UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Desarrollo de dos librerías Java: JavaCheckCode y JavaCheckStyle

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autora: Ana Navarrete Durante

Directores: Josep Silva Galiana
David Insa Cabrera

2015/2016

Agradecer en primer lugar a mis padres y hermanos por todo el apoyo y confianza que me han mostrado.
Dedicarles este proyecto a mis padres, por ser como son y porque, sin ellos, no sería nada de lo que soy hoy.
A mis compañeros, principalmente a mi amigo Diego, por acompañarme en estos cinco años de carrera.
A Josep y a David por toda su paciencia y dedicación.
A mis amigas por estar cuando se las necesita.

A todos vosotros gracias.

Resumen

En *Java*, como en la mayoría de lenguajes de programación, existe el concepto de librería. Una librería Java se puede entender como un conjunto de clases que facilitan operaciones y tareas ofreciendo al programador funcionalidad ya implementada y lista para ser usada través de una Interfaz de Programación de Aplicaciones, comúnmente abreviada como API (por el anglicismo *Application Programming Interface*). Las librerías *Java* nos permiten reutilizar código, es decir, podemos hacer uso de las clases, métodos y atributos que componen la librería evitando así tener que implementar nosotros mismos esas funcionalidades.

Las librerías Java, dentro de una comunidad de desarrollo, tienen como objetivo cubrir funcionalidades comunes que suelen presentarse al desarrollar software. El presente trabajo tiene como principal propósito desarrollar dos librerías Java que han sido llamadas *JavaCheckCode,* la cual permite medir la eficiencia de un algoritmo, y *JavaCheckStyle, la cual permite* evaluar el estilo de programación de un algoritmo (en relación a un estándar o norma concreta).

Para dar cumplimiento a este proyecto se desarrolla un análisis del contexto educativo en el que se desarrolla el diseño e implementación de estas librerías. Así mismo, se estudian todas las fases que necesita cada librería, con el fin de difundirlas de forma libre e independiente.

Las librerías desarrolladas en este proyecto son actualmente distribuidas como código públicamente accesible, abierto y gratuito. Para ello, se ha desarrollado un portal web independiente con un repositorio de versiones para su difusión y distribución.

Palabras clave: Java, librerías, eficiencia, estilo y algoritmo.

Abstract

In *Java*, as in most programming languages, there exists the concept of libraries. A library can be understood as a set of classes that facilitate operations and tasks offering functionality already implemented and ready to be used by the programmer through an application programming interface, commonly abbreviated as API (Application Programming Interface). The Java libraries allow us to reuse code, i.e. we can use its classes, methods and attributes that make up the library thus avoiding having to implement these features ourselves.

The Java libraries, within a community development, are intended to cover common functionalities that usually occur when developing software. This degree project has as a main purpose to identify different needs to develop two Java libraries that are call *JavaCheckCode,* which measures the efficiency of an algorithm and *JavaCheckStyle,* which evaluates the style of an algorithm (relative to a concrete standard or norm).

To fulfill this project, we develop an analysis of the educational context in which the design and implementation of these libraries is developed. In addition, all phases each library needs are studied, in order to freely and independently disseminate them.

The libraries developed in this project are currently distributed and publicly accessible, free and open source. To do so, an independent web portal with repository versions for broadcast and distribution has been developed.

Keywords: Java, bookstores, efficiency, style and algorithm.

Resum

A Java, com en la majoria de llenguatges de programació, existeis el concepte de llibreries. Una llibreria a Java es pot entendre com un conjunt de classes, que faciliten operacions i tasques oferint funcionalitat ja implementada i llista per a ser usada al programador a través d'una Interface de Programació d'Aplicacions, comunament abreviada com API (per l'anglicisme *Application Programming Interface*). Les llibreries a Java ens permeten reutilitzar codi, és a dir que podem fer ús de les classes, mètodes i atributs que componen la llibreria evitant així haver d'implementar nosaltres mateixos eixes funcionalitats.

Les llibreries Java dins d'una comunitat de desenvolupament, tenen com a objectiu cobrir funcionalitats comunes que solen presentar-se al desenvolupar programari. El present treball té com principal propòsit identificar diferents necessitats per a desenvolupar dues llibreries Java que han sigut cridades JavaCheckCode que permet mesurarà l'eficiència d'un algoritme i JavaCheckStyle que permet avaluar l'estil de programació d'un algoritme (en relació a un estàndard o norma concret).

Per a donar compliment a aquest projecte es desenvolupa una anàlisi del context educatiu en què es desenvolupa el disseny i implementació d'aquestes llibreries. Així mateix, s'estudien totes les fases que necessita cada llibreria, amb la finalitat de difondre-les de forma lliure i independent.

Les llibreries desenvolupades en este projecte són actualment distribuïdes com a codi públicament accessible, obert i gratuït. Per a això, s'ha desenvolupat un portal web independent amb un repositori de versions per a la seua difusió i distribució.

Paraules clau: Java, llibreries, eficiència, estil i algoritme.

Tabla de contenidos

1 Introducción

1.1 Motivación

Partiendo de un análisis de necesidades, se pretende desarrollar dos librerías Java y ponerlas a disposición de todos los usuarios para ser usadas libremente. Las librerías tendrán por nombre JavaCheckCode y JavaCheckStyle*, y serán* independientes de cualquier aplicación e independientes entre sí.

El propósito de las librerías es dotar al programador de nuevas funcionalidades que permitan descubrir información sobre un fichero Java. En particular, se pretende proveer un API para analizar la eficiencia y el estilo de un código Java dado.

Para la fase de desarrollo y programación del proyecto ha sido utilizada la metodología de desarrollo clásica o en cascada. Más adelante, se especificará cómo se ha adoptado esta metodología en el proyecto.

La fase de especificación de requisitos de la aplicación se desarrollará siguiendo la "Guía del IEEE para la Especificación de Requerimientos Software" [IEEE, 98] por ser el estándar en la especificación de requisitos.

En la fase de análisis se estudiarán los distintos aspectos de diseño Web actuales, analizando las ventajas inconvenientes que supone la utilización de cada uno de ellos en el proyecto actual.

En la fase de análisis se estudiarán los distintos aspectos que hay que tener en cuenta para poder realizar las diferentes librerías Java, analizando las ventajas que supone la utilización de cada uno de ellos en el proyecto actual.

En la fase de implementación de la aplicación se han utilizado dos de las tecnologías actualmente más extendidas para el desarrollo de las librerías y los portales web:

- **Java:** Todo el código fuente está realizado en Java.
- **HTML:** Se ha utilizado en el desarrollo de los dos sitios web creados para poder distribuir las librerías JavaCheckCode y javaCheckStyle.

1.2 Objetivos

Una vez se ha diseñado el proceso a seguir para alcanzar el objetivo final, se han definido funcionalidades concretas y concisas que se desearía que tuvieran las nuevas librerías.

La primera librería, JavaCheckCode, es una herramienta cuya finalidad es poder evaluar la eficiencia de un algoritmo dado de manera fácil y rápida.

Se pretenden abarcar diferentes objetivos según los diferentes tipos de datos en un fichero Java. La librería debe ser capaz de mostrar para un fichero, clase o método de Java:

- Mediante un análisis estático:
  - El número de condiciones totales existentes en el código.
  - El número de líneas de código sin tener en cuenta comentarios.
  - La máxima profundidad de los bucles.
- Mediante un análisis dinámico:
  - El tiempo de ejecución de un método dado.

La segunda librería, JavaCheckStyle, es una librería que permitirá evaluar el estilo de un archivo de manera educativa. Está pensada para que los estudiantes tengan unas reglas mínimas de estilo al realizar algoritmos de programación. La librería deberá ofrecer reglas de estilo y, para un código Java dado, obtener:

- El número de errores de estilos.
- Mostrar en pantalla los errores de estilo.
- Crear un informe detallado de los errores de estilo.

Cualquier usuario que quiera obtener esta información, no necesitará programarlo, sino que únicamente necesitará importar las librerías JavaCheckCode y JavaCheckStyle.

2 Conceptos básicos

En este apartado se establecerán las definiciones de los conceptos fundamentales utilizados en el proyecto. El objetivo de esta definición de conceptos es facilitar la comprensión del texto por parte del lector, independientemente de su nivel de familiarización con las tecnologías empleadas o con el mundo de la ingeniería informática.

Javadoc: [Wikipedia, 2016] Javadoc es una utilidad de Oracle para la generación de documentación de APIs en formato HTML a partir de código fuente Java. Javadoc es el estándar de la industria para documentar clases de Java. La mayoría de los IDEs los generan automáticamente.

API: [Wikipedia, 2016] La interfaz de programación de aplicaciones, abreviada como API (del inglés: *Application Programming Interface*), es el conjunto de subrutinas, funciones y procedimientos (o métodos, en la programación orientada a objetos) que ofrece cierta biblioteca o librería para ser utilizado por otro software como una capa de abstracción.

Paquete: [Wikipedia, 2016] Un Paquete en Java es un contenedor de clases que permite agrupar las distintas partes de un programa, definiendo la ubicación de dichas clases en un directorio de estructura jerárquica.

Clase: [Wikipedia, 2016] Una clase es una plantilla para la creación de objetos de datos según un modelo predefinido. Las clases se utilizan para representar entidades o conceptos, como los sustantivos en el lenguaje. Cada clase es un modelo que define un conjunto de variables (el estado), y métodos apropiados para operar con dichos datos (el comportamiento). Cada objeto creado a partir de la clase se denomina instancia de la clase.

Método: [Wikipedia, 2016] Un método es una subrutina cuyo código es definido en una clase y puede pertenecer tanto a una clase, como es el caso de los métodos de clase o estáticos, como a un objeto, como es el caso de los métodos de instancia.

Atributo: [Wikipedia, 2016] Un atributo es una especificación que define una propiedad de un objeto, elemento o archivo. También puede referirse o establecer el valor específico para una instancia determinada de los mismos.

Visitador: Un visitador es una utilidad para obtener el contenido interno de ficheros, clases y métodos. Es una forma rápida de conocer todo lo que hay en el interior de un fichero Java sin tener que implementar ninguna función.

LOC: Abreviatura de *Lines Of Code,* es una métrica de software que se utiliza para medir el tamaño de un programa, contando el número de líneas de código que tiene el código fuente del programa.

Complejidad Ciclomática: [Wikipedia 2008] Es una métrica de software que proporciona una medición cuantitativa de la complejidad lógica de un programa. La complejidad ciclomática define el número de caminos independientes dentro de un fragmento de código.

JavaParser: Es una librería Java creada para analizar el código fuente y la estructura de un fichero Java. De manera que JavaParser parsea todas las líneas de código mediante un visitador. JavaParser es un componente ideal para el análisis del código fuente. Por ello, las dos librerías harán uso de JavaParser para el análisis del código fuente.

Estilo: Es un término que describe convenciones para escribir en ciertos lenguajes de programación, aplicaciones, etc. El estilo de la librería JavaCheckStyle se medirá según los requerimientos necesarios para su uso.

Checkstyle: [Ivanov, 2016] Es una herramienta de desarrollo para ayudar a los programadores a escribir código Java adhiriéndose a un estándar de codificación. CheckStyle es altamente configurable y puede apoyar casi cualquier estándar de codificación.

- **SUN Style:** este documento refleja los estándares de lenguaje de codificación de Java que se presentan en la especificación del lenguaje Java de Sun Microsystems.
- **Google Style:** definición completa de estándares de codificación de Google para el código fuente en el lenguaje de programación Java. Un archivo fuente de Java se define como Google Style cuando se adhiere a todas sus normas.

3 Análisis previo

El presente proyecto parte de la necesidad de conocer información sobre un fichero Java. Dicha información no es directamente accesible con la tecnología actual, y es necesario un estudio para su extracción. El proyecto proporciona dos librerías independientes para conseguir extraer automáticamente la información que se necesita (nivel de eficiencia y adecuación a un estilo de programación de un código Java dado). Para la distribución de las librerías se desarrollan dos sitios web, explicando en cada uno de ellos la funcionalidad de una de las librerías. Algunos de los objetivos del proyecto son:

- Disponer de dos librerías documentadas y de total acceso para el usuario.
- Proporcionar ejemplos de las diferentes funcionalidades de cada una de las librerías.
- Documentar las librerías con JavaDoc.
- Establecer enlaces con todo el código fuente de las librerías para poder utilizarlas de forma específica.

El objetivo principal del presente proyecto es extender la funcionalidad de los desarrolladores a la hora de programar, ayudándoles a resolver de forma más rápida su trabajo como programadores.

Para extender la funcionalidad se observaron ciertas necesidades que no estaban disponibles en la comunidad Java y se estudió la forma de desarrollar dos librerías que puedan cubrir las carencias detectadas.

Las áreas que se van a desarrollar en este proyecto son las siguientes:

- La eficiencia de un algoritmo presenta varias características cómo el número de condiciones, LOC, profundidad máxima de bucles y el tiempo total de ejecución de un método, para medir el correcto funcionamiento de un método de programación. Se estudia la manera de poder implementar las necesidades anteriormente nombradas y proporcionar al usuario la documentación necesaria para averiguar la eficiencia que tiene un método desarrollado.
- El estilo de un fichero Java debe ser limpio y debe mantener unas mínimas características para poder tener una apariencia clara a la hora de leer un programa. Se desea proporcionar al

usuario, una serie de errores, si los tiene, de formas de representar el código fuente de un programa.

- Se va a implementar un sitio web que permitirá a un usuario acceder a todo lo que necesita para poder utilizar las librerías que vamos a implementar.

Se puede observar que hay otras librerías que miden el estilo y la eficiencia de un fichero tratándolo de forma diferente a la que nosotros queremos. Por ejemplo, para medir el estilo encontramos la librería CheckStyle, muy funcional, pero no es lo que nos interesa para resolver nuestras necesidades, ya que tiene muchas cosas en cuenta que a nosotros no nos convendrán, por ejemplo: la forma de introducir comentarios JavaDoc, paquetes mal importados, etc.

Estos dos ejemplos de librerías no nos ayudan a conseguir nuestros objetivos. Por lo que nos interesa realizar las librerías JavaCheckCode y JavaCheckStyle para aportar nuevas funcionalidades que no están desarrolladas actualmente.

3.1 Selección de tecnologías

Como parte del análisis previo, se ha realizado un estudio para evaluar las tecnologías necesarias para este tipo de librerías

A continuación, se enumeran uno a uno los lenguajes y tecnologías utilizadas en este proyecto:

- ***Java:*** Es un lenguaje de programación orientado a objetos y actualmente uno de los lenguajes más usados. Java se conoce principalmente por la gran cantidad de componentes que abarca. Las dos librerías serán desarrolladas íntegramente en Java.
- ***JavaParser:*** Es una librería Java creada para analizar el código fuente y la estructura de un fichero Java. De manera que JavaParser parsea todas las líneas de código mediante un visitador. JavaParser es un componente ideal para el análisis del código fuente. Por ello, las dos librerías harán uso de JavaParser para el análisis del código fuente.
- ***HTML:*** Es el lenguaje que se emplea para definir la estructura de una página web. Es una manera fácil para darle forma a la información que queremos ofrecer al usuario mediante etiquetas que el navegador puede interpretar.
- ***CSS:*** La página web que realicemos dispondrá de una página de estilos CSS. Hoy en día las hojas de estilos CSS son la manera más elegante de diseñar entornos Web, dado que suponen un alto grado de separación entre el contenido de la aplicación y su presentación al usuario. Además, la utilización de hojas de estilo CSS es imprescindible en cualquier aplicación Web para que sea estándar según el *World Wide Web Consortium* [W3C, 1994].
- ***Checkstyle:*** [Ivanov, 2016] Es una herramienta de desarrollo para ayudar a los programadores a escribir código Java adhiriéndose a un estándar de codificación. CheckStyle es altamente configurable y puede apoyar casi cualquier estándar de codificación.

La librería javaParser es especialmente relevante en el desarrollo del proyecto. Su uso nos facilitará la extracción de mucha información que las librerías necesitan usar. Concretamente, con esta librería se puede obtener toda la información de un fichero utilizando un objeto llamado CompilationUnit. A partir de él, podremos obtener información y guardarla en las diferentes clases a desarrollar en el presente trabajo.

Un pequeño ejemplo de cómo declarar y usar alguna función de CompilationUnit:

```
file = new File(filePath);
CompilationUnit cu = JavaParser.parse(file);
Visitor visitor = new Visitor();
int beginLine = cu.getBeginLine();
int endLine = cu.getEndLine();
visitor.visit(cu, null);
```

El objeto file es de tipo File, una clase que nos proporciona información de los archivos que le pasamos por parámetro. En este caso, la ruta donde se encuentra el archivo con el que queremos trabajar.

CompilationUnit: Es una de las clases de Javaparser para poder visitar todo lo que hay dentro de un fichero Java. En el ejemplo de código obtenemos a partir de la CompilationUnit la primera y última línea y las guardamos en dos variables.

Tras analizar las diferentes maneras de usar JavaParser, se concluyó que lo más conveniente es implementar un visitador para poder obtener los datos necesarios, y a partir de ellos medir el tamaño y la eficiencia de un programa a partir de distintas métricas (LOC, número de condiciones, etc.).

Como conclusión se puede comentar que aunque sea viable utilizar directamente esta librería para realizar una métrica de eficiencia en un archivo Java, esa labor es tediosa, puesto que requiere realizar diferentes tratamientos de datos para conseguir la información necesaria para el estudio. Por este motivo, se decidió desarrollar la librería JavaCheckCode.

4 Metodologías empleadas

En este apartado se va a explicar la metodología que se va a utilizar para el desarrollo de las librerías, explicando brevemente cada una de las fases del modelo que se han de seguir.

4.1 Modelo en cascada

El modelo clásico o en cascada es el más utilizado en los desarrollos software debido a que es muy sencillo de comprender y utilizar, ya que consta de cinco fases claramente diferenciadas.

El modelo en cascada está constituido por cinco fases secuenciales, de modo que para empezar una fase se tiene que haber terminado la anterior, y las salidas de la fase anterior serán las entradas de la fase actual. El modelo en cascada tiene un gran inconveniente: si en una de las fases iniciales se comete un error, éste se arrastra por todas las fases hasta el final, y la corrección del error cometido puede tener un gran coste. A continuación se exponen las fases del modelo en cascada:

- **Especificación de requisitos:** La fase de análisis de requisitos consiste en realizar un estudio pormenorizado de las necesidades del sistema, de los objetivos que debe cumplir, de los resultados que se deben obtener. En definitiva consiste en saber qué se necesita.
- **Análisis:** En esta fase se especifican formalmente las decisiones tomadas en la fase de análisis de requisitos, es la fase en la que se decide cómo se abordará el problema en las fases siguientes, el problema se analiza para su mejor tratamiento en la fase posterior. Por tanto, en esta fase se realizan decisiones técnicas que afectarán sobre todo a la fase de implementación.
- **Implementación:** En esta fase se realiza la codificación del software mediante un lenguaje de programación. La fase toma como entradas los diagramas de la fase de diseño y se codifica cada uno de ellos dando como resultado el software que es la solución del problema abordado.
- **Pruebas:** Esta fase consiste en someter al software generado en la fase anterior a una serie de pruebas exhaustivas para comprobar que no se han cometido errores y que su funcionamiento es el correcto.

- **Implantación:** Consiste en implantar el software generado en el sistema físico donde se va a utilizar. Esta fase también puede comprender la instrucción de los usuarios finales en el manejo de dicho software.

El modelo en cascada ha sido el elegido para la realización de este proyecto porque los requerimientos no iban a cambiar durante el desarrollo de la aplicación, y este modelo presenta bastantes ventajas si se saben los requisitos con exactitud antes de empezar.

5 Especificación de requisitos

5.1 Introducción

En este apartado se va a realizar una especificación de requerimientos siguiendo el estándar IEEE 830 [IEEE 98] destinado a la especificación de requerimientos software. En este caso la especificación de requisitos se enfocará en el desarrollo de dos librerías Java para recibir información sobre un fichero Java.

5.1.1 Propósito

El propósito de este proyecto es realizar la especificación completa de los requisitos que deben satisfacer las librerías. Se realizará una exposición clara y concisa de los requerimientos que deben tener las librerías y se abordarán sus posibles limitaciones y funcionalidades.

5.1.2 Ámbito

La aplicación que se va a desarrollar está orientada a ser utilizada en cualquier ámbito (doméstico, pequeñas empresas, docente, etc). Se accederá a las dos librerías a través de una página web, por lo que deben ser intuitivas y fáciles de comprender para su uso.

Las librerías serán de uso público, por lo que cada usuario tendrá disponibles todas las funciones de la librería para poder utilizarlas de forma independiente.

5.1.3 Definiciones, acrónimos y abreviaturas

HTML: ***H****yper* ***T****ext* ***M****arkup* ***L****anguage*. Lenguaje de marcas diseñado para estructurar textos y presentarlos en forma de hipertexto, que es el formato estándar de las páginas Web. La escritura de lenguaje HTML se puede realizar con cualquier editor de texto ASCII[1], aunque existen editores especiales que facilitan la creación de documentos HTML gracias a la marcación de etiquetas y al espaciado automático.

XML: *e****X****tensible* ***M****arkup* ***L****anguage*. Lenguaje de marcas extensible. Es un metalenguaje extensible de etiquetas desarrollado por el *World Wide Web Consortium* (W3C). Es una simplificación y adaptación del SGML y permite definir la gramática de lenguajes específicos (de la misma manera que HTML es a su vez un lenguaje definido por SGML).

[1] **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange (*Código Estadounidense Estándar para el Intercambio de Información*) es un código de caracteres basado en el alfabeto latino.

Por lo tanto XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. La utilidad del XML no se limita únicamente a Internet, sino que se propone como un estándar para el intercambio de información semi-estructurada entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo, etc.

CSS: ***C****ascading* ***S****tyle* ***S****heets.* Las hojas de estilo en cascada son un lenguaje formal usado para definir la presentación de un documento estructurado escrito en HTML o XML. El *World Wide Web Consortium* (W3C) es el encargado de formular la especificación de las hojas de estilo que servirá de estándar para los agentes de usuario o navegadores. La idea principal del desarrollo de CSS es separar la estructura de un documento de su presentación.

Java: Es un lenguaje de programación de propósito general, simultáneo en la ejecución de múltiples tareas interactivas y orientado a objetos. Java ofrece el código de casi todas sus librerías nativas para que los desarrolladores puedan conocerlas y estudiarlas en profundidad, o bien ampliar su funcionalidad, beneficiándose a ellos mismos y a los demás.

Usabilidad: La Organización Internacional para la Estandarización (ISO) dispone de dos definiciones de usabilidad:

- La norma ISO/IEC 9126 dice: "La usabilidad se refiere a la capacidad de un software de ser comprendido, aprendido, usado y ser atractivo para el usuario, en condiciones específicas de uso".
- La norma ISO/IEC 9241 dice: "Usabilidad es la efectividad, eficiencia y satisfacción con la que un producto permite alcanzar objetivos específicos a usuarios específicos en un contexto de uso específico".

Accesibilidad: La accesibilidad indica la facilidad con la que algo puede ser usado, visitado o accedido en general por todas las personas, especialmente por aquellas que poseen algún tipo de discapacidad.

5.1.4 Referencias

Para la elaboración del presente documento se ha seguido el estándar IEEE especificado en los siguientes documentos:

[IEEE 98] IEEE Std 830 - IEEE Guide to Software Requirements Specifications. IEEE Standards Board. 345 Eas 47 th Street. New York, NY 10017, USA.1998.

5.2 Descripción general

En los sucesivos apartados se detallan los requerimientos de cada función y todo lo que necesitan para poder obtener resultados. Las dos librerías que vamos a realizar se llamarán JavaCheckcode y JavaCheckStyle.

Con la librería JavaCheckCode podremos obtener la eficiencia de un fichero, clase o método determinado, a partir de conocer el número de condiciones, líneas de código fuente, profundidad máxima de bucles y el tiempo que tarda en ejecutarse.

Por otro lado, en la librería JavaCheckStyle se podrá obtener un informe de errores de estilo del código fuente en el que estemos interesados.

5.2.1 Perspectiva del producto

Las librerías a desarrollar deben permitir la utilización de todas las funciones implementadas, a través de un manual de usuario y unas especificaciones de uso. El usuario tendrá la información que requiera dentro de las diferentes posibilidades que le ofrecen las librerías.

5.2.2 Funciones del producto

Las dos librerías que se van a desarrollar deben satisfacer las siguientes funcionalidades:

JavaCheckCode:

Número de condiciones de un fichero Java: Una de las principales cosas que hay que tener en cuenta para medir la eficiencia de un método son las condiciones. Para ello, nuestra librería ofrecerá al usuario el número de condiciones (if, for, switch, foreach y while) que hay en el fichero.

Número de líneas de código de un fichero Java: De la misma forma, el usuario podrá obtener el número de líneas de código de un

fichero, sin contar los comentarios, ya que ese tipo de líneas no influye a la hora de ejecutar un fichero java.

Máxima profundidad de bucles en un fichero Java: Una de las funcionalidades de la librería y una de las características que hay que tener en cuenta para medir la eficiencia de un algoritmo, es la profundidad máxima de un bucle y se pretende acceder a cada uno de los bucles que haya en el fichero, para poder recoger la información de todos ellos y quedarnos con el máximo valor de anidamiento de bucles.

Número de condiciones de una clase: El usuario, a partir de la clase de la cual le interese obtener información, podrá averiguar el número de condiciones de la misma.

Número de líneas de código de una clase: Una característica fundamental de la librería es poder obtener solo la información que nos interese de una clase específica que el usuario introducirá como argumento de la función. Una vez se sepa la clase a tratar, obtendremos el número de líneas de código, obviando los comentarios.

Máxima profundidad de una clase: Como ya hemos hablado anteriormente, la profundidad es algo que tenemos que tener en cuenta para el coste de ejecución de un programa. Para ello, se realizará la una función equivalente a la del fichero, pero para una clase. Diferenciando de esta forma las diferentes clases que pudieran estar en el fichero.

Número de condiciones, líneas de código y profundidad máxima de un método: Se realizarán las mismas comprobaciones como las que hemos nombrado anteriormente para los métodos, pero para poder realizar estas consultas los usuarios necesitarán conocer el nombre de la clase donde está declarado el método de la cual quieran obtener la información.

Tiempo de ejecución de un fichero: El usuario podrá averiguar el tiempo que tarda en ejecutarse un programa. La librería ofrecerá información inmediata sobre el tiempo de ejecución, una de las principales medidas para calcular la eficiencia.

JavaCheckStyle:

Errores de formato en un programa: Con la librería JavaCheckStyle se podrán averiguar errores de estilo que hay en el fichero. El usuario tendrá información de los errores y de esa forma, podrá corregirlos.

El requisito funcional de la librería JavaCheckStyle es poder generar un informe de errores siguiendo las reglas de estilo del estándar de Java que utilizaremos en nuestra librería.

Posteriormente de haber creado la librería, si se necesitan más comprobaciones, deberá estudiar todas las funcionalidades de CheckStyle para poder utilizar las que más le interese.

5.2.3 Características del usuario

Los usuarios que utilicen la librería serán usuarios con conocimientos informáticos de programación en Java y que tengan la necesidad de obtener los datos que ofrecen las librerías.

Pueden haber distintos usuarios que accedan a las librerías, todos podrán utilizar todas las funciones que ésta ofrece, aunque solo los desarrolladores informáticos podrán aprovechar el acceso al código fuente de las mismas, para incrementar la funcionalidad.

Por tanto distinguimos entre dos tipos de usuarios de las librerías:

- Usuarios que utilizarán la funcionalidad de la librería. En este caso, simplemente importarán la librería en sus proyectos Java. Para ello, solo necesitan un .jar sin el código fuente.
- Usuarios que modificarán o extenderán la funcionalidad de la librería. En este caso, necesitan acceder al código fuente de la misma y, por tanto, necesitan un .jar con el código fuente.

5.2.4 Obligaciones generales

No nos encontramos con ninguna obligación general en la utilización de las librerías.

5.2.5 Asunciones y dependencias

No existen asunciones ni dependencias en el proyecto. Cualquiera que utilice las librerías solo necesita importarlas a su proyecto. No será necesario el uso de ningún otro paquete para utilizar las librerías.

5.3 Requerimientos específicos

5.3.1 Requerimientos funcionales

Las librerías que se van a desarrollar estarán disponibles en dos páginas web diferentes, cada una de las cuales aportará todas sus funcionalidades específicas.

Las dos librerías tendrán el mismo estilo de página y las mismas secciones, ya que tendrán el mismo objetivo: informar al usuario de todo lo que necesite para utilizar la librería.

En los sitios web de cada librería aparecerán al menos las siguientes secciones:

- **JavaCheckCode/JavaCheckStyle:** Será una página informativa para el usuario donde se explicará para qué sirve la librería y se enumerarán las funciones más importantes.
- **Documentación:** En esta sección se podrá descargar un manual de usuario en formato PDF (Portable Document Format), para que cualquier usuario pueda utilizar las librerías.
- **Javadoc:** El usuario podrá acceder a partir de esta sección a la API de las librerías.
- **Downloads:** La página web ofrecerá la descarga de las librerías y su código fuente. De esta forma, los usuarios podrán darle un uso específico.

5.3.2 Requerimientos de interfaz externos

5.3.2.1 Interfaz de usuario

Como en cualquier página Web que desee cumplir los estándares, la interfaz gráfica debe estar estrechamente sujeta al concepto de usabilidad, de manera que el usuario encuentre con rapidez y fluidez lo que deseé, a lo largo de un recorrido amigable e intuitivo para el mismo. Ante todo, la interfaz gráfica debe caracterizarse por la facilidad de uso y la sencillez.

Los enlaces sujetos a las diferentes funcionalidades dentro de la página Web deberán ser bien visibles. A su vez, se deberá respetar los estándares establecidos en relación a usabilidad, menús, estilos, colores, etc.

5.3.2.2 Interfaces hardware

No son necesarias y, por tanto, no se han definido.

5.3.2.3 Interfaces software

Una librería en Java se puede entender como un conjunto de clases que poseen una serie de métodos y atributos. Lo realmente interesante de estas librerías para Java es que facilitan muchas operaciones. De una forma más completa, las librerías Java nos permiten reutilizar código, es decir, podemos hacer uso de los métodos, clases y atributos que componen las librerías, evitando así tener que implementar nosotros mismos esas funcionalidades.

Las librerías que se van a desarrollar se pueden utilizar en cualquier proyecto Java con el simple hecho de importar sus paquetes.

Lo más interesante de las librerías Java, es que se pueden crear y hacer uso de ellas en el interior de nuestros proyectos. Básicamente un paquete en Java puede ser una librería, sin embargo una librería Java completa puede estar conformada por muchos paquetes más. Al importar un paquete podemos hacer uso de las clases, métodos y atributos que lo conforman.

Para poder utilizar las librerías que queremos desarrollar, se deberá saber que para importar librerías Java se usará la palabra clave import en el programa editor que cada usuario utilice, y seguidamente, insertar la "ruta" del paquete o clase que se desee agregar al proyecto.

Las interfaces de las librerías son las siguientes:

JavaCheckCode:

- Número de condiciones de un fichero Java.

```
public int getConditionCount() throws Exception
```

- Número de líneas de código de un fichero Java.

```
public int getLOC() throws Exception
```

- Máxima profundidad de bucles en un fichero Java.

```
public int getMaxDepth() throws Exception
```

- Número de condiciones de una clase.

```
public int getClassConditionCount(String className) throws
Exception
```

- Número de líneas de código de una clase.

```
public int getClassLOC(String className) throws Exception
```

- Máxima profundidad de una clase.

```
public int getClassMaxDepth(String className) throws
Exception
```

- Número de condiciones de un método.

```
public int getMethodConditionCount(String className, String
methodSignature) throws Exception
```

- Número de líneas de código de un método.

```
public int getMethodLOC(String className, String
methodSignature) throws Exception
```

- Máxima profundidad de un método.

```
public int getMethodMaxDepth(String className, String
methodSignature) throws Exception
```

- Tiempo de ejecución de un fichero.

```
private long getExecutionTime(Class<?> clazz, Object obj,
String methodSignature, Object[] methodArgs)
```

JavaCheckStyle:

- Errores de formato en un programa.

```
public JavaCheckStyle(String filePath) throws Exception
```

- Número de errores de formato en un programa.

```
public String getErrors() throws Exception
```

- Informe de errores de formato en un programa.

```
public String getStyleReport()
```

5.3.2.4 Interfaz de comunicaciones

No tendremos que tener en cuenta este punto para la realización de nuestro proyecto.

5.3.3 Requerimientos de eficiencia

No se han definido requerimientos de eficiencia mínimos. Sin embargo, es deseable maximizar la eficiencia de las librerías. Por eso, se realizará un estudio intensivo de testeo para poder abordar todos los casos posibles y garantizar que la implementación de las librerías sea lo más eficiente posible.

5.3.4 Obligaciones de diseño

El aspecto más importante a tener en cuenta en el diseño de la librería, es la claridad descriptiva y funcional de todas las especificaciones que necesita un usuario para poder utilizar la librería. Por ello, las dos librerías contarán con una API documentada con JavaDoc con explicaciones detalladas de cada clase.

5.3.4.1 Estándares cumplidos

En nuestras librerías no tenemos que cumplir ningún estándar, pero sí habrá que investigar sobre la guía de estilo de Java para la librería JavaCheckStyle, ya que hay que seguir un criterio para medir el estilo de un fichero.

Por otro lado, las librerías respetarán las normas propuestas por varios autores de reconocido prestigio en temas de usabilidad y accesibilidad:

- En cuanto a los requerimientos de usabilidad, se cumplirán los diferentes estándares definidos por Jakob Nielsen [Nielsen,

2000]. Jakob Nielsen es considerado por muchos [Romero, 2006] [Encinar, 2006] [Wikipedia, 2007] el mayor experto en usabilidad del mundo.

- Los sitios web también cumplirán las normas indicadas en los artículos de Eduardo Manchón [Manchón, 2005] acerca de fuentes, colores, distribución del texto, etc.

5.3.4.2 Limitaciones hardware

No se establecerán limitaciones hardware para poder utilizar las dos librerías.

5.3.5 Atributos

5.3.5.1 Seguridad

La seguridad es algo que los desarrolladores del lenguaje se toman muy en serio, continuamente liberan actualizaciones que corrigen o previenen este tipo de problemas.

En el desarrollo de las librerías se tendrá en cuenta que las librerías propuestas no produzcan agujeros de seguridad debido a su acceso al código fuente de los ficheros.

5.3.5.2 Integridad

No se requieren acciones de integridad para realizar las librerías.

6 Análisis y Diseño de la aplicación

Habiendo estudiado los requerimientos expuestos en el punto anterior, el análisis y diseño resultantes de dicho estudio se centran en orientar las librerías hacia un ámbito público, intentando desarrollar librerías de fácil uso y comprensión.

Se van a desarrollar librerías con un diseño cuidado y a la vez tradicional, con una usabilidad muy alta para evitar que los usuarios se encuentren con problemas de comprensión al utilizarlas.

6.1 Diseño de página

En este punto se analizarán los factores a tener en cuenta a la hora de diseñar las librerías y los portales web.

El diseño de un JavaDoc se realiza a partir de una utilidad de Oracle para la generación de documentación de APIs en formato HTML a partir de código fuente Java. Este diseño se hace de forma automática desde la herramienta eclipse que se utilizará para implementar las librerías. Para generar la documentación de un proyecto automáticamente se tendrán que seguir unas normas a la hora de realizar los comentarios JavaDoc dentro del mismo, por ejemplo:

```
/**
 * The class checks the style of a file, creating a document with all
the errors found.
 * @author Ana Navarrete Durante
 */
```

El diseño de los portales web se realiza a partir de páginas, siendo el elemento central de los portales, ya que cualquier información hacia el usuario se mostrará en ellas.

Los portales web mantendrán una estructura uniforme en cada una de sus páginas para intentar no confundir al usuario y mantener un cierto orden y facilidad de uso.

A continuación se muestran detallados una serie de aspectos a tener en cuenta.

6.1.1 Espacio de pantalla

No se tendrá que pensar en un diseño de pantalla porque las páginas se crean automáticamente.

Los estándares de diseño de librerías Java emplean la siguiente distribución en la especificación de su API:

- En la parte izquierda se situarán dos menús. En el menú superior se mostrará un listado con los distintos paquetes que tendrá la librería. En el menú inferior, se visualizarán todas las clases y se accederán a ellas mediante enlaces.
- En lo que queda de pantalla a partir de los dos menús nombrados anteriormente, se encontrará toda la información que hayamos seleccionado en los menús. Se documentará todo lo necesario para poder utilizar las librerías, detallando todas las funciones creadas.

En el diseño de los portales web se va a intentar mostrar en las páginas la máxima cantidad de información posible de una manera clara y ordenada.

Las normas de diseño de páginas Web indican que la parte de la página correspondiente a información debería ocupar entre un 60% y un 80% del área visible de la página, dejando el resto para zonas de navegación y opciones [Nielsen, 2000]. El portal web que se va a realizar dedicará aproximadamente un 80% del espacio a información destinada al usuario, mientras que aproximadamente el 20% restante se dedicará a la zona de menú para navegar entre páginas.

La distribución de los objetos en las páginas se realizará basándose en el hecho de que los usuarios cuando navegan por páginas Web tienden siempre a buscar automáticamente determinados elementos en determinados lugares de las páginas. Por ello en el portal web se va a emplear la siguiente distribución:

- En la parte superior de las páginas se situará el menú de navegación entre las distintas páginas. Este menú se encontrará en el mismo lugar en todas las páginas para facilitar la navegación.
- El título de la página siempre se incluirá al principio de las mismas, en la parte central y con un tipo de letra más grande que el resto, de este modo al usuario le será fácil identificar en qué página se encuentra.
- La información buscada se ubicará en el centro de la página. Este es el lugar donde normalmente los usuarios intentan ubicar la información cuando navegan por páginas Web. Esta información puede ser de varios tipos: imágenes, texto o ejemplos de implementación.

Esta distribución se encontrará en el mismo lugar en todas las páginas para facilitar la navegación. Normalmente, esta posición del menú de navegación es la esperada por los usuarios.

6.1.2 Navegación

Las librerías permitirán el acceso a cualquier paquete, clase o método de las mismas por parte del usuario únicamente pulsando en el enlace.

Se situará el menú de navegación en la parte izquierda de la pantalla dispuesto de manera vertical, ofreciendo el listado de todos los paquetes y clases que ofrezcan las librerías.

Los portales web permitirán el acceso a cualquier página de los mismos por parte del usuario únicamente pulsando un botón.

Las páginas normalmente no contendrán botones para ir a determinadas páginas como la página principal, ya que en todas las páginas de los portales estará disponible el menú en su parte superior.

6.1.3 Resolución de pantalla

No tendremos que tener en cuenta la resolución de pantalla, ya que se generan las librerías automáticamente y este aspecto es tenido en cuenta por el generador.

Por otro lado, la mayoría de los monitores actuales son de 17 pulgadas, por lo que se va a poder utilizar una resolución de pantalla de 1024x768 píxeles como resolución mínima en el portal web. Esta resolución es la resolución adecuada para el portal web, ya que la utilización de una resolución menor supondría no poder mostrar toda la información esencial en la parte visible de las páginas.

6.1.4 Colores

Los colores que presentarán las librerías es el que utiliza el estándar al crear un JavaDoc. Dispone de una usabilidad muy alta, el fondo de las librerías será blanco, dejando los enlaces de todos los paquetes, clases y métodos en color azul y resaltando cada título en negrita.

A continuación se expondrán los valores numéricos de los colores de los elementos más importantes:

- Color de fondo de la página: #ffffff (Blanco)

- Color de enlaces: #4D7A97 (Azul)
- Color de los bordes y separadores: #4D7A97 (Azul)
- Color del texto y títulos: #000000 (Negro)

Por otro lado, para el diseño de colores de las páginas del portal web se usa una gama de colores claros, para que así no entorpezcan la lectura de la información ni distraigan al usuario.

En los portales Web en los que se quiera disponer de una usabilidad muy alta, el fondo de las páginas que contienen texto debería ser normalmente blanco, a no ser que se quiera separar distintas partes del contenido. Esto es así porque el lector cuando observa un texto sobre un color de fondo le resta automáticamente importancia a dicho texto, este es un proceso cognitivo del usuario para evitar procesar información no relacionada con sus objetivos [Manchón, 2004].

El color de fondo de las páginas va a ser blanco y el del menú horizontal negro. El uso de estos colores no supondrá una desviación de la atención del usuario.

6.1.5 Vinculación

La vinculación entre los distintos menús que forman la librería se va a realizar por medio de enlaces con el nombre de cada uno de los paquetes, clases o métodos.

La única variación que se puede visualizar es en la parte donde se muestra toda la información. Habrá un menú superior que dispondrá de botones para cambiar la forma de visualización de pantalla:

El menú dispondrá de 4 opciones, y estas serán:

- Package
- Tree
- Deprecated
- Index

Los botones del menú serán de un tono azulado acorde con el resto de colores de la aplicación y el texto de color blanco.

Por la parte de los portales web, la vinculación entre las distintas páginas que forman el portal web se va a realizar por medio de botones con un texto descriptivo que formarán enlaces a las distintas páginas del portal.

En la barra superior será donde se ubiquen todos estos enlaces, en forma de menú horizontal.

El menú dispondrá de 4 opciones, y estas serán:

- About
- Documentation
- Downloads
- Contact

6.1.6 Conclusiones del diseño de página

Como se ha expuesto en los puntos anteriores de esta memoria, en el diseño de las librerías y de los portales web ha primado la facilidad de uso, ya que el diseño de la librería se crea automáticamente y no se modificará y en los portales web la usabilidad y simplicidad a la hora de encontrar toda la información aportada. Hay que mencionar que el diseño es familiar al usuario y eso es una característica para el objetivo del proyecto.

6.2 Diseño del contenido

Los resultados que los usuarios esperan obtener de una librería y de un portal web cuando los utilizan forman parte del contenido de la misma.

Este contenido debe ser claro y descriptivo, por eso se tendrá que realizar todo tipo de comentarios JavaDoc para detallar toda la información de las librerías de la forma más correcta y un lenguaje adecuado en toda la información que contendrán los portales web. En el caso de que estas dos premisas no se cumplieran el usuario no podría obtener la información que necesite en el portal web, ni podría utilizar la librería de manera cómoda, esto desembocaría en una pérdida importante de usabilidad.

6.2.1 Lenguaje claro

Cualquier librería y portal web debe contener textos escritos en un lenguaje claro y fácilmente entendible por el usuario al que va dirigido.

Cada función o clase de las librerías poseerá toda la información necesaria para su uso. Describirá detalladamente el resultado de la función, sus argumentos y las posibles excepciones que pudieran ocurrir durante su ejecución. En los portales web se describirá de una forma

clara toda la información necesaria para utilizar las librerías y poder obtenerlas de manera fácil y familiar para el usuario.

6.2.2 Conclusiones sobre el contenido

Las librerías y los portales web deben ofrecer un contenido claro, conciso y fácilmente legible por parte del usuario.

En este proyecto es una parte importante, ya que el contenido para el usuario será la base para que pueda interpretar todas las funciones que se realicen en la librería.

Los textos que se mostrarán en el contenido de cada función van a ser muy cortos y explicativos, no serán textos de contenido, por lo que será suficiente con que sean claros y sencillos.

7 Implementación

En este capítulo se introducirá al lector en las distintas herramientas que se han utilizado para el desarrollo de la aplicación. Seguidamente, se hará una explicación detallada de cada función de las librerías, detallando cómo se ha realizado y los problemas que han surgido durante el desarrollo.

7.1 Tecnologías de soporte a la aplicación

En este apartado se describe el lenguaje utilizado en la programación de la librería.

7.1.1 El lenguaje Java

Java es un lenguaje de programación y una plataforma informática comercializada por primera vez en 1995 por Sun Microsystems. Es un lenguaje orientado a objetos que fue desarrollado para tener pocas dependencias de implementación. El lenguaje de programación Java permite simplificar el código y evita la necesidad de copiar y pegar muchas veces un mismo procedimiento.

Java es multiplataforma y permite hacer que una misma aplicación escrita una sola vez funcione en todos los entornos ya que el ejecutable de Java no lo ejecuta el sistema operativo sino la máquina virtual JVM (Java Virtual Machine).

El propósito general de utilizar este lenguaje, a parte de la familiaridad del mismo por utilizarlo en toda la carrera, es por ser expandible. Java tiene la posibilidad de ofrecer una gran cantidad de librerías que los programadores independientes han puesto a disposición de la comunidad para crear funcionalidades a aplicaciones o a empresas.

No todos los procedimientos que se pueden llegar a necesitar están contenidos en las librerías nativas de Java, hay muchísimas librerías de programadores independientes y empresas que amplían su funcionalidad. Cuando se finalice este proyecto habrá dos librerías más accesibles para todos los desarrolladores.

A la hora de decidirse por un lenguaje de programación se valoraron las innumerables ventajas que proporciona Java, así como también se valoró que es el lenguaje más utilizado para realizar la programación de librerías [Wikipedia JAVA].

7.1.2 HTML

HTML es un lenguaje de marcación diseñado para estructurar textos y presentarlos en forma de hipertexto, que es el formato estándar de las páginas Web. Gracias a Internet y a los navegadores del tipo Internet Explorer, Opera, Firefox o Netscape, HTML se ha convertido en uno de los formatos más populares que existen para la construcción de documentos y también de los más fáciles de aprender.

HTML es una aplicación de SGML conforme al estándar internacional ISO 8879. XHTML es una reformulación de HTML 4 como aplicación XML 1.0, y que supone la base para la evolución estable de este lenguaje. Además, XHTML permite la compatibilidad con los agentes de usuario que ya admitían HTML 4 siguiendo un conjunto de reglas.

HTML es una herramienta básica y fundamental para el desarrollo de páginas Web y aplicaciones Web. Constituye la base de toda página Web. El lenguaje HTML puede ser creado y editado con cualquier editor de textos básico, aunque existen editores avanzados con unas características muy adecuadas para la escritura de código HTML [Wikipedia. HTML].

Para la elaboración de las páginas Webs se ha necesitado esta herramienta, con la que hacemos un uso mínimo de la misma pero si necesario para poder difundirla vía online.

7.1.3 CSS

Las hojas de estilo en cascada CSS (Cascading Style Sheets) son un lenguaje formal usado para definir la presentación de un documento estructurado escrito en HTML o XML (y por extensión en XHTML). El W3C es el encargado de formular la especificación de las hojas de estilo que servirá de estándar para los agentes de usuario o navegadores.

El objetivo principal que se pretende al utilizar CSS es separar la estructura de un documento de su presentación.

En la página Web que se ha desarrollado se utiliza una hoja de estilos CSS, ya que esto presenta innumerables ventajas:

- Mejora la accesibilidad de la página.
- Permite realizar cambios en la presentación de la página rápidamente.
- Permite la utilización de divisiones para realizar la maquetación de las páginas.

- Se consigue reducir bastante el tamaño de las páginas.
- Se evita la existencia de código fuente duplicado correspondiente a los estilos.

Estos motivos son suficientemente importantes como para que en la programación de la Web se utilicen hojas de estilo.

7.1.4 XML

XML[2] es un metalenguaje extensible de etiquetas desarrollado por el W3C. Es una simplificación y adaptación del SGML y permite definir la gramática de lenguajes específicos (de la misma manera que HTML es a su vez un lenguaje definido por SGML). Por lo tanto, XML no es realmente un lenguaje en particular, sino una manera de definir lenguajes para diferentes necesidades. Algunos de estos lenguajes que usan XML para su definición son XHTML, SVG, MathML.

XML no fue creado únicamente para su aplicación en Internet, sino que se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas. Se puede usar en bases de datos, editores de texto, hojas de cálculo y casi cualquier cosa imaginable.

XML es una tecnología sencilla, que tiene a su alrededor otras que la complementan y la hacen mucho más completa y con unas posibilidades mucho mayores. Tiene un papel muy importante en la actualidad ya que permite la compatibilidad entre sistemas para compartir la información de una manera segura, fiable y fácil [Wikipedia. XML].

La estructura de un fichero XML se crea por medio de etiquetas, dentro de las cuales se colocan los valores asociados a cada etiqueta. Al utilizar texto plano, es compatible con todo tipo de sistemas y es el modo que tenemos de mandar peticiones a la cámara.

La utilización del lenguaje XML es necesaria en la librería *JavaCheckStyle*, debido a que debemos utilizar una librería externa *CheckStyle* que nos aporta multitud de funciones de estilo y solo escogemos las que más nos interesan para el proyecto.

7.1.5 Aplicaciones utilizadas para la programación

A continuación se procederá a exponer una breve explicación sobre las aplicaciones utilizadas para programar en los distintos lenguajes.

[2] eXtensible Markup Language (lenguaje de marcas extensible).

7.1.5.1 Eclipse Luna

Como editor para el lenguaje Java, CSS y HTML se ha utilizado Eclipse Luna, ya que también lo había utilizado anteriormente en varias asignaturas y es una plataforma fácil de usar y te facilita mucha información a la hora de programar.Eclipse es un programa informático compuesto por un conjunto de herramientas de programación de código abierto multiplataforma para desarrollar Aplicaciones de Cliente Enriquecido.

Esta plataforma típicamente ha sido usada para desarrollar entornos de desarrollo integrados (del inglés IDE). Eclipse es también una comunidad de usuarios, extendiendo constantemente las áreas de aplicación cubiertas.

Eclipse fue desarrollado originalmente por IBM como el sucesor de su familia de herramientas para *VisualAge*. Eclipse es ahora desarrollado por la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y un conjunto de productos complementarios, capacidades y servicios.

Este editor proporciona una interfaz sencilla y muy amigable para programar. Tiene funcionalidades para ayudar al programador en las tareas, las más importantes son:

- Auto completado de símbolos, como son por ejemplo las comillas.
- Cuando se está escribiendo el nombre de una variable Java aparece la opción de auto completar el nombre.
- Al escribir el nombre de una función Java muestra información de los parámetros que necesita.
- Marcación de etiquetas, como por ejemplo cuando se escribe una llave cerrada el programa marca la llave de apertura con la que se corresponde.
- Explorador de variables y clases.

7.2 Descripción de la aplicación

En el presente apartado se detallan con minuciosidad las distintas funcionalidades que poseen las librerías creadas.

Este apartado también aborda los distintos problemas que han surgido en la realización de las librerías, así como las distintas soluciones utilizadas para la resolución de dichos problemas.

7.2.1 Visión general de las librerías

Se han desarrollado dos librerías Java, una para poder medir la eficiencia de un algoritmo y la otra librería para medir un correcto estilo del proyecto Java. El usuario recibirá la información captada por las diferentes funciones que se le solicite a las librerías.

Las librerías creadas son de acceso libre e independiente. Asimismo, cada librería se puede utilizar de la manera deseada por el usuario.

Las librerías son capaces de realizar todos los objetivos marcados y cada una de ellas incluye la utilización de librerías externas para alcanzar los resultados deseados:

JavaCheckCode

- Información de un fichero Java.
- Información de una clase.
- Información de un método.
- Medición del tiempo de ejecución de un algoritmo.

Se ha comentado anteriormente que las librerías dependen de otras librerías externas que facilitan información y funcionalidad de las librerías creadas. La librería JavaCheckCode utiliza varias funcionalidades pertenecientes a la librería JavaParser:

- Información detallada de ficheros Java.
- Información de todas las clases integradas en un fichero Java.
- Información de todos los métodos integrados en una clase de un fichero Java.
- Información de todas las condiciones y bucles que hay en el interior de todos los métodos.

A partir de todos los resultados que recogemos de la librería *JavaParser* hemos podido implementar la funcionalidad de nuestras funciones.

JavaCheckStyle

- Número de errores de estilos de un fichero Java.

- Mostrar en pantalla los errores de estilo de un fichero Java.
- Crear un informe con los errores de estilo de un fichero Java.

Como hemos comentado anteriormente, nos ha sido de gran utilidad poder tener acceso a una librería libre de Java "CheckStyle", que mide diferentes exigencias de estilo de un fichero Java.

7.2.2 JavaCheckCode

Como comentamos anteriormente, las librerías de Java mantienen un estándar en su formato. Realizamos una captura de la página principal de la librería.

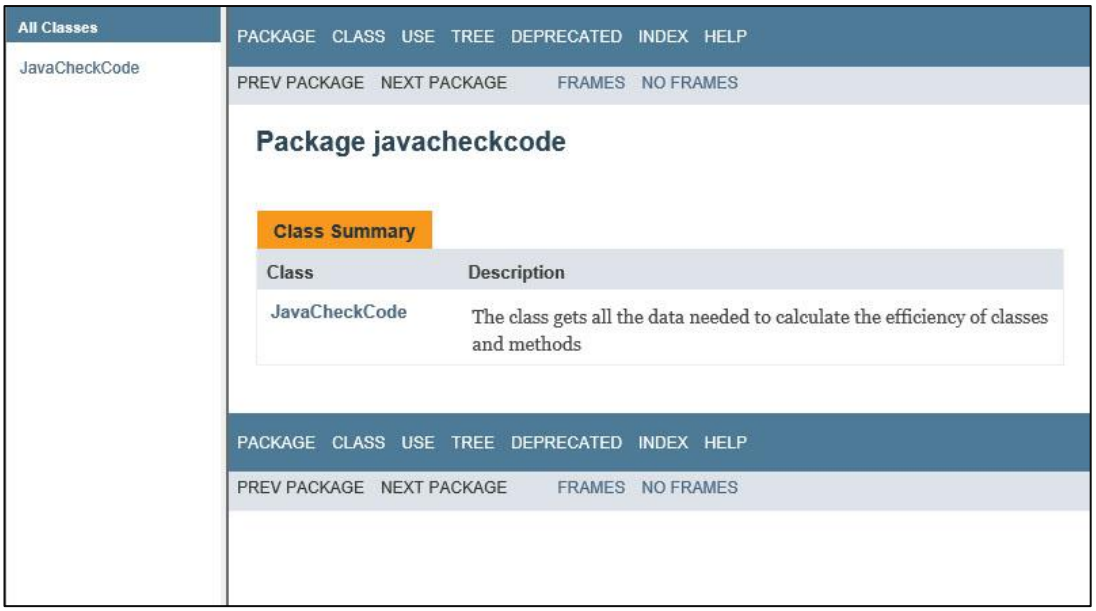


Figura 1. Portada principal librería JavaCheckCode

Todos los usuarios del sistema disponen de todas las funciones mediante el cual un usuario puede moverse por cualquier funcionalidad de la librería. Al hacer clic en el menú de la izquierda, nos ofrece todas las funciones que se encuentran en la librería.

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
java.lang.Object	createObject(java.lang.String className, java.lang.Object[] args) Gets a constructor Object using the class name and arguments.
int	getClassConditionCount(java.lang.String className) Gets the number of conditions of the class.
int	getClassLOC(java.lang.String className) Gets the number lines of code of the class.
int	getClassMaxDepth(java.lang.String className) Gets the maximun depth of the class.
int	getConditionCount() Gets the number of conditions of the file.
void	getExecutionTime(java.lang.Object obj, java.lang.String methodSignature, java.lang.Object[] methodArgs) The execution time of file when the object is passed as an argument.
void	getExecutionTime(java.lang.String className, java.lang.Object[] args, java.lang.String methodSignature, java.lang.Object[] methodArgs) The execution time of file when the method belongs to an object we have to create us.
void	getExecutionTime(java.lang.String className, java.lang.String methodSignature, java.lang.Object[] methodArgs) The execution time of file when the method is static.
int	getLOC() Gets the number lines of code of the class.

Figura 2. Funciones de la librería JavaCheckCode

En cada una de las funciones de la librería se encuentra una pequeña descripción de lo que ésta realiza. A continuación vamos a detallar como se han realizado las diferentes funcionalidades.

Antes de pasar por cada una de las funciones, debemos explicar la razón de por qué hemos utilizado la librería JavaParser para el desarrollo de nuestra librería JavaCheckCode.

JavaParser

Es un analizador de Java 1.5 con la generación de AST y apoyo visitante. Con la API, se puede acceder a varios elementos del código fuente como la clase, métodos, declaraciones, JavaDoc, comentarios, etc.

También se pueden cambiar estos elementos y guardarlos en un archivo nuevo. Esto permite esencialmente una transformación o generación de código fuente. En nuestro proyecto lo vamos a utilizar para leer el código fuente de Java de un determinado algoritmo, y generar código de implementación.

JavaParser es bastante fácil de usar si se está familiarizado con el patrón de diseño visitante. Se cubren muchas de las necesidades que hemos tenido para desarrollar esta librería. Vamos a ir explicando cómo hemos recogido esa información con JavaParser.

Realizamos una captura para que se pueda observar lo potente que es esta librería y muchas de las funciones que se pueden utilizar.

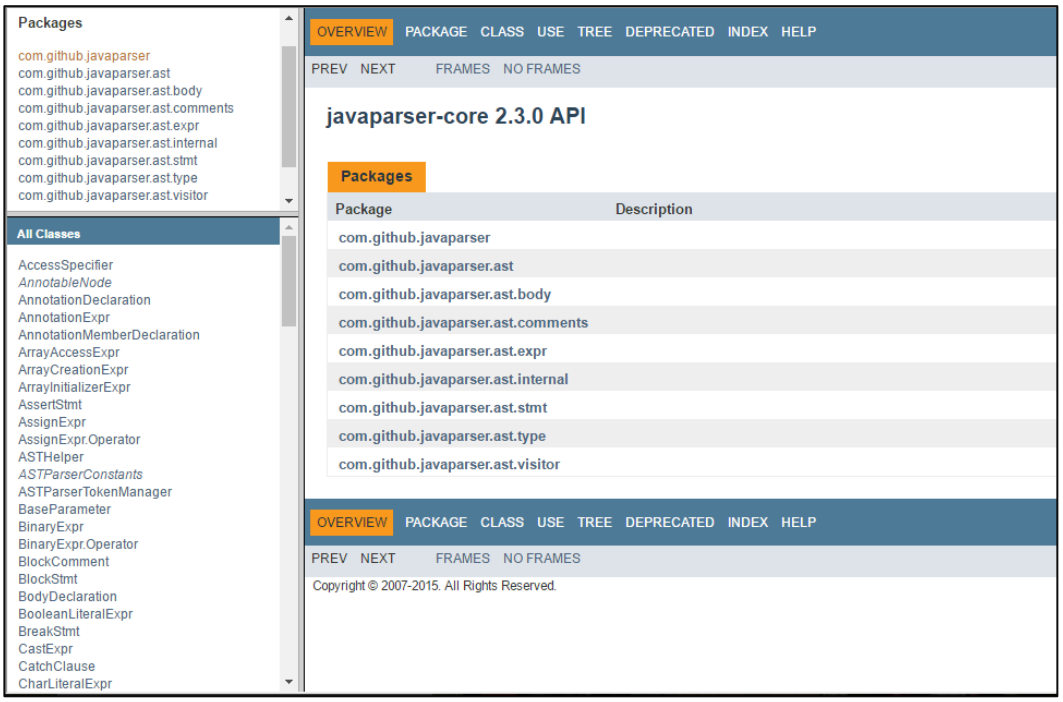


Figura 3. Portada principal librería JavaParser

7.2.2.1 Tratamiento de un fichero Java

La librería necesita utilizar un fichero Java. A partir de él, se construye su árbol de sintaxis abstracta con JavaParser. Finalmente, mediante un objeto de la clase Visitor realizamos la llamada visit. Con esta llamada podemos obtener toda la información deseada que haya en el interior del fichero java.

Para poder acceder a todos los datos que tenemos que visitar hemos tenido que realizar las siguientes líneas de código.

```
CompilationUnit cu = JavaParser.parse(file);
Visitor visitor = new Visitor();
visitor.visit(cu, null);
```

Se puede observar en el bloque de código que para el correcto funcionamiento de la librería hay que tener en cuenta:

- CompilationUnit: Es la llamada a una de las clases de Javaparser para poder visitar todo lo que hay dentro de un fichero Java.
- File: Se puede observar que uno de los parámetros que exige la creación del objeto CompilationUnit es el fichero (tipo File), donde se le indica el fichero a parsear.
- Visitor: Es la clase de más peso de la librería, ya que a partir de ella recogemos toda la información para poder tratarla según las especificaciones seguidas en el proyecto.
- Llamada al método visit: Esta última línea de código, realiza la llamada al método visit de la librería JavaParser para poder visitar todo el código fuente del fichero.

7.2.2.2 CompilationUnit

Una de las clases más importantes de la librería es la clase Visitor. Los objetos de la clase visitor atraviesan el árbol de sintaxis abstracta de un fichero Java. Al atravesar el fichero, podemos realizar funciones para modificar el propio fichero Java. Por ejemplo, en la clase Visitor implementamos diferentes funciones con las que contamos el número de atributos de un fichero Java o añadimos atributos, argumentos, etc.

Comenzaremos hablando del tipo de objeto CompilationUnit de la librería JavaParser que vamos a visitar:

```
public void visit(CompilationUnit cu, Object args){
      if(cu.getPackage() != null)
            packageName = cu.getPackage().getName().getName();
      else packageName="";
      int beginLine = cu.getBeginLine();
      int endLine = cu.getEndLine();
      fichero.setPackage(packageName);
      fichero.setBounds(beginLine, endLine);
      super.visit(cu, args);
}
```

Por lo que hemos comentado anteriormente, el objeto CompilationUnit hace referencia a un fichero, y con él podemos averiguar todo lo que hay dentro.

Vamos a realizar una breve explicación de lo que queremos obtener con este método:

- Paquetes: Tenemos que tener en cuenta si el fichero Java está contenido dentro de un paquete, para ello lo obtenemos y lo guardamos en una clase creada para nuestra librería llamada Fichero, una de las funciones creadas dentro de esta clase ha sido para realizar esta comprobación.
- Líneas: Obtenemos la línea donde empieza y acaba el fichero para posteriormente tratarlas.

7.2.2.3 ClassDeclaration

Con el objeto ClassDeclaration conseguimos saber toda la información que hay en el interior de nuestra clase.

Para ello se realiza una llamada al método visit para recorrer todos los elementos y conseguir la estructura deseada de la clase que queremos visitar.

```
public void visit(ClassOrInterfaceDeclaration classDeclaration, Object args)
{
	linesClass = 0;
	int beginLine = classDeclaration.getBeginLine();
	int endLine = classDeclaration.getEndLine();
	String nameClass = classDeclaration.getName();
	Clazz auxClass = new Clazz();
	this.clazz = auxClass;
	this.clazz.setName(nameClass);
	fichero.addClazz(auxClass);
	this.clazz.setLoc(beginLine, endLine);

	this.visited(classDeclaration, args);
}
```

Con este método podemos averiguar diferentes características de una clase utilizando las funciones de la librería JavaParser.

Cuando visitamos una clase interna de un fichero, queremos conseguir toda la información necesaria para las propiedades de nuestra librería y necesitamos obtener la siguiente información:

- Líneas: La primera y última línea de una clase, desde que se declara hasta la llave final.

- Nombre: Nombre de la clase que estamos visitando, para ir guardando todos los datos necesarios en la clase "Clazz" creada para nuestra librería.

Añadimos todas las clases que se encuentran en un fichero y para ello tenemos creada una lista en la clase "Fichero" para guardar la información necesaria.

7.2.2.4 MethodDeclaration

Para el siguiente objeto de la librería JavaParser se utiliza la misma función de visitar los métodos que hay en un fichero.

El siguiente código de ejemplo corresponde a la llamada del método visit de la librería JavaParser para obtener toda la información que hay en el interior de cada uno de los métodos que vamos a visitar:

```
public void visit(MethodDeclaration methodDeclaration, Object args) {
      linesMethod = 0;
      currentDepth = 0;
      parameters = methodDeclaration.getParameters();
      int beginLine = methodDeclaration.getBeginLine();
      int endLine = methodDeclaration.getEndLine();
      String methodName= methodDeclaration.getName();
      parameters = methodDeclaration.getParameters();
      if (parameters == null)
         p = "[]";
      else p = parameters.toString();
      p = p.substring(1, p.length()-1);
      String signature = methodName + "("+p+")";
      Method methodAux = new Method();
      methodAux.setSignature(signature);
      this.clazz.addMethod(methodAux);
      this.method = methodAux;
      this.method.setLoc(beginLine, endLine);

      this.visited(methodDeclaration, args);
}
```

Se puede observar en el código todas las líneas necesarias para obtener la información deseada para las funciones especificadas en nuestra librería.

Vamos a realizar una breve explicación de todo lo que necesitamos saber de los métodos que visitamos utilizando la librería JavaParser:

- Parámetros: Necesitamos conocer los diferentes parámetros que contiene el método visitado.
- Líneas: Obtenemos la primera y última línea del método que estemos visitando, para poder tratarlas posteriormente en una clase realizada para obtener el número exacto de líneas de código que forman el método.
- Nombre: Tenemos que tener en cuenta los diferentes métodos que hay en el interior de un fichero y para ello debemos guardar los nombres de todos ellos creando un objeto "Método" realizado para nuestra librería.

Al igual que las clases se añaden en un fichero, los métodos se deben añadir a una lista creada en la clase Clazz. Esta clase se utiliza para guardar toda la información que hay en el interior de una clase. Por este motivo, debemos ir añadiendo todos los métodos que se encuentren en el interior de la clase que estemos visitado.

7.2.2.5 Tipos de bucles

Una de las medidas necesarias que se propusieron como objetivo en nuestro proyecto es tener información de los bucles que hay en el interior de un fichero Java.

Todos los bucles que aparezcan en el fichero Java sobre los que queramos obtener información van a realizar las mismas iteraciones y modificar los mismos parámetros.

Vamos a mostrar el código que utilizamos para obtener toda la información de los bucles que visitemos:

- For
- Foreach
- While
- Do While

```
public void visit(WhileStmt ws, Object args) {
      this.fichero.setCond(this.fichero.getCond() + 1);
      this.clazz.setCond(this.clazz.getCond() + 1);
      this.method.setCond(this.method.getCond() + 1);
      currentDepth++;
      if (currentDepth > this.fichero.getDepth())
            this.fichero.setDepth(currentDepth);
      if (currentDepth > this.clazz.getDepth())
            this.clazz.setDepth(currentDepth);
      if (currentDepth > this.method.getDepth())
            this.method.setDepth(currentDepth);
      super.visit(ws, args);
      currentDepth--;
}
```

Como ejemplo de uno de ellos, hemos seleccionado el método que visita un bucle While.

7.2.2.6 Tipos de condiciones

Al igual que los bucles, medir el número de condiciones sirve como métrica para medir el tamaño y la complejidad de un código Java.

Cuando visitamos las diferentes condiciones que hay en los ficheros, clases y métodos, solo necesitamos conocer la cantidad que hay. Para ello, incrementamos el número de condiciones que hay en cada uno de ellos.

Además de los bucles ya mencionados anteriormente, también se tiene que tener en cuenta la cantidad de condiciones de:

- Switch
- If

Mostamos el código correspondiente a estas funciones, siendo idénticas a las tres primeras líneas del código mostrado anteriormente.

```
public void visit(SwitchStmt ss, Object args) {
      this.fichero.setCond(this.fichero.getCond() + 1);
      this.clazz.setCond(this.clazz.getCond() + 1);
      this.method.setCond(this.method.getCond() + 1);
      super.visit(ss, args);
}
```

```
public void visit(IfStmt is, Object args) {
      this.fichero.setCond(this.fichero.getCond() + 1);
      this.clazz.setCond(this.clazz.getCond() + 1);
      this.method.setCond(this.method.getCond() + 1);
      super.visit(is, args);
}
```

7.2.2.7 Características de un fichero

Hemos tenido que realizar todas las explicaciones necesarias que se realizan en la clase Visitor para que se pueda entender de qué manera recogemos todos los datos mediante la librería JavaParser.

En este apartado vamos a explicar las diferentes funciones que obtienen información de un fichero y qué cosas hay que tener en cuenta para obtener los datos necesarios.

A continuación detallamos las funciones principales de nuestra librería y la forma en la que conseguimos toda la información.

7.2.2.7.1 Condiciones de un fichero

La primera función de la librería que vamos a explicar es la que obtiene el número de condiciones que hay en un fichero Java.

Como ya hemos comentado en puntos anteriores, para obtener el número de condiciones que hay en el interior de un fichero necesitamos conocer el número total de bucles y condiciones.

Cada vez que se visita un fichero y se recorre una condición o bucle se modifica el valor que almacena el número de condiciones que hay en el fichero, aumentándolo en uno.

Toda la información se va modificando y añadiendo a la clase Fichero creada para la librería *JavaCheckCode*. Asimismo, todos los cambios y consultas realizadas de un fichero, se guardaran ahí.

Por lo tanto en nuestra librería, todos los datos están guardados en la clase Fichero, ya que todas las consultas se realizan a partir de ese objeto.

7.2.2.7.2 Número de líneas de código de un fichero

Por otro lado, hay que recordar lo que hemos comentado anteriormente sobre el número de líneas.

Cuando visitamos el fichero nos guardamos el número de línea inicial y final, y a partir de esa información sabemos en qué línea tenemos que empezar a leer y hasta donde tenemos que llegar para poder obtener el número exacto de líneas del fichero.

Para realizar una consulta de este tipo, no nos hemos podido conformar con el número de líneas totales que hay en un fichero, teniendo en cuenta comentarios y líneas en blanco.

Hemos tenido que tener en cuenta tres características que podemos encontrar en un fichero para poder mostrar al usuario, el número de líneas de código que hay.

Estos son los tres casos que debemos contemplar para el tratamiento de comentarios:

- Comentario de línea (//)
- Comentario (/*)
- Comillas (“”)

Para realizar un recuento correcto de líneas, llegamos a la conclusión de que cuando empezáramos a leer un comentario de línea, teníamos que llegar hasta un salto de línea y no aumentar el contador cuando se trata de este primer caso, sino hay ninguna instrucción antes de empezar el comentario en línea.

Con el comentario de varias líneas, tenemos que tener en cuenta que cuando empieza con los caracteres seguidos (/*) sin estar dentro de unas comillas, tenemos que llegar a otro grupo de caracteres como (*/). De esta forma, podemos saber que lo que haya entre esos caracteres no se tiene que tener en cuenta como número de líneas.

Y por último, nos dimos cuenta de que dentro de unas comillas también podría haber (barra-barra) o (barra-asterisco) y puede llegar a confundir al tratar los casos anteriores. Para ello, cuando hay unas comillas, se espera a comprobar todo lo que hay en su interior hasta llegar al cierre de comillas, teniendo en cuenta que dentro de ellas los caracteres de comentarios, no se pueden tratar como tal.

7.2.2.7.3 Profundidad máxima de un fichero

Recordando puntos anteriores, cuando visitamos el fichero y encontramos un bucle aumentamos la profundidad, ya que al ver un bucle se tiene que aumentar en uno el parámetro de profundidad y de esa forma tenemos que tener en cuenta la profundidad máxima que

haya en ese momento, para comprobar cuál de las dos es mayor y actualizar así la profundidad máxima si hiciera falta.

Cuando hay bucles anidados, hay que tener en cuenta todos los que hay en el interior del bucle más externo e ir actualizando la profundidad.

7.2.2.8 Características de una clase

Hemos tenido que realizar las mismas operaciones de los ficheros con las clases.

En este caso hemos tenido que tener en cuenta una cosa más. En un fichero pueden aparecer varias clases con el mismo nombre y hay que tener en cuenta la ruta del nombre de una clase.

Principalmente, hay que tener en cuenta si el fichero contiene un paquete, si es el caso, el nombre del paquete y la clase a visitar es el nombre de la clase completo. De lo contrario, sería solo el nombre de la clase.

Si hay varias clases dentro de un fichero añadidas a una lista dentro del fichero, para ir añadiendo el nombre de las clases visitadas, introducimos el símbolo "$" al nombre de la clase anteriormente visitada y seguidamente el nombre de la clase que se está visitando actualmente.

Ya teniendo en cuenta esta característica comenzamos a describir las distintas implementaciones para averiguar todo lo que hay en el interior de una clase.

7.2.2.8.1 Condiciones de una clase

Al poder haber varias clases en un fichero, hay que realizar un reseteado de las variables que utilizan en común todas las clases.

Antes de visitar una clase, inicializamos el valor de número de condiciones, para tener en cuenta las condiciones de la clase donde estemos.

Cada una de las condiciones o bucles que el visitador encuentra en el interior de una clase van aumentando en uno el número de condiciones totales de la misma.

Actualizando, en este caso, todos los valores en el objeto Clazz. Para que, cuando estemos tratando con la librería JavaCheckCode, no necesitemoslidiar con la clase Visitor y el usuario solo requiera realizar llamadas al objeto de tipo Clazz.

7.2.2.8.2 Número de líneas de código de una clase

Al igual que con los ficheros, a partir de la primera y la última línea de la clase realizamos una búsqueda de los diferentes casos que no tenemos que tener en cuenta para contar como número de líneas de código dentro de una clase.

7.2.2.8.3 Profundidad máxima de los bucles de una clase

Para conseguir la profundidad máxima de una clase, tenemos el caso en el que hay más de una clase en el interior de un fichero. Por ese motivo, tenemos que resetear el parámetro de profundidad máxima para que, cada vez que el visitador encuentre una clase, el contador de la profundidad máxima esté a cero.

Por lo demás, se realizan los mismos cambios cuando en el interior de una clase nos encontramos con un bucle. Incrementamos el contador de profundidad y al terminar de visitar la condición disminuimos el contador.

7.2.2.9 Características de un método

En el siguiente punto vamos a realizar las mismas operaciones nombradas anteriormente que hemos implementado para los ficheros y las clases.

En este caso, también puede haber varios métodos con el mismo nombre pero con parámetros diferentes.

Hemos tenido que tener en cuenta la signatura completa del método para poder diferenciar métodos con el mismo nombre. Cuando encontramos un método visitando una clase, utilizamos la librería JavaParser para averiguar los parámetros del método y concatenar una cadena String con el nombre del mismo y los parámetros que contiene entre paréntesis.

De esta forma podemos estar seguros de que no haya confusiones con las comprobaciones realizadas al solicitar información de un método específico.

Al conocer el nombre del método, lo añadimos a la lista de métodos de la clase en la que se encuentre para empezar a visitar el método.

7.2.2.9.1 Condiciones de un método

Para poder obtener toda la información sobre las condiciones que hay en el interior de un método tenemos que tener en cuenta que puede haber muchos métodos y hay que resetear el número de condiciones cada vez que visitemos un método.

De esta forma nos aseguramos de que cuando se visite el método, empiece desde cero.

Posteriormente, cada vez que se encuentre con una condición o bucle se aumentará el contador en uno y al final de ser visitado tendrá todas las condiciones que se encuentran dentro de él.

Actualizamos los valores guardándolos en una clase llamada Método creada para el funcionamiento de la librería JavaCheckStyle.

7.2.2.9.2 Número de líneas de código de un método

Realizamos los mismos accesos al método para obtener la primera y última línea del bloque de un método.

Para realizar todo el proceso de contar el número de líneas de código que hay hemos tenido que realizar una clase "ContarLOC", con esta clase tenemos en cuenta todos los casos para ceñirnos a la lectura de las líneas de código que nos interesa.

Una de las ventajas de Java es que se puede realizar un código y utilizarlo en cualquier clase, sin tener que repetir el mismo código en todas, por ese motivo hemos realizado la clase ContarLOC para utilizarla en todos los casos que necesitamos.

7.2.2.9.3 Profundidad máxima de un método

Hemos tenido en cuenta el mismo desarrollo para los métodos y, como en el caso anterior, también debemos realizar un reseteo de la profundidad máxima.

Guardamos datos únicos de cada método en la clase Method creada para esta librería.

7.2.2.10 Tiempo de ejecución

```
private long getExecutionTime(Class<?> clazz, Object obj, String methodSignature, Object[] methodArgs)
{
  try {
        JavaCompiler.compileFile(this.projectPath, new File(this.pathFile));
        String[] paramTypes = {};
        Class<?>[] copyParamTypes = {};
        String objctClass = "";
        int bracketIni = methodSignature.lastIndexOf("(");
        String methodName = methodSignature.substring(0, bracketIni);
        int bracketEnd = methodSignature.lastIndexOf(")");
        String paramType = methodSignature.substring(bracketIni, bracketEnd);
        paramTypes = paramType.split(",");
        if(paramTypes != null){
        for(int i=0; i<paramTypes.length; i++){
                objctClass = paramTypes[i];
                copyParamTypes[i] = getClass(objctClass);
        }
  }
  Introspector introspector = new Introspector(new String[] { this.projectPath });
  java.lang.reflect.Method method = introspector.getDeclaredMethod(clazz, methodName, copyParamTypes, false);
  long tiempo = this.ejecutar(obj, method, methodArgs);                    return tiempo;
}
```

Para obtener el tiempo de ejecución de un fichero necesitamos obtener el objeto constructor de la clase donde se encuentra el método a ejecutar, el nombre del método y sus argumentos, para tratar con estos parámetros necesitamos utilizar la clase Introspector.

Introspector es una clase que te permite inspeccionar código Java. En este caso concreto, la llamada `getDeclaredMethod` obtiene el método que estamos buscando y lo guarda en la variable `method`. A continuación, ejecuta el método `method` llamando a `ejecutar`. El método `ejecutar` devuelve el tiempo que ha tardado en ejecutarse.

7.2.3 JavaCheckStyle

Antes de empezar a explicar todas las funciones que hemos necesitado para desarrollar la librería *JavaCheckStyle* y poder medir el estilo de un fichero Java, queremos mostrar la apariencia final de la misma.

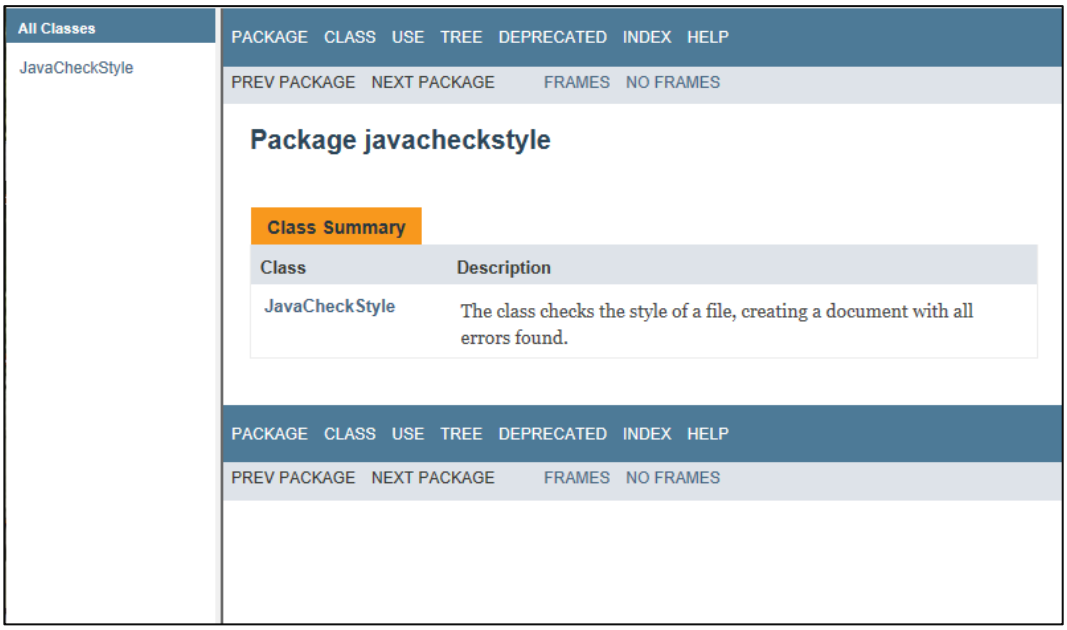
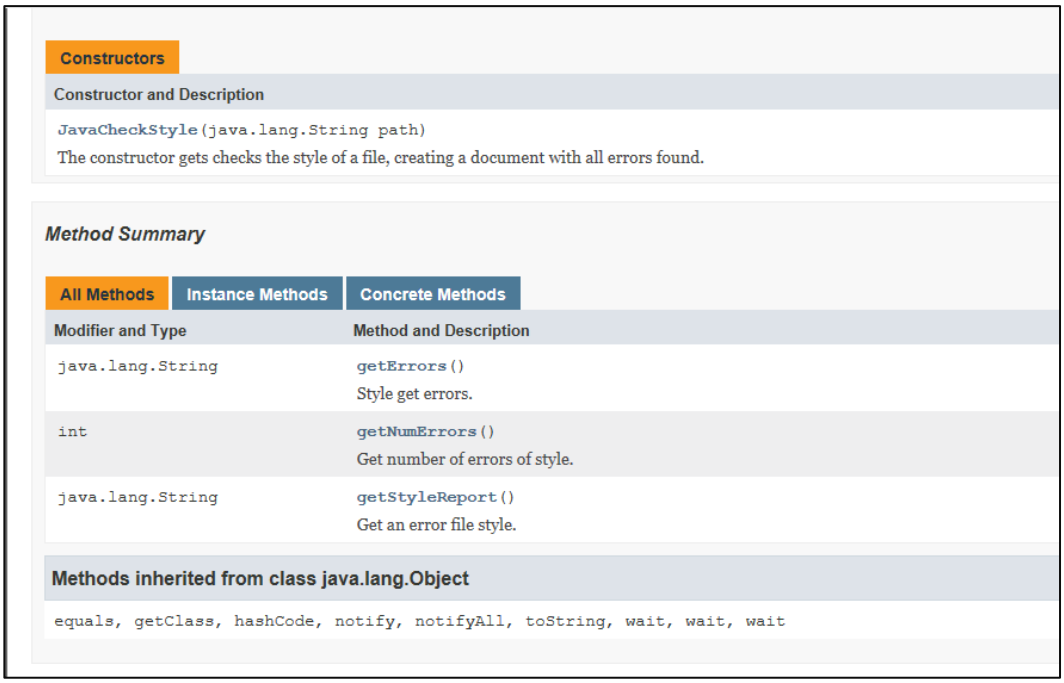


Figura 4. Portada principal librería JavaCheckStyle

Todos los usuarios pueden moverse por cualquier funcionalidad de la librería. Al hacer clic en el menú de la izquierda, podemos acceder a todas las funciones que se encuentran en la librería.

Se puede observar la similitud entre las dos librerías, por ser un API creado automáticamente desde Eclipse en ambos casos.

Seguidamente, vamos a mostrar una captura con todas las funciones creadas para utilizar esta librería y obtener información sobre el estilo de un fichero Java.



Constructors

Constructor and Description
`JavaCheckStyle(java.lang.String path)` The constructor gets checks the style of a file, creating a document with all errors found.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`java.lang.String`	`getErrors()` Style get errors.
`int`	`getNumErrors()` Get number of errors of style.
`java.lang.String`	`getStyleReport()` Get an error file style.

Methods inherited from class java.lang.Object

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Figura 5. Funciones de la librería JavaCheckStyle

En cada una de las funciones de la librería se encuentra una pequeña descripción de lo que ésta realiza. A continuación vamos a detallar cómo se han realizado las diferentes funcionalidades.

Antes de pasar por cada una de las funciones, debemos explicar la razón de por qué hemos utilizado la librería CheckStyle para el desarrollo de nuestra librería *JavaCheckStyle*.

7.2.3.1 Uso de la librería JavaCheckCode

A continuación mostramos un ejemplo de uso de la librería JavaCheckCode para medir la eficiencia de un fichero Java.

El código de ejemplo siguiente, muestra todas las llamadas a todas las funciones de la librería.

```
public class Tester {

	public static void main (String args []) throws Exception
	{
		String filePath = "C:\\TFG\\Prueba\\pruebaTFG\\Mult1.java";
		String className = "pruebaTFG.Mult1";
		Object[] argClass = {};
		String methodSignature = "multiplicacion()";
		Object[] arg = {};

		JavaCheckCode checker = new JavaCheckCode(filePath);

		checker.getConditionCount();
		checker.getLOC();
		checker.getMaxDepth();

		checker.getClassConditionCount(className);
		checker.getClassLOC(className);
		checker.getClassMaxDepth(className);

		checker.getMethodConditionCount(className,
		methodSignature);
		checker.getMethodLOC(className, methodSignature);
		checker.getMethodMaxDepth(className, methodSignature);
		checker.getExecutionTime(className,argClass,
		methodSignature, arg);
	}
}
```

CheckStyle

Para realizar nuestra librería investigamos de qué forma podríamos conseguir información de estilos de una forma fácil.

De entre las soluciones posibles, sin duda, la librería CheckStyle es la más adecuada porque nos facilita todo lo que necesitamos para hacer las mediciones de estilo de un fichero Java.

Se pueden observar todas las funciones que introduce esta librería en una captura donde se aprecian diferentes funcionalidades de la misma.

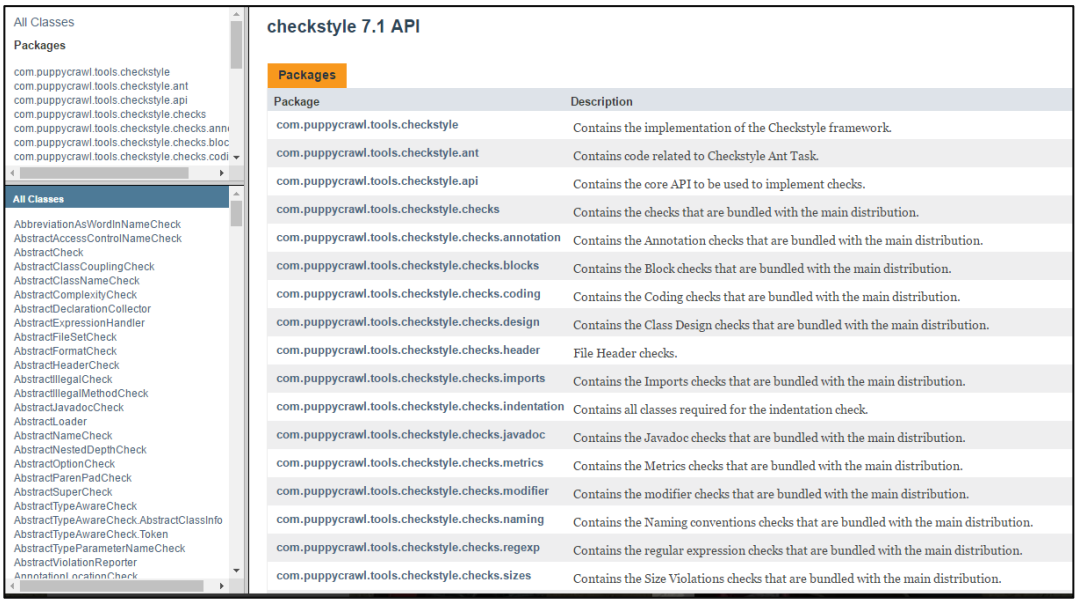


Figura 6. Funciones de la librería CheckStyle

Se puede observar en la figura 6 que hay multitud de opciones para comprobar el estilo de un fichero Java.

Hemos tenido que utilizar el lenguaje de programación XML para poder implementar las funciones escogidas de CheckStyle en nuestra librería.

Hemos realizado una búsqueda de los criterios de estilo en Java y encontramos una guía de estilo de Google para Java.

Esta guía sirve como la definición completa de estándares de codificación de Google para el código fuente en el lenguaje de programación Java. Un archivo fuente de Java tiene que adherirse a las normas de estilo que hemos concretado en nuestra librería.

El documento que hemos tenido en cuenta se centra principalmente en las reglas que sigue Google para el estilo de los archivos Java.

Para la creación de nuestra librería hemos tenido en cuenta una pequeña parte de esta guía, ya que la funcionalidad de nuestra librería está enfocada para utilizarse de manera docente y hay que tener en cuenta los límites de un estudiante a la hora de realizar un formato correcto.

Vamos a explicar el XML creado para nuestra librería. Primero vamos a mostrar el código realizado para cada función.

7.2.3.2 Funciones de formato en XML

Hemos tenido en cuenta las limitaciones que puede tener un estudiante para poder implementar un archivo fuente en Java, por este motivo no se pueden medir varias características de estilo.

Principalmente hemos utilizado el check **LineLength**:

```
<module name="LineLength">
      <property name="max" value="80"/>
      <property name="ignorePattern" value="^ *\* *[^ ]+$"/>
 </module>
```

Con este código restringimos que el algoritmo que queramos comprobar tenga una longitud inferior a 80 columnas en una línea.

A parte de ser una especificación de Java, la justificación de por qué hemos utilizado este check es porque las líneas largas son difíciles e incómodas de leer en las herramientas de desarrollo y porque muchas veces los programadores han limitado espacio de la pantalla para el código fuente.

Con esta declaración no tenemos en cuenta los imports, paquetes y comentarios del código fuente, ya que no podemos saber qué longitud pueden tener los mismos y no nos preocupa para poder entender el código.

La siguiente comprobación de estilo que hemos utilizado es **BooleanExpressionComplexity**. Con este check podemos indicarle el número máximo de comparaciones booleanas que puede haber en una condición en el algoritmo que estemos comprobando.

```
<module name="BooleanExpressionComplexity">
      <property name="max" value="4"/>
 </module>
```

Demasiadas condiciones conducen a un código que es difícil de leer y por lo tanto depurar y mantener. Por defecto, si no se indica ningún valor se permiten tres expresiones booleanas como máximo.

Hay que ser exigentes cuando un archivo Java tiene muchas líneas de código y varios programadores tienen que utilizarlo, pero en el caso de los estudiantes hay que tener un margen a la hora de exigirles unas especificaciones.

El próximo check que utilizamos en nuestra librería es **Indentation**, que sirve para comprobar el sangrado entre líneas del código fuente.

```
<module name="Indentation">
      <property name="basicOffset" value="8"/>
      <property name="lineWrappingIndentation" value="8"/>
 </module>
```

Con el anterior código le obligamos al usuario a que tenga tabulado el código. El valor por defecto que tiene esta función son cuatro espacios, pero en nuestro caso vamos a evaluar por tabulaciones.

Por ese motivo el valor que hemos introducido es ocho, equivalente a los espacios que utiliza un tabulador.

Otra de las funciones de la librería CheckStyle que utilizamos, es el check **EmptyBlock**. Esta llamada comprueba que no se encuentren en el código fuente bloques vacíos de instrucciones.

```
<module name="EmptyBlock">
      <property name="option" value="text"/>
      <property name="tokens" value="LITERAL_WHILE, LITERAL_TRY,
LITERAL_CATCH, LITERAL_DO, LITERAL_IF, LITERAL_ELSE, LITERAL_FOR,
LITERAL_SWITCH"/>
</module>
```

Este código sirve para configurar la verificación de la política de texto y tratar todos los tipos de bloques que suelen existir en los ficheros Java.

Es una forma de obligar al usuario a realizar un código limpio y fácil de entender. No tiene sentido que se queden bloques sin completar, si no se van a utilizar.

El último check utilizado en nuestra librería *JavaCheckStyle* es **UnusedImports**, para verificar que no hay imports que no se utilizen.

Checkstyle utiliza un algoritmo simple, pero muy fiable para informar sobre las declaraciones de importación no utilizadas. Una declaración de importación se considera sin usar si:

- No se hace referencia en el archivo.
- El algoritmo no es compatible con la importación de comodín como import java.io. *
- Es un duplicado de otra importación. Esto es cuando una clase se importa más de una vez.
- La clase importada es del paquete java.lang. Por ejemplo java.lang.String.
- La clase importada es del mismo paquete.

```
<module name="UnusedImports">
```

Hay muchas veces que utilizamos objetos en un determinado momento y al instanciar un objeto debemos importar el paquete que hace referencia a todo el código que implementa ese objeto. Posteriormente, eliminamos la llamada a ese objeto por algún motivo, pero no nos acordamos de eliminar el import requerido para su uso.

Por ese motivo, queremos tener en cuenta que el usuario cuide todos los cambios realizados en el programa, para no tener que cargar cosas innecesarias para la ejecución del mismo.

7.2.3.3 Errores de estilo

Antes de empezar a explicar todas las funciones creadas en la librería *JavaChekStyle*. Debemos explicar el siguiente código:

```
public static String styles(String pathFile)
{
 String result ="";
 String path ="";
 try
 {
        path = "./tmp/Result.txt";
        com.puppycrawl.tools.checkstyle.Main.main("-o", path, "-c", "/Style.xml", pathFile);
        result = Misc.read(path);
 }
 catch (IOException e)
 {
        e.printStackTrace();
 }
 return result;
}
```

La librería CheckStyle es una herramienta que valida el estilo del código. Utilizamos CheckStyle para:

- Aportar higiene al código
- Estandarizar el desarrollo
- Asegurar una fácil lectura

Para poder utilizar la librería CheckStyle necesitamos conocer la ruta del fichero a comprobar. CheckStyle nos da la opción de generar un informe de errores de estilo y vemos que puede ser interesante como utilidad en nuestra librería.

Para ello se necesita "-o" para realizar un fichero de salida y "-c" especifica la ubicación del archivo que define los módulos de configuración.

Para poder obtener los errores de estilo, debemos crear el constructor de la librería con todos los datos necesarios para crear un informe y poder leer a partir de él.

```
public JavaCheckStyle(String filePath) throws Exception{
   this.pathfile = filePath;
   result = "./tmp/Result.txt";
   this.reader = new BufferedReader(new FileReader(this.result));
}
```

Con la creación del código anterior ya podemos trabajar con el informe creado para obtener todos los datos que queremos que ofrezca la librería *JavaCheckStyle*.

```
public String getErrors() throws Exception
```

```
{
   while((this.line = this.reader.readLine()) != null)
         this.text += line + "\n";
   return text;
}
```

Necesitamos leer línea a línea los errores para poder mostrarlos por pantalla y generar una imagen limpia y fácil de entender para el usuario.

7.2.3.4 Número de errores de estilo

Para obtener el número de errores también necesitamos leer línea a línea para poder leer el número total de errores que se han generado en el informe.

```
public String getErrors() throws Exception
{
   while((this.line = this.reader.readLine()) != null){
   if(line.startsWith("["))
         cont++;
   }
   return text;
 }
```

7.2.3.5 Informe de errores de estilo

Para poder obtener el informe de errores de estilo, es necesario hacer la llamada a la primera función explicada styles, ya que es la clase externa que nos ayuda a generar el informe de errores a partir de las especificaciones realizadas en el fichero XML de la librería *JavaCheckStyle*.

Lo único que necesitamos es instanciar la clase Style con el método styles.

```
public String getStyleReport(){
   return Style.styles(this.pathfile);
}
```

7.2.3.6 Uso de librería JavaCheckStyle

Queremos añadir un ejemplo de uso de la librería JavaCheckStyle para medir la eficiencia de un fichero Java.

El código de ejemplo siguiente muestra todas las llamadas a todas las funciones de la librería.

```
public class Tester {
    public static void main (String args []) throws Exception
    {
         String filePath = "C: \\TFG\\Prueba\\pruebaTFG\\Mult1.java";
         JavaCheckStyle checkStyle = new JavaCheckStyle(filePath);
         System.out.println(checkStyle.getErrors());
        checkStyle.getNumErrors();
        checkStyle.getStyleReport();
    }
}
```

8 Conclusiones

La existencia de librerías en Java se ha convertido, para los programadores de este lenguaje, en una necesidad básica que requiere de métodos efectivos y fiables.

Existe una gran variedad de librerías en la comunidad Java, pero siempre se necesitan nuevas funcionalidades para mejorar la calidad del software y poder solventar distintos problemas que aún no se habían desarrollado, como puede ser la eficiencia de un algoritmo y el formato de estilo del mismo.

El desarrollo de nuevas librerías y sobre todo el desarrollo de las nuevas tecnologías han propiciado nuevas métricas, más modernas y con unas funcionalidades impensables hace sólo unos años.

La combinación de todas estas librerías libres proporcionan innumerables ventajas, como pueden ser la disponibilidad de funciones ya creadas y la de obtener información de una forma fácil y segura.

Estas nuevas librerías, por sí mismas, o en combinación con otras librerías Java, permiten realizar tareas más eficientes. Por ejemplo, se pueden incluir en cualquier aplicación y ejecutarse a partir de un botón.

La combinación de las librerías JavaCheckCode y JavaCheckStyle con algunos de los métodos de otras librerías, proporcionan un nivel más alto de eficiencia a la hora de desarrollar cualquier aplicación.

Para la realización del presente trabajo ha sido necesario, en primer lugar, aprender los aspectos referentes a las librería JavaParser y CheckStyle, a partir de las cuales se ha desarrollado el proyecto.

Ha sido necesario consultar y estudiar los manuales y APIs para poder utilizarlas de forma correcta, las cuales han sido de gran ayuda a la hora de trabajar, ya que una vez aprendido su funcionamiento resulta realmente rápido y fácil de implementar las funciones.

Los principales resultados obtenidos en la elaboración de este proyecto son:

- Dos librerías Java independientes provistas de un API con JavaDoc

- Un sitio web diseñado específicamente para la distribución de cada librería.

La construcción de estas librerías ha sido un reto dado el uso de la abstracción de Java y la introspección avanzada que se necesita para analizar y extraer la información que necesitamos de las clases de Java. Gracias a estas librerías, cualquier proyecto puede analizar la eficiencia del código Java y su estilo de manera automatizada.

Las librerías JavaCheckCode y JavaCheckStyle van a ser usadas en la asignatura de Lenguajes, Tecnologías y Paradigmas de la Programación (LTP) de la Universitat Politècnica de València para evaluar los ejercicios y proyectos entregados por los alumnos. Esta evaluación es automática y permite al profesor conocer aspectos no funcionales del código (eficiencia y estilo) automáticamente.

9 Ampliaciones futuras

Cabe mencionar las posibles mejoras o ampliaciones que podrían realizarse a las librerías desarrolladas, bien para dotarlas de nuevas funcionalidades, bien para mejorar algunas funcionalidades ya existentes.

Una mejora interesante sería conseguir que la librería *JavaCheckStyle* funcionase con más verificaciones, es decir, ampliar más tipos de estilos para comprobar nuevos formatos. Esto posibilitaría que la librería midiera más especificaciones de estilo de Java.

Una mejora importante sería poder elegir qué especificación de estilos seguir (Google Style, Sun Style, etc.). Igualmente, se podría aceptar un estilo como correcto si sigue alguna especificación. Para ello, habría que implementar varias y que el comprobador viera si el código del alumno se ciñe a alguna de ellas. Esto implicaría una validación híbrida de estilos.

Utilizamos en nuestra librería JavaCheckStyle la especificación realizada de Google Style, una de la ampliaciones futuras sería realizar lo mismo pero usando la guía de Sun Style para ceñirnos a otro tipo de formato de estilo.

Para el objetivo final de esta librería se han creado todas las funciones necesarias para medir la eficiencia de un algoritmo utilizando varias métricas. Como extensión futura, se pueden considerar otras métricas como puede ser la complejidad ciclomática, el número de parámetros de los métodos, número de atributos de las clases, etc.

10 Bibliografía

[HTML, 2003]: World Wide Web Consortium W3C (MIT, ERCIM, Keio); Página Oficial lenguaje HTML; www.w3.org/MarkUp/; 2003

[IEEE, 84] IEEE Std 830 - IEEE Guide to Software Requeriments Specifications. IEEE Standards Board. 345 Eas 47 th Street. New York, NY 10017, USA.1984.

[Nielsen J., 2000]: Jakob Nielsen; Libro *"Usabilidad. Diseño de sitios Web"*; Editorial Prentice may; Pearson Educación S.A; C/ Núñez de Balboa, 120 Madrid; 2000.

[Nielsen J., 2006]: Jakob Nielsen; Libro *"Prioritizing Web Usability"*; Editorial New Riders Pub; 1249 8th Street Berkeley, CA 94710; 2006.

[Wikipedia, 2007]: Wikipedia la enciclopedia libre; Artículo; *"Java"*, *"Biblioteca (informática);* http://es.wikipedia.org/wiki/Portada; 2007.

[Wikipedia, 2016]: Wikipedia la enciclopedia libre; *"Javadoc";* 28 de enero de 2016; https://es.wikipedia.org/Javadoc; 2016.

[Wikipedia, 2016]: Wikipedia la enciclopedia libre; *"Paquete Java";* 28 de marzo de 2016; https://es.wikipedia.org/paqueteJava; 2016.

[Wikipedia, 2016]: Wikipedia la enciclopedia libre; *"Interfaz de programación de aplicaciones"*, *"Biblioteca (informática);* 28 de enero de 2016; https://es.wikipedia.org/Interfaz de programación de aplicaciones; 2016.

[Wikipedia, 2008]: Wikipedia la enciclopedia libre; Artículo; *"Complejidad ciclomática"; 10 de febrero de 2016;* https://es.wikipedia.org/wiki/Complejidad_ciclomática; 2008.

[Wikipedia, 2016]: Wikipedia la enciclopedia libre; Artículo; *"Método"*, *"Biblioteca (informática);* 2016.

[Wikipedia, 2016]: Wikipedia la enciclopedia libre; Artículo; *"Clase"*, *"Biblioteca (informática); 25 de julio de 2016;* https://es.wikipedia.org/Clase(Informática); 2016.

[Wikipedia, 2016]: Wikipedia la enciclopedia libre; Artículo; *"Atributo"*, *"Biblioteca (informática); 25 de julio de 2016; https://es.wikipedia.org/Atributo(Informática);* 2016.

[Desconocido, 2016]: Desconocido "API Java"; 10 de febrero de 2016; https://docs.oracle.com/javase/7/docs/api/ ; 2016.

[Ivanov, 2016]: Roman Ivanoc "CheckStyle"; 15 de abril de 2016; http://www.checkstyle.sourceforge.net; 2016.

[Ivanov, 2016]: Roman Ivanoc "API CheckStyle"; 15 de abril de 2016; http://www.checkstyle.sourceforge.net/apidocs/index.html; 2016.

[Desconocido, 2014]: Desconocido "Google Java Style Guide"; 3 de mayo de 2016; https://google.github.io/styleguide/javaguide.html; 2014.

[Vilmar, 2007]: Julio Vilmar Gesser "API JavaParser"; 6 de febrero de 2016; http://www.javadoc.io/doc/com.github.javaparser/javaparser-core/; 2007.

[Peñaloza, 2011]: Jorge.E Peñaloza,"Cálculo del tiempo de ejecución de un método"; 10 de marzo de 2016; http://jorgep.blogspot.com.es/2011/02/calculo-del-tiempo-de-ejecucion-de-un.html; 2011

ANEXO I – Manual de usuario JavaCheckCode

1 Introducción

El presente apartado tiene como objetivo ser un manual de la librería JavaCheckCode para analizar la eficiencia de un fichero Java. A continuación se explica, de manera sencilla e ilustrada, el manejo de las funciones de la librería para los usuarios que vayan a utilizarla.

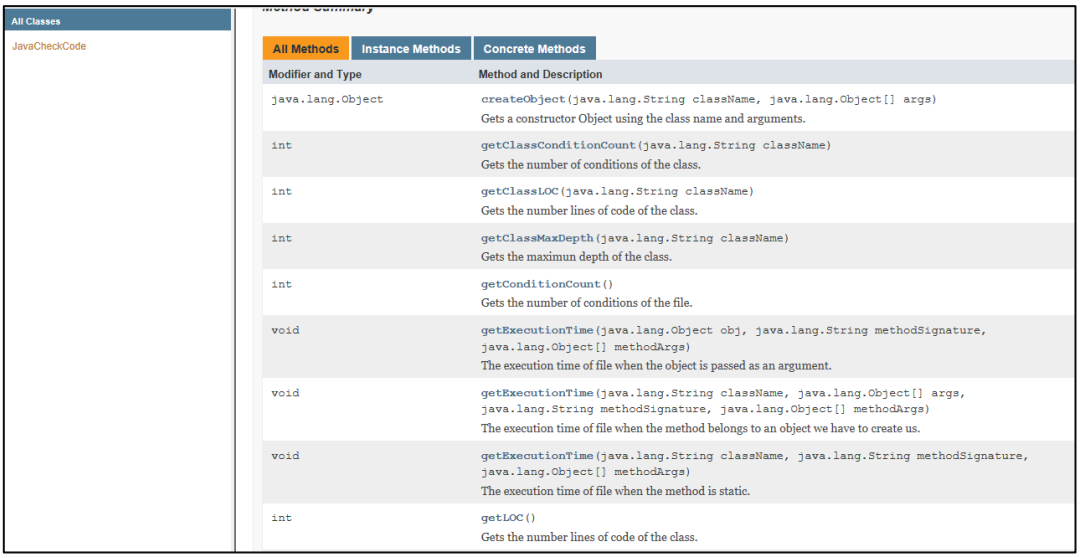
2 Funcionalidades

La librería *JavaCheckCode* ofrece las siguientes funcionalidades:

- Contar el número de condiciones de un fichero Java.
- Contar el número de líneas de un fichero Java.
- Obtener la profundidad máxima de los bucles de un fichero Java.
- Contar el número de condiciones de una clase.
- Contar el número de líneas de una clase.
- Obtener la profundidad máxima de los bucles de una clase.
- Contar el número de condiciones de un método.
- Contar el número de líneas de un método.
- Obtener la profundidad máxima de los bucles de un método.
- Mostrar el tiempo de ejecución de un algoritmo.

3 Entrada en la API JavaCheckCode

La página de la API de JavaCheckCode te muestra todas las funciones que se pueden utilizar y una breve descripción de lo que realizan cuando se ejecuta a cada uno de los métodos.



All Classes

JavaCheckCode

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method and Description
java.lang.Object	createObject(java.lang.String className, java.lang.Object[] args) Gets a constructor Object using the class name and arguments.
int	getClassConditionCount(java.lang.String className) Gets the number of conditions of the class.
int	getClassLOC(java.lang.String className) Gets the number lines of code of the class.
int	getClassMaxDepth(java.lang.String className) Gets the maximun depth of the class.
int	getConditionCount() Gets the number of conditions of the file.
void	getExecutionTime(java.lang.Object obj, java.lang.String methodSignature, java.lang.Object[] methodArgs) The execution time of file when the object is passed as an argument.
void	getExecutionTime(java.lang.String className, java.lang.Object[] args, java.lang.String methodSignature, java.lang.Object[] methodArgs) The execution time of file when the method belongs to an object we have to create us.
void	getExecutionTime(java.lang.String className, java.lang.String methodSignature, java.lang.Object[] methodArgs) The execution time of file when the method is static.
int	getLOC() Gets the number lines of code of the class.

Figura 1. Funciones de la librería JavaCheckCode

Se puede observar todos los parámetros que se necesitan para poder ejecutarlos y ver qué tipo de información se va a mostrar.

Se podrá descargar la librería para utilizar todas las funciones ya creadas, modificar los diferentes métodos o ampliar la funcionalidad de la librería según el interés del desarrollador.

Para poder ejecutar la librería JavaCheckCode necesitamos conocer la ruta donde se encuentra el algoritmo que queremos comprobar y con esa información podemos crear un objeto JavaCheckCode para invocarlas funciones que se encuentran desarrolladas en su interior.

Un ejemplo de uso sería la siguiente:

```
String filePath = "C: \\TFG\\Prueba\\pruebaTFG\\Mult1.java";
JavaCheckCode checker = new JavaCheckCode(filePath);
```

4 Calcular el número de condiciones de un fichero

Una vez creado el objeto checker como lo hemos llamado en el ejemplo anterior, podemos invocar el método realizado en la librería que nos devuelve el número de condiciones del fichero Java que le hayamos pasado como parámetro.

La llamada a la función para obtener el número de condiciones de un fichero es:

```
checker.getConditionCount();
```

Con esta llamada se obtiene un tipo entero que indica el número exacto de condiciones que hay en el código fuente.

5 Calcular el número de líneas de código de un fichero

Para obtener el número de líneas de código de un fichero Java, se tiene que utilizar el objeto checker para realizar la llamada a esta función. También se obtiene un entero con el número total de líneas de código que hay en el fichero.

```
checker.getLOC();
```

6 Calcular la Profundidad máxima de los bucles de un fichero

Para poder obtener la profundidad máxima de los bucles de un fichero, necesitamos llamar a la función getMaxDepth de la misma forma que llamamos a las funciones anteriores.

```
checker.getMaxDepth();
```

7 Calcular el número de condiciones de una clase

En este apartado vamos a trabajar con la información de una clase, y para ello necesitamos conocer el nombre de la clase que queremos comprobar.

Además de necesitar la ruta donde está el fichero, tenemos que declarar un objeto de tipo String con el nombre completo de la clase. Un ejemplo de cómo inicializar esta variable es:

```
String filePath = "C:\\TFG\\Prueba\\pruebaTFG\\Mult1.java";
String className = "pruebaTFG.Mult1";
```

Como se puede ver en el ejemplo mostrado en el código, tenemos que conocer el nombre completo de la clase. En este caso, "pruebaTFG" es el paquete donde se encuentra la clase y "Mult1" es el nombre de la clase que queremos visitar.

Como ya comentamos en puntos anteriores, debemos conocer la ruta completa de la clase. En este caso si quisiéramos comprobar una clase que se encuentre dentro de la clase Mult1, deberíamos insertar el nombre de la siguiente manera:

- pruebaTFG.Mult1$NombreClase

De esta forma, como tenemos el nombre completo de la clase, podemos asegurarnos de que vamos a comprobar la clase deseada, en el caso de que tuviéramos dos clases con el mismo nombre.

La forma correcta para poder ejecutar esta función a partir de los datos explicados anteriormente es:

```
checker.getClassConditionCount(className);
```

8 Calcular el número de líneas de código de una clase

Para obtener el número de líneas de código de una clase necesitamos conocer las mismas variables que hemos comentado en el punto anterior. Conociendo la ruta del fichero donde se encuentra la clase y el nombre completo de la clase que queremos ejecutar, podemos averiguar el número de líneas de código que compone la clase.

La función necesaria para calcular las líneas de código de una clase es:

```
checker.getClassLOC(className);
```

Con la siguiente instrucción obtendremos un entero que indica el número total de líneas de código de la clase, sin tener en cuenta líneas en blanco ni comentarios.

9 Calcular la profundidad máxima de los bucles de una clase

Para obtener la siguiente información, seguimos necesitando conocer la ruta del proyecto donde se encuentra la clase y el nombre completo.

Esta función nos devolverá un número entero que indica la profundidad máxima de todos los bucles que se encuentren en los métodos de la clase. Para poder obtener esta información el único parámetro que se tiene que conocer es el nombre completo de la clase.

```
checker.getClassMaxDepth(className);
```

10 Calcular el número de condiciones de un método

Para poder averiguar todos los datos de un método necesitamos conocer:

- La ruta donde se encuentra el fichero en el que esta implementado el método que queremos comprobar.
- El nombre de la clase donde se encuentra el método.
- La signatura del método.

```
String filePath = "C:\\TFG\\Prueba\\pruebaTFG\\Mult1.java";
String className = "pruebaTFG.Mult1";
String methodSignature = "multiplicacion()";
```

Tenemos que conocer la signatura del método, ya que puede haber en el interior de una clase métodos con el mismo nombre, pero que necesitan parámetros diferentes.

En el código de ejemplo que hemos mostrado se puede apreciar que el método que queremos comprobar no contiene parámetros, pero en el caso de que los tuviera, por ejemplo dos enteros, se debería de inicializar de la siguiente forma:

```
String methodSignature = "multiplicacion(int, int)";
```

De esta forma, la librería JavaCheckCode visitará todos los métodos que contiene la clase y te devolverá los datos del método que coincida con la asignatura que le pasamos como parámetro.

La llamada a la función para obtener el entero con el número total de condiciones que se encuentran en el interior del método es:

```
JavaCheckCode checker = new JavaCheckCode(filePath);
checker.getMethodConditionCount(className, methodSignature);
```

Como se puede comprobar en la llamada de la función, necesitamos pasarle como parámetros el nombre completo de la clase y la signatura del método.

11 Calcular el número de líneas de código de un método

En la siguiente llamada que vamos a explicar, necesitamos conocer la misma información que utilizamos en el punto anterior.

Para ello, llamamos con el objeto JavaCheckCode al nombre de la función con los parámetros correspondientes.

```
checker.getMethodLOC(className, methodSignature);
```

12 Calcular la profundidad máxima de los bucles de un método

Si queremos conocer la profundidad máxima de un método tenemos que utilizar las mismas variables explicadas en los puntos anteriores, ya que debemos conocer la ubicación donde se encuentra el método.

La llamada correspondiente a la función que devuelve el número de profundidad máxima de los bucles, que se encuentran en el método que le pasamos por parámetro es:

```
checker.getMethodMaxDepth(className, methodSignature);
```

13 Calcular el tiempo de ejecución de un algoritmo

Para obtener el tiempo de ejecución de un método dependemos de la estructura de la función, ya que dependiendo de cómo sea le debemos facilitar diferentes parámetros.

- Método estático
- Método que utilice el constructor de la clase
- Objeto constructor de la clase

Para calcular el tiempo de ejecución necesitamos conocer todos los argumentos del constructor de la clase, nombres de clase y nombre y argumentos del método.

13.1 Método estático

En este caso necesitaríamos conocer el nombre de la clase y el nombre y argumentos del método que vamos a utilizar.

```
String className = "pruebaTFG.Mult1";
String methodSignature = "multiplicacion()";
Object[] arg = {};
```

Al tener un método estático solo necesitamos conocer los tres parámetros mostrados en el código y llamar a la función para quecalcule el tiempo de ejecución.

La llamada se realizará de la siguiente forma:

```
checker.getExecutionTime(className, methodSignature, arg);
```

Con este código se observará en pantalla el tiempo que ha tardado en ejecutarse el método que le decimos.

13.2 Método con argumentos del constructor de la clase

Cuando en el interior de un método utilizamos el constructor de la clase que se encuentra, debemos inicializar las diferentes variables necesarias, para ejecutar la función.

```
String className = "pruebaTFG.Mult1";
Object[] argClass = {};
String methodSignature = "multiplicacion()";
Object[] arg = {};
```

En el ejemplo mostrado en el código mostramos los argumentos como vacíos, pero en el caso de que tuvieran valores se introducirían esos valores dentro de las llaves y se realizaría la llamada a la función de la siguiente manera:

```
checker.getExecutionTime(className,argClass,methodSignature, arg);
```

Con esta llamada se tratan internamente los datos pasados para crear un objeto constructor que se necesita para obtener el tiempo de ejecución.

13.3 Objeto constructor pasado por parámetros

Por otro lado, podemos tener un algoritmo que utilice un objeto como parámetro.

En este caso, la llamada a la función tendrá el aspecto siguiente:

```
checker.getExecutionTime(obj, methodSignature, arg);
```

Aunque cada llamada se realice de forma diferente, en todos ellos obtendremos el tiempo que tarda en ejecutarse el algoritmo.

14 Ejemplo completo de código

Queremos mostrar un ejemplo completo de todas las funciones que se pueden realizar en el mismo fichero para que se pueda apreciar de una forma general lo que se puede conseguir con la librería *JavaCheckCode*.

```
public class Tester {

	public static void main (String args []) throws Exception
	{
		String filePath = "C:\\ TFG\\Prueba\\pruebaTFG\\Mult1.java";
		String className = "pruebaTFG.Mult1";
		Object[] argClass = {};
		String methodSignature = "multiplicacion()";
		Object[] arg = {};

		JavaCheckCode checker = new JavaCheckCode(filePath);

		checker.getConditionCount();
		checker.getLOC();
		checker.getMaxDepth();

		checker.getClassConditionCount(className);
		checker.getClassLOC(className);
		checker.getClassMaxDepth(className);

		checker.getMethodConditionCount(className,
		methodSignature);
		checker.getMethodLOC(className, methodSignature);
		checker.getMethodMaxDepth(className, methodSignature);
		checker.getExecutionTime(className,argClass,
		methodSignature, arg);
	}

}
```

Ejecutando el código anterior podremos conseguir todos los datos que nos facilita la librería *JavaCheckCode* de una forma fácil y rápida.

ANEXO II – Manual de usuario JavaCheckStyle

1 Introducción

El presente apartado tiene como objetivo ayudar al usuario a utilizar la librería JavaCheckStyle. A continuación se explica, de manera sencilla e ilustrada, el manejo de las funciones de la librería para los usuarios que vayan a utilizarla.

2 Funcionalidades

La librería JavaCheckStyle ofrece las siguientes funcionalidades:

- Mostrar en pantalla los errores de estilo encontrados en el fichero
- Número de errores de estilo
- Obtener un informe de errores de estilo en formato txt

3 Entrada a la API JavaCheckStyle

La página de la API de *JavaCheckStyle* muestra todas las funciones que se pueden utilizar y una breve descripción de lo que realizan cuando se ejecuta cada uno de los métodos.

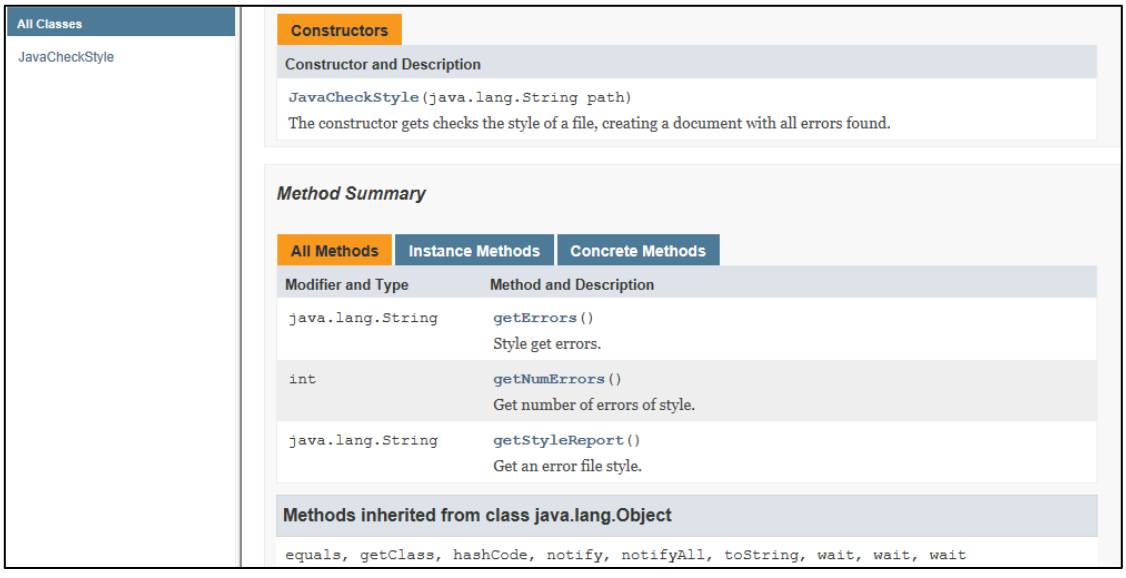


Figura 1. Funciones de la librería JavaCheckStyle

Se pueden observar todos los parámetros que se necesitan para poder ejecutarlos y ver qué tipo de información se va a mostrar.

Se podrá descargar la librería para utilizar todas las funciones ya creadas, modificar los diferentes métodos o ampliar la funcionalidad de la librería según el interés del desarrollador.

Para poder ejecutar la librería JavaCheckStyle necesitamos conocer la ruta donde se encuentra el algoritmo que queremos comprobar. Con esa información podemos crear un objeto JavaCheckStyle para invocar las funciones que se encuentran desarrolladas en su interior.

Un ejemplo de uso sería de esta forma:

```
String filePath = "C:\\TFG\\Prueba\\pruebaTFG\\Mult1.java";
JavaCheckStyle checkStyle = new JavaCheckStyle(filePath);
```

4 Errores de estilo

Con la primera función que vamos a explicar se podrá observar en pantalla los diferentes errores de estilo que hayan en el fichero.

Para poder utilizar esta función hace falta crear el objeto *JavaCheckStyle* y después llamar al método getErrors de la siguiente forma.

```
System.out.println(checkStyle.getErrors());
```

5 Número de errores de estilo

Si queremos obtener un entero con el número de errores de estilo que hay en total en todo el fichero Java, tampoco necesitamos ningún parámetro.

Para poder obtener el número de errores de estilo se necesita realizar la siguiente llamada:

```
checkStyle.getNumErrors();
```

6 Informe de errores de estilo

La última función que vamos a explicar genera un informe en formato txt en la ruta que se indica en el constructor de la librería *JavaCheckStyle*.

```
public JavaCheckStyle(String filePath) throws Exception{
   this.pathfile = filePath;
   result = "./tmp/Result.txt";
   this.reader = new BufferedReader(new FileReader(this.result));
}
```

Se crea una carpeta llamada temp en la misma ruta donde se encuentre el fichero Java y automáticamente se generará el fichero en formato texto con los errores encontrados en el fichero.

La llamada a la función se haría de la siguiente manera:

```
   checkStyle.getStyleReport();
```

7 Ejemplo completo de código

Para finalizar el siguiente manual, mostramos un ejemplo completo de todas las funciones que se pueden realizar en el mismo fichero para que se pueda apreciar de una forma general lo que se puede conseguir con la librería *JavaCheckStyle*.

```
public class Tester
{
   public static void main (String args []) throws Exception{
         String filePath = "C:\\TFG\\Prueba\\pruebaTFG\\Mult1.java";
         JavaCheckStyle checkStyle = new JavaCheckStyle(filePath);
         System.out.println(checkStyle.getErrors());
   checkStyle.getNumErrors();
   checkStyle.getStyleReport();
   }
}
```

Ejecutando el código anterior podremos conseguir todos los datos que nos facilita la librería *JavaCheckStyle* de una forma fácil y rápida.