Universidad
Rey Juan Carlos

Escuela Técnica Superior de Ingeniería Informática

JRater, una aplicación de consola para la corrección automática de prácticas de programación

Memoria del Trabajo Fin de Grado en Ingeniería de Computadores

Autor:

Alejandro Quesada Mendo

Tutores: Mayte González de Lena Alonso, José F. Vélez Serrano

Septiembre 2020

Agradecimientos

Quiero agradecerle a los tutores el apoyo y la idea sobre la que se ha construido este proyecto. Y también, a aquellos docentes que han mantenido vivas mis ganas de seguir aprendiendo.

Sobre todo quiero agradecerle este proyecto a mis padres, que siempre me han apoyado y ayudado a cumplir mis objetivos.

Resumen

Las corrección de prácticas y exámenes de programación suele resultar un proceso tedioso para los docentes. Este proceso suele requerir del uso de diversas herramientas de manera simultánea y corregir a cada práctica individualmente. Es un proceso potencialmente automatizable.

Para intentar resolver el problema se ha decidido desarrollar un proyecto doble, por un lado, una aplicación de consola capaz de procesar, de manera automática, proyectos de estudiantes a partir de proyectos de referencia que incluyen tests unitarios desarrollados por los docentes. Por otro lado, una aplicación web capaz de gestionar los proyectos de referencia y de actuar a modo de interfaz gráfica de la aplicación de consola. Ambos proyectos trabajan conjuntamente, pero en esta memoria solo se comenta el desarrollo de la primera parte.

La aplicación de consola resultante es capaz de procesar, a partir de un proyecto de referencia, proyectos de estudiantes individualmente o en conjuntos de manera simultánea. Como resultado, ofrece informes de compilación y test en un formato amigable para la web. También es capaz de realizar análisis completos antiplagio de conjuntos de proyectos.

Índice general

Índice de figuras

Nomenclatura

EDA	Estructuras de Datos Avanzadas
JRE	Java Runtime Environment, o Entorno de Ejecución de Java
JSON	JavaScript Object Notation
POM	Project Object Model, fichero de descripción de dependencias de un proyecto Maven
UML	Unified Modeling Language, o Lenguaje Unificado de Modelado
XML	eXtensible Markup Language, o Lenguaje de Marcado Extensible

Capítulo 1

Introducción

Es una práctica común en algunas asignaturas de ingeniería informática, que el método de evaluación consista en la entrega de prácticas y exámenes de programación [1]. La corrección de dichas prácticas y exámenes puede resultar una tarea tediosa para el profesorado, ya que el proceso de corrección requiere varias aplicaciones abiertas simultáneamente y una serie de pasos manuales a repetir para cada uno de los estudiantes.

Este trabajo se centra en el caso particular de la asignatura de Estructuras de Datos Avanzadas (o EDA), impartida en el tercer curso de Ingeniería Informática en el campus de Vicálvaro de la Universidad Rey Juan Carlos [2]. El objetivo general de esta asignatura consiste en introducir las propiedades y el funcionamiento de algunas de las estructuras de datos avanzadas más importantes, así como su aplicación para resolver eficientemente determinados problemas.

Esta asignatura tiene un enfoque esencialmente práctico. La evaluación consiste en la entrega de una serie de prácticas evaluables y un examen final. En estas pruebas se exige a los estudiantes que programen alguna funcionalidad de una estructura de datos o que usen las estructuras de datos estudiadas para resolver un problema de forma eficiente.

La asignatura se imparte usando Java [3] como lenguaje de programación y Netbeans [4] como entorno de desarrollo para el lenguaje Java. Ambas tecnologías permiten la implementación y ejecución de tests unitarios de forma sencilla tanto para estudiantes como docentes. Por otra parte, la entrega de prácticas y exámenes se realiza íntegramente a través del Aula Virtual [5] de la universidad.

Un test o prueba unitaria es una función cuyo objetivo es comprobar el correcto funcionamiento de una unidad de código. Por ejemplo, el Algoritmo 1 comprueba si el método de encolado de una cola `enqueue(1)` es correcto.

El proyecto sobre el que trata esta memoria es la primera mitad de un proyecto doble, que engloba dos trabajos de fin de grado. Por un lado, se ha desarrollado la lógica del proyecto, explicada en esta memoria. Y por otro lado, una aplicación web que utiliza dicha lógica como base para funcionar. En este documento no se trata el tema del desarrollo de la aplicación web, pero sí es necesario saber de su existencia para entender el porqué de algunas decisiones de diseño, implementación y despliegue.

Durante este capítulo se va a introducir el problema a resolver, así como cuál ha sido la motivación para la realización del proyecto, una enumeración de los objetivos del mismo y una vista sobre cuáles eran las soluciones ya existentes al problema planteado. Por último, se realiza una introducción a la estructura de este documento.

Algoritmo 1 Ejemplo de test unitario

```
@Test
void testEnqueue() {
    int size = queue.size();
    for (int i=1;i<=10;i++) {
        queue.enqueue(i);
        size++;
        assertEquals(size, queue.size());
    }
}
```

1.1. Motivación

El trabajo nace motivado por un intento de mejorar el sistema de corrección de la asignatura de EDA.

El uso de pruebas automáticas como apoyo a la evaluación de este tipo de asignaturas de programación supone una gran ayuda tanto para los estudiantes como para los docentes. Los estudiantes son capaces de comprobar si el código que están generando cumple con las especificaciones requeridas por el enunciado. Los docentes, por otra parte, son capaces de evaluar rápidamente la corrección del código de los estudiantes.

No obstante, aunque gracias a las pruebas automáticas el proceso de corrección puede haberse agilizado, sigue siendo poco eficiente: para evaluar a cada estudiante, el docente debe tener abierta el Aula Virtual, desde donde descargar los proyectos de los estudiantes y Netbeans para abrir cada uno de los proyectos de uno en uno. Para cada proyecto, debe compilar el proyecto y ejecutar todas las pruebas automáticas, revisando especialmente aquellas que fallen, cerrar el proyecto y puntuar al estudiante en el Aula Virtual.

Desde el punto de vista del estudiante, las pruebas automáticas también pueden llegar a ser de gran utilidad, sin embargo, no son capaces de comprobar, por ejemplo, si el estudiante está usando librerías no permitidas para la realización de las prácticas, lo que supondría que una práctica no fuese válida.

La corrección de este tipo de asignaturas implica que los docentes van a tener que ejecutar el código de los estudiantes para comprobar su correcta implementación, confiando en la ausencia de *malware* (software malicioso) en dicho código. Una manera de evitar que esto comprometiese el estado de la máquina del docente sería usar entornos seguros en los que testear los códigos de los estudiantes.

Sería interesante la existencia de una aplicación que intentara automatizar al máximo el proceso de corrección. De forma que los docentes de asignaturas de programación solo tengan que subir los proyectos desarrollados por los estudiantes a la aplicación y, como resultado, obtengan un informe global sobre el resultado de la ejecución de las pruebas, un informe individual sobre cada uno de los estudiantes y un informe anticopia para evitar fraude. De este modo, se ahorraría mucho tiempo de ejecución manual de las pruebas y se conseguiría una visión general de los resultados la práctica o examen. Además, se evitarían riesgos que comprometiesen la seguridad de la máquina en la que se ejecuta la aplicación.

Por otro lado, también sería interesante que los estudiantes, usando la aplicación, obtuviesen *feedback* (retroalimentación) real sobre la corrección de sus prácticas antes de entregarlas de forma oficial a través del Aula Virtual, de manera automática.

1.2. Objetivos

Los objetivos generales de la aplicación a desarrollar durante el proyecto son:

- Crear una aplicación que agilice el proceso de corrección de prácticas y exámenes en asignaturas de programación, permitiendo la revisión de los resultados compilación y de la ejecución de los test unitarios mediante la creación de informes sobre el resultado de la corrección.
- Ofrecer *feedback* a los estudiantes sobre el grado de validez de la práctica que está realizando antes de entregarla de forma oficial a través del Aula Virtual.
- Obtener un informe anticopia de las prácticas y exámenes entregados por parte de los estudiantes.
- Obtener una aplicación multiplataforma que pueda ejecutarse independientemente del sistema operativo subyacente.
- Obtener una aplicación robusta y fácil de mantener de cara a su futuro mantenimiento.

1.3. Estado del arte

Antes de empezar con el desarrollo de la aplicación que cumpla con los objetivos que se acaban de comentar, se ha hecho una revisión de las soluciones disponibles ya existentes:

- DOMjudge [6], es un software automático preparado para la realización de concursos de programación. Está preparado para procesar las respuestas de forma automática e implementa interfaces tanto para el jurado como para los equipos. Esta opción se descartó porque la configuración resultaba compleja, ya se había intentado con anterioridad y no permite la implementación del sistema anticopia.
- Acepta el reto [7], es otro sistema preparado para la realización de concursos de programación, al igual que DOMjudge. El problema principal de esta plataforma es que no permite subir problemas, en este caso prácticas y exámenes, por lo que el profesorado no podría usarla para corregir sus propios enunciados.
- Mooshak [8] es una herramienta web capaz de evaluar de forma automática la corrección de programas. La razón por la que se descartó Mooshak es que no funciona usando pruebas automáticas, sino que compara "manualmente" la salida estándar del programa con la salida esperada, normalmente, cadenas de caracteres o números enteros.

1.4. Estructura de la memoria

A continuación, se describe brevemente la estructura del resto del documento:

En el capítulo 2, Análisis del problema, se realiza una descripción completa del sistema que se quiere construir a partir de diagramas de casos de uso y de actividad. También se realiza una descripción detallada de los requisitos funcionales y no funcionales del proyecto. Por último, se explica la metodología y cómo se ha adaptado al caso particular de este proyecto.

En el capítulo 3, Diseño e implementación, se explica cómo se ha desarrollado la aplicación que cumpla con los objetivos y requisitos previamente explicados. Se realiza un análisis sobre las herramientas usadas durante el desarrollo y el porqué de su uso. Se explican, de forma detallada, cada una de las partes software que conforman la solución desarrollada. Finalmente, se expone en qué consiste el proceso de despliegue de la aplicación y en qué tecnologías se apoya.

En el capítulo 4, Métricas, se explican las medidas que se han ido tomando a lo largo del desarrollo del proyecto, relativas a tiempos de desarrollo, tiempos de ejecución y calidad del software.

Para finalizar, en el capítulo 5, Conclusiones, se presentan las conclusiones del trabajo realizado, así como sus posibles mejoras y ampliaciones en el futuro y las lecciones aprendidas durante el desarrollo del proyecto.

Capítulo 2

Análisis del problema

Durante este capítulo se intenta obtener un modelo lo más completo posible del problema que se desea resolver. Para ello, primero se introduce el sistema actual, se modela el problema a resolver con ayuda de diagramas UML (`Unified Modeling Language` o Lenguaje Unificado de Modelado) y también se realiza una recapitulación de aquellos requisitos funcionales y no funcionales que se desea que la solución implemente. Por último, se explica cuál ha sido la metodología seguida durante el desarrollo, cómo se ha adaptado a este caso particular y cuáles han sido los motivos de su elección.

2.1. Descripción del sistema actual

Actualmente el sistema de corrección de la asignatura se basa principalmente en un *script* (secuencia de comandos) *bash* (intérprete de comandos) desarrollado por uno de los docentes de la asignatura. Para entender su funcionamiento se explican los siguientes términos:

- Directorio "Por corregir": es un directorio que contiene todos aquellos proyectos de los estudiantes que todavía no han sido procesados por el *script*, es decir, sin corregir.
- Directorio "Corregidos": es un directorio que contiene todos los proyectos de los estudiantes ya procesados.
- Proyecto de referencia: es un proyecto Java a partir del cual se corrigen los proyectos de los estudiantes. No contiene carpeta "src", pero sí carpeta "test". Es importante porque así el docente se asegura de que se ejecutan los mismos tests para todos los estudiantes. Gracias al proyecto de referencia, el docente también puede detectar si un estudiante estaba haciendo uso de librerías externas no permitidas en el enunciado de una práctica o examen porque generaría un fallo en la compilación.
- Directorio "Alumno temporal": es el directorio en el que se descomprimen de uno en uno los ficheros que contienen los proyectos de los estudiantes.
- Directorio "Proyecto compilado": directorio en el que se copian los ficheros del proyecto de referencia y el código fuente del estudiante. Sobre ese directorio se ejecutan las operaciones de compilación y test.

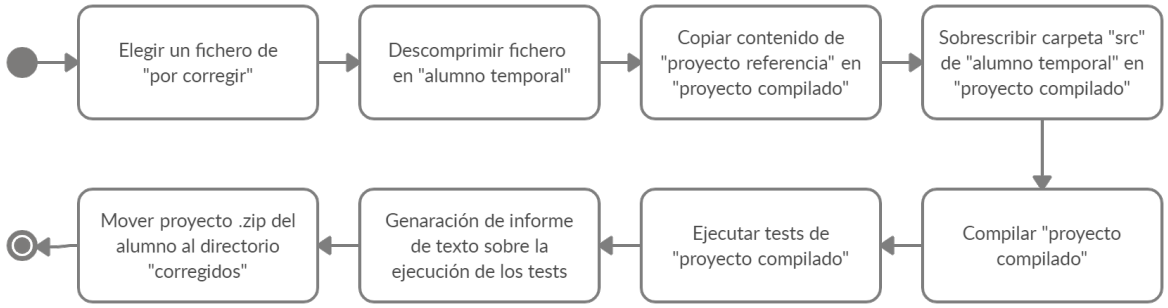

Figura 1: Diagrama de actividad del *script* usado por los docentes

La Figura 1 muestra el diagrama de actividad del *script*.

De esta forma, si se quiere corregir un proyecto usando el sistema actual, se ha de abrir una terminal desde la que ejecutar el *script*, una sesión del Aula Virtual desde la que descargar los proyectos de los estudiantes y desde la que puntuar a los estudiantes tras la ejecución del *script*. También se ha de abrir una sesión de Netbeans (o un editor de texto cualquiera) para poder visualizar el código del estudiante. Además todo el sistema depende de que el sistema operativo desde el que se realiza la corrección sea una distribución de Linux [9].

2.2. Modelo del problema a resolver

En esta sección se intenta dar una idea general del modelo del problema mediante el uso de diagramas de casos de uso y diagramas de actividad para cada uno de los casos de uso especificados.

La Figura 2 muestra el diagrama de casos de uso de la aplicación.

Existen dos posibilidades para la identidad del actor del diagrama de casos de uso. O bien es un usuario de la aplicación, en particular un docente de la asignatura de EDA, o bien es la aplicación web. En cualquier caso, los casos de uso son los mismos.

En cuanto al caso de uso “Autoevaluación de una práctica” contempla el caso de que la aplicación corrija un única práctica durante su ejecución.

El caso de uso “Autoevaluación de un conjunto de prácticas” se refiere a la posibilidad de corregir un conjunto de prácticas de manera simultánea en una única ejecución del programa.

Por último, el caso de uso “Generación de informe anticopia de un conjunto de prácticas”, que depende del caso de uso “Autoevaluación de un conjunto de prácticas”, consistiría en la posibilidad de obtener información acerca de las similitudes, fraude, entre un conjunto de prácticas de estudiantes.

Se quiere que la aplicación opere de manera similar a como lo hacía el sistema descrito el Apartado 2.1, es decir, como un *script*. De esta forma, el usuario de la aplicación solo tiene que ejecutar el programa y la autoevaluación correspondiente concluirá una vez el programa finalice su ejecución, sin necesidad de interactuar con él a través de la consola.

Esto es así ya que, al estar pensado para trabajar en tándem con la aplicación web, es más sencillo que la aplicación no necesite más de un comando para poner a ejecutar las autoevaluaciones que necesite.

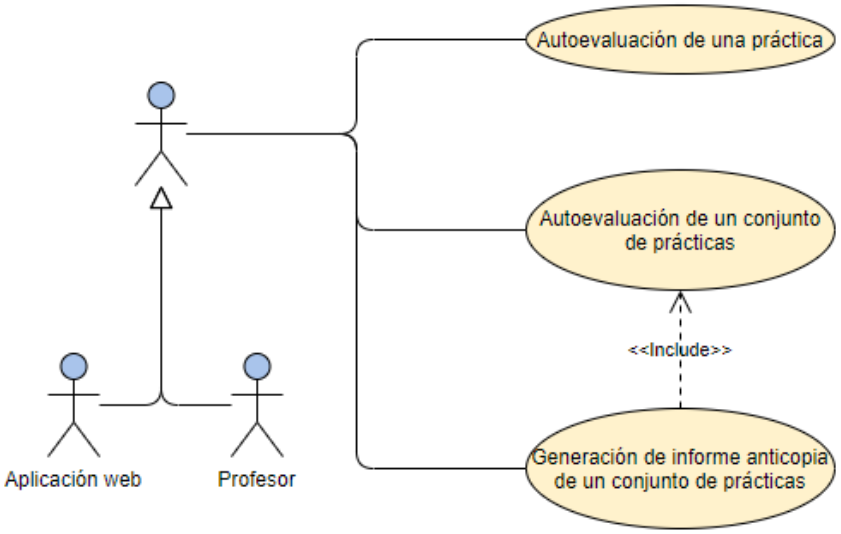

Figura 2: Diagrama de casos de uso

2.3. Requisitos

En esta sección se van a enumerar y detallar cada uno de los requisitos funcionales y no funcionales recogidos durante la fase de elicitación de requisitos.

2.3.1. Requisitos funcionales

Los requisitos funcionales definen una función que se desea que un sistema software incluya. Los requisitos funcionales de esta aplicación son los siguientes:

- Dados un proyecto de referencia y una práctica de estudiante, el sistema debe permitir realizar la autoevaluación de dicha práctica (entendiendo autoevaluación como el proceso de ejecutar los tests del proyecto de referencia sobre el código del proyecto del estudiante).
- Dados un proyecto de referencia y un conjunto de prácticas de estudiantes, el sistema debe permitir realizar la autoevaluación del conjunto de prácticas.
- Dado un conjunto de prácticas, el sistema debe permitir obtener un informe anticopia del conjunto de prácticas, una vez se haya realizado su autoevaluación.
- El sistema no debe bloquearse en caso de que la autoevaluación de una práctica conlleve demasiado tiempo.
- El sistema debe funcionar como un *script*, de manera que cuando finalice la ejecución, hayan terminado de ejecutarse las autoevaluaciones convenientes.
- El sistema debe generar un informe de autoevaluación por cada práctica que autoevalúe.

2.3.2. Requisitos no funcionales

Los requisitos no funcionales son aquellos que no se refieren a la funciones del sistema, sino a sus propiedades y restricciones del sistema en su totalidad. Los requisitos no funcionales del sistema son los siguientes:

- El sistema debe poder ejecutarse con independencia del sistema operativo subyacente.
- El sistema debe poder realizar autoevaluaciones de más de 70 prácticas de manera simultánea.

2.4. Metodología

Para la realización del proyecto se ha seguido una adaptación del modelo en *waterfall* (cascada) descrito por Winston W. Royce en su ensayo de 1970 titulado *Managing the Development of Large Software Systems* [10].

Este modelo consiste en un procedimiento lineal que se caracteriza por dividir el proceso de desarrollo del proyecto en una serie de fases bien diferenciadas. Cada una de estas fases se ejecuta una única vez. Los resultados de cada una de estas fases sirven como punto de partida para la fase siguiente.

Originalmente, Royce propuso un modelo de siete fases, no obstante, hoy en día se suele aplicar una versión del modelo original basado en cinco fases: Análisis, Diseño, Implementación, Verificación y Mantenimiento.

- Análisis. En esta fase se analizan las necesidades de los usuarios finales del producto para determinar las características del software a desarrollar, y se especifica todo lo que debe hacer el sistema sin entrar en detalles técnicos.
- Diseño. En esta etapa se describe las estructura interna del software, y las relaciones entra las distintas entidades que lo componen.
- Implementación. En esta fase se programan los requisitos especificados previamente, así como las pruebas unitarias que comprueben el correcto funcionamiento del software.
- Verificación. Esta fase se centra en probar el funcionamiento la lógica interna del software y las funciones externas del mismo.
- Mantenimiento. Esta fase es la que se encarga de actualizar el proyecto cuando este lo necesite, solucionando fallos, introduciendo nuevas funcionalidades... según las necesidades legales, por petición del cliente, etc.

Esta metodología separa el software a desarrollar de forma que sea posible que un equipo pueda trabajar de forma simultánea sin pisarse. En el caso particular este proyecto, al haber una única persona encargada de realizar el desarrollo de todas las fases, se ha adaptado la metodología para agilizar el proceso.

La fase de análisis es peculiar porque los usuarios finales de la aplicación son también los tutores del proyecto. De modo que, en las tutorías para hablar del proyecto, en lugar de discutir solamente cómo debía comportarse el sistema, también se habló de cómo enfocar el diseño e implementación de la aplicación. Es decir, la fase de análisis y la fase de diseño se mezclaron en una única fase.

El resto de fases del proyecto se caracterizan por contar con un único programador, de forma que la complejidad del desarrollo del proyecto disminuye bastante. En cuanto a la fase de mantenimiento, comenzará una vez se empiece a utilizar el software. En el apartado 5.3 se comentan posibles mejoras a desarrollar en posteriores proyectos.

Por último, mencionar que, para la gestión del proyecto se ha uso la plataforma GitHub [11]. GitHub es un sistema de gestión de proyectos y control de versiones que permite trabajar en colaboración con otras personas desde cualquier máquina con acceso a Internet. Es una herramienta que permite ver a cualquier colaborador actual o futuro cuál ha sido el desarrollo de un proyecto. Durante la fase de mantenimiento se podrá descargar y modificar el software desde GitHub en caso de necesidad.

Capítulo 3

Diseño e implementación

En este capítulo se describe la solución creada que cumple con los objetivos y requisitos que se han planteado previamente el Capítulo 2.3. Para ello, primero se presentan cuáles han sido las herramientas utilizadas para la construcción de la solución y en qué medida han sido importantes. Después se realiza una descripción detallada de cada una de las partes software que componen la solución. Finalmente, se comenta cómo debe ser el proceso de despliegue de la aplicación.

3.1. Herramientas utilizadas

En esta sección se va a hacer un recorrido por todas las herramientas y tecnologías que se han usado, en mayor o menor medida, durante el desarrollo del proyecto y su posterior puesta en producción.

3.1.1. Java e IntelliJ IDEA

Java es un lenguaje de programación y una plataforma informática comercializada por primera vez en 1995 por Sun Microsystems. Actualmente, es uno de los lenguajes de programación más populares [12] ya que permite el desarrollo de todo tipo de aplicaciones. Se eligió Java para este proyecto porque ya estaba familiarizado con el desarrollo de aplicaciones web en este lenguaje.

Por otro lado, a parte de ser un lenguaje de programación, también es un entorno de ejecución para programas desarrollados en este lenguaje. A dicho entorno de ejecución se le conoce como JRE (Java Runtime Enviroment). Este proyecto está desarrollado usando Java 8. No obstante, se necesita la versión 11, o superior, de JRE para su correcta ejecución, ya que JPlag [13], el software antiplagio elegido mencionado en el Apartado 3.1.4, está desarrollado usando Java 11.

IntelliJ IDEA [14] es un entorno de desarrollo integrado para el desarrollo de programas informáticos. Está desarrollado por la compañía JetBrains [15]. Es un entorno multilenguaje y multiplataforma pero especialmente pensado para desarrollar proyectos en lenguaje Java.

3.1.2. Maven

Maven [16] es una herramienta software para la gestión y construcción de proyectos en Java. Es una herramienta de código abierto que se creó con el objetivo de simplificar los procesos de compilación y generación de ejecutables a partir del código fuente, similar a Ant. Maven utiliza un fichero XML llamado Project Object Model (POM) para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos.

Es muy útil sobre todo a la hora de trabajar con librerías externas, basta con añadir la fuente en el fichero POM marcado como `dependency` y a la hora de compilar un proyecto, Maven se encarga de comprobar si dicha librería está descargada y, en caso necesario, descargarla de su repositorio oficial.

3.1.3. Linux

Linux (o GNU/Linux, más correctamente) es un sistema operativo de tipo Unix [17] multiplataforma, multiusuario y multitarea. Linux es el sistema operativo por excelencia en la red en servidores y supercomputadores a nivel mundial. Es importante para este proyecto porque se quiere ejecutar la solución desarrollada en uno de los servidores web de los que dispone la universidad. Además, la *shell* (intérprete de comandos) de Linux pone a disposición de los desarrolladores una serie de instrucciones que pueden facilitar la gestión de ficheros dentro de la aplicación.

3.1.4. JPlag

JPlag es un software antiplagio multilenguaje de código abierto. Permite detectar similitudes entre una serie de ficheros de código. Al ejecutar JPlag, éste genera un informe global sobre el plagio detectado en todos los proyectos analizados y un conjunto de informes individuales para cada uno de los proyectos analizados remarcando las posibles similitudes con el resto de proyectos analizados.

Permite modificar la sensibilidad de detección y elegir qué directorios específicos se quiere analizar. Además, funciona con varias versiones de Java [18] y otros lenguajes (C [19], C++ [20], C#[21] y Python [22]) y texto plano. Es una herramienta fácil de usar y rápida en su ejecución.

Como se ha mencionado previamente, es una aplicación desarrolla usando Java 11, por lo tanto requiere JRE 11 o superior para poder ejecutarse.

3.1.5. JRaterWeb

JRaterWeb [23] es la aplicación web desarrollada para la segunda mitad del proyecto conjunto. Es el actor de todos los casos de uso de esta aplicación. Este proyecto está pensado para trabajar de forma independiente, pero sobre todo como motor de la lógica de JRaterWeb.

3.1.6. Ant

Ant es una herramienta usada en programación para la realización de tareas mecánicas y repetitivas, normalmente durante la fase de compilación y construcción. Es, por tanto, un software para procesos de automatización de compilación, similar a Maven.

Tiene la ventaja de no depender de las órdenes del terminal de cada sistema operativo, sino que se basa en archivos de configuración XML y clases Java para la realización de las distintas tareas.

Es la herramienta utilizada para compilar y ejecutar los tests de los proyectos de los alumnos. A partir de los resultados de dichas tareas, se construyen los informes individuales de autoevaluación.

3.1.7. Docker

Docker [24] es una plataforma de software que permite crear, probar e implementar aplicaciones rápidamente.

Docker empaqueta software en unidades estandarizadas llamadas contenedores. Dentro de ellos podemos alojar todas las dependencias que nuestra aplicación necesite para ser ejecutada: empezando por el propio código, las librerías del sistema, el entorno de ejecución o cualquier tipo de configuración. Desde fuera del contenedor solo se requiere el servicio de Docker ejecutando en la máquina local.

Los contenedores son la solución al problema habitual, por ejemplo, de moverse entre entornos de desarrollo como puede ser una máquina local o en un entorno real de producción. Se puede probar de forma segura una aplicación sin preocuparse de que el código se comporte de forma distinta en función del entorno.

Los contenedores, además, conforman un sistema completamente aislado del sistema operativo que los contiene, de forma que si se ejecutase algún tipo de *malware* dentro del contenedor bastaría con eliminarlo y volver a lanzarlo. Y, dado que para lanzar un contenedor Docker no hace falta poner en marcha un sistema operativo, como en las máquinas virtuales tradicionales, este proceso tarda de unos pocos minutos a incluso segundos.

El servicio de Docker que se ejecuta en el ordenador se encarga de traducir las peticiones que realiza el contenedor al sistema operativo nativo, de forma que no hace falta un hipervisor que actúe a modo de interfaz, y tampoco hace falta levantar un sistema operativo completo dentro de los contenedores. El servicio de Docker también se encarga de reservar más o menos memoria para los contenedores de forma dinámica.

Los contenedores Docker son esenciales para la puesta en producción de la aplicación y evitar que el servidor desde el que se ejecute la aplicación pueda corromperse por un ataque informático.

3.2. Arquitectura del software

A lo largo de esta sección se va a describir cómo ha sido la etapa de diseño e implementación de todas las partes que conforman la solución. Primero se analiza la estructura del

Figura 3: Fases del proceso de autoevaluación

programa desde un punto de vista general, en qué consiste su funcionamiento y cuáles son las distintas partes que lo conforman.

Posteriormente se hace énfasis en cada una de dichas partes, analizando en profundidad cómo funcionan, en qué medida son importantes dentro de la aplicación y cómo se comunican entre sí.

3.2.1. Estructura del programa

El programa desarrollado funciona de manera similar al *script* mencionado en el Apartado 2.1. La idea de su funcionamiento es que los proyectos a evaluar vayan pasando por una serie de fases hasta que, eventualmente, se obtengan los informes de autoevaluación correspondientes.

Es un proceso parecido a una cadena de montaje. Primero, se genera el proyecto temporal. A continuación, se compila el proyecto temporal. Posteriormente, se ejecutan los tests. Por último, se genera el informe de autoevaluación. Opcionalmente, si se estuviese corrigiendo un conjunto de prácticas, se generaría también el informe antiplagio.

Por lo tanto, las distintas fases por las que pasa una práctica durante su autoevaluación son: construcción, compilación, testeo, generación de informe y comprobación de antiplagio, tal y como se muestra en la Figura 3. Este diagrama es muy útil para distinguir cuáles son las distintas partes que van a conformar el programa final y para dar una idea general de cómo será el diagrama de clases de la aplicación.

Cada una de las fases puede implementarse de muchas maneras distintas, en función de la tecnología escogida, el estilo de programación o el tipo de proyecto que se esté evaluando. En el caso particular de este proyecto, como ya se ha comentado, se van a evaluar proyectos Java desarrollados en Netbeans. No obstante, en un futuro se podría querer evaluar proyectos Maven, o incluso proyectos escritos en otros lenguajes como Python.

Teniendo esto en cuenta, el programa se ha diseñado de forma que sea lo más escalable posible. De esta forma, si el día de mañana se amplía el programa a otro tipo de proyectos o lenguajes, no debería ser necesario replantear el proyecto sino, simplemente, añadir las nuevas funcionalidades.

Para lograr este objetivo, para cada una de las fases se ha generado un paquete que contiene, por un lado, una interfaz, y por otro, las distintas implementaciones de dicha interfaz, ahora mismo, únicamente una.

Por ejemplo, en el paquete de la fase Compilación habrá una interfaz llamada `Compiler`, y una clase que implemente la interfaz `Compiler` por cada tipo distinto de compilación, en este caso particular `JavaAntCompiler`. En un futuro se podrían añadir las clases `JavaMavenCompiler` o `PythonCompiler`, por ejemplo.

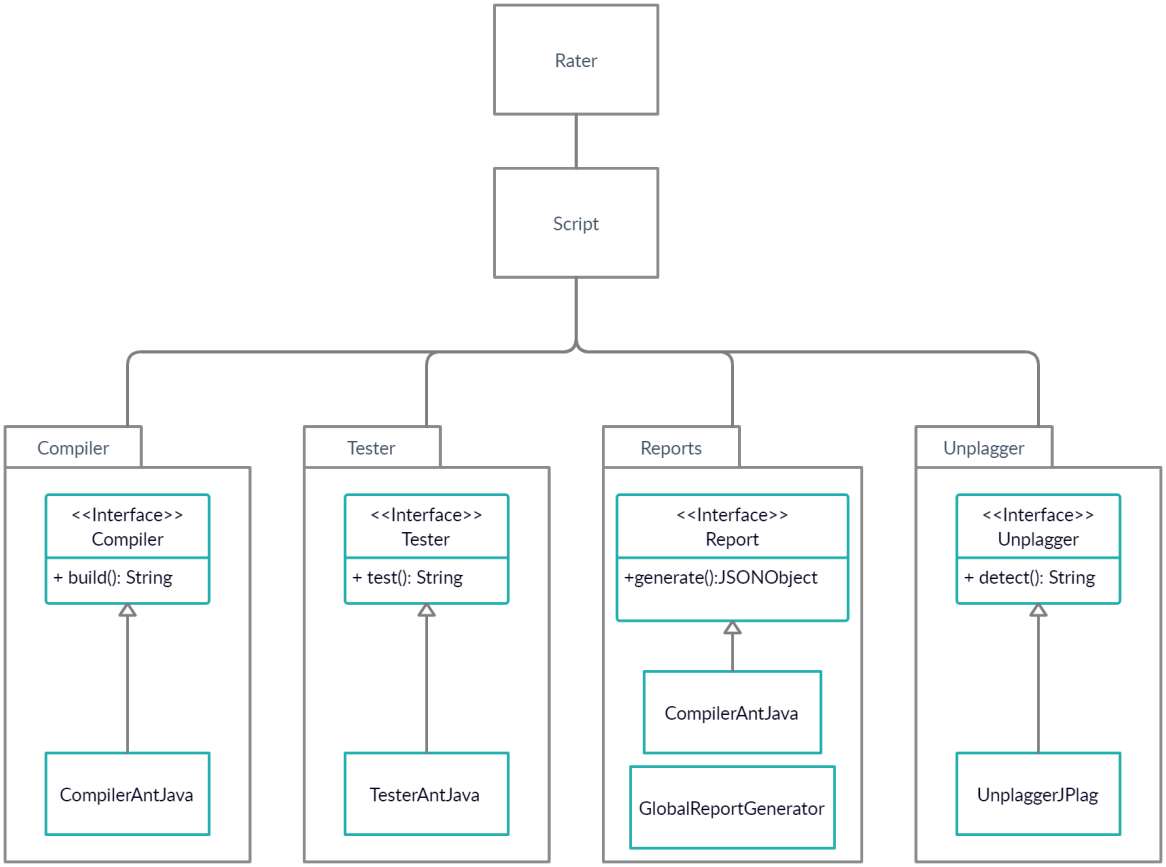

Figura 4: Diagrama de clases de la aplicación

La Figura 4 muestra el diagrama de clases del proyecto, de acuerdo a lo mencionado en los párrafos anteriores.

Nótese que la clase `Script` es la que se encarga de orquestar el funcionamiento del programa, ya que contiene las instancias de las clases que se encargarán de completar las distintas fases de la autoevaluación.

Los paquetes `Compiler`, `Tester`, `Reports` y `Unplagger` se corresponden con las últimas cuatro fases del diagrama de la Figura 3. Las clases contenidas en dichos paquetes funcionan de manera autónoma entre sí y es la clase `Script` la que se encarga de facilitar los datos de entrada en función de la salida de la fase anterior, de manera similar a una *pipeline* (cadena de procesos conectados de forma tal que la salida de cada elemento de la cadena es la entrada del próximo) [25].

Por otra parte, la clase `Rater` es la que contiene el `main` (programa principal) de la aplicación. Es la clase encargada de decidir, en función de los argumentos de entrada, qué tipo de autoevaluación se quiere realizar. Para ello, se ha tenido que decidir una forma de distinguir los distintos casos de uso y qué información necesita el programa para ejecutar cada uno de ellos:

- Para el caso de uso "Autoevaluación de una práctica" se ha elegido el argumento "-p" de práctica o *project*. Se necesita la ruta al proyecto de referencia, la ruta al proyecto del estudiante y el nombre de la práctica (para anotarlo en el informe).

- En cuanto al caso de uso "Generación de un informe anticopia de un conjunto de prácticas" se ha escogido el argumento "-a" de anticopia. Se necesita la ruta al directorio que contiene los proyectos de los estudiantes y la ruta al ejecutable JPlag.
- Por último, para el caso de uso "Autoevaluación de un conjunto de prácticas" se ha escogido el comando "-d" de directorio o *directory*. Se necesita la ruta al proyecto de referencia, la ruta al directorio que contiene los proyectos de los estudiantes, el nombre de la práctica y la ruta al ejecutable Jplag.

3.2.2. Script

La clase `Script` es la que se encarga de gestionar los proyectos a procesar de forma que pasen por las fases correspondientes en el orden correcto. Todos los métodos de la clase `Script` son estáticos, es decir, no hace falta instanciar un objeto para poder ejecutarlos.

Los tres métodos principales se corresponden con los casos de uso de la Figura 2. Estos métodos son: `antiplag`, `rateProject` y `rateDirectory`.

El método `antiplag` recibe como parámetros la ruta al directorio que contiene los proyectos de los estudiantes, y la ruta a JPlag. Comprueba si el directorio de los proyectos está comprimido en formato `zip` o no y, después, crea un objeto de tipo `UnplaggerJplag` que se encarga de comprobar el plagio y de generar un informe.

El método `rateProject` es más complejo que el anterior. La Figura 5 muestra el diagrama de actividad del método. Mientras que la Figura 6 muestra un ejemplo de su traza de ejecución del método.

Para la fase de compilación se crea un objeto de tipo `CompilerAntJava`. Para la fase de ejecución de los tests, un objeto de tipo `TesterAntJava`. Y para la fase de generación del informe, un objeto de tipo `SingleProjectReportGenerator`.

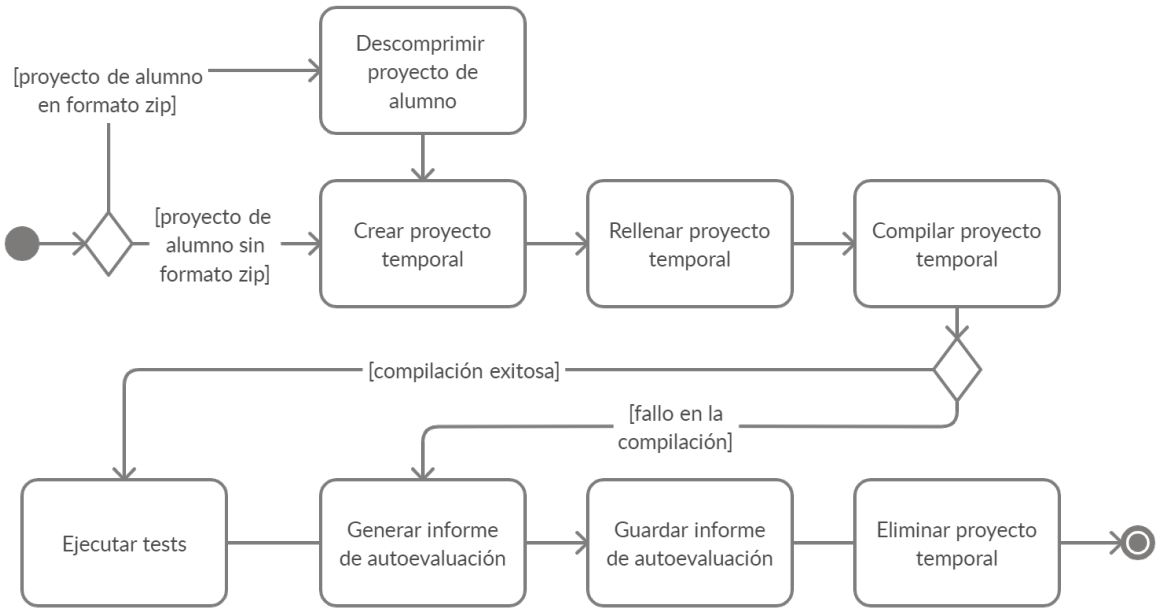

Figura 5: Diagrama de actividad del método `rateProject`

```
Rating project projects/students/practica_1

Rating student project: practica_1 ...
: Building project...
: Executing tests...
: JSON file saved in: _projects/students/practica_1/build_test_report.json
```

Figura 6: Ejemplo de ejecución del método `rateProject`

Nótese, que en función del éxito en la fase de compilación, se ejecuta, o no, la fase de tests. Se ha tomado esta decisión ya que no es posible ejecutar los tests de un proyecto cuya compilación ha fallado, y de este modo se ahorra en tiempo de ejecución.

Por último, el método `rateDirectory` se encarga de procesar un conjunto de proyectos de manera simultánea. La Figura 7 muestra su diagrama de actividad.

Este último método se apoya sobre los dos anteriores para llevar a cabo su tarea. Para realizar las autoevaluaciones de los proyectos individuales utiliza llamadas al método `rateProject` mencionado anteriormente.

Para la implementación de este método se han usado técnicas de programación concurrente. De esta forma, se consigue que las autoevaluaciones individuales se realicen todas a la vez, en lugar de una después de otra de manera secuencial. De este modo, se consigue una ejecución mucho más eficiente.

El uso de este tipo de programación aumenta ligeramente la complejidad del código. Se han de incorporar mecanismos de sincronización a lo largo de la ejecución del programa para evitar que los hilos de ejecución que se ponen en marcha se estorben entre sí, comprometiendo al ejecución del método. Esta situación se conoce como condición de carrera y sucede cuando dos hilos de ejecución distintos intentan acceder a un mismo recurso al mismo tiempo.

En este proyecto sucede, por ejemplo, al intentar guardar los informes de evaluación en una lista. Si dos hilos intentan ejecutar el método `add` de la lista a la vez, es posible que no se ejecute correctamente y la lista no almacene la información como se deseaba.

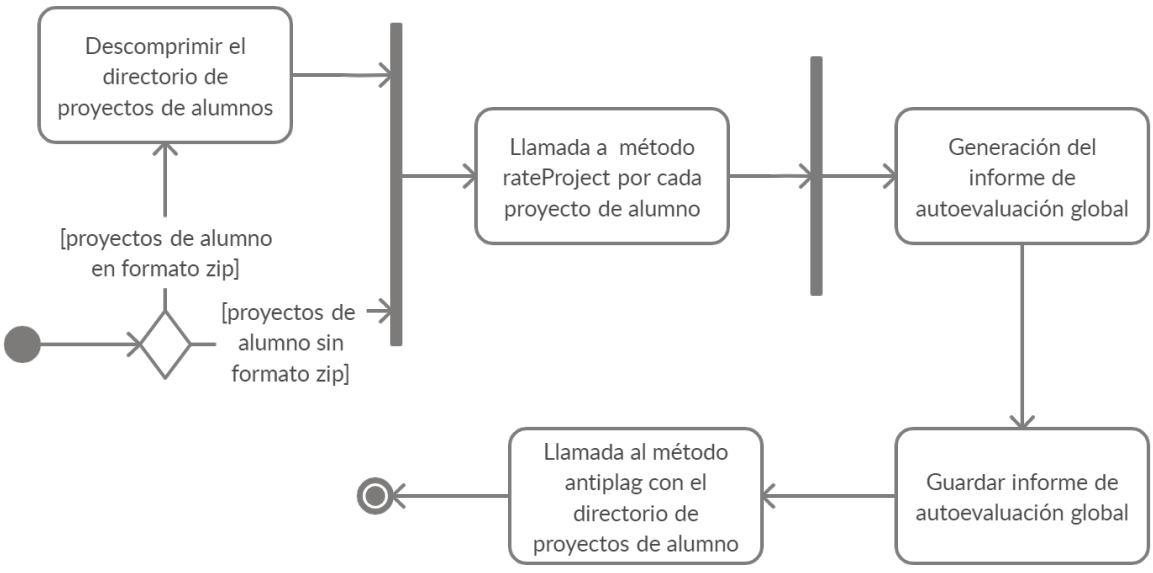

Figura 7: Diagrama de actividad del método `rateDirectory`

```
Rating directory of projects: ./students.zip

Rating student project: ALEX QUESADA MENDO ...
Rating student project: JORGE GONZALEZ LOPEZ ...
Rating student project: MARTA MARTIN MARTINEZ ...
Rating student project: PEPE PEREZ GARCIA ...
JORGE GONZALEZ LOPEZ: Building project...
MARTA MARTIN MARTINEZ: Building project...
PEPE PEREZ GARCIA: Building project...
ALEX QUESADA MENDO: Building project...
ALEX QUESADA MENDO: Executing tests...
MARTA MARTIN MARTINEZ: Executing tests...
JORGE GONZALEZ LOPEZ: Executing tests...
PEPE PEREZ GARCIA: Executing tests...
MARTA MARTIN MARTINEZ: JSON file saved in: ./students/students/MARTA_MARTIN_MARTINEZ_3186451684_assignsubmission/build_test_report.json
1/4 - Rated: MARTA MARTIN MARTINEZ
PEPE PEREZ GARCIA: JSON file saved in: ./students/students/PEPE_PEREZ_GARCIA_25416318615631_assignsubmission/build_test_report.json
2/4 - Rated: PEPE PEREZ GARCIA
JORGE GONZALEZ LOPEZ: JSON file saved in: ./students/students/JORGE_GONZALEZ_LOPEZ_25416318615631_assignsubmission/build_test_report.json
3/4 - Rated: JORGE GONZALEZ LOPEZ
ALEX QUESADA MENDO: JSON file saved in: ./students/students/ALEX_QUESADA_MENDO_3546315_assignsubmission/build_test_report.json
4/4 - Rated: ALEX QUESADA MENDO

Global report JSON saved in projects/references/4/global_report.json

Checking plagiarism at ./students/students
4 submissions parsed successfully!
Writing results to: ./students/Jplag

---- RATING COMPLETE ----
```

Figura 8: Ejemplo de ejecución del método `rateDirectory`

Se han usado dos mecanismos de sincronización:

- Por un lado, se ha usado una objeto de tipo `Object` a modo de cerrojo para proteger el acceso a la lista de informes individuales.
- Por otra parte, para la gestión de los hilos que se encargan de realizar las autoevaluaciones individuales, se ha usado un `ExecutorService`. Esta herramienta se encarga de crear un conjunto finito de hilos, y poner a ejecutar un método determinado en cada hilo, mientras haya hilos disponibles.

  Cuando la ejecución de un método en un hilo termina, el `ExecutorService` se encarga de recoger el resultado de la ejecución, y lanzar una nueva ejecución distinta en ese mismo hilo. Esta forma de gestionar los hilos mejora la eficiencia del programa, ya que no hace falta eliminar el hilo y volver a crear uno nuevo, sino que se reutilizan los creados al principio. La creación y eliminación de hilos, aunque es menos costosa que la creación y eliminación de procesos, también consume tiempo y recursos del procesador.

  Una vez que se terminan de procesar todas las autoevaluaciones individuales se elimina la instancia del `ExecutorService`, que a su vez se encarga de eliminar los hilos creados.

La Figura 8 es un ejemplo de la traza de ejecución del método `rateDirectory()`. Como se puede observar, los distintos proyectos de alumnos van avanzando por las fases del método, pero no siempre en el mismo orden, ya que cada uno se está evaluando en un hilo de ejecución distinto.

3.2.3. Compiler

El paquete `Compiler` es el encargado de realizar la fase de compilación del proyecto. Contiene una interfaz llamada `Compiler`, que cuenta con un único método: `build()`. Por

otra parte, en este paquete también hay una clase, que implementa la interfaz `Compiler`, llamada `CompilerAntJava` .

La clase `CompilerAntJava` tiene un único atributo de tipo `File` llamado `temporalDir`, que almacena una referencia al directorio del proyecto temporal generado en la fase anterior. Por otra parte, implementa un único método, el de la interfaz que implementa la clase.

El método `build()` se encarga de ejecutar la compilación del proyecto temporal con la ayuda de Ant.

Ant da varias opciones en cuanto a objetivos para procesar un proyecto. Un objetivo consiste en un conjunto de tareas fijas que se ejecutan de manera secuencial y que están definidas con un nombre determinado.

En el caso de este proyecto nos interesan especialmente los objetivos `compile` y `test` para las fases de compilación y test, respectivamente. Otros ejemplos de objetivos que ofrece Ant son `dist`, que crea un ejecutable `.jar` con el contenido del proyecto o `clean`, que hace lo mismo pero borrando todos los directorios temporales antes de generar el ejecutable.

Para la compilación del proyecto el comando a ejecutar es el siguiente:

```
ant -f ruta/al/proyecto/temporal/build.xml compile | grep ``BUILD''.
```

El comando `-f` indica que a continuación se especifica cuál es el fichero `build.xml` del proyecto. Dicho fichero contiene información sobre cómo debe realizarse la compilación del proyecto. Después, se especifica la ruta global a dicho fichero. Por último, se ejecuta el comando `grep ``BUILD''`, ya que la única información que se necesita es si la compilación del proyecto fue exitosa o no.

El método `build()` se encarga de ejecutar dicho comando y de recoger el resultado que se imprimiría por la salida estándar. Para ello se usa un objeto de tipo `ProcessBuilder`, que es capaz de ejecutar comandos como si se ejecutasen en una terminal y de recoger el resultado por medio de un objeto de tipo `BufferedReader`.

3.2.4. Tester

El paquete `Tester`, como su propio nombre indica, es el encargado de realizar la fase de ejecución de los tests. Al igual que el paquete `Compiler`, contiene una interfaz con un único método. En este caso, el método `test()`. Por otra parte, también contiene una implementación de dicha interfaz para Ant y Java, `TesterAntJava` .

Esta clase, `TesterAntJava` , tiene una estructura prácticamente idéntica a `CompilerAntJava` . Se apoya en dos objetos de tipo `ProcessBuilder` y `BufferedReader` respectivamente, para ejecutar un comando y procesar la salida estándar.

Lo complicado de la implementación de este método fue el comando a ejecutar. Para la ejecución de los tests de un proyecto, Ant ofrece el objetivo `test`, que se encarga de compilar el proyecto, si previamente se han ejecutado los tests satisfactoriamente. La ejecución del comando `ant compile` imprime por pantalla mucha información, pero solo algunas partes de dicha información interesan de cara a la posterior generación del informe de compilación y tests.

Antes de comentar la información obtenida al ejecutar `ant compile` cabe introducir los siguientes términos:

Algoritmo 2 Ejemplo de la traza de ejecución de un fichero de test cuando la ejecución es exitosa

```
[junit] Testsuite: practica2.CollisionAnalyzerTest
[junit] analyzeQuadraticHash
[junit] Prueba cuadratica = 5640305
[junit] analyzeLinearHash
[junit] Prueba Lineal = 140097624
[junit] analyzeDoubleHash
[junit] Doble Hash = 102426731
[junit] Tests run:3, Failures:0, Errors:0, Skipped:0, Time elapsed:
   16.017 sec
[junit]
[junit] ------------- Standard Output ---------------
[junit] analyzeQuadraticHash
[junit] Prueba cuadratica = 5640305
[junit] analyzeLinearHash
[junit] Prueba Lineal = 140097624
[junit] analyzeDoubleHash
[junit] Doble Hash = 102426731
[junit] ------------- ---------------- ---------------
```

- `TestCase`, es una prueba unitaria. En Java equivale a un método anotado con `@Test` similar al ejemplo del Algoritmo 1. El resultado de la ejecución de un `TestCase` puede ser fallido o exitoso.
- `TestSuite`, es una clase Java que contiene un conjunto de `TestCases`. Normalmente en las `TestSuites` se agrupan aquellos `TestCases` encargados de probar una parte software en particular.

El Algortimo 2 es un ejemplo de una traza de ejecución de un fichero de test, en el que la ejecución de todas las pruebas individuales ha sido exitosa.

Por otra parte, el Algoritmo 3 es un ejemplo de una traza de ejecución de un fichero de test, en la que hay pruebas individuales cuya ejecución ha sido fallida.

La información necesaria de cada `TestSuite` es:

- El propio nombre de la `TestSuite` (o nombre de la clase que contiene los tests).
- El número de `TestCases` ejecutados, fallidos, erróneos y omitidos.
- En caso de fallo, para cada `TestCase`, el motivo del fallo y el lugar del código en el que ha sucedido.

Analizando en profundidad las trazas anteriores y cuál es la información que hay que guardar, se llega a la conclusión de que solo son necesarias aquellas líneas que contengan las siguientes cadenas de caracteres: "`Testsuite`", "`Testcase`", "`Tests run`", "`[junit] java.`", "`[junit] junit.`" y "`at`". Con ayuda del comando `grep` obtenemos tan solo la información deseada.

Algoritmo 3 Ejemplo de la traza de ejecución de un fichero de test cuando la ejecución no es exitosa

```
junit] Testsuite: material.maps.HashTableMapLPTest
[junit] offset
[junit] Tests run: 1, Failures:1, Errors:0, Skipped:0, Time elapsed
   :0.181 sec
[junit]
[junit] ------------- Standard Output ---------------
[junit] offset
[junit] ------------- ---------------- ---------------
[junit] Testcase: testBucketSize(material.maps.HashTableMapLPTest):
        FAILED
[junit] null
[junit] junit.framework.AssertionFailedError
[junit]         at material.maps.HashTableMapLPTest.testBucketSize(
   HashTableMapLPTest.ja va:37)
[junit]         at java.base/jdk.internal.reflect.
   NativeMethodAccessorImpl.invoke0(Nativ e Method)
[junit]         at java.base/jdk.internal.reflect.
   NativeMethodAccessorImpl.invoke(Native MethodAccessorImpl.java:62)
[junit]         at java.base/jdk.internal.reflect.
   DelegatingMethodAccessorImpl.invoke(De legatingMethodAccessorImpl.
   java:43)
[junit]
[junit] Test material.maps.HashTableMapLPTest FAILED}
```

Por lo tanto, el comando a ejecutar en el método `test()` de la clase `TesterAntJava` es el siguiente:

```
timeout 30 ant -f ruta/al/proyecto/temporal/build.xml test | grep -E "TestSuite|
Testcase|Tests run|[junit] java.|[junit] java.|at''.
```

La sentencia `timeout 30` obliga a detener la ejecución de los tests para evitar que el programa caiga en un bucle infinito y se bloquee.

Por último, una vez se ha capturado la información importante y se eliminan las etiquetas "`[junit]`", como resultado se obtienen las trazas de los Algoritmos 4 y 5 cuando un `TestSuite` se ejecuta de manera exitosa y fallida, respectivamente.

Ésta es la información con la que contará el paquete `Reports` para generar los informes de compilación y tests.

Algoritmo 4 Traza de la ejecución exitosa de un test, guardando únicamente la información necesaria

```
Testsuite: practica2.CollisionAnalyzerTest
Tests run:3, Failures:0, Errors:0, Skipped:0, Time elapsed:16.017 sec
```

Algoritmo 5 Traza de la ejecución fallida de un test, guardando únicamente la información necesaria

```
Testsuite: material.maps.HashTableMapLPTest
Tests run: 1, Failures:1, Errors:0, Skipped:0, Time elapsed:0.181 sec
Testcase: testBucketSize(material.maps.HashTableMapLPTest):
   FAILED
junit.framework.AssertionFailedError
        at material.maps.HashTableMapLPTest.testBucketSize(
           HashTableMapLPTest.ja...
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.
           invoke0(Nativ...
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.
           invoke(Native...
        at java.base/jdk.internal.reflect.
           DelegatingMethodAccessorImpl.invoke(De...
```

3.2.5. Reports

El paquete `Reports` se encarga de generar, a partir de los datos obtenidos en las fases de compilación y test, los informes de autoevaluación de las prácticas procesadas. En el caso de que se realice la autoevaluación de un conjunto de prácticas, también se encarga de generar un informe del resultado general de la autoevaluación, a parte de cada uno de los informes individuales.

El resultado que se obtiene después de que este paquete procese las prácticas a evaluar es un informe individual para cada práctica procesada en formato JSON (JavaScript Object Notation) [26]. JSON es un formato de texto sencillo que se usa para el intercambio de datos. Se ha escogido este formato para los informes ya que es fácil de interpretar a simple vista. Además, es un lenguaje usado en Internet para el intercambio de información y JRater está pensado para funcionar junto con una aplicación web.

En el Algoritmo 6 se incluye un ejemplo de un texto en lenguaje JSON que contiene datos sobre los estudiantes matriculados en una asignatura.

Algoritmo 6 Ejemplo de información representada en formato JSON

```
{
    [
        {
            "nombre": "Alejandro Quesada",
            "curso": 4,
            "grado": "GII+GIC"
        },
        {
            "nombre": "Sergio Gutiérrez",
            "curso": 3,
            "grado": "GIS"
        }
    ]
}
```

El paquete `Reports` cuenta, al igual que los dos anteriores, con una interfaz, `Report`, con un único método, `generate()`. Por otra parte, hay dos clases que implementan la interfaz `Report`, `SingleReportGenerator` y `GlobalReportGenerator`. La clase `SingleReportGenerator` se encarga de generar informes de autoevaluación de las prácticas individuales. Por otro lado, la clase `GlobalReportGenerator` genera un informe que resume el resultado general de la autoevaluación de un conjunto de prácticas.

Para generar los ficheros JSON de los informes, se ha usado una librería externa llamada Json Simple [27]. Esta librería incluye las clases `JSONObject` y `JSONArray`. La clase `JSONObject` representa una objeto JSON, que en el fichero aparece entre llaves. La clase `JSONArray` representa un array de `JSONObjects`, que en los ficheros JSON se representa con ayuda de los corchetes.

La clase `JSONObject` incorpora el método `toJSONString` que devuelve un objeto de tipo String con el contenido que debería tener el fichero JSON. A partir de ese objeto se genera el fichero JSON que contiene el informe.

La Figura 9 representa el flujo del algoritmo que sigue la clase `SingleReportGenerator` para construir el `JSONObject` que representa el informe.

Por otra parte, el Algoritmo 7 es un ejemplo de un informe de autoevaluación en formato JSON, que se obtiene como resultado de aplicar el algoritmo de la Figura 9.

Algoritmo 7 Ejemplo de informe de autoevaluación en formato JSON

```
{
    "studentName": "Alejandro Quesada Mendo",
    "projectName": "Práctica 1: Árboles binarios",
    "date": "02/08/2020 12:36",
    "build": "BUILD SUCCESFULL",
    "test":
        [
            {
                "testsuite": "practica2.CollisionAnalyzerTest",
                "correct": 3,
                "total": 3,
            },
            {
                "testsuite": "material.maps.HashTableMapLPTest",
                "correct": 0,
                "total": 1,
                "testcases":
                    [
                        {
                            "testName": "testBucketSize",
                            "cause": "FAILED",
                            "trace": "at material.maps.
                               HashTableMapLPTest.te..."
                        }
                    ]
            }
        ]
}
```

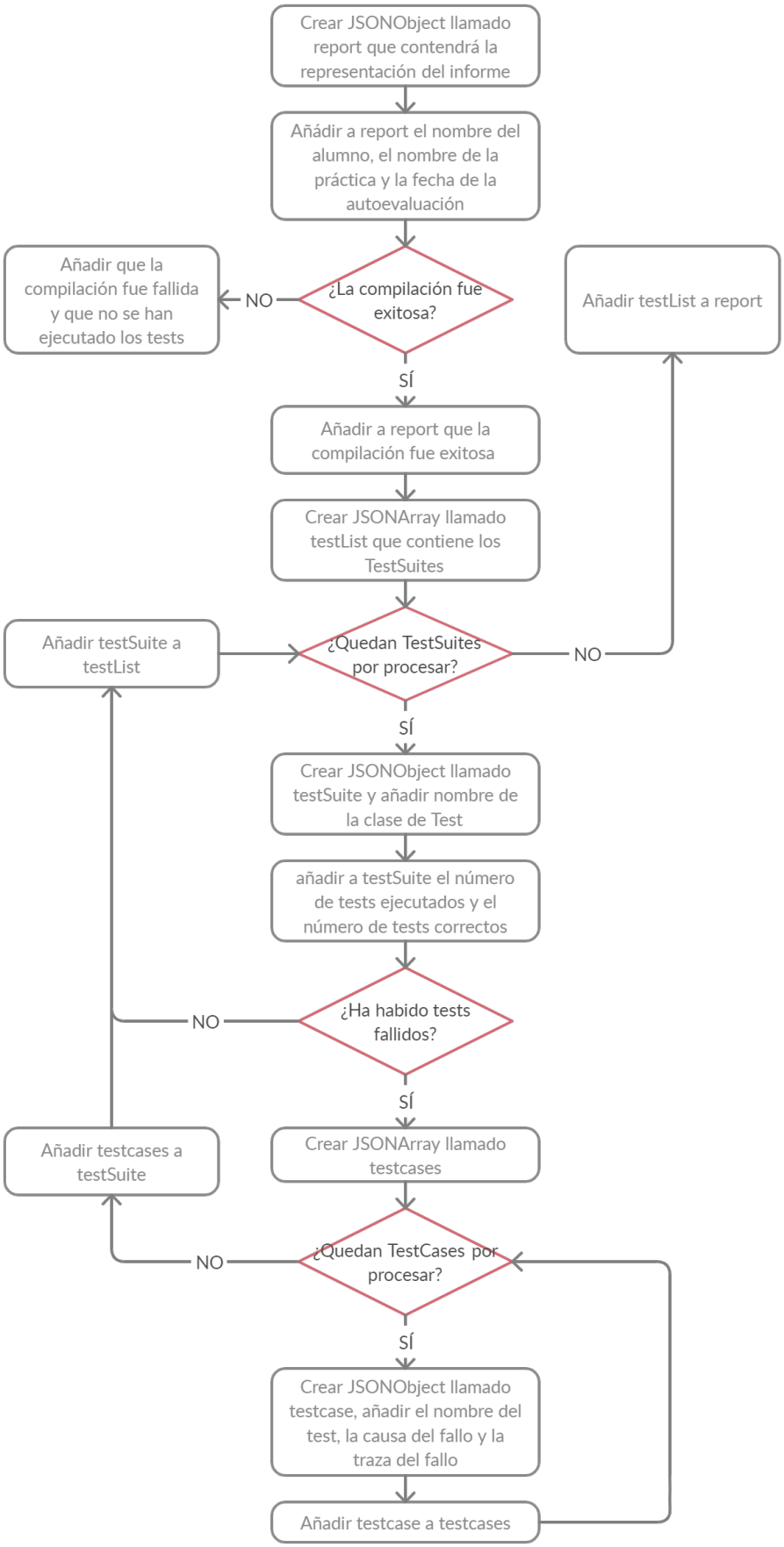

Figura 9: Diagrama de flujo del algoritmo de generación de los informes individuales

La clase `GlobalReportGenerator` opera de manera distinta. Para la generación del informe individual parte de la lista de `JSONObjets` que ha generado la clase `SingleReportGenerator`. Esta clase genera un informe en formato JSON que incluye el nombre de la práctica, la fecha de la evaluación y, para cada una de las prácticas individuales procesadas, el nombre del estudiante, y el numero de tests cuya ejecución ha resultado exitosa.

Este es un ejemplo de un informe global generado: El Algoritmo 8 es un ejemplo de un informe global generado por la clase `GlobalReportGenerator`.

Algoritmo 8 Ejemplo de informe global en formato JSON

```
{
    "projectName": "Práctica 1: Árboles binarios",
    "date": "02/08/2020 12:36",
    "students":
        [
            {
                "studentName": "Alejandro Quesada Mendo",
                "build": "BUILD SUCCESSFULL",
                "test":
                    [
                        {
                            "testSuite": "practica2.
                               CollisionAnalyzerTest",
                            "correct": 3,
                            "total": 3
                        },
                        {
                            "testSuite": "material.maps.
                               HashTableMapLPTest",
                            "correct": 0,
                            "total": 1
                        }
                    ]
            },
            {
                "studentName": "Sergio Gutiérrez López",
                "build": "BUILD SUCCESSFULL",
                "test":
                    [
                        {
                            "testSuite": "practica2.
                               CollisionAnalyzerTest",
                            "correct": 1,
                            "total": 3
                        }
                    ]
            }
        ]
}
```

3.2.6. Unplagger

Por último, el paquete `Unplagger` se encarga de realizar las comprobaciones antifraude cuando se evalúan un conjunto de prácticas a la vez.

La estructura del paquete es similar a la de los paquetes anteriores. En primer lugar, una interfaz, llamada `Unplagger` que cuenta con un único método, `detect()`. En segundo lugar, una clase que implementa dicha interfaz, llamada `UnplaggerJPlag`.

La clase `UnplaggerJPlag` está implementada de manera similar a las clases `CompilerAntJava` y `TesterAntJava` , usando `ProcessBuilder` para ejecutar un comando y `BufferedReader` para capturar la salida estándar de la ejecución de dicho comando.

En este caso, se ejecuta el software de antiplagio introducido en el Apartado 3.1.4, JPlag. Esta herramienta es muy amplia y permite personalizar el tipo de análisis antiplagio en función de las necesidades de cada proyecto. Para este proyecto en particular, solo interesa analizar las carpetas que contienen el código fuente desarrollado por los estudiantes, no el proyecto entero, ya que eso incluye los tests, que no están programados por los estudiantes, sino por los docentes.

El comando a ejecutar mediante el objeto `ProcessBuilder` es el siguiente:

`java -jar jplag.jar -l java19 -r ruta/al/directorio/de/resultados -S src -s ruta/a/los/proyectos`.

El argumento `-l java19` indica que se van a analizar proyectos escritos en Java 9 o inferior. El argumento `-r` indica que a continuación se especifica la ruta en la que se quieren escribir los resultados, si dicha ruta no existe, se crea automáticamente. El argumento `-S` indica el nombre de la carpeta de cada proyecto que se quiere analizar. Por último, el argumento `-s` indica la ruta al directorio que contiene los proyectos a analizar.

3.3. Despliegue del software: Dockerización

La puesta en producción de la aplicación es el último paso de la fase de implementación. Para ello se utilizan dos contenedores Docker que se comunican entre sí, uno alberga la base de datos y otro la aplicación web. En este apartado solo se trata el caso particular del contenedor de la aplicación web, que será en el que se ejecute JRater.

Los contenedores Docker se construyen a partir de imágenes. Una imagen es una "plantilla" que le indica al servicio de Docker cómo debe ser el contenedor a construir. Prácticamente todas las imágenes utilizan Linux como base para ejecutar las aplicaciones que contienen los contenedores Docker. Existen dos formas de construir un contenedor a partir de una imagen.

La primera opción, que en este caso particular se usa para desplegar el contenedor que alberga la base de datos, es utilizar una imagen oficial desde los servidores de Docker. Estas imágenes oficiales estas listas para contener, por ejemplo, un sistema operativo Linux o una base de datos sin tener que programar nada, desde la terminal del sistema operativo.

La segunda opción para construir un contenedor a partir de una imagen es "programándola" para adaptarla a las necesidades de cada caso particular. Para ello, hace falta un Dockerfile, un fichero que describe las características que tendrá el contenedor una vez

desplegado. Los Dockerfiles se construyen a partir de una imagen oficial y, mediante comandos, se pueden configurar los puertos abiertos del contenedor, qué programas instalar, cómo será el sistema de ficheros, desde qué directorio se ejecutarán las instrucciones y, finalmente, si se quiere ejecutar alguna acción una vez finalizada la construcción del contenedor. Éste es el método que se ha usado para desplegar el contenedor de la aplicación web.

El Algortimo 9 es un ejemplo reducido de un Dockerfile que lanza una aplicación web de Spring [28].

El contenedor de la aplicación web de este proyecto se crea a partir de una imagen oficial de OpenJDK, en particular de la versión 11 (porque JPlag requiere Java 11 para funcionar). Sobre esta imagen base se instala `zip`, `unzip` y `ant`, necesarios para que la aplicación web y JRater funcionen correctamente. Después, se añaden los ficheros de configuración de Netbeans, y los ejecutables (la aplicación web, JRater y JPlag). Finalmente se abre el puerto 8080 y se pone en marcha la aplicación web.

De esta forma ya están definidas los dos contenedores que componen el proyecto al completo. Pero en lugar de tener que lanzar ambos contenedores por separado y configurarlos individualmente, Docker proporciona un servicio llamado Docker-compose. Este servicio permite desplegar y configurar un conjunto de contenedores que vayan a a trabajar de forma conjunta a la vez. La principal ventaja de usar Docker-compose para el despliegue de la aplicación y de la base de datos, es que permite la creación de `Networks`. Las `Networks` son redes internas que la permiten comunicación entre distintos contenedores.

Para desplegar los contenedores usando Docker-compose, hay que crear un fichero `docker-compose.yml`. En este fichero se especifica la versión del servicio que va a usar para construir los contenedores, las posibles redes de interconexión entre los distintos servicios y por último cuáles son dichos servicios.

Para cada uno de los servicios contenidos en el Docker-compose hay que especificar, como mínimo, a partir de qué imagen se construyen. Además, en este caso particular también se especifican los puertos abiertos, las `Networks` y algunas variables de entorno en el caso de la base de datos. Por último, también se especifica el orden y condiciones de despliegue: primero se ha de poner en marcha la base de datos y, una vez que ésta funcione correctamente, se pone en marcha la aplicación web. De esta forma no hay problemas de conexión entre los contenedores y el despliegue es más robusto.

Algoritmo 9 Ejemplo de fichero Dockerfile para desplegar una aplicación web

```
FROM openjdk:11

WORKDIR /webapp

COPY WebApp.jar /webapp/app.jar

EXPOSE 8080

RUN apt-get update
RUN apt-get install -y zip unzip ant

CMD ["java" , "-jar", "app.jar"]
```

El Algoritmo 10 es un ejemplo reducido de un fichero `docker-compose.yml`.

La Figura 10 muestra el diagrama de despliegue del proyecto. Como se puede observar, la comunicación entre los dos servicios se realiza de forma interna dentro del Docker Host. Tanto el servicio que contiene la aplicación, como el servicio que contiene la base de datos se construyen a partir de una imagen base, OpenJDK y MySQL, respectivamente. Ambos servicios cuentan con un volumen cada uno. El volumen del servicio de la aplicación web contiene los directorios y ficheros sobre los que operan JRater y JPlag. Por otro lado, el volumen del servicio de la base de datos contiene los datos almacenados en ésta.

Para desplegar toda esta arquitectura, una vez están escritos los ficheros Dockerfile y Docker-compose, basta con ejecutar una única instrucción desde el terminal del sistema operativo: `docker-compose up --build`.

Automáticamente el servicio de Docker se encargará de descargar e instalar todo lo necesario y, a continuación, de desplegar ambos servicios.

Esta facilidad para poner en marcha aplicaciones es otro de los motivos por los que Docker ha sido la herramienta escogida para realizar el despliegue y puesta en producción del proyecto.

Algoritmo 10 Ejemplo reducido de fichero Docker-compose que despliega una aplicación web y una base de datos interconectadas

```
version: '2.1'

services:
    mysql:
        image: mysql:latest
        ports:
            - "3306:3306"
        networks:
            jrater-network:

    spring:
        build: ./App
        ports:
            - "8080:8080"
        depends_on:
            mysql:
        networks:
            jrater-network:

networks:
    jrater-network:
```

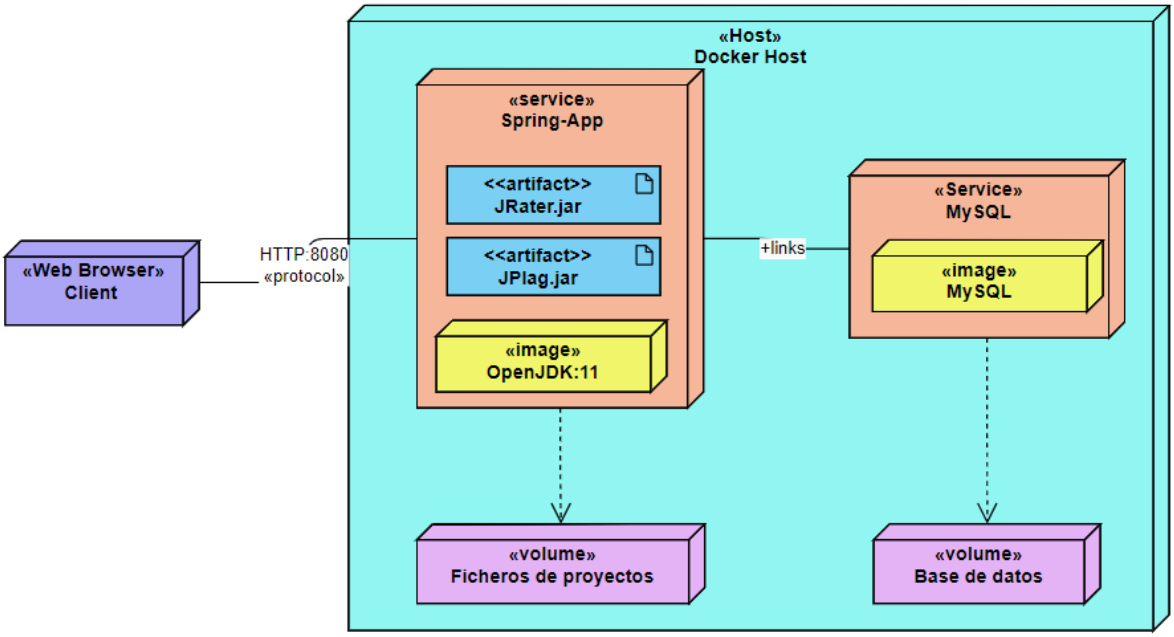

Figura 10: Diagrama de despliegue del proyecto

Capítulo 4

Métricas

Durante este capítulo se explican las distintas medidas que se han acumulado durante el desarrollo del proyecto, relativas a tiempos de ejecución, pero también tiempos de desarrollo de las distintas fases del proyecto. Por otra parte, también se mencionan métricas sobre la calidad y complejidad del software desarrollado.

4.1. Métricas temporales

Las métricas temporales hacen referencia a los tiempos de ejecución de la aplicación y a los tiempos de desarrollo de la misma.

Los tiempos de ejecución que se incluyen a continuación pueden servir como referencia pero son poco significativos, ya que pueden variar notablemente en función de la máquina en la que se ejecute la aplicación. Esto se debe a que JRater ejecuta la corrección de varios proyectos de manera concurrente en varios hilos de ejecución. Por eso, los tiempos en una máquina cuya CPU tenga, por ejemplo, 4 núcleos y 8 hilos, serán mucho más largos que en un servidor web con varias CPUs y más de 200 hilos.

No obstante, los tiempos de ejecución para los distintos casos de uso en un ordenador portátil con 8 GB de RAM, y un procesador intel i5-8250u (con 4 núcleos y 8 hilos) a 1,60GHz son:

- Autoevaluación de un único proyecto: 5,27 segundos.
- Autoevaluación de un conjunto de 15 proyectos: 12.78 segundos.
- Autoevaluación de un conjunto de 30 proyectos: 16.04 segundos.

Por otra parte, están los tiempos de desarrollo del proyecto. La Figura 11 es un diagrama de Gantt que representa la duración de cada una de las distintas actividades en las que se ha dividido el proyecto.

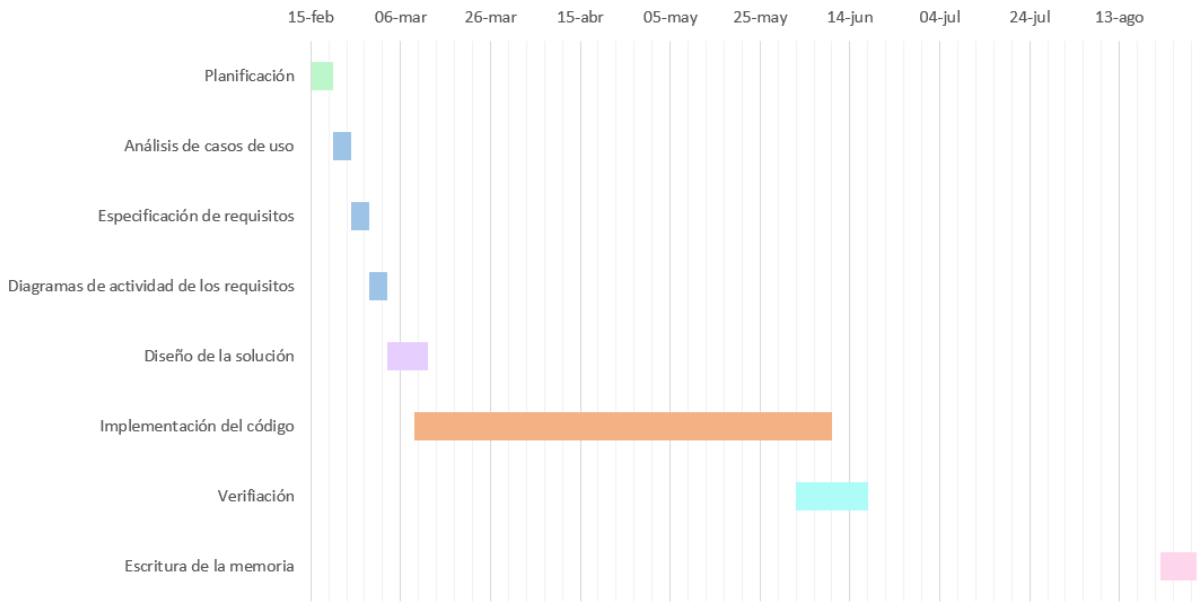

Figura 11: Diagrama de Gantt del proyecto

Como se puede observar, desde el final de la fase de planificación/verificación hasta el comienzo de las escritura de esta memoria hay un espacio de más de dos meses. Esto se debe a que durante ese periodo, se estuvo desarrollando el proyecto de la aplicación web, que duró desde el 15 de febrero hasta el 30 de agosto.

Por último, la Figura 12 muestra la tabla de hitos asociada al diagrama de Gantt de la Figura 11.

4.2. Métricas sobre calidad del software

Para evaluar la calidad y complejidad del software se ha utilizado una herramienta llamada SonarQube [29]. SonarQube es una plataforma desarrollada en Java que permite realizar análisis de código de manera automatizada. Con esta herramienta se pueden realizar auditorías exhaustivas del código de un proyecto en pocos segundos, obteniendo como

Nombre actividad	Fecha inicio	Duración en días	Fecha fin
Planificación	15-feb	5	20-feb
Análisis de casos de uso	20-feb	4	24-feb
Especificación de requisitos	24-feb	4	28-feb
Diagramas de actividad de los requisitos	28-feb	4	03-mar
Diseño de la solución	03-mar	9	12-mar
Implementación del código	09-mar	93	10-jun
Verifiación	02-jun	16	18-jun
Escritura de la memoria	22-ago	25	16-sep

Figura 12: Tabla de hitos asociada al diagrama de Gantt

resultado una serie de informes que evalúan, desde distintos puntos de vista, la calidad del software desarrollado.

Para el análisis con SonarQube, la propia compañía recomienda utilizar tecnología Docker para levantar un pequeño servidor web, que será el que se encargue de realizar el análisis [30]. Para ello, basta con descargar la imagen oficial de SonarQube mediante el comando `docker pull sonarqube` y, después, poner en marcha el contenedor Docker exponiendo el puerto 9000, desde el que se realiza la conexión al servidor, mediante el comando `docker run -d --name sonarqube -p 9000:9000 sonarqube`.

Una vez el servidor web está en funcionamiento, SonarQube permite realizar el análisis en sí de distintas formas, en función de la naturaleza de cada proyecto. En este caso particular, al ser un proyecto Maven, se ha usado el método especifico para proyectos Maven. Según la guía oficial, basta con ir a la raíz del proyecto y ejecutar el siguiente comando: `mvn sonar:sonar -Dsonar.host.url=http://localhost:9000 -Dsonar.login=token`.

Nótese que se accede a la dirección `http://localhost:9000`, esto es porque se han enlazado los puertos 9000 del contenedor Docker y de la máquina local, de forma que accediendo al puerto 9000 de la máquina local, se accede al puerto 9000 del contenedor, donde está esperando peticiones el servidor web de SonarQube. El *token* (ítem) del segundo argumento se genera al crear un proyecto dentro del propio servidor web.

En la Figura 13 se puede obtener una vista general del resultado del análisis, tal y como lo muestra SonarQube.

Como se puede observar, se evalúa la calidad del proyecto desde el punto de vista de la fiabilidad o robustez, la seguridad y el futuro mantenimiento.

El código desarrollado ha obtenido la puntuación más alta posible en todas las categorías que incluye el análisis. Esto quiere decir que el proyecto desarrollado es fiable, robusto y, sobre todo, fácil de mantener para futuros desarrolladores.

Por otro lado, SonarQube indica que no ha encontrado pruebas que comprueben el correcto funcionamiento del proyecto. Ésta es una de las posibles mejoras que se podrían desarrollar en futuros trabajos, tal y como se comenta en el Apartado 5.3. Por otra parte, ha encontrado un 8 % de duplicación en el código, esta duplicación se encuentra en las clases `CompilerAntJava` , `TesterAntJava` y `UnplaggerJPlag`, cuya estructura y funcionamiento es similar.

Además de puntuar dichos apartados, esta herramienta expone los motivos por lo que considera que hay partes de código que pueden acarrear conflictos y, además, explica cómo solucionar dichos problemas incluyendo ejemplos de cómo hacerlo correctamente.

Por eso, se ha seguido un proceso iterativo en el que se han ido resolviendo los problemas que esta herramienta encontraba de manera progresiva. Según se iban resolviendo los problemas de fiabilidad y mantenimiento, la puntuación SonarQube otorga aumentaba hasta llegar a la situación reflejada en la Figura 13.

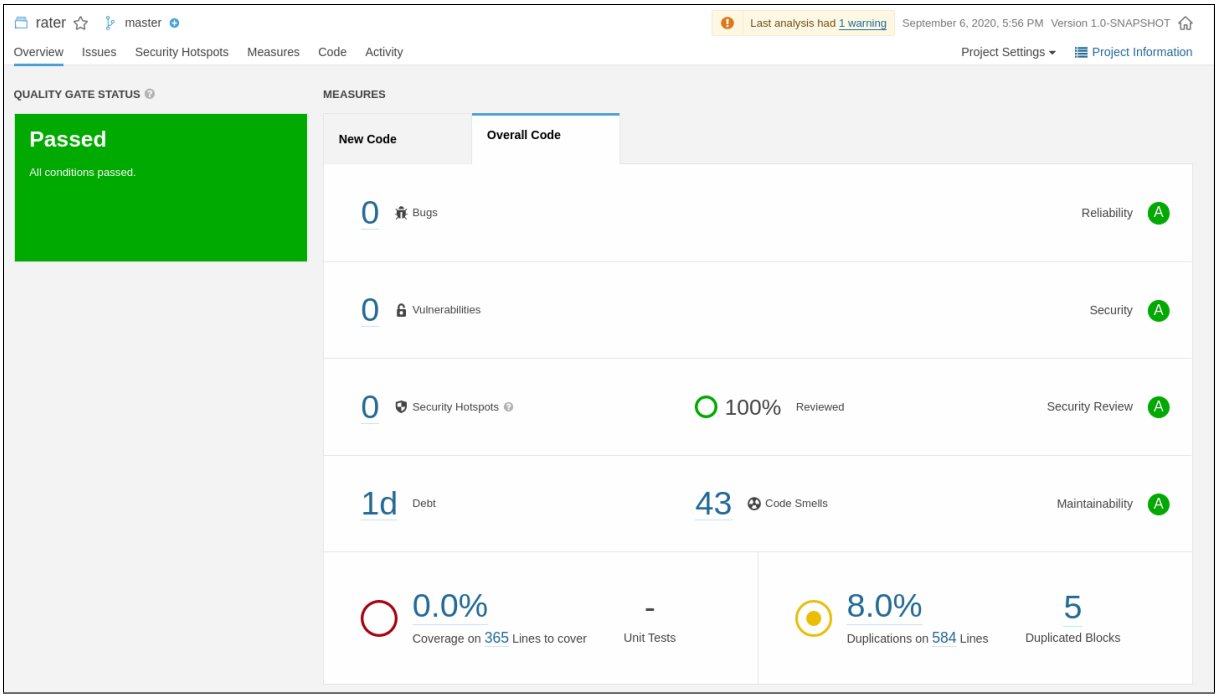

Figura 13: Análisis de calidad con SonarQube

Capítulo 5

Conclusiones

A lo largo de este capítulo se resumen los principales objetivos cumplidos y lecciones aprendidas durante la realización del proyecto. Finalmente, se propone un conjunto de posibles trabajos para continuar con el desarrollo del proyecto, orientado a futuros desarrolladores.

5.1. Objetivos conseguidos

El proyecto sobre el que trata esta memoria cumple con los objetivos descritos en la introducción de la misma respecto a servir como herramienta de autoevaluación para prácticas de programación, tanto para estudiantes como docentes.

En concreto, el proyecto desarrollado cumple con los objetivos generales planteados:

- Agiliza el proceso de corrección de prácticas o exámenes en asignaturas de programación. Se apoya sobre Ant para ejecutar de manera automática tests de referencia y construye un informe de corrección a partir del resultado de dicha ejecución.
- Ofrece *feedback* sobre los resultados de corrección de una práctica o conjunto de prácticas en un formato legible para una aplicación web, en concreto, JSON. Para ello, utiliza la librería Json Simple que permite interactuar con ficheros JSON como si fueran objetos Java.
- Permite obtener un informe anticopia de las prácticas y/o exámenes entregados por los estudiantes. Se usa la herramienta JPlag para la generación de informes anticopia adaptados al tipo de proyectos que se realizan en la asignatura de EDA.
- Es una aplicación multiplataforma que pueda ejecutarse independientemente del sistema operativo subyacente. Para poder ejecutar la aplicación es necesario que esté instalado el entorno de ejecución de aplicaciones Java, versión 11 o superior.
- Se ha desarrollado una aplicación fiable, robusta y fácil de mantener de cara a su futuro mantenimiento, según los análisis ejecutados por SonarQube.

5.2. Lecciones aprendidas

En esta sección se van a comentar algunas partes del proceso de desarrollo que más tiempo han consumido por falta de información y que puedan ser de utilidad para personas que estén desarrollando proyectos parecidos a este, o que utilicen tecnologías similares.

5.2.1. Netbeans y sus ficheros de configuración

La ejecución de los tests de proyectos Netbeans usando Ant fue una tarea complicada. Sucedía que, usando como ejemplo un proyecto Netbeans que fue facilitado los tutores y el comando `ant test`, desde la raíz del proyecto, no se ejecutaban los tests.

Se probó a crear un pequeño proyecto Netbeans de prueba, y sí que se ejecutaban los tests sin problema. Después se probó a abrir el proyecto de prueba (con el que se estaban teniendo problemas) en Netbeans y ejecutar los tests desde el propio entorno de desarrollo y sí que se ejecutaban correctamente. Así que se probó otra vez desde la consola usando Ant y en este caso sí se ejecutaban los test, sorprendentemente.

A continuación, se comprobó qué cambios podía haber sufrido el proyecto tras haberlo abierto desde Netbeans. Lo que se encontró fue un fichero llamado `private.properties` que guardaba la siguiente información "`user.properties.file=/home/mayte/netbeans/8.2/build.properties`". Esta ruta apuntaba a la localización de los ficheros de compilación de una máquina de la tutora de este proyecto, y cuando Ant buscaba dicho fichero no lo encontraba y, por lo tanto, no ejecutaba los tests.

La solución a este problema fue modificar ese fichero en los proyectos de referencia de la aplicación, apuntando a la ruta "`webapp/build.properties`" dentro del contenedor Docker donde se ejecuta la aplicación.

5.2.2. Usar SonarQube desde el principio del proyecto

SonarQube es una herramienta muy útil para solucionar fallos y malas prácticas no detectadas durante el desarrollo del proyecto. En el caso de este proyecto, se uso únicamente al final del desarrollo para el análisis sobre calidad de software del apartado 4.2. Una vez se realizo en el análisis hubo un proceso iterativo de corrección de fallos y nuevos análisis hasta que la calidad del proyecto fue aceptable.

Si se hubiese usado SonarQube desde el comienzo de la fase de desarrollo, el proceso iterativo para solventar los problemas que detecta la herramienta habría sido mucho más ágil, ya que se hubiera aprendido de las malas prácticas y errores en lugar de repetirlas.

Además, el proyecto contaría con un historial de la calidad del mismo a lo largo del tiempo, que permitiese, en caso de necesidad, volver a una versión en la que la calidad fuera mejor.

5.2.3. Requisitos de la aplicación antes de Dockerizar

Por último, una buena práctica que hubiese ahorrado tiempo y varios intentos a la hora de dockerizar la aplicación es apuntar, a lo largo del desarrollo, cuáles eran los requisitos

software que necesitaba la aplicación para funcionar correctamente, y también qué versión de dichos requisitos.

En el caso particular de esta aplicación, se necesitaban los programas `zip` y `unzip` de Linux, Ant y Java Runtime Enviroment (JRE) versión 11 o superior, ya que JPlag no se puede ejecutar con una versión de Java inferior.

Al no haber apuntado estos requisitos, las primeras versiones del contenedor fallaban estrepitosamente al intentar ejecutar algunas partes de la aplicación.

5.3. Trabajos futuros

En esta sección se introducen una serie de posibles ampliaciones y mejoras del proyecto para futuros desarrolladores.

5.3.1. Pruebas de Software

Una posible mejora de cara a la fiabilidad y mantenimiento del proyecto es la implementación de pruebas automáticas que comprueben el correcto funcionamiento de la aplicación desarrollada.

Este tipo de pruebas son especialmente útiles de cara a mantener y ampliar el proyecto. Son una manera rápida de comprobar si los cambios que se realizan comprometen el correcto funcionamiento del proyecto o no.

5.3.2. Soporte para nuevos lenguajes

Actualmente, la aplicación solo está enfocada a trabajar con proyectos Java desarrollados en Netbeans. No obstante, el diseño de esta aplicación está pensado para que sea escalable y, en un futuro, se pueda trabajar con distintos tipos de proyectos en Java o, incluso, otros lenguajes.

Esto es posible ya que para la ejecución de los tests lo que se hace es llamar internamente a Ant para que realice dicha ejecución. Por lo tanto, siempre que un lenguaje implemente tests unitarios, es susceptible de ser compatible con JRater.

El soporte para nuevos lenguajes sería interesante para que esta aplicación sirviese en otras asignaturas que se desarrollan en otros lenguajes como C++ o Python, o con proyectos Maven, por ejemplo.

Bibliografía

[1] *Guías docentes de las asignaturas Programación Orientada a Objetos y Estructuras de Datos Avanzadas del curso 2019/2020*. URL: https://gestion3.urjc.es/guiasdocentes/ (vid. pág. 1).

[2] *Web oficial de la Universidad Rey Juan Carlos*. URL: https://www.urjc.es/ (vid. pág. 1).

[3] *Web oficial de Java*. URL: https://www.java.com/ (vid. pág. 1).

[4] *Web oficial de Netbeans*. URL: https://netbeans.org/ (vid. pág. 1).

[5] *Web del Aula Virtual de la Universidad Rey Juan Carlos*. URL: https://www.aulavirtual.urjc.es/ (vid. pág. 1).

[6] *Web oficial de DOMjudge*. URL: https://www.domjudge.org/ (vid. pág. 3).

[7] *Web oficial de Acepta el reto*. URL: https://www.aceptaelreto.com/ (vid. pág. 3).

[8] *Web oficial de Mooshak*. URL: http://mooshak.inf.um.es/ (vid. pág. 3).

[9] *Web oficial de Linux*. URL: https://www.linux.org/ (vid. pág. 6).

[10] Winston W. Royce. "Managing the Development of Large Software Systems". En: *ICSE '87* (1987), págs. 328-338. DOI: http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf (vid. pág. 8).

[11] *Web oficial de Github*. URL: https://github.com/ (vid. pág. 9).

[12] *Los 10 lenguajes de programación más usados según Github*. URL: https://hardware360.net/2020/07/01/los-10-lenguajes-de-programacion-mas-populares-segun-github/ (vid. pág. 11).

[13] *Repositorio oficial de JPlag*. URL: https://github.com/jplag/jplag (vid. pág. 11).

[14] *Web oficial de IntelliJ IDEA*. URL: https://www.jetbrains.com/idea/ (vid. pág. 11).

[15] *Web oficial de JetBrains*. URL: https://www.jetbrains.com/ (vid. pág. 11).

[16] *Web Oficial de Maven*. URL: https://maven.apache.org/ (vid. pág. 12).

[17] *Wikipedia Unix*. URL: https://es.wikipedia.org/wiki/Unix (vid. pág. 12).

[18] *Versiones de Java*. URL: https://www.arquitecturajava.com/las-versiones-de-java/ (vid. pág. 12).

[19] *Wikipedia del lenguaje C*. URL: https://es.wikipedia.org/wiki/C_(lenguaje_de_programaci%C3%B3n) (vid. pág. 12).

[20] *Wikipedia del lenguaje C++*. URL: https://es.wikipedia.org/wiki/C%2B%2B (vid. pág. 12).

[21] *Wikipedia del lenguaje C Sharp*. URL: https://es.wikipedia.org/wiki/C_Sharp (vid. pág. 12).

[22] *Web oficial de Python*. URL: https://www.python.org/ (vid. pág. 12).

[23] Alejandro Quesada Mendo. "Trabajo de Fin de Grado: Aplicación web para la corrección de prácticas de programación". En: (2020) (vid. pág. 12).

[24] *Web oficial de Docker*. URL: https://www.docker.com/ (vid. pág. 13).

[25] *Wikipedia de pipeline*. URL: https://es.wikipedia.org/wiki/Tuber%C3%ADa_(inform%C3%A1tica) (vid. pág. 15).

[26] *Web oficial de JSON*. URL: https://www.json.org/json-en.html (vid. pág. 22).

[27] *Web oficial de JSON-Simple*. URL: https://code.google.com/archive/p/json-simple/ (vid. pág. 23).

[28] *Web oficial de Spring*. URL: https://spring.io/ (vid. pág. 27).

[29] *Web oficial de SonarQube*. URL: https://docs.sonarqube.org/latest/ (vid. pág. 32).

[30] *Análisis de código con SonarQube con la ayuda de Docker*. URL: https://www.federico-toledo.com/analisis-de-codigo-con-sonarqube/ (vid. pág. 33).