos de manifiesto, herramientas de conversión de proyecto java a plug-in, asistentes que permiten exportarlos e importarlos y asistentes para crear nuevos. El PDE Build facilita la automatización de los procesos de construcción de plug-ins, en resumen, crea un script que busca proyectos relevantes a partir de la información proporcionada por diferentes archivos del plug-in, el objetivo de este script es poder enviar un archivo con los proyectos necesarios para construir el plug-in de una forma sencilla. Por último, el PDE API Tools proporciona herramientas a los desarrolladores para ayudar a mantener el plug-in [10].

Una de las características más importantes del PDE es su asistente que facilita la tarea de crear el plug-in, a partir de este asistente se generan todos los archivos necesarios para empezar a crear el plug-in además se tiene la opción de seleccionar diferentes modelos básicos de plug-in como uno que crea vistas, uno básico que escribe un mensaje de Hello World o un editor. Al crear el plug-in se generan los archivos de manifiesto que servirán para añadir otros

plug-ins, APIs o librerías que se vayan a utilizar a la hora de crearlo. Además, se permitirá probar el plug-in creando una nueva aplicación Eclipse en la que se puede ejecutar, probar y depurarlo, cuando ya está creado usando el asistente el siguiente paso sería ir programando en el execute usando Java las funcionalidades que se quieran en el plug-in, por ejemplo, crear una ventana a la que pasarle datos. Para probarlo sería tan sencillo como pulsar "run as an Eclipse application" en el desplegar que aparece en el botón de ejecutar de la barra de herramientas y entonces se abrirá una nueva aplicación de Eclipse en la que su barra de herramientas habrá un icono de eclipse y si se pulsa se ejecutará el plug-in.

A la hora de crear el plug-in y que este tenga acceso a la consola se utilizará el paquete ui.console de Java con el cual se podrá leer las consolas existentes y poder buscar texto en su contenido. Para realizar las búsquedas dependerán de los ConsoleListeners unos objetos que recibirán el patrón o texto a buscar, para esto este objeto irá leyendo todas las consolas que se vayan generando desde que se creó y leyendo su contenido el cual será analizado con el patrón y se guardará el número de ocurrencias de este. Usando este método se podrá tener acceso a métricas como el número de errores o las veces que ha sido ejecutado el código. Para conseguir todas las métricas que se quieren en este proyecto se necesitará tener acceso al código fuente, para esto se utilizarán diferentes APIs como el IWorkbench por ejemplo o IWorkbenchPage. Primero, será necesario tener acceso al workbench y a partir de este conseguir guardar en una variable la ventana activa actual y la página. Después de estos pasos se puede acceder a todos los editores de código existentes y parsearlos en strings en los que se podrá realizar búsquedas de expresiones regulares que permitan, por ejemplo, calcular el número de clases o métodos actualmente en el código.

2.1.2 JAVA

El lenguaje de programación en el que se ha realizado este proyecto es Java, este es un lenguaje de propósito general que sirve tanto para desarrollar aplicaciones como de escritorio o móviles entre otros. Java es un lenguaje de programación orientado a objetos, esto son instancias [16] de una clase que comparten campos y métodos. Las operaciones en los lenguajes orientados a objetos se basan en estas instancias. Las clases a su vez son una combinación de atributos y método los cuales adquieren valor al crear los objetos.

2.1.3 JFREECHART

JFreechart, que se puede conseguir en [15], es una librería gratuita que permite la creación e integración en aplicaciones de gráficos a la vez de poder generarlos en diferentes tipos de formatos como un componente de java swing o en un formato de imagen JPG o PNG [15]. Esta librería nos permite generar gráficos de líneas, de barras o circulares. La creación de estos gráficos se basa en asignar un dataset con los diferentes valores para ejes x e y, el cual se pasará de argumento al gráfico junto al título y los títulos de los diferentes ejes.

2.1.4 JSON

JSON (JavaScript Object Notation), cuya página oficial es [7], es un formato ligero de intercambio de datos, el tamaño de los archivos es poco pesado, incluso más que por ejemplo XML. Es fácil de escribir y leer por humanos y de analizar y generar por máquinas. JSON es un formato de texto completamente independiente de cualquier lenguaje de programación, pero está basado en la sintaxis de objetos de JavaScript. JSON se construye sobre las siguientes dos estructuras [7]:

- Una lista ordenada de valores. En la mayoría de los lenguajes se conoce como array o lista.

- Una colección de parejas nombre/valor. En diferentes lenguajes estos aparecen de varias formas como objeto o diccionario.

La sintaxis de JSON se compone de duplas nombre y valor como {“regex”: “Exception in thread”} en que el nombre o llave siempre será de formato string pero el valor puede ser string, un número, otro objeto JSON, un array o un booleano. Mientras que los objetos JSON son listas de duplas separadas por comas y entre llaves, los arrays son listas de diferentes valores separados por comas y entre corchetes. Un ejemplo de sintaxis JSON es la Figura 1 en la que se pueden observar diferentes tipos de ejemplo de objetos de JSON, el primero “array” es una lista de números, luego hay varias duplas compuestas por un string y por diferentes tipos de valor como un número, otro string, valor nulo o un booleano. Por último, el ejemplo object es un ejemplo de objeto formado a partir de otros objetos.

```
{
  "array": [
    1,
    2,
    3
  ],
  "boolean": true,
  "color": "gold",
  "null": null,
  "number": 123,
  "object": {
    "a": "b",
    "c": "d"
  },
  "string": "Hello World"
}
```

Figura 1. Ejemplo de sintaxis JSON

2.2 METODOLOGÍA

La metodología que se usará en este proyecto para análisis de requisitos y para la planificación del proyecto y trabajo es scrum, una metodología ágil. El manifiesto ágil surge en 2001 por un grupo de diecisiete representantes de diferentes empresas del desarrollo de software por la necesidad de una alternativa a los desarrollos de software pesados [11]. Las metodologías basadas en el manifiesto Ágil presentan algunas ventajas como adaptarse mejor a los cambios de requisitos del cliente.

El manifiesto ágil se sustenta en cuatro valores [12]:

- Valoramos más a los individuos y su interacción que a los procesos y las herramientas.
- Valoramos más el software que funciona que la documentación exhaustiva.
- Valoramos más la colaboración con el cliente que la negociación contractual.
- Valoramos más la respuesta al cambio que el seguimiento de un plan.

Dentro de los diferentes métodos de desarrollo ágil en este proyecto se intentará seguir scrum, el cual es un marco de trabajo que ayuda a las personas, equipos y organizaciones a crear productos a partir de soluciones adaptativas para problemas complejos [18], aun cuando algunas de sus características no se pueden llevar a cabo en este tipo de proyecto debido a ser trabajo individual. Las cuatro características principales de scrum son [13]:

- Equipos autónomos y autoorganizados que comparten su conocimiento de forma abierta y aprenden juntos.
- Una estrategia de desarrollo incremental, en lugar de la planificación completa del producto.
- Basar la calidad del resultado en el conocimiento tácito de las personas y su creatividad; no en la calidad de los procesos empleados.
- Solapar las diferentes fases del desarrollo, en lugar de realizarlas una tras otra en un ciclo secuencial o “de cascada”.

En la realización de este proyecto se sigue un desarrollo incremental, empezando en lo básico y siguiendo un desarrollo basado en un incremento continuo añadiendo ideas al plug-in que se especifican en reuniones semanales entre el autor y director y co-director de este proyecto. En estas reuniones se plantea el trabajo a realizar y objetivos a conseguir en la siguiente semana.

3. ANÁLISIS DE REQUISITOS

En este capítulo se hablará de las historias de usuario que será el primer apartado y de la organización temporal del proyecto que será el segundo. El primero nos servirá como documento de requisitos y el segundo apartado permitirá visualizar como se ha ido realizando el trabajo en el tiempo durante este proyecto.

3.1 HISTORIAS DE USUARIO

Para facilitar la tarea de analizar los requisitos de este proyecto se usarán historias de usuario basadas en las de scrum, las cuales se pueden desarrollar de manera más rápida que un documento de análisis de requisitos normal y se pueden generar a medida que surgen nuevas funcionalidades durante el desarrollo. Las historias de usuario son descripciones sencillas de las funcionalidades que se implementarán y de los criterios que nos servirán para aceptarlas.

Las historias de usuario usadas en scrum pueden sustituir a un documento de requisitos. Se basan en descripciones cortas y simples de características escritas desde el punto de vista de la persona que las quiere. Una de las ventajas de escribir historias de usuario es que se puede mantener un diferente grado de detalle, algunas englobando una mayor o menor funcionalidad. Estas historias se componen de una descripción y de los criterios de aceptación de éstas [14].

Las historias de usuario que se usarán en este proyecto seguirán una estructura compuesta por:

- Un identificador de la historia de usuario y un nombre que la describa.
- Una descripción de en lo que consiste esta historia y una pequeña explicación de lo que debería hacer el plug-in para llevar a cabo esta historia.
- Una condición o prueba de aceptación que nos permitirá saber si el plug-in resuelve esta historia de usuario de una forma satisfactoria.

Identificador	HU1
Nombre	Identificación de usuario
Descripción	Las métricas que se extraen dependen de cada usuario por lo que es necesario que el sistema guarde un identificador de usuario que nos permita saber a quién pertenecen las métricas extraídas. Para esto se creará una ventana al iniciar el plug-in que permita introducir el ID de usuario, no se dejará realizar ninguna acción sobre eclipse mientras no se introduzca uno válido y si se introduce un identificador incorrecto aparecerá un mensaje en la ventana de aviso y se volverá a pedir otro ID.
Criterio de aceptación	Existirá un formulario donde se introduzca, aparte de otros campos como el intervalo de tiempo donde ocurrirá cada cálculo de métricas o el archivo donde estén todas las expresiones regulares, el identificador que nos permitirá distinguir el usuario del cual se calcularán las métricas.

Identificador	HU2
Nombre	Calcular métricas del proceso
Descripción	Para saber cómo realiza sus tareas el programador es necesario tener unas métricas que den información sobre esto. Algunos ejemplos de métricas, podría ser el número de líneas de código fuente que escribe por minuto, el número de pruebas o de comentarios en el código.
Criterio de aceptación	Para métricas de proceso, en el mismo formulario donde se introduce el identificador, y cada cuanto intervalo de tiempo se deben calcular, el plug-in pedirá una lista con las métricas a buscar las cuales estarán especificadas mediante expresiones regulares en un archivo JSON, en el que además estará incluido su nombre, tipo y una sencilla descripción.

Identificador	HU3
Nombre	Calcular métricas de la ejecución de pruebas
Descripción	A la hora de conocer en qué forma ha realizado su trabajo el programador una parte importante es conocer los resultados de las pruebas. Por ello, se pretende disponer de métricas que analicen los resultados de las pruebas que hacer el estudiante durante su proceso de trabajo. Para llevar a cabo esta tarea se usará JUnit que es una librería que permite realizar de forma sencilla diferentes test o pruebas sobre los métodos y clases Java.
Criterio de aceptación	Se utilizará de forma similar a las métricas de consola puesto que los resultados de JUnit son presentados en una ventana que es considerada otra consola más. Para buscar métricas de este tipo habría que hacerlo de forma similar a las de consola, por ejemplo, para errores sería buscando como expresión regular el mensaje de error que debería devolver el test de Junit que se quiera contar.

Identificador	HU4
Nombre	Calcular métricas de los procesos de compilación fallidos
Descripción	Se pretende generar información sobre el tipo de problemas que encuentra el estudiante a la hora de compilar sus programas. Para ello será necesario acceder a la salida de la consola durante su proceso de trabajo y generar métricas descriptivas de los errores de compilación.
Criterio de aceptación	Para métricas de consola, en el mismo formulario donde se introduce el identificador el plug-in se recibirá el archivo JSON con las expresiones regulares, tipos de métricas, nombre y descripciones de cada una. Después se buscará en todas las consolas generadas la expresión regular y se llevará la cuenta de cuantas veces aparece.

Identificador	HU5
Nombre	Visualización gráfica de los valores de las métricas
Descripción	Para facilitar la visualización de los resultados de las métricas se deberán poder crear unas gráficas que permitan ver el progreso o como varían los

	valores de estas. La creación de estas permitirá a su vez comparar métricas del mismo usuario o de diferentes usuarios.
Criterio de aceptación	Se crea una aplicación independiente del plug-in que presenta un formulario por el cual se le podrá pasar un archivo JSON con los resultados calculados. Después, se generará otro formulario con una lista desplegable en la que se podrá seleccionar las métricas de las que se crearán las gráficas, estas serán diagramas de barras en las que el eje x se representará el intervalo de tiempo en el que se han calculado las métricas y el eje y será el valor de la métrica en ese momento.

Identificador	HU6
Nombre	Almacenamiento de resultados
Descripción	Los resultados de cada usuario deberán de ser almacenados en archivos JSON en los cuales se deberá encontrar los resultados de todas las métricas buscadas, con su descripción y el tiempo en el que han sido calculadas, además del identificador del usuario del cual se han calculado.
Criterio de aceptación	Las métricas se irán calculando según intervalos de tiempo automáticamente, cada vez que ocurra esto se irá escribiendo el archivo JSON actualizando los resultados añadiendo las métricas con su resultado calculado, su identificador, nombre y descripción.

Identificador	HU7
Nombre	Envío de resultados
Descripción	El evaluador necesitará tener acceso a los valores de las métricas de cada sesión de trabajo del alumno para poder analizarlos con la aplicación que crea para generar gráficas. Para esto se deberá implementar una funcionalidad que envíe el archivo JSON con esta información y que el profesor lo pueda recibir.
Criterio de aceptación	Se generará una funcionalidad accesible a través de un nuevo botón en la barra de herramientas de Eclipse que lanzará un proceso para procesar el archivo de JSON con los valores de las métricas y mediante una conexión SSH lo subirá a un servidor donde el profesor tenga acceso a él.

Identificador	HU8
Nombre	Plug-in Eclipse
Descripción	Todas las funcionalidades deben estar disponibles a través de un plug-in de Eclipse de modo que puedan estar integradas en este IDE y puedan monitorizar y cuantificar el trabajo del programador mientras desarrolla su actividad de forma habitual.
Criterio de aceptación	Se puede importar el plug-in correctamente en el ordenador del alumno y devolver unas métricas correctas. Para importar el plug-in desde Eclipse se deberá seleccionar Añadir nuevo software en el apartado de ayuda de la barra de tareas, seleccionar añadir y elegir el archivo del plug-in, otra forma de realizar esto será añadiendo el archivo del plug-in a la carpeta de eclipse/plugins en el directorio de Eclipse.

3.2 ORGANIZACIÓN TEMPORAL DEL PROYECTO

En este apartado se tratará el orden de realización de las historias de usuario, su prioridad para ser realizadas y si se han completado o siguen pendientes. Para esto se creará la Tabla 1 que representa el product backlog de historias de usuario basado en scrum, lo que es

una lista ordenada de los elementos necesarios para ir mejorando el producto y conseguir el objetivo [18], en esta tabla se recopilan los requisitos iniciales del proyecto además de su estado, prioridad, orden y output.

HISTORIAS DE USUARIO	PRIORIDAD	ORDEN	ESTADO	OUTPUT
IDENTIFICACIÓN	5	1	Completado	El plug-in almacenará la ID de usuario
MÉTRICAS DEL PROCESO	2	3	Completado	Las métricas del proceso se irán calculando y guardando
MÉTRICAS DE EJECUCIÓN	3	2	Completado	Las métricas de ejecución se irán calculando y guardando
MÉTRICAS DE CONSOLA	3	2	Completado	Las métricas de consola se irán calculando y guardando
VISUALIZACIÓN DE RESULTADOS	4	5	Completado	Una aplicación que recibe el archivo JSON de métricas y te visualizar sus datos en forma de gráfica
ALMACENAMIENTO DE RESULTADOS	6	4	Completado	Todas las métricas calculadas se almacenarán en un archivo JSON
ENVIO DE RESULTADOS	7	6	Completado	Mediante un protocolo SSH el evaluador recibirá el JSON
PLUG-IN ECLIPSE	1	7	Completado	Un JAR con el plug-in exportado que se pueda importar en eclipse para poder utilizarlo

Tabla 1. Product Backlog

La Tabla 2 presentará un diagrama de Gantt que permitirá ver el tiempo aproximado que ha ocupado cada historia de usuario. El tiempo será dividido en semanas de trabajo. Las historias de usuario se irán realizando según su prioridad empezando con las más prioritarias y acabando por las de menos.

Historia de Usuario	Semanas																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Identificación																		
Métricas del proceso																		
Métricas de Ejecución																		
Métricas de Consola																		
Visualización de Resultados																		
Almacenamiento de Resultados																		
Envío de Resultados																		
Plug-in Eclipse																		

Tabla 2. Diagrama de Gantt

4. DISEÑO E IMPLEMENTACIÓN

En este capítulo se tratará el trabajo de diseño e implementación para realizar el plug-in. Primero se expondrá las interacciones básicas que deberá realizar el plug-in y el diseño de este. Por último, se tratará la implementación del sistema en el que se darán a conocer los diferentes algoritmos y técnicas utilizados para recrear las diferentes funcionalidades que se esperan.

4.1 DISEÑO E IMPLEMENTACIÓN DEL REPOSITORIO DE MÉTRICAS JSON

Este apartado expondrá el diseño e implementación seguidos para la utilización de archivos JSON y como serán enviados al evaluador. Los diferentes archivos que llevan información como el inicial donde se reciben las características las métricas que hay que calcular o en el que se escriben los resultados de dichos cálculos seguirán un formato de tipo JSON y para transmitir estos archivos se usará una conexión SSH, el plug-in dará la posibilidad de elegir donde se va a enviar el archivo y que archivo.

El archivo JSON de entrada representado en la Figura 2 seguirá un diseño basado en una lista de métricas que son un objeto que está formado por cinco atributos: la expresión regular la cual es indispensable ya que es ella la que le dice al plug-in que es lo que debe buscar, una pequeña descripción que facilita el entendimiento del objetivo detrás de esa expresión regular, un número que permita distinguir si se buscará en consola o en el código fuente, un identificador que diferencia una métrica de otro y el nombre de esta.


```
[
  {
    "metrica":{
      "regex":"Exception in thread",
      "descripcion":"Errores",
      "tipoMetrica":"0"
    }
  },
  {
    "metrica":{
      "regex":"\\s*(public|private)\\s+class\\s+(\\w+)\\s+((extends\\s+\\w+)|(implements\\s+\\w+( ,\\w+)*))?\\s*\\{",
      "descripcion":"Clases",
      "tipoMetrica":"1"
    }
  }
]
```


Figura 2. Formato JSON del fichero de entrada de las métricas

El archivo JSON de salida se compondrá de una lista formada por objetos como el de la Figura 3, estos objetos se formarán a su vez por una lista de métricas de consola y otra de código además del tiempo en el que fue calculado, el ID de usuario, el nombre de la métrica, su identificador y una pequeña descripción.

```
[
  {
    "Metricas Consola":[
      {
        "Description":"Errores",
        "Value":0
      }
    ],
    "Time":4.544E-4,
    "Metricas Codigo Fuente":[
      {
        "Description":"Clases",
        "Value":"3"
      }
    ],
    "ID":"1"
  }
]
```


Figura 3. Formato JSON en el que son almacenadas las métricas

Para poder enviar los datos se usará el protocolo SSH, para implementar esta funcionalidad se añadió un botón en las herramientas de eclipse que llama a un nuevo handler. Al principio, se lanza un JFrame en el cual se reciben los datos necesarios para realizar una conexión SSH, el nombre de usuario, el host, el puerto y la contraseña. Otro dato será la dirección el archivo a enviar el cual estará ya rellenado si estado usando el cálculo de métricas, si no, se puede elegir usando un botón que lo acompaña. Como se ve en la Figura 4 se inicializa el conector SSH gracias a los datos de la interfaz, se envía el archivo que este en la dirección guardada usando el método whenUploadFileUsingJsch_thenSuccess cambiándole el nombre a resultados.json, se cierra la conexión y también la interfaz que se había creado para realizar este envío de datos.

```
Frame frame=new Frame();


frame.setResizable(false);
frame.setLocation(0,0);
frame.setExtendedState(JFrame.MAXIMIZED_BOTH);
File abre=null;
String direccion="";
SSHConnector sshConnector = new SSHConnector();
try {
    sshConnector.connect(USERNAME, PASSWORD, HOST, Integer.parseInt(PORT));


    direccion=campoDireccion.getText();

} catch (NumberFormatException e1) {
    // TODO Auto-generated catch block
    JOptionPane.showMessageDialog(frame, "ERROR DE FORMATO DE NUMEROS");
    e1.printStackTrace();
} catch (IllegalAccessException e1) {
    // TODO Auto-generated catch block
    JOptionPane.showMessageDialog(frame, "ERROR DE CONEXIÓN");
    e1.printStackTrace();
} catch (JSchException e1) {
    // TODO Auto-generated catch block
    JOptionPane.showMessageDialog(frame, "ERROR AL INTRODUCIR LOS DATOS");
    e1.printStackTrace();
}

try {



    sshConnector.whenUploadFileUsingJsch_thenSuccess(direccion, "resultados.json");
    sshConnector.disconnect();

    frameInicial.dispose();
```

Figura 4. Conexión SSH

4.2 DISEÑO E IMPLEMENTACIÓN DEL MOTOR PARA CALCULAR MÉTRICAS

Se creará una métrica para cada dupla de descripción y expresión regular, como base para su creación se generó una clase abstracta métrica la cual implementaba los métodos y atributos básicos que tendrían todos los tipos ya sea métrica de código o de consola. Como se ve en el siguiente diagrama de clases UML de la Figura 5 se ve como las clases de métrica de consola o código fuente extienden la clase Metrica original y la adaptan para realizar el cálculo necesario o para recibir algún objeto necesario como el ConsoleListener.

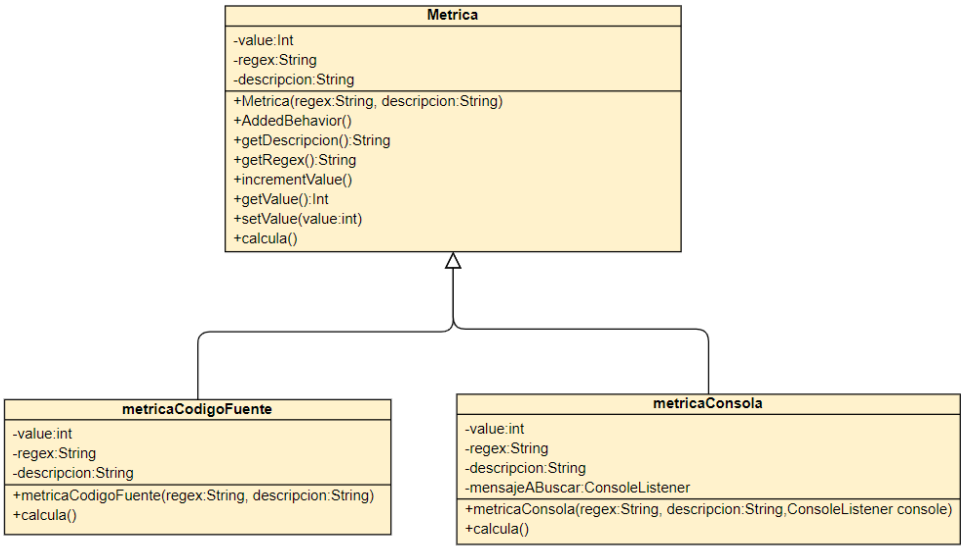

Figura 5. Diagrama UML de las clases Métricas

La implementación del plug-in empieza por presentar una ventana donde se puede introducir el ID de usuario, la ubicación del archivo JSON donde vienen las métricas a buscar y el intervalo de tiempo. Para realizar esto, se utiliza la librería swing de Java que nos permite crear interfaces gráficas sencillas, después de introducir estos datos se deberá usar el botón de Enviar Datos para que se active un event listener en el que el plug-in recibe los datos y llama a una clase lectorJSON que nos permite traducir ese archivo de JSON en atributos y variables que puedan ser usadas.

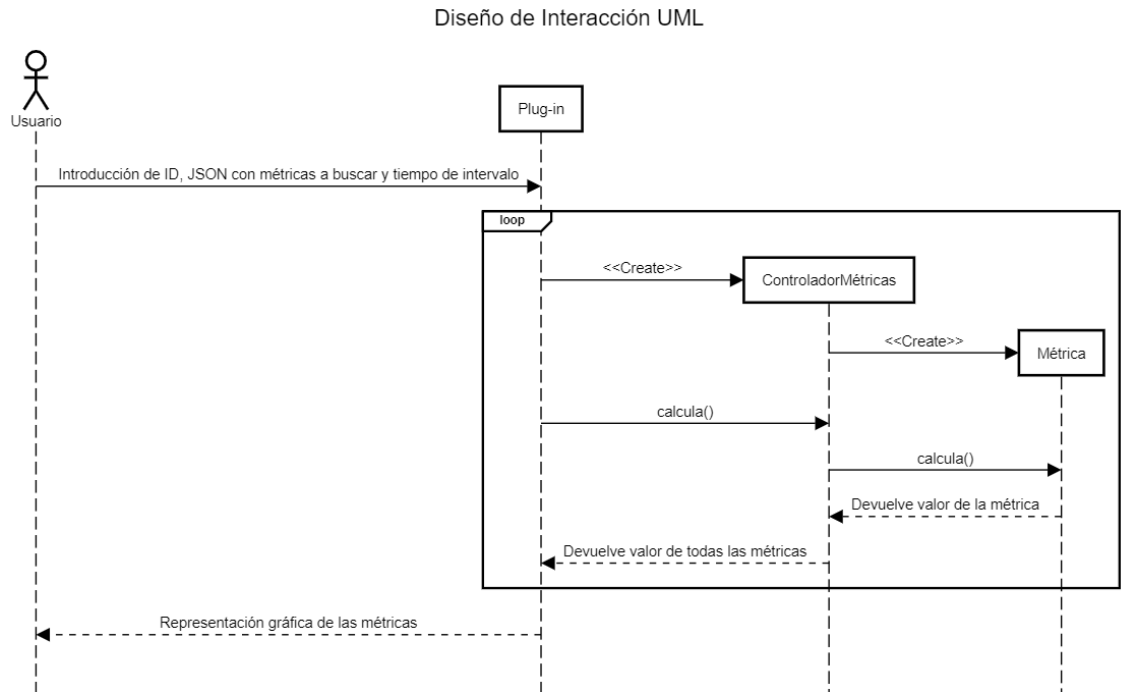

Figura 6. Diagrama de Interacciones UML

Como se ve en la Figura 6 el usuario deberá introducir diferentes argumentos, como el ID del programador y la ubicación del JSON con las diferentes métricas a buscar, además de un intervalo de tiempo que indicará cada cuanto tiempo se deberá realizar esta búsqueda. El plug-in lo que hará será generar unos controladores que permiten agrupar las métricas y no tener que trabajar con ellas de forma directa una a una. A este controlador se le irá mandado cada intervalo que calcule todas las métricas por lo que llamará a todas las métricas para que generen su valor y luego lo retornen cuando sea necesario. Cada intervalo el controlador devolverá el valor de todas las métricas al plug-in y cuando sea requerido el plug-in generará una representación gráfica de las métricas especificadas y su evolución en el tiempo.

Antes de empezar el cálculo de las métricas la clase lectorJSON de la Figura 7 primero recibirá una dirección donde deberá estar un archivo JSON que contenga para cada métrica la expresión regular que busca, una breve descripción y si es una métrica que se busque en código fuente o consola representado por un identificador numérico. Para esto se llamará a un método parseMetricaObject que identifique cada campo para cada métrica y lo traduzca.

```
public LectorJSON(String direccion) {
    JSONParser jsonParser = new JSONParser();

    try (FileReader reader = new FileReader(direccion))
    {
        //Read JSON file
        Object obj = jsonParser.parse(reader);
        JSONArray listaMetricas = (JSONArray) obj;
        listaMetricas.forEach( emp -> parseMetricaObject( (JSONObject) emp ) );

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (ParseException e) {
        e.printStackTrace();
    }
}
```

Figura 7. Código del constructor de LectorJSON

Después existirá un método parseMetricaObject(), Figura 8, que se encarga de leer las métricas y de dividir sus expresiones regulares, descripciones, nombres e identificadores según el tipo que sean, añadiendo cada una en arrays diferentes. Para las métricas que son buscadas en consola se crea a su vez un consoleListener que es un objeto indispensable que nos permitirá ir buscando en la consola las diferentes expresiones regulares y que se guarda también en una lista.

```
private void parseMetricaObject(JSONObject metrica) {

    JSONObject metricaObject = (JSONObject) metrica.get("metrica");
    String regex = (String) metricaObject.get("regex");
    String descripcion = (String) metricaObject.get("descripcion");
    String nombre = (String) metricaObject.get("nombre");
    String tipoMetrica =  (String) metricaObject.get("tipoMetrica");
    String id =(String) metricaObject.get("idMetrica");
    int tipoMetr=Integer.parseInt(tipoMetrica);
    if(tipoMetr==0) {
        regexConsola.add(regex);
        descripConsola.add(descripcion);
        nameConsola.add(nombre);
        idConsola.add(id);
        ConsoleListener c=new ConsoleListener(regex);
        consoleListeners.add(c);
    }else if(tipoMetr==1) {
        regexCode.add(regex);
        descripCode.add(descripcion);
        nameCode.add(nombre);
        idCode.add(id);
    }

}
```

Figura 8. Método parseMetricaObject() de LectorJSON

A partir de aquí el plug-in generará unos controladores, representados en la Figura 9, para cada tipo de métrica que nos permitan trabajar sobre estas globalmente y no individualmente a la vez que guarda el tiempo en el que están siendo calculadas. Para esto, el

controlador recibirá la lista de expresiones regulares y de descripciones e irá creando cada métrica, además si se van a utilizar métricas de consola deberá recibir una lista de consoleListeners.

```
public controladorMétricas(ArrayList<String> expresionesRegulares, ArrayList<String> descripciones, ArrayList<String> nombres,
        ArrayList<String> identificadores, double tiempo) {
    this.tiempo=tiempo;
    Metrica m=null;
    for(int i=0;i<expresionesRegulares.size();i++) {

        m=new metricaCodigoFuente(expresionesRegulares.get(i),descripciones.get(i),nombres.get(i),identificadores.get(i));

        metricas.add(m);
    }
}
public controladorMétricas(ArrayList<String> expresionesRegulares, ArrayList<String> descripciones,ArrayList<String> nombres,
        ArrayList<String> identificadores, double tiempo,ArrayList<ConsoleListener> consolas) {
    this.tiempo=tiempo;
    Metrica m=null;
    for(int i=0;i<expresionesRegulares.size();i++) {

        m=new metricaConsola(expresionesRegulares.get(i),descripciones.get(i),nombres.get(i),identificadores.get(i),consolas.get(i));
        metricas.add(m);
    }

}

public void calculaMetricas() throws CoreException, IOException {
    for(int i=0;i<metricas.size();i++) {
        metricas.get(i).calcula();
    }
}
```

Figura 9. Dos tipos de constructores dependiendo del tipo de métrica

Los dos tipos de métrica se extienden de una clase abstracta Métrica, la cual se compone de unas variables y unos métodos que permiten unas funcionalidades básicas como devolver la descripción, devolver, incrementar o modificar el valor o realizar el cálculo. Esta clase Métrica deberá recibir tres String, la expresión regular, la descripción, el nombre y el identificador.

El calculo de las métricas se realizará cada vez que pasen los segundos especificados en el intervalo de tiempo que recibe el plug-in, para que esto no moleste a la hora de programar se usan hilos, de esta forma se podrá escribirá y ejecutará código mientras por detrás opera el plug-in. El hilo comienza como se ve en la Figura 10, se generan los controladores de métricas a partir de los datos leídos del archivo JSON y del tiempo actual y se mandará a estos controladores que calculen todas sus métricas, este proceso se irá repitiendo cada intervalo de tiempo. A la hora de calcular las métricas estas se deben calcular en primer plano porque si no las métricas de código fuente no podrían tener acceso a la ventana del workbench para poder leer el código fuente, por esto, se usa un "*Display.getDefault().asyncExec"* haciendo que mientras se esté ejecutando este parte del código el plug-in deje de estar en segundo plano.

```
@Override
public void run() {

    while(true) {
        double tiempoCalc=(System.nanoTime()-tiempo)/1000000000;
        controladorMétricas contConsola=new controladorMétricas(regexConsola, descripConsola,
                nameConsola,idConsola,  tiempoCalc,consoleListeners);
        controladorMétricas contCode=new controladorMétricas(regexCode, descripCode,nameCode,idCode,
                tiempoCalc);

        Display.getDefault().asyncExec(new Runnable() {
            @Override
            public void run() {

                try {
                    contCode.calculaMetricas();
                    contConsola.calculaMetricas();
                } catch (CoreException | IOException e1) {
                    // TODO Auto-generated catch block
                    e1.printStackTrace();
                }
```

Figura 10. Inicio del hilo

Dependiendo del tipo de métrica se deberán realizar dos tipos de cálculos, uno que acceda a la consola y otro al código fuente, por eso los argumentos que reciben los controladores en la Figura 10 son diferentes ya que dependiendo del tipo de métrica se necesitarán unos datos u otros. En la Figura 11 se puede observar como para realizar el cálculo de métricas de código fuente se necesita acceso primero a la ventana de Eclipse y así poder leer diferentes partes de esta, en concreto los editores de texto. El siguiente sería leerlos como ficheros y convertirlos en un String, cuando ya se ha realizado la conversión se pueden usar las clases Pattern y Matcher para contabilizar las veces que aparece la expresión regular de la métrica en el String con el texto de los editores.

```
public void calcula() throws CoreException, IOException {


    IWorkbenchWindow win = PlatformUI.getWorkbench().getActiveWorkbenchWindow();
    win = PlatformUI.getWorkbench().getActiveWorkbenchWindow();

    IWorkbenchPage page = win.getActivePage();
    IEditorReference[] editores= page.getEditorReferences();
    for(int i=0;i<editores.length;i++) {
        IEditorPart editor=editores[i].getEditor(true);
        IEditorInput input = editor.getEditorInput();
        if (input instanceof FileEditorInput) {
            IFile file = ((FileEditorInput) input).getFile();
            InputStream is = file.getContents();
            InputStreamReader isReader = new InputStreamReader(is);
            BufferedReader reader = new BufferedReader(isReader);
            String fichero="";
            String temp=reader.readLine();
            while (temp != null) {
                fichero=fichero+"\n"+temp;
                temp=reader.readLine();

            }
            Pattern p = Pattern.compile(super.getRegex());
            Matcher m = p.matcher(fichero);

            while(m.find()) {
                value++;
            }


            reader.close();

        }

    }
```

Figura 11. Calculo de métricas de código fuente

Por otro lado, las métricas de consola necesitarán el uso de consoleListeners que irán comprobando en cada consola las veces que aparece el texto deseado. Como se puede

observar en la Figura 12 se recibe un patrón que es el texto que se busca en consola y se añade un consoleListener con este patrón a las consolas. Gracias al método consolesAdded se tiene acceso a todas las consolas que se añaden después de la creación del ConsoleListener, por esto solo existirá uno por patrón que será creado en cuanto se ejecute el plug-in y se compartirá por todas las métricas iguales. Cada vez que se necesite calcular una métrica de consola se llamará a un método que puede leer el valor del ConsoleListener en ese momento.

```
public ConsoleListener(String patron) {
    pattern = patron;
    matchCount = 0;
    ConsolePlugin.getDefault().getConsoleManager().addConsoleListener(this);
}

@Override
public void consolesAdded(IConsole[] consoles) {
    // TODO Auto-generated method stub
    for(IConsole console : consoles) {
        if(console instanceof TextConsole) {
            ((TextConsole) console).addPatternMatchListener(this);

            if(console.getName().contains("javaw.exe")) {
                this.erroresJavaw++;
            }else if(console.getName().contains("javac.exe")) {
                this.erroresJavac++;
            }

        }
    }
}
```

Figura 12. ConsoleListener

Cuando acaba el calculo en cada intervalo el hilo continúa modificando el archivo JSON de resultados añadiendo los nuevos datos, para esto se crean los objetos y listas de JSON necesarios que guarden la información de todas las métricas calculadas. Cuando ya se ha añadido toda la información nueva a la lista JSON con todas las métricas se crea un nuevo archivo de resultados en el que se escribe esta. Para finalizar, se obliga al hilo a dormirse durante el tiempo establecido en el intervalo de tiempo recibido, el proceso se repite indefinidamente hasta que se cierre la ventana de Eclipse.

```
            JSONObject metricas = new JSONObject();
            metricas.put("ID", ID);
            metricas.put("Time", tiempoCalc);
            metricas.put("Metricas Consola", listaConsolaJSON);
            metricas.put("Metricas Codigo Fuente", listaCodigoJSON);

            listaMetricasJSON.add(metricas);
            //Write JSON file
            String userHomeFolder = System.getProperty("user.home");
            File textFile = new File(userHomeFolder,"\\" +"salida"+ID+".json");
            archivo=userHomeFolder+ "\\" +"salida"+ID+".json";

            try (FileWriter file = new FileWriter(textFile)) {
                file.write(listaMetricasJSON.toJSONString());
                file.flush();
            } catch (IOException e) {
                e.printStackTrace();
            }
            listaControladoresConsola.add(contConsola);
            listaControladoresCode.add(contCode);
        }
    });

    try {
        Thread.sleep(1000*this.intervalo);
    } catch (InterruptedException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
```

Figura 13. Fin del hilo y escritura de los resultados

4.3 DISEÑO E IMPLEMENTACIÓN DE LAS INTERFACES DE USUARIO

En este apartado del capítulo 4 se tratarán el diseño e implementación de las interfaces que componen el plug-in y la aplicación que presenta gráficas a partir de los resultados de las métricas. En la Figura 14 se puede observar en que orden aparecen las interfaces. Cuando se inicializa el plug-in se mostrará una interfaz en la que se deben rellenar los datos que permitan la ejecución, el identificador de usuario, un intervalo de tiempo y un archivo JSON con las métricas a buscar, que se deberá seleccionar a partir de una nueva interfaz que permite buscar archivos. Después de un tiempo ejecutando el plug-in el usuario podrá enviar los resultados gracias a un botón en la barra de tareas de tareas de Eclipse, esto lanza una nueva interfaz en la que se deben introducir los datos requeridos para realizar una conexión SSH, usuario, contraseña, host y puerto. Además, estará rellenado un campo extra con la dirección del archivo que se quiere enviar la cual se podrá cambiar gracias a un botón que genera una ventana igual a la que permite seleccionar el archivo JSON al iniciar el plug-in. Cuando ya se ha recibido el archivo de resultados se puede ejecutar la aplicación para visualizar gráficas a partir de las métricas. En un principio, se debe seleccionar el archivo de resultados gracias a una ventana similar a las usadas anteriormente, si el archivo es correcto se lanzará una ventana en la que podrán ver y generar gráficas a partir de ese archivo.

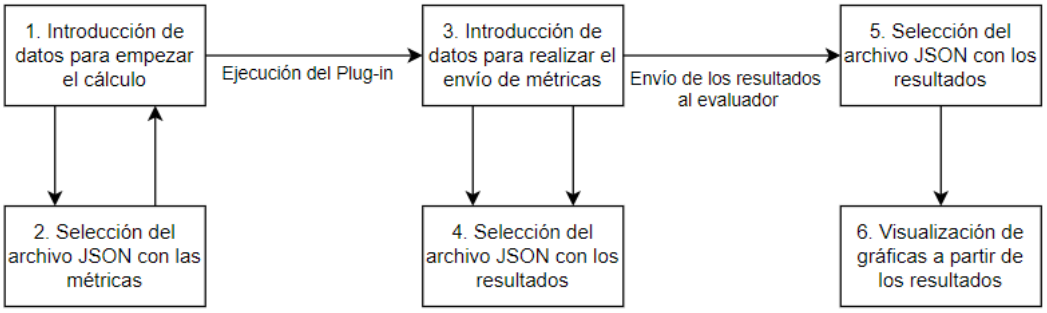

Figura 14. Modelo de Navegación entre interfaces de usuario

Cuando se quiere empezar a calcular las métricas se debe pulsa un botón, el encuadrado en verde de la Figura 16, en la barra de tareas de Eclipse, lo siguiente que debe hacer el usuario es rellenar unos datos necesarios para la ejecución del plug-in en la interfaz de la Figura 15. Para recibir los datos se usan dos JTextFields que permiten la escritura de texto por parte tanto del programa como del usuario, cada uno esta acompañado por un JLabel, una especie de etiqueta, en el que se expresa lo que debe ser escrito en cada campo, el identificador de usuario y el intervalo de tiempo en segundo en el que se calcularán las métricas, debajo están dos botones, el primero sirve para lanzar otra interfaz que permita elegir el JSON archivo con las métricas que se va a usar y el último botón permite que plug-in lea los datos y empiece a realizar los cálculos.

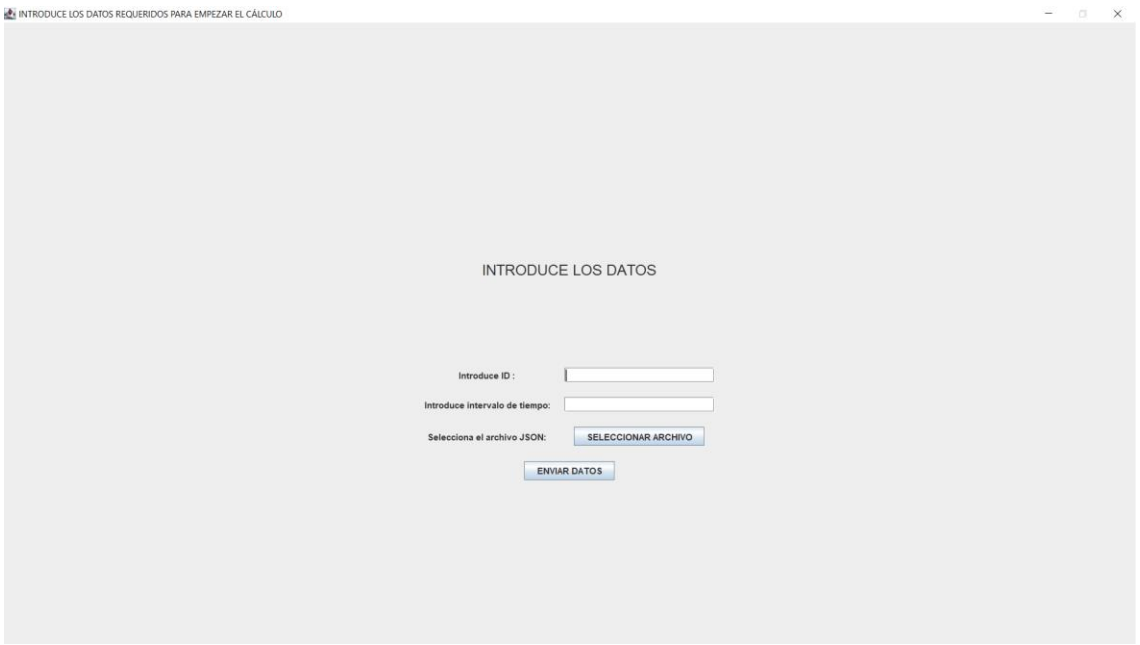

Figura 15. Interfaz de inicio del plug-in

Para que el usuario pueda decidir cuando iniciar a calcular las métricas y cuando se enviarán al evaluador se hace uso de botones que son añadidos a la barra de herramientas de

Eclipse. La creación de estos botones se lleva a cabo gracias al archivo XML plugin.xml, archivo de manifiesto, en el que se define la estructura del plug-in y como se añade a Eclipse. Se añadió dos command que definen a cada botón dentro de la etiqueta toolbar, se les otorga un icono, el comando que al que deben llamar y un mensaje que aparezca cuando se pasa el cursor por encima, en el primer botón, el verde, sería “Calcula métricas”, en el rojo sería “Envía Métricas”. Cada uno de estos comandos a los que se llama a su vez ejecuta un handler diferente, así iniciando el cálculo o la conexión SSH.

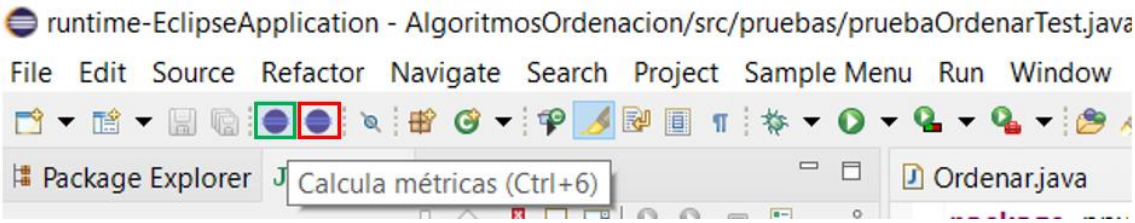

Figura 16. Botones del plug-in en la barra de herramientas

Para realizar la conexión SSH se proporciona al usuario una interfaz, Figura 17, en la que se introducen los datos que se necesitan para realizar la conexión, los cuales son el nombre de usuario, el HOST al que se quiere acceder, el puerto que se va a usar y la contraseña, existe un último dato que es la dirección del archivo que se quiere mandar, este estará ya completado si se ha ejecutado el cálculo de métricas, si no, aparece en blanco, al lado hay un botón que lanzará una ventana de selección de archivo para poder cambiar la dirección si es necesario como la usada anteriormente al iniciar el plug-in para seleccionar el archivo con las métricas. Por último, el botón de enviar dato si se ejecuta comprobará los datos, saltando un mensaje si son incorrectos o si no se puede realizar la conexión, establece una conexión SSH y envía el archivo al destino.

Figura 17. Interfaz conexión SSH

Para facilitar la visualización de las métricas en forma de gráficas por parte del evaluador se creará una aplicación pueda leer el archivo de resultados en formato JSON, gracias a que cuando se inicia se lanza una interfaz que permite seleccionar el fichero adecuado, y crear gráficas a partir de este. Esta aplicación empieza con una clase Main que genera una ventana en el que se puede seleccionar el archivo JSON a leer y que lo reciba otra clase que lo pueda traducir. Por último, le enviará la lista de las métricas a una clase que generará el frame donde se crearán las gráficas el cual será explicado más adelante en este apartado.

```java
public class Main {

    public static void main(String[] args) throws FileNotFoundException {


        LectorJSON lector=null;
        Frame frame=new Frame("ELIGE EL ARCHIVO CON LAS MÉTRICAS");


        frame.setResizable(false);
        frame.setLocation(0,0);
        frame.setExtendedState(JFrame.MAXIMIZED_BOTH);

        JFileChooser file=new JFileChooser();
        file.showOpenDialog(frame);
        File abre=file.getSelectedFile();
        if(abre!=null)
        {
            FileReader archivos=new FileReader(abre);
            lector= new LectorJSON(archivos);
        }



        ArrayList<Intervalo>intervalos =lector.getIntervalos();

        FrameMetrica frame2=new FrameMetrica(intervalos);

    }

}
```

Figura 18. Clase Main de la aplicación

La clase creada para parsear el JSON es similar a la usada en el plug-in pero con pequeñas variaciones para adaptarla al cambio de formato como el número de lista puesto que en el fichero de resultados se compone de una lista de intervalos que a su vez se forman por el tiempo en el que se calculó, el ID de identificación de usuario y dos listas de métricas calculadas, unas dependientes de la consola y otras del código fuente, cada una de estas se divide en una descripción y un valor. El constructor del parser recibe el fichero JSON, convierte la lista de intervalos en un array y, por último, manda como argumento cada intervalo al método que deberá traducir su información.

```java
public class LectorJSON {


    ArrayList<Intervalo> intervalos=new ArrayList<Intervalo>();


    @SuppressWarnings("unchecked")

    public LectorJSON(FileReader file) {
        JSONParser jsonParser = new JSONParser();

        try (FileReader reader = file)
        {
            //Read JSON file
            Object obj = jsonParser.parse(reader);
            JSONArray listaMetricas = (JSONArray) obj;
            listaMetricas.forEach( emp -> parseMetricaObject( (JSONObject) emp ) );

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException e) {
            e.printStackTrace();
        }
    }
```

Figura 19. Constructor de la clase que permite leer el archivo JSON

El método recibe el objeto Métrica, de este extrae la ID del usuario, el tiempo en el que se realizó el cálculo y crea un objeto de la clase intervalo. Después, lee las dos listas y se las manda como argumento al método parseLista() que traduce cada instancia de las listas a un objeto de la clase métrica y las añade al intervalo, cada uno se va añadiendo a una lista de intervalos completando la traducción del JSON.

```
private void parseMetricaObject(JSONObject metrica) {


    String ID = (String) metrica.get("ID");
    double tiempo = (double) metrica.get("Time");
    int IDInt=Integer.parseInt(ID);

    Intervalo inter=new Intervalo(IDInt, tiempo);
    JSONArray listaMetricasConsola = (JSONArray)metrica.get("Metricas Consola");
    JSONArray listaMetricasCodigo = (JSONArray)metrica.get("Metricas Codigo Fuente");

    listaMetricasConsola.forEach( emp -> parseLista( inter,(JSONObject) emp ) );
    listaMetricasCodigo.forEach( emp -> parseLista( inter,(JSONObject) emp ) );
    intervalos.add(inter);

}

private void parseLista(Intervalo inter, JSONObject metrica) {
    String description = (String) metrica.get("Description");
    long valuelong = (long) metrica.get("Value");
    String name = (String) metrica.get("Name");
    int idmetrica =Integer.parseInt( (String) metrica.get("IdMetrica"));
    int value=(int) valuelong;


    inter.addMetrica(description, value, name,idmetrica);
}
public ArrayList<Intervalo> getIntervalos() {
    return intervalos;
}
```

Figura 20. Método que traducen la información del JSON

Cuando se tienen todos los datos en la aplicación se genera la interfaz, para esta se usará java.swing, lo primero será generar un comboBox que deje seleccionar el nombre de la métrica de la que se quiere generar la gráfica, lo segundo una etiqueta como titulo que explique como funciona la interfaz, por último un panel donde se puedan añadir las gráficas y un scroll vertical para poder desplazarse por ellas. Se genera una gráfica desde el principio correspondiente a la primera métrica para esto se usa una librería JFreeChart que usa un dataset con los datos del archivo JSON de resultados, cada vez que se seleccione un nombre en la lista del comboBox se dibujará una nueva gráfica y se puede desplazar entre ellas usando el scroll vertical.

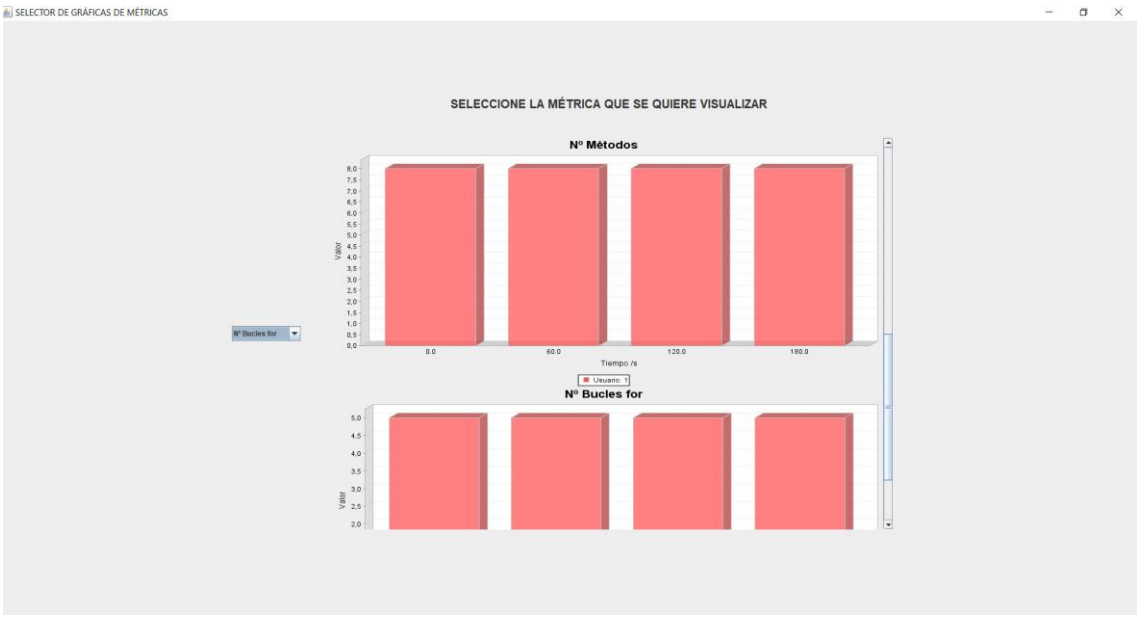

Figura 21. Interfaz de muestra de gráficas

5. EVALUACIÓN Y PRUEBAS

Para comprobar que el plug-in cumple con las funcionalidades establecidas en un principio se deberá realizar una series de pruebas, las cuales serán enumeradas en este capítulo. A parte de comprobar si las funcionalidades son correctas nos permiten encontrar errores de implementación que deberán ser solventados.

5.1 PRUEBAS SOBRE EL PLUG-IN

En este apartado se tratarán las pruebas ejecutas sobre la parte más importante del proyecto, el plug-in, y que podría dar lugar al mayor número de errores. A la hora de probar el plug-in se fueron realizando varias pruebas durante el desarrollo para comprobar que los diferentes fragmentos del código funcionaban correctamente, como utilizar diferentes expresiones regulares para comprobar que calcula correctamente las métricas o escribir código en la ventana de eclipse creada al ejecutar el plug-in y ver como varían las métricas, todo esto correspondiente al código fuente. Con respecto a la consola, se hicieron pruebas similares buscando diferentes tipos de errores o mensajes en la consola y fueron ejecutados varias veces diferentes códigos que mostrarán mensajes por consola para comprobar como variaba el valor de las métricas. Para comprobar que las métricas se calculaban siguiendo el intervalo necesario de tiempo se le fueron pasando varios de estos con diferentes números.

Para comprobar que funciona el plug-in correctamente se dará la tarea de a varios programadores de hacer tres algoritmos de ordenación, Quicksort, Mergesort y Heapsort, los cuales se estudian durante la carrera de Ingeniería Informática, se les dará una lista no ordenada de enteros que deberán ser ordenados de menor a mayor mediante estos algoritmos y posteriormente escritos en consola separados por comas. A su vez, en un archivo JSON se les pasarán 12 como los de la Tabla 3, en esta tabla se puede ver la expresión regular que define cada métrica y su tipo que es donde se debe buscar. Las métricas permitirán monitorizar el trabajo realizado por el programador como el número de métodos en el código fuente o el comprobar que los números se han ordenado correctamente en consola, cuantas veces se ha probado el código y cuantas veces correctamente. Las siguientes pruebas se realizarán mediante la utilización Junit que nos permitirán construir diferentes test para comprobar errores en los algoritmos de ordenación y usar el plug-in para hacer cálculos sobre estos errores.

Métrica	*Expresión que buscar*	*Tipo*
Número de Clases	\\s*(public|private)\\s+class\\s+(\\w+)\\s+((extends\\s+\\w+)|(implements\\s+\\w+(,\\w+)*))?\\s*\\{	Codigo Fuente
Número de Métodos	(public|protected|private|static|\\s)+[\\w\\<\\>\\[\\]]+\\s+(\\w+) *\\([^\\)]*\\)*(\\{?|[^;])	Codigo Fuente
Número de Bucles for	.*for\\s*(.*;.*;.*).*	Codigo Fuente
Número de Comentarios	(/*([^*]|[\\r\\n]|(*+([^*/]|[\\r\\n])))**+/)|(//.*)	Codigo Fuente
Número de Errores	Exception in thread	Consola

Número de comprobaciones JUnit fallidas	Se buscarán tres mensajes diferentes, uno para cada uno de los tres algoritmos de ordenación que pudiera fallar	Consola
Número de Errores index out of bounds	ArrayIndexOutOfBoundsException	Consola
Número de Errores null pointer	NullPointerException	Consola
Número de Errores stack overflow	StackOverflowException	Consola
Número de Comprobaciones si la lista ha sido ordenada	Lista ordenada	Consola

Tabla 3. Métricas que se calcularán en la prueba

Después que los programadores hayan terminado las pruebas de los algoritmos y la ordenación de la lista sea correcta se hará la encuesta de la Tabla 4, la cual recibirán todos los participantes para que puedan dar su opinión sobre el plug-in y la aplicación que genera las gráficas. Las preguntas se centran sobre todo en la facilidad de uso de diferentes aspectos del proyecto como el plug-in o la aplicación para visualizar las gráficas, además de si el uso del plug-in a la vez que se programa resulta en algún tipo de molestia. Por último, podrán dejar algún comentario sobre aspectos que se podrían mejorar o que se echan en falta.

Encuesta	**Facilidad para iniciar el plug-in**	**No dificulta la labor de programación**	**Comodidad de uso de las interfaces**	**Se adecuan las métricas al trabajo realizado**	**Facilidad de uso de la aplicación**	**Las gráficas se realizan de forma adecuada**	**Comentarios extra**
1	Ha resultado sencillo	Ningún problema	Cómodas	Se actualiza correctamente adecuándose a los cambios realizados	Fácil de usar	Muestran la información de forma adecuada	Sugerencias por parte del plug-in para solucionar errores
2	Es fácil de iniciar	No he sentido ninguna molestia a la hora de programar	Intuitivas	Las métricas reflejan los cambios realizados en el código y las ejecuciones de consola	Resulta sencillo	La información se muestra adecuadamente, aunque me gustaría que se pudiera cambiar el formato y el tamaño	Que se puedan personalizar las gráficas

3	Es intuitivo	No ha habido ningún problema	Sencillas de usar	Se calculan los datos correctamente	Fácil de usar	La visualización es correcta, pero estaría bien que las líneas guía fueran más visibles	
4	Es fácil de instalar y de iniciar	En absoluto	Son sencillas de usar	Se adecuan correctamente	Intuitivo y fácil de usar	La información está bien expuesta y se aprecia la diferencia entre barra	En vez de una lista despegable con los nombre de las métricas en la aplicación fuera un checkbox

Tabla 4. Encuesta

5.2 PRUEBAS JUNIT

En este apartado se explicará más detalladamente en qué consisten las pruebas finales realizadas en el plug-in usando JUnit. Para realizar las pruebas se proporcionó a los tester un archivo .jar con el plug-in que se puede añadir a Eclipse yendo a la pestaña de ayuda y clickando en añadir nuevo software, se abre una pestaña en la que escribir la dirección o elegir el archivo que se quiera añadir. Además, se les envió un archivo JSON con las métricas de la Tabla 3 que sirven para realizar pruebas de los diferentes tipos de métricas y si el calculo de una cantidad alta de ellas dificulta a la hora de programar provocando que el workbench de Eclipse se bloqueé o algún otro problema. Por último, también se les proporcionó un pequeño documento de texto donde se explica como se usa el plug-in y el paquete de clases java necesario para realizar las pruebas sobre los algoritmos de ordenación, tanto la clase donde se deben escribir los algoritmos, Figura 22 , como la clase que usa JUnit para probarlos.

```
public class Ordenar {
        ArrayList<Integer> metodo1;
        ArrayList<Integer> metodo2;
        ArrayList<Integer> metodo3;


        public Ordenar(ArrayList<Integer> desordenado) {
            metodo1= desordenado;
            metodo2= desordenado;
            metodo3= desordenado;
        }


        public ArrayList<Integer> metodoOrdenacion1(){
            //TODO

            return null;
        }

        public ArrayList<Integer> metodoOrdenacion2(){
            //TODO

            return null;
        }

        public ArrayList<Integer> metodoOrdenacion3(){
            //TODO

            return null;
        }
```

Figura 22. Clase donde se escriben los algoritmos

La clase pruebaOrdenaTest usa JUnit para realizar test sobre los métodos de ordenación como se ve en la Figura 23. Cuando se empiezan los test el primer paso es ejecutar el método que tiene un @BeforeAll, en este caso se crea un array de números ordenados del uno al diez y otro con los mismos números, pero desordenados usando un algoritmo que coloca en cada hueco de la lista un número aleatorio del array. Para finalizar este método se pasa como argumento a la clase Ordenar la lista desordenada. Las siguientes tres pruebas se ejecutarán en orden y serán similares pero cada uno dedicado a comprobar el resultado de un algoritmo diferente. Lo primero que se hace en cada uno de ellos es ejecutar el método de ordenación correspondiente y guardar el resultado de la ejecución de este. Por último, se recorren la lista ordenada desde el principio y la resultante de aplicar el algoritmo y se comprueban que los elementos de los dos arrays son iguales, si no, se lanza un error con un aviso que especifica el método que no se ha ordenado correctamente. Estos mensajes de error aparecen en la consola de JUnit la cual puede leer el plug-in para calcular métricas compartiendo su tipo con las métricas de consola normales. Gracias a lo explicado anteriormente se puede utilizar JUnit como se hace usualmente para comprobar el funcionamiento de un código y a su vez como una forma de controlar la labor del programador gracias al uso de este plug-in para guardar un registro de cuantas veces se ha ejecutado este test, cuantas veces ha fallado cada algoritmo y como se han ido produciendo en el tiempo estos errores, lo cual más adelante se le puede presentar como una gráfica al evaluador gracias a la aplicación.

```
@BeforeAll
public static void  beforeAll() {
    desordenado=new ArrayList<Integer>();
    for(int i=0;i<10;i++) {
        desordenado.add(i);
    }
    ordenadoComparar=new ArrayList<Integer>();
    for(int i=0;i<10;i++) {
        ordenadoComparar.add(i);
    }
    int index, temp;
    Random random = new Random();
    for (int i = desordenado.size() - 1; i > 0; i--)
    {
        index = random.nextInt(i + 1);
        temp = desordenado.get(index);
        desordenado.set(index,desordenado.get(i));
        desordenado.set(i,temp);
    }
    o=new Ordenar(desordenado);
}
@Test
@DisplayName("First test")
void firstTest()  {
    try {
        o.metodoOrdenacion1();
        ArrayList<Integer> temporal=o.getMetodo1();
        for(int i=0;i<temporal.size();i++) {
            assertEquals("No ordenado correctamente en el metodo 1",temporal.get(i),ordenadoComparar.get(i));
        }
    }catch(Exception e) {
        fail();
    }
}
```

Figura 23. Clase que prueba los algoritmos

6. CONCLUSIONES

En este capítulo se muestran las conclusiones alcanzadas tras realizar el proyecto. El capítulo incluye un apartado inicial que analiza el grado de cumplimiento de los objetivos establecidos en el Capítulo **¡Error! No se encuentra el origen de la referencia.** Por otra parte, se expondrán en el segundo apartado los posibles futuros trabajos que se pueden desarrollar a partir de este proyecto.

6.1 ANÁLISIS DEL CUMPLIMIENTO DE LOS OBJETIVOS

Este proyecto partía con un objetivo principal que se enunció en el capítulo inicial como *"desarrollar un plug-in para Eclipse IDE que permita monitorizar el trabajo de estudiantes en sus tareas de programación y calcular métricas que permitan al profesor cuantificar distintos aspectos de sus sesiones de trabajo"*. Este objetivo se dividió en los siguientes subobjetivos más concretos cuyo grado de cumplimiento se analiza a continuación:

- **Crear diferentes métricas.** Para resolver este objetivo se usó archivos JSON que permitían guardar, como se establece en el Capítulo 1, la expresión regular y el tipo de métrica que es, se ha implementado un método diferente para cada tipo de métrica. El archivo JSON puede variar en número de métricas puesto que se puede utilizar cualquier expresión regular que se quiera. Había que resolver dos problemas, acceder a la consola y acceder al código fuente que estaba siendo desarrollado por el programador, puesto que la mayoría de las métricas dependen de una u otra. Para el cálculo de las métricas de consola se necesitó usar ConsoleListeners que reciben un String a buscar en todas las consolas que se generan después de su creación, en cuanto a las métricas del código fuente a partir de la API IWorkbench y otras relacionadas con esta se consigue acceder a los editores de texto y a partir de allí se pueden usar por ejemplo expresiones regulares para calcular el número de clases, métodos, comentarios, etc.

- **Que el profesor pueda crear sus propias métricas.** Este objetivo se cumplió gracias al uso de archivos JSON, ya que permiten modificar el contenido de las métricas por parte del evaluador cambiando la expresión regular. Esta funcionalidad es gracias a la no dependencia del plug-in de unas métricas ya establecidas desde un principio si no que son definidas desde cualquier archivo JSON que siga el formato que se pide. Se le da la opción al profesor de buscar la expresión regular en dos sitios diferentes, en código fuente y en consola. Las métricas de código fuente dan la opción de buscar usando expresiones regulares o un texto directamente en el código del alumno, a su vez las métricas de consola permiten buscar texto en todas la consolas que se generen, lo que da la opción de calcular tanto número o tipo de errores como diferentes String que se necesiten escribir, por otra parte, ya que los resultados de los test de JUnit se escriben en una ventana que funciona como otra consola se puede calcular diferentes métricas sobre estas pruebas.

- **Que las métricas puedan cuantificar el proceso de trabajo y el producto final del mismo.** Este objetivo se alcanzó con la implementación de la capacidad de calcular métricas en diferentes sitios como la consola o el código fuente, el calculo en intervalos, que se produce gracias al uso de hilos que se duermen durante el tiempo especificado y cuando se despiertan calculan las métricas y vuelven a dormir, y al uso de JUnit para poder extraer información sobre el funcionamiento correcto del programa. Tanto para el cálculo de métricas en JUnit como en consola se usó ConsoleListener que permiten contar el número

de veces que ocurre un String en todas las consolas, incluyendo en la que se muestran los errores de JUnit. Para el cálculo de las métricas de código fuente se debió acceder al IWorkbench y otros componentes de la ventana de Eclipse a partir de los cuales poder convertir todo el texto de los editores en un String y poder buscar en este las diferentes expresiones regulares.

- **Comunicación alumno y profesor.** Para cumplir este objetivo se usó el protocolo SSH que permite realizar una conexión entre el usuario y el evaluador, alumno y profesor respectivamente. En primer lugar, se modificó el archivo xml del plug-in para generar un nuevo botón a parte del que inicializa el cálculo de las métricas y ligarlo a una nueva clase handler, la cual genera una interfaz gráfica que permite introducir los datos del servidor que recibe la información, que son puerto, host, usuario, contraseña y dirección donde está el archivo de resultados. La comunicación se realiza mediante una API que permite la utilización del protocolo SSH para mandar un archivo con los resultados de las métricas a partir de los datos que se reciben en la interfaz.

- **Representación gráfica de los valores de las métricas.** Con el fin de resolver este objetivo se creó una nueva aplicación que pudiera leer los datos de los resultados y generar gráficas a partir de estos. se genera una pequeña interfaz donde se debe seleccionar el archivo con las métricas finales generado, después se lee el archivo y se guarda la información de este en listas que permiten tener acceso a los diferentes datos de cada métrica. Para presentar las gráficas se creó una interfaz sencilla que da la opción de seleccionar la métrica que se quiera. Para generar la gráfica se usó la librería JFreeChart que tomando como eje x el tiempo y como eje y el valor crea una tabla de barras. En un principio se elige la métrica y crea un dataset en el que se incluye cada tiempo, id de usuario y valor, si hubiera varios identificadores de usuario se crearía una barra de un color diferente en cada intervalo de tiempo, una para el valor de cada usuario. Se crea la gráfica a partir del dataset y unos títulos que describan los valores de cada eje. Por último, se añade la gráfica a la interfaz sin borrar el resto de las gráficas generado anteriormente, pudiendo usar un scroll para moverse entre ellas.

- **Guardar un repositorio con los valores de las métricas para cada uno de los usuarios.** Este objetivó se solventó gracias a la creación de un servidor en el equipo del evaluador en el que se recibirán gracias a la conexión SSH todos los archivos JSON de resultados. Cada vez que se acabe de programar el alumno debe de darle al botón de la barra de tareas y rellenar la información, esto permite la subida al servidor de los archivos de todos los alumnos.

6.2 TRABAJOS FUTUROS

Los trabajos futuros que tomen como base este proyecto podrían implementar nuevos tipos de funcionalidades como que el propio plug-in devuelva feedback indicando cómo resolver problemas identificados por las métricas (cómo corregir errores de compilación, recordarle que debe probar más a menudo su implementación, etc.). Además, cabría plantearse nuevos mecanismos de comunicación entre el equipo utilizado por el alumno y el empleado por el profesor para que este puede monitorizar en tiempo real el trabajo que hacen los estudiantes.

ANEXO I. ACRÓNIMOS

IDE- Integrated Development Environment
JDT- Java Development Tools
PDE- Plug-in Development Environment
JSON- Java Script Object Notation
API- Application Programming Interfaces

REFERENCIAS

[1] FERRARIS, Diego Rafael Llanos. *Fundamentos de informática y programación en C*. Editorial Paraninfo, 2010.

[2] KONSYNSKI, Benn R., et al. PLEXSYS-84: An integrated development environment for information systems. *Journal of Management Information Systems*, 1984, vol. 1, no 3, p. 64-104.

[3] MARÍN, Sindy Johana Martínez; ARAMBURO, Santiago Arango; VELÁSQUEZ, Jorge Robledo. El crecimiento de la industria del software en Colombia: Un análisis sistémico. *Revista eia*, 2015, vol. 12, no 23, p. 95-106.

[4] MUÑOZ, Roberto, et al. Determinando las dificultades en el aprendizaje de la primera asignatura de programación en estudiantes de ingeniería civil informática. En *Memorias del XVII Congreso Internacional de Informática Educativa, TISE*. 2012. p. 120-126.

[5] https://help.eclipse.org/2020-09/index.jsp Autor: Eclipse Foundation Última consulta: 30/01/2021

[6] GEER, David. Eclipse becomes the dominant Java IDE. *Computer*, 2005, vol. 38, no 7, p. 16-18.

[7] https://www.json.org/json-en.html Autor: Última consulta: 23/12/2020

[8] https://wiki.eclipse.org/Eclipse_Project Autor: Eclipse Foundation Última consulta: 19/12/2020

[9] https://www.eclipse.org/eclipse/ Autor: Eclipse Foundation Última consulta: 10/12/2020

[10] https://www.eclipse.org/pde/ Autor: Eclipse Foundation Última consulta: 16/01/2021

[11] https://agilemanifesto.org/history.html Autor: Jim Highsmith Última consulta: 27/12/2020

[12] https://www.scrummanager.net/bok/index.php?title=El_manifiesto_%C3%A1gil Autor: Certificación Scrum Master Última consulta: 12/01/2021

[13] https://scrummanager.net/files/scrum_master.pdf Autor: Marta Palacio Última consulta: 12/01/2021

[14] https://scrum.mx/informate/historias-de-usuario Autor: Scrum México Última consulta: 12/01/2021

[15] https://www.jfree.org/jfreechart/ Autor: David Gilbert Última consulta: 14/01/2021

[16] BARNES, David J.; KÖLLING, Michael; BRENTA, Blanca Irene. Programación orientada a objetos con Java. Pearson Educación, 2007.

[17] https://www.cs.odu.edu/~zeil/cs350/f17/Public/IDEs/index.html Autor: Steven J Zeil, Old Dominion University Última consulta: 24/02/2021

[18] https://www.scrum.org/ Autor: Scrum Última consulta: 25/01/2021