eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Ingeniería del Software

Trabajo de Fin de Grado

Framework para refactorización de código Java

Autor

Iñaki García Noya

2021

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Ingeniería del Software

Trabajo de Fin de Grado

Framework para refactorización de código Java

Autor

Iñaki García Noya

Directora

Maider Azanza Sese

Resumen

Este documento corresponde a la memoria del Trabajo de Fin de Grado con título "*Framework* para refactorización de código Java" en colaboración con la empresa INDABA CONSULTORES S.L. El proyecto se ubica dentro de la rama de calidad del *software*, concretamente en el mantenimiento del mismo. Se ha desarrollado un programa que permite obtener el árbol de llamadas de un proyecto Java, complementando el árbol con métricas de métodos y de clases. En primer lugar, se analiza el código fuente y compilado de una aplicación Java, se construye el árbol de llamadas y se añaden métricas que analizan la calidad del mismo, solicitadas por la empresa. El objetivo del proyecto es remplazar un trabajo manual que realizaba la empresa con un proceso automatizado y libre de errores.

Para llevar a cabo este proyecto, se ha realizado un estudio sobre las diferentes opciones que existen, evaluando cada una de las opciones y argumentando la elección de cada una de las herramientas utilizadas. Por último se han realizado pruebas exhaustivas que verifican el correcto funcionamiento del programa y asegurar un buen rendimiento en su ejecución, siguiendo los requisitos definidos por la empresa.

Índice general

Índice de figuras

Índice de tablas

Índice de códigos

1. CAPÍTULO

Introducción

La producción de *software* de calidad es uno de los mayores retos que tiene el sector industrial hoy en día. El avance descomunal y frenético de las últimas tecnologías como la inteligencia artificial (IA) o el *Big Data*, genera un incremento del número de líneas de los proyectos y a su vez de su complejidad. Esta complejidad hace que el desarrollo de los proyectos sea arduo y lento, con la consecuencia de que el mantenimiento del *software* acapare entre el 60 % y el 90 % del ciclo de vida de un proyecto [Pressman, 2009]. Durante esta fase de mantenimiento se realizan diferentes tareas: búsqueda de fallos, añadir nuevas características y modificar el código después de que el software se haya distribuido. Esta complejidad hace que sea necesario disponer de una buena gestión de la calidad de los proyectos, por lo que se hace uso de un sin fin de herramientas para mantener esa calidad durante el ciclo de vida de los proyectos.

Este Trabajo de Fin de Grado (TFG en adelante), se ha realizado en colaboración con la empresa INDABA CONSULTORES S.L. que se especializa en soluciones personalizadas en el diseño y desarrollo de aplicaciones. En esta empresa, se realizan muchas tareas de modificación de código (*refactoring* en adelante), para gestionar la calidad de aplicaciones. Para encontrar oportunidades de *refactoring* dentro de una clase; como por ejemplo extraer un método de una clase, realizaban una tabla CSV introduciendo de manera manual las llamadas que realizaban los métodos entre ellos. De esta manera, se podía obtener una vista general del diseño del programa y detectar posibles errores de diseño y oportunidades de mejora. También, se complementaba esa tabla con ciertas métricas que analizaban la calidad del código las cuales se calculaban a mano, para disponer de más información sobre los métodos o para detectar posibles malos diseños (Véase el Capítu-

lo 2). Este proceso es costoso de realizar a mano, ya que no se conoce la estructura del código de antemano al ser los programas son de clientes externos y es laborioso seguir el recorrido de todas las llamadas del programa. Además, como se trata de un proceso manual, se necesita mucho tiempo para obtener todos los datos y los mismos pueden contener errores. Por si fuera poco, durante el proceso de *refactoring*, el cliente puede sugerir cambios lo que conllevaría volver a calcular todos los datos de la tabla.

En este documento se detalla el proyecto realizado describiendo el desarrollo y el diseño de una herramienta que permite realizar un análisis sobre proyectos Java, para obtener de manera automática un árbol de llamadas junto a métricas especificadas por la empresa. Este árbol de llamadas se puede interpretar como un gráfico de llamadas, pero no se ha podido implementar la visualización gráfica del mismo. Los gráficos de llamadas [Jász. et al., 2019] son gráficos dirigidos que representan las relaciones de control entre los métodos de un programa. Todo ello se complementa con el estudio previo realizado sobre todas las herramientas que se utilizan, describiendo importancia del programa dentro de la calidad del *software*, indicando planificación para lograr los objetivos del proyecto, las pruebas realizadas para verificar los requisitos y todos los problemas que se tuvieron durante el desarrollo.

Gracias al uso de esta herramienta, se consigue analizar un proyecto Java de manera automática realizando un análisis sobre el árbol de llamadas con ayuda de filtros (entre ellos, métricas definidas por la empresa). De esta forma se sustituye un proceso que se realizaba en la empresa de manera manual y cuya realización conllevaba tiempo, por una herramienta automática. Por consiguiente, se consigue reducir considerablemente el tiempo de la gestión de calidad que realiza la empresa en este ámbito, se reduce el número de fallos que conllevaba el proceso manual y se dispone de una personalización para realizar análisis más específicos.

El objetivo de esta memoria es aunar todos los conocimientos adquiridos durante el proyecto, presentar la primera planificación y explicar el desarrollo del proyecto. Todo ello se ha estructurado en capítulos, representando cada una de las fases del ciclo de vida del proyecto:

- **Capítulo 2:** contexto en el cual se ubica el proyecto, describiendo la empresa, los conceptos básicos para entender el entorno del proyecto y el estado anterior del proyecto. De esta manera se ubica el proyecto en el sector actual de la informática y en la empresa.
- **Capítulo 3:** planificación inicial del proyecto. En ella, se describe con detalle el

producto final esperado del proyecto y la planificación inicial con la gestión del proyecto: alcance, tiempo, riesgos, calidad, comunicación e interesados.

- **Capítulo 4:** estudio y análisis de las diferentes herramientas que se han utilizado para la implementación del proyecto. Se realiza también una breve descripción sobre los conceptos básicos para entender las herramientas. Después de presentar por cada herramienta una descripción y una comparativa de sus ventajas y desventajas, se argumentan las herramientas finales que se han utilizado.
- **Capítulo 5:** diseño que se ha utilizado para implementar el proyecto en la fase de desarrollo.
- **Capítulo 6:** desarrollo de la implementación del proyecto. Se explican las funcionalidades principales del programa, indicando las opciones que se plantearon durante el desarrollo. También, se indica como se utiliza la herramienta.
- **Capítulo 7:** pruebas exhaustivas realizadas durante el desarrollo del proyecto para garantizar que funciona para las mismas y también que se cumplen los requisitos definidos en la planificación. Se indican todos los errores encontrados y las decisiones que se tomaron para solucionar dichos errores.
- **Capítulo 8:** seguimiento y control del proyecto, donde se redacta la segunda planificación del proyecto y los motivos de su realización. Por otra parte, se analizan las desviaciones finales respecto a la primera y segunda planificaciones.
- **Capítulo 9:** conclusiones finales del proyecto, enumerando las lecciones aprendidas durante la realización del proyecto y las líneas de avance futuras relacionadas con el mismo.

2. CAPÍTULO

Antecedentes

En este capítulo se presentan los conceptos en los que se basa el TFG y así como la descripción de la empresa y el estado actual de la misma. Estos conceptos están presentes durante todo el documento y es necesario su comprensión para poder entender el proyecto. En primer lugar, se presenta la empresa y el estado actual de la misma (2.1). Después, se sitúa el estado del arte del proyecto definiendo los conceptos en los que se basa (2.2), dejando claro la importancia del campo en el que se ha trabajado. Estos conceptos se adquirieron en la fase de *Administración del campo (AC)*. Finalmente, se realiza una descripción de la situación de partida del proyecto, describiendo la motivación para realizar esta proyecto y la relación previa de la empresa con el alumno (2.3).

2.1. Descripción de la empresa

INDABA CONSULTORES S.L. es una empresa especializada en la consultoría informática que se creó para ofrecer al mercado soluciones informáticas para la ayuda a la toma de decisiones y a la gestión del conocimiento, incorporando las últimas tecnologías informáticas disponibles en los mercados internacionales. Desde el año 2000, ha ido ampliando el espectro de cobertura de su oferta de productos y servicios, conforme las necesidades de sus clientes se lo han ido reclamando.

INDABA CONSULTORES sabe que la capacidad tecnológica es primordial para ser competente en el mercado y que todos los miembros de la organización son necesarios y deben estar debidamente preparados para desempeñar eficientemente el papel que les corresponde en ese proceso: búsqueda y rastreo de información, evaluación y selección,

negociación, aprendizaje y asimilación tecnológica. Cabe destacar la certificación en la administración y desarrollo sobre la plataforma Liferay[1].

Desde su creación forma parte de la asociación de empresas GAIA[2] y desde el año 2005 a ESLE[3], asociación de empresas de software libre de Euskadi. Destacar también que Indaba ha mantenido constantemente relaciones con centros de formación como son el Instituto de Máquina Herramienta, Tknika y la Universidad del País Vasco (UPV/EHU). Esta serie de asociaciones favorece la cooperación en proyectos de I+D+i.

2.2. Estado del arte

2.2.1. Calidad del *software*

La **calidad del *software*** [Bourque and R.E. Fairley, 2014] es el grado el cual un producto de software cumple los requisitos establecidos; sin embargo, la calidad depende del grado en que esos requisitos establecidos representan con precisión las necesidades, deseos y expectativas de las partes interesadas. Define y evalúa la adecuación de los procesos de *software* para proporcionar pruebas que establezcan la confianza en que los procesos de software son apropiados y producen productos de *software* de calidad adecuada para los fines previstos. Desarrollar *software* con calidad es algo crucial, pero prevenir y mejorar la calidad del *software* durante la fase de mantenimiento es algo crítico [Lacerda et al., 2020]. Para ello, existen diferentes herramientas (programas, *plug-in*) para ayudar al desarrollador a realizar estas tareas. Java es el lenguaje de programación que mayor número de herramientas dispone para realizar estas tareas. En la Figura 2.1 se muestra un gráfico donde se indica por cada lenguaje de programación, cuantos estudios avalan la anterior sentencia.

[1] Liberay: https://www.liferay.com/es/
[2] GAIA: http://www.gaia.es/
[3] ESLE: http://www.esle.eu/es

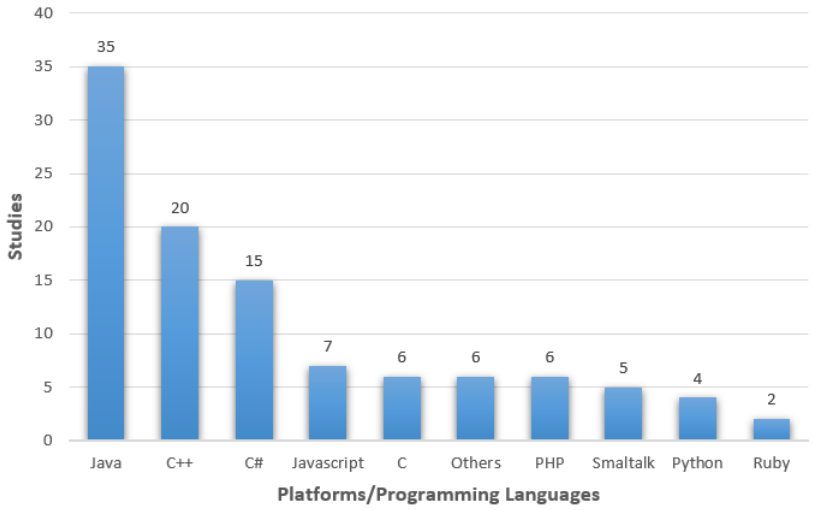

Figura 2.1: Plataformas/lenguajes de programación más utilizadas para el desarrollo de herramientas [Lacerda et al., 2020]

El **mantenimiento del *software*** es una actividad esencial para cualquier sistema de software. Entre el 60 % y el 90 % de los costes del *software* están relacionados con las actividades de mantenimiento: reparación de fallos de diseño e implementación, adaptación del *software* a un entorno diferente (*hardware*, sistema operativo) y añadir o modificar funcionalidades. El mantenimiento del software es difícil debido a la falta de documentación útil, por lo que los programas con código fuente largo y extenso se convierten en la única vía de información fiable.

Este mantenimiento del *software* se puede prolongar si no se identifican los ***code smell***s. Los ***code smell*** son violaciones de los principios del diseño del código. Incrementan la deuda técnica afectando a la evolución y mantenimiento del *software*. Se pueden identificar en muchos contextos: requisitos, pruebas, arquitectura y servicios. Para eliminar estos *code smell* se utilizan diferentes mecanismos de ***refactoring***. También, se pueden emplear métricas que ayudan a la identificación de los mismos (véase el apartado 2.2.2).

Contextos de *code smell*

Existen diferentes enfoques para detectar *code smell*, a continuación se presentan algunos de ellos [Lacerda et al., 2020]:

- **Percepción humana:** enfoque manual normalmente basado en diferentes pautas seguidas por los desarrolladores para detectar defectos de diseño.

- **Métricas:** se utiliza para evaluar/medir diferentes elementos del código fuente (p. ej.: atributos, parámetros...) para tomar decisiones.
- **Reglas:** combinación de reglas, expresiones lógicas y métricas. Cada regla es específica para ciertos *code smell* concretos de manera manual o automática.
- **Búsqueda probabilística:** haciendo uso de diferentes algoritmos y reglas directamente en el código fuente. La mayoría de técnicas de esta categoría aplican *machine learning* y lógica difusa (*fuzzy logic*).
- **Visualización:** se combina la experiencia del desarrollador con el proceso de detección automatizado. En algunos casos, si el *software* es muy complejo la representación gráfica del artefacto de *software* puede ser una solución para hacer frente a la complejidad.

Problemas con el *refactoring*

Se identifica un proceso de *refactoring* cuando se mejora el sistema *software* aplicando transformaciones que no modifiquen el comportamiento previo del sistema. Ayuda a que el código sea más comprensible y elimina los posibles problemas, además de mejorar la calidad interna del sistema. También se utiliza para crear más módulos y estructuras específicas del sistema. A pesar de que se puede aplicar este proceso en otros muchos contextos (p. ej.: modelos UML, esquemas de bases de datos...), es difícil identificar qué tipo de *refactoring* aplicar y cuándo tienen que ser aplicados esos cambios. La relación entre los *code smell* y el *refactoring* no es uno a uno, ya que se puede aplicar más de un tipo de *refactoring* para el mismo *code smell*.

Otro de los problemas que sufre, es que los beneficios de la calidad del *software* que se obtiene mediante estas prácticas se diluyen a menudo a raíz de los altos costes y la baja prioridad en comparación con las correcciones de errores y la implementación de nuevas características. Esto se debe a que el 40 % del tiempo que se invierte en el mantenimiento de un sistema *software* le pertenece al coste de entender el funcionamiento del sistema y arquitectura [Lacerda et al., 2020].

Sin embargo, si se detectan los *code smell* en una fase inicial del proyecto, se reduce considerablemente el coste del mantenimiento del *software*. Además, muy pocas herramientas disponen de la capacidad de analizar proyectos muy grandes (millones de líneas de código).

Contextos de *refactoring*

Como ya se ha comentado, el *refactoring* se puede aplicar en diferentes contextos, pero ninguno es siempre la mejor opción ya que siempre se disponen de diferentes opciones. A continuación se presentan algunas de las diferentes oportunidades para realizar este proceso:

- **Métricas de calidad:** la mayoría de métricas están asociadas a la cohesión, acoplamiento y distancia entre fragmentos de código.
- **Precondición:** primero se evalúa una condición antes de aplicar el *refactoring*. La condición puede estar relacionada con métricas.
- **Agrupación:** utiliza algoritmos para calcular la similitud y combinación de fragmentos de código.
- **Gráficos:** se combina con el *slicing* de código y el análisis dinámico (Véase el Capítulo 4) para encontrar oportunidades de cambios. Se utiliza para detectar cambios en SPL (*Software Product Line*) y modelos.
- **Patrones:** identifica patrones en el diseño del código y se aplican los cambios siguiendo las indicaciones de patrones de diseño.

2.2.2. Métricas *software*

La medición del *software* es un componente esencial para hacer buena ingeniería del *software*. Con el rápido desarrollo de software a gran escala, la complejidad del software crece rápidamente, lo que hace que la calidad sea cada vez más difícil de controlar. Para ello, es importante incluir métricas en los proyectos informáticos [Honglei et al., 2009]. El desarrollo de *software* incluye documentos de requisitos, diseño de los programas, implementación y pruebas, todo ellos con la capacidad de que puedan ser medibles y analizados mediante diferentes mecanismos. Las **métricas** de *software* aportan una medición del *software* y valores cuantitativos a los atributos que intervienen en el producto o proceso. Además, el conjunto de métricas elegido debe basarse en un modelo de calidad bien definido, que incorpore ideas de muchos de los actuales de calidad estándar (desde modelos jerárquicos fijos (*Boehm, McCall's FCM*) hasta enfoques más flexibles como el *Goal-Question-Metric (GQM)*) [Harrison et al., 1997]. Estos modelos flexibles ayudan a aclarar qué aspectos de la calidad se tienen en cuenta y por qué.

Clasificación

Las métricas se pueden clasificar en 3 grupos diferentes [Honglei et al., 2009]:

- **Métricas de procedimiento:** hacen énfasis en el procedimiento del desarrollo *software* (p. ej.: duración de un procedimiento, el coste, si los métodos utilizados son eficaces...). Incluye la mejora del procedimiento y las predicciones para el procedimiento futuro.
- **Métricas de proyecto:** tienen como función entender y controlar la situación y el estado del proyecto (p. ej.: escala, coste, carga de trabajo...). Su objetivo principal es ajustar el proyecto para evitar los problemas o riesgos, y ayudar a optimizar los planes de desarrollo.
- **Métricas de producto:** sirven para entender y controlar la calidad el producto (p. ej.: reusabilidad, mantenibilidad, complejidad...). Su objetivo es predecir y manipular la calidad del producto, midiendo el producto desde la fase de requisitos hasta la fase de mantenimiento.

Ventajas

A continuación se presentan las principales ventajas que conlleva el uso de las métricas en procesos y productos *software*:

- **Comprensión:** permite entender el proyecto, producto, proceso o recurso de manera más intuitiva y rápida. Selecciona y determina la línea de base adecuada para la evaluación, la previsión, el control y mejora.
- **Previsión:** midiendo las relaciones entre proyecto, producto, procedimiento y recursos, se puede estimar la tendencia de futuro. De esta manera, permite realizar plan de acciones sobre esas previsiones.
- **Evaluación:** evalúa la situación ”real” del proyecto, producto o proceso, que podría ayudar a evaluar el desarrollo mediante ciertos estándares de calidad.
- **Control:** permite calcular las desviaciones que sufre el desarrollo del *software* con la planificación inicial. Además, puede identificar los lugares donde se podrían producir defectos y luego ajustar el plan para encontrar el control real.

2.2.3. Sistemas de apoyo a la decisión

Los sistemas de apoyo a la decisión (*Decision Support System (DSS)*) [Donzelli, 2006], combinan las capacidades de los humanos y ordenadores, con el objetivo de mejorar la capacidad de las decisiones. Estos sistemas están penetrando en sectores cada vez más complejos, desde la detección de fraudes en los seguros hasta la adquisición de sistemas militares, pasando por la planificación de emergencias.

En los sectores que utilizan sistemas informáticos, se puede observar que con el paso de los años la cantidad de datos que tienen que controlar aumenta de una manera desmesurada. Esta gran carga de datos genera un gasto de tiempo y dinero, lo que puede llevar al fracaso a los sectores industriales ya que no son capaces de realizar una correcta gestión. A raíz de este problema, muchas empresas están integrando DSS para realizar la gestión de los datos.

Dentro de DSS se ubica la **analítica visual** (*Visual Analytics (VA)*) [Reddivari et al., 2013], que permite a un usuario establecer una conexión con una máquina para interactuar con la información relevante. También puede generar visualizaciones interactivas para la comprensión, el razonamiento y la toma de decisiones sobre la base de conjuntos de datos muy amplios y complejos [Keim et al., 2008]. Con la integración de este análisis se quieren conseguir los siguientes objetivos:

- Sintetizar la información y obtener conocimientos a partir de datos masivos, dinámicos, ambiguos y a menudo contradictorios.
- Detectar lo esperado y descubrir lo inesperado.
- Proporcionar evaluaciones oportunas, defendibles y comprensibles.
- Comunicar las evaluaciones de forma eficaz para que se tomen medidas.

VA se basa en diferentes áreas de investigación como la minería de datos, la gestión de datos, la fusión de datos la estadística y la ciencia de la cognición (entre otras).

2.3. Situación de partida

Antes de plantear el TFG, el alumno y la empresa tuvieron una relación con anterioridad, ya que el alumno realizó prácticas entre los meses de junio y agosto del año 2020. En

estas prácticas, también se desarrolló una nueva herramienta y también tenía relación con la calidad del *software*. Aun así, no tenía relación con este TFG.

El TFG comenzó el 22 de enero de 2021 con una reunión entre los interesados del proyecto. Antes de concretar esa reunión, la directora del proyecto propuso 2 posibles TFG a realizar por el alumno. Finalmente, el alumno escogió el TFG que se redacta en esta memoria por los siguientes motivos:

- El lenguaje de programación es Java, y se dispone de mucha experiencia en el mismo.
- El proyecto trata temas de calidad del *software*, una rama de ingeniería del *software* que interesa al alumno.
- Se habían cursado prácticas en la empresa, por lo que ya se conocía la metodología de trabajo.
- El proyecto era interesante, novedoso y con cierta dificultad, lo cual generó una motivación extra de cara al desarrollo del mismo.

Antes de utilizar el producto final desarrollado en este TFG, la empresa utilizaba una tabla en formato CSV para obtener el árbol de llamadas de una clase. Para ello, se creaba un nuevo CSV por cada clase que se quería analizar y se introducían todos los datos a mano. A continuación, se muestra un ejemplo de una tabla creada por Beatriz Pérez, directora del proyecto por parte de INDABA.

Acceso	Método	LOC	Nivel	Llamado por
public	generarPreviewReport(String, ReinstatementTram, Expedient, boolean)	139	0	NADIE
private	obtenerDatosCabecera(String, ResourceBundle, ReinstatementTram, Expedient, boolean, boolean, boolean)	50	1	Llamada de 3 metodos
private	getTextoCabeceraBySubreport(String, ResourceBundle, boolean, boolean)	39	2	Nadie más
private	crearDireccionCabecera(Expedient, DatosCabecera)	14	2	Nadie más
private	montarNombreCabecera(DatosCabecera, NotificationRecipientPOJO)	47	3	Nadie más
private	rellenarFirmasSubreport(String, ReportLauncher, ResourceBundle, InputStream, InputStream)	16	1	Nadie más
private	obtenerDatosReport218InicioPreview(ReinstatementTram, ResourceBundle, DecimalFormat, DecimalFormat	43	1	Nadie más
private	textoFijoInicioRei(String, ResourceBundle, Datos218Inicio)	9	2	Llamada de otro +
private	verSiTieneAtrasosBloqueados(DecimalFormat, Datos218Inicio, ExpedientImproperCharges)	10	2	Llamada de otro +
private	**generarSolicitudFraccionamientoImporteFijo**	22	2	Llamada de otro +
private	obtenerRecExpIniReiResolRei(Datos218Inicio, DatosPlantillaResolNotifRei, Recursos)	18	3	Nadie más
private	obtenerFechaResolRevOrigenIniReiOResolRei(ReinstatementTram, Datos218Inicio, DatosPlantillaResolNoti	19	2	Llamada de 3 metodos
private	obtenerContenidoDocIniRei(Datos218Inicio, ExpedientImproperCharges, Recursos)	37	2	Llamada de otro +
private	iniReiGetDataInforEconYCompens(Datos218Inicio, ExpedientImproperCharges, Recursos, boolean, Long, b	25	3	Nadie más
private	obtenerCompenRgiOPcvSolCorrec(DatosPlantillaResolNotifRei, Datos218Inicio, Recursos, Request)	18	4	Llamada de otro +
private	obtenerPropuestaSolicitud(String, Request)	12	5	Llamada de otro +
private	reiGetCsrAnuladoPorRevCorrecRec(Recursos)	15	5	Llamada de otro +
private	obtenerCiCompensadosSol(List<ExpedientImproperCharges>)	7	6	Llamada de otro +
private	obtenerCompenRgiSolCorrec(DatosPlantillaResolNotifRei, Datos218Inicio, Recursos, List<Long>, List<Expe	31	5	Nadie más
private	inforEconGetCompenRgiOPcvOMixta(List<Long>, String)	16	6	Muchos
public	getLastExpReiTramNoAnulada(List<ExpedientReimbursementsGEP>)	21	6	Muchos
private	rellenarReiCompensacionesRgi(List<ExpedientReimbursementsGEP>, String)	12	6	Muchos
private	rellenarContenidoTablaCompenRgiIniOResol(DatosPlantillaResolNotifRei, Datos218Inicio, List<ExpedientRe	10	6	Nadie más
private	obtenerCompenPcvSolCorrec(DatosPlantillaResolNotifRei, Datos218Inicio, Recursos, List<Long>, List<Expe	31	5	Nadie más

Figura 2.2: Tabla creada de manualmente por la empresa para analizar el árbol de llamadas

La tabla contiene las siguientes columnas:

- **Acceso:** tipo de acceso del método (*public*, *private* o *protected*). Aporta información extra sobre el método.
- **Método:** nombre del método, junto a sus parámetros de entrada.
- **LOC:** líneas de código del método. Se calcula restando la posición final del método, con la cabecera del código. Por consiguiente, aparte de las líneas de código, también se incluyen los comentarios y las líneas vacías.
- **Nivel:** jerarquía de niveles del método. Estos niveles indican cuantas llamadas se han realizado con anterioridad para llegar a ese método. Por ejemplo, si el método A llama al método B, el método A es de nivel 0 mientras que el método B es de nivel 1.
- **Llamado por:** se indica si ha sido llamado por alguien, si ha sido llamado por muchos o si no ha sido llamado. No se indica por qué método ni clase es llamado.

Observando la tabla, se puede observar que contiene información limitada porque incluye muy poca información para poder obtener diferentes conclusiones para realizar *refactoring* sobre la clase. Además, la creación de esta tabla de manera manual conlleva otras desventajas que se han enumerado previamente.

3. CAPÍTULO

Planificación

En este capítulo se detalla la planificación del proyecto, describiendo el alcance, el tiempo, la dedicación, comunicación, información y los interesados. El objetivo de esta planificación es que al final del proyecto se satisfagan los objetivos definidos del proyecto, procurando que las incidencias tengan el menor impacto posible. Para ello es imprescindible planificar adecuadamente lo que se va a realizar y definir un plan de gestión de riesgos sufriendo el menor número de incidencias. Durante el desarrollo del proyecto, algunos de los puntos definidos en una primera instancia han sido modificados, y estas desviaciones están recogidas en el Capítulo 8.

3.1. Descripción del producto

El programa tiene como objetivo, ayudar en la toma de decisiones para la mejora de la calidad de un proyecto Java. Para ello, se analiza el proyecto Java y se obtiene como resultado el árbol de llamadas y diferentes métricas, que favorecen las modificaciones para el *refactoring* de código. Los gráficos de llamadas [Jász. et al., 2019] son gráficos dirigidos que representan las relaciones de control entre los métodos de un programa. Los nodos del gráfico denotan los métodos, mientras que una arista del *NodoA* al *NodoB* indica que un método *A* invoca al método *B*.

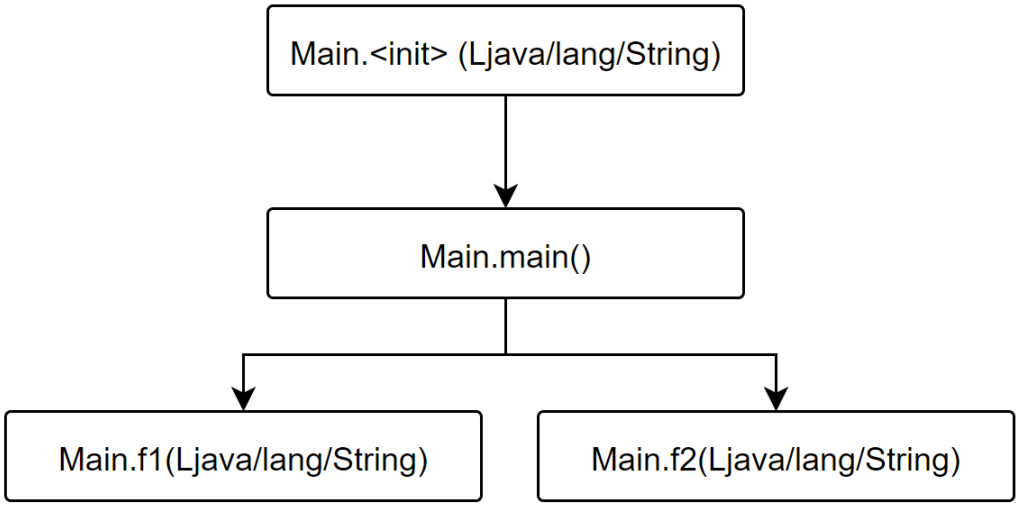

Figura 3.1: Ejemplo de gráfico de llamadas

En este proyecto, solo se guardan las llamadas que se realizan desde los métodos de una clase a otros métodos de esa misma clase. Aunque se puede realizar el análisis sobre las llamadas a métodos de otras clases, limitar el análisis a las llamadas internas ha sido un requisito definido por la empresa. Los gráficos de llamadas son la base en diferentes análisis de *software*:

- **Análisis del flujo de control:** *Control Flow Graph (CFG)* es la representación gráfica del flujo de control o computación durante la ejecución de un programa. Mientras que los diagramas de flujo tradicionales presentan los elementos con cuadrados, diamantes y hexágonos, los CFG se representan con nodos. Generalmente se utilizan en análisis estático, así como en aplicaciones de compiladores, ya que pueden representar con precisión el flujo dentro de un programa.

- ***Program slicing:*** [Mark, 1981] método para abstraer la información de los programas. Consiste en ”partir” un subconjunto del comportamiento de un programa, con el fin de que el corte reduzca el programa a una forma mínima que sigue produciendo ese comportamiento. El programa reducido, llamado ”*slice*”, es un programa independiente que garantiza fielmente el programa original dentro del dominio del subconjunto de comportamiento especificado.

- **Análisis de seguridad:** mantener la seguridad en los programas informáticos es primordial, y sobre todo cuando estos programas utilizan librerías externas. Como se comentará más en detalle en la memoria (véase el apartado 4.2.7), las librerías

que utilizan los proyectos pueden suponer un agujero de seguridad en el programa. Teniendo en cuenta este problema y otras ventajas (como la comprensión del programa), hace que el empleo de gráficos de llamadas sea muy positivo. Existen diferentes estudios que respaldan la mejora de la seguridad con el uso de gráfico de llamadas, como la detección de *malware* en Android [Feng et al., 2014] o la extracción de gráficos de ejecución en caja gris para la detección de anomalías [Gao et al., 2004].

- **Predicción de *bug*s**: existen diferentes mecanismos, como la predicción basada en pesos ([Turhan et al., 2008]) o la identificación de diferentes patrones siguiendo modelos ([Christodorescu and Jha, 2003], [Antal et al., 2021]), que permiten predecir qué *bug*s puede disponer el programa.

- **Compresión del programa:** permite conocer el diseño de un programa sin tener conocimiento previo.

- **Optimización del programa**

Una vez formado el gráfico, se obtienen una serie de métricas que aportan información sobre la calidad de la clase y métodos, y todo ello se exporta a ficheros para su lectura y análisis.

3.2. Gestión del Alcance

El proyecto tiene como meta principal optimizar el análisis de la calidad de diferentes proyectos para la empresa INDABA CONSULTORES S.L. Aun así, el alcance de este proyecto no está definido al completo, ya que muchas de las tareas están sujetas a cambios, por lo que se contempla la posibilidad de cambiar el alcance durante el proyecto. Esto se debe a que las tecnologías a utilizar no están bien definidas, ni tampoco se conoce con total exactitud las herramientas que están disponibles. Incluso, cabe la posibilidad de que exista una herramienta que cumpla con los requisitos definidos, por lo que no tendría sentido la planificación. Para evitar que ocurra ese riesgo, se va a definir un ciclo de vida que permita tener un control continuo sobre el proyecto, con la ayuda de reuniones periódicas con los interesados del proyecto. También será necesario realizar una buena búsqueda de información previa. Para el cumplimiento de todos los objetivos definidos en el Capítulo 3.2.1, es necesario definir correctamente el alcance de los mismos. Para ello, se descomponen las tareas del proyecto en paquetes de trabajo, y se definen sus requisi-

tos. Además, estas tareas deben cumplir unos estándares de calidad (3.5) y es necesario controlar los riesgos identificados (3.4) con sus planes de acción.

3.2.1. Objetivos

El objetivo principal de este proyecto es desarrollar una herramienta que permita generar un árbol de llamadas de un conjunto de clases. A pesar de este ser el objetivo principal, está acompañado de otros objetivos, que colaboran en la consecución del objetivo principal. El cumplimiento de los objetivos secundarios encarecería la calidad del TFG, pero no forman parte del objetivo principal del proyecto. Los objetivos principales que se pretenden alcanzar son los siguientes:

- **Analizar las herramientas existentes para refactoring de código:** realizar un estudio en aquellas herramientas que permitan obtener el árbol de llamadas de una clase, conocer sus fortalezas y deficiencias. De esta manera, también se asegura que la herramienta que se quiere crear no esté ya creada, por lo que el proyecto carecería de sentido.
- **Analizar herramientas existentes para obtener métricas:** Obtener métricas que sean útiles al *refactoring* (por ejemplo, LOCs, complejidad ciclomática, etc.), que permitan tomar decisiones respecto en los diferentes contextos.
- **Obtener llamadas entre clases y métodos:** teniendo como entrada una clase (fichero *.class*) o un conjunto de clases (fichero *.jar*), se obtendrán las relaciones clase-clase y método-método(s). Gracias a ello, se podrán observar las dependencias que tiene el código, y ayudará a mejorar el *refactoring* del proyecto.
- **Representación gráfica de las llamadas entre clases y métodos:** Cuando se obtengan las llamadas, se utilizarán esos datos para representarlos de manera gráfica, creando así el gráfico de llamadas. Con el uso de la interfaz gráfica, el usuario podrá visualizar los datos con mayor claridad, disminuyendo el tiempo en el que obtiene las conclusiones para su modificación.
- **Filtrado de las llamadas:** Cuando se disponga de la representación gráfica de las llamadas, se aplicará un filtrado para obtener un resultado diferente. Este filtrado se realizará modificando la estructura de la representación, o haciendo uso de las métricas del código analizado (LOC, número de líneas, nivel del método...). De esta manera, el usuario podrá obtener más conclusiones, que utilizará posteriormente para mejorar la calidad del código.

- **Utilizar la herramienta en entorno real:** Utilizar la herramienta en un proyecto real de la empresa INDABA CONSULTORES S.L., para obtener conclusiones de su aplicación, mejoras y trabajo a futuro. Realizando las correspondientes pruebas, se minimiza el riesgo de que en un futuro se encuentren errores, lo que conlleva un menor coste en el mantenimiento.

Objetivos secundarios

Una vez conseguidos los correspondientes objetivos principales, se procederá a conseguir los objetivos secundarios. Se definen como objetivos secundarios ya que su función es mejorar los resultados obtenidos con la realización de los objetivos principales.

- **Añadir más filtros:** Se añadirán más filtros que los definidos, incluso se estudiará la posibilidad de añadir herramientas externas que proporcionen más datos para el filtrado (por ejemplo: SonarQube [1]). Gracias a ello, se podrá mejorar la refactorización del código, mejorando de esa manera la calidad del código.

- **Mejorar la calidad de las interfaces gráficas:** mejorar la calidad de los datos mostrados de manera gráfica, para mejorar la usabilidad de la herramienta. De esta manera, los usuarios que no hayan utilizado la herramienta, podrán adaptarse a la misma en un tiempo menor.

Objetivos del alumno

Además de los objetivos que tiene el propio proyecto, el alumno también se plantea ciertos objetivos, con el fin de nutrirse de conocimiento durante el desarrollo del proyecto:

- **Mejorar el trabajo en equipo:** se realizarán muchas reuniones telemáticas que se realizarán con la empresa y la universidad a raíz de la envergadura y ambigüedad del proyecto. El objetivo es mejorar la realización de estas reuniones, con el fin de recopilar la mayor cantidad de información en cada una de las reuniones e intentando minimizar lo máximo posible la duración de las mismas. Para ello, se pretende mejorar la redacción de actas, la preparación previa de las reuniones y el manejo de las reuniones.

[1] SonarQube: https://www.sonarqube.org/

- **Mejorar la captura de requisitos:** como el proyecto dispone de un alcance ambiguo, es necesaria una buena captura de requisitos. Es importante adquirir experiencia en la captura de los mismos, para evitar posibles desviaciones futuras en otros proyectos. Este aspecto no se trabaja mucho durante el grado y casi todos los proyectos lo sufre, lo cual justifica su importancia.

- **Transición laboral:** adquirir la mayor cantidad de experiencia posible en el entorno laboral de la empresa, para que la transición laboral futura sea más sencilla. También, tener en cuenta los consejos que indique la directora del proyecto, ya sean referentes al proyecto como al entorno laboral.

3.2.2. Requisitos

En la fase inicial del proyecto, después de realizar varias reuniones, se redactan los requisitos del proyecto y son validados por la responsable de la empresa. Es necesario que estos requisitos estén bien definidos para que el proyecto sufra la menor cantidad de desviaciones posibles. Además, será útil para poder dar fin al proyecto en su fase final, ya que el proyecto se cierra con el cumplimiento mínimo de los requisitos, y es extensible con la definición de más requisitos.

Requisitos funcionales

Como la herramienta a implementar dispone de diferentes funciones, se han clasificado estos requisitos en diferentes grupos.

- **Análisis**

  - **IR-01:** El sistema recibe código compilado como parámetro de entrada y analiza el árbol de llamadas de cada clase.
  - **IR-02:** El sistema deberá mostrar el árbol de llamadas, clasificando cada método por su nivel, LOC, tipo de resultado, línea en la que se encuentra dentro de la clase y por qué otros métodos es llamado.
  - **IR-03:** El sistema deberá crear un reporte por cada clase que recibe por parámetro.

- **Visualización**

- **IR-04:** El sistema permitirá crear y modificar el diseño de la aplicación que recibe como parámetro de forma gráfica, y comprobará su viabilidad. Es decir, comprobará si el nuevo diseño se puede compilar sin tener problemas.
- **IR-05:** El sistema mostrará los datos que ha obtenido como resultado de manera gráfica, mediante distintos tipos de gráficos. Estos datos son los obtenidos en la fase de análisis.

- **Filtrado**
  - **IR-06:** El sistema permitirá filtrar las clases y directorios que obtiene como parámetro de entrada. (p. ej.: excluir clases)
  - **IR-07:** El sistema permitirá filtrar la representación visual de los gráficos con diferentes filtros. (p. ej.: clases abstractas)

- **Exportación**
  - **IR-08:** El sistema podrá exportar los análisis en diferentes formatos (por ejemplo, CSV).

Requisitos no funcionales

- **Rendimiento:**
  - **IR-09:** Las clases que el sistema tiene que analizar tendrán un tamaño máximo de 100.000 líneas.
  - **IR-10:** Las clases por proyecto que el sistema tiene que analizar serán un máximo de 2000.
  - **IR-11:** El sistema no podrá realizar un análisis superior a 20 minutos, y deberá mostrar un *feedback* al usuario.
  - **IR-12:** El sistema realizará análisis de ficheros Java de versiones superiores o iguales a la 1.6, compilados y sin compilar.

- **Usabilidad:**
  - **IR-13:** Cualquier informático de la empresa, debería ser capaz de utilizar la herramienta en menos de 15 minutos, con la ayuda de una guía.
  - **IR-14:** El sistema debe proporcionar mensajes de error que sean informativos y orientados a usuario final. Para asegurar de este requisito, se pedirá *feedback* a miembros de la empresa.

- **Fiabilidad:**
  - **IR-15:** El sistema deberá de disponer de unos casos de prueba que aseguren el correcto funcionamiento. Estos casos de prueba comprobarán la corrección de las métricas calculadas por la herramienta.
- **Coste:**
  - **IR-16:** Todas las herramientas que se integren en el sistema deberán ser de código abierto y gratuitas.
- **Modificabilidad:**
  - **IR-17:** Como se quiere extender con otras funcionalidades en el futuro, el código deberá de ser extensible/adaptable, de manera que sea fácil añadir funcionalidades.

Requisitos de la empresa

Por parte de la empresa también se han puesto unos requisitos, que afectan más al proceso que al producto.

- **IR-18:** Disponer de un control de las horas realizadas. De esta manera, la empresa podría realizar un seguimiento y poder asegurar el cumplimiento de las mismas.
- **IR-19:** Realizar reuniones periódicas para poder realizar un seguimiento y control más exhaustivo.
- **IR-20:** Redactar un manual que explique cómo utilizar la herramienta, y añadirlo al repositorio del código (GitHub[2]).

Exclusiones del proyecto

- **IE-01:** No se tiene que probar que el sistema analice todos los diferentes tipos de herramientas de software para la gestión y construcción de proyectos (Ant, Gradle...).
- **IE-02:** El sistema no tiene que analizar todos los archivos de un proyecto, solo los archivos *.class*.

[2]GitHub: https://github.com/

- **IE-04:** No se tienen que realizar casos de prueba para todos los métodos, pero sí de los algoritmos importantes del código (cálculo de LOC...).

- **IE-05:** El sistema no podrá tener dependencias con plataformas como GitHub o GitLab como forma para obtener los proyectos a analizar, ya que muchos proyectos de la empresa son propietarios. Es decir, los clientes de la empresa no quieren alojar su código en un repositorio de manera pública.

3.2.3. Fases del proyecto

Como ya se ha comentado en apartados anteriores, el proyecto dispone de un alcance ambiguo. Muchos de los requisitos presentados pueden presentar unas dependencias con módulos que no se pueden comprobar. Por ende, es necesario realizar un buen estudio de las herramientas para su futura implementación, por lo que la gestión del tiempo se realiza cuando se disponen de los conocimientos mínimos sobre los diferentes campos del proyecto. Por consiguiente, el ciclo de vida del proyecto es de tipo **Incremental**.

Gracias a este tipo de planificación del tiempo de las tareas del proyecto, se alcanza un crecimiento progresivo de las funcionalidades del proyecto. Consiste en realizar procesos en ciclos y al final se realiza una entrega al cliente del proyecto. De esta manera, la persona que trabaja en el proyecto tiene un *feedback* directo con el cliente. Gracias a ello, la directora del proyecto por parte de la empresa (Beatriz Pérez) y la directora del proyecto por parte de la UPV/EHU (Maider Azanza) podrán saber en todo momento el estado del proyecto, pudiendo indicar cambios que no lastren el proceso del proyecto. Además, también se definen posibles fases del proyecto en el caso de que el proyecto siguiese un ritmo adecuado, y se tuviera la posibilidad de realizarlo.

A la hora de planificar un proyecto con un ciclo de vida incremental, es difícil estimar las fechas de inicio y fin de cada una de las fases, pero se ha realizado una estimación aproximada gracias al diagrama de Gantt (Figura 3.5). Estas estimaciones tienen en cuenta las fechas claves del curso del alumno. A continuación se presentan las diferentes fases o incrementos del proyecto:

El objetivo de cada incremento es disponer de un prototipo que se pueda exponer a los interesados del proyecto en una reunión, que contiene unas características y funcionalidades que le diferencias del resto de prototipos. Aun así, estos prototipos están interconectados.

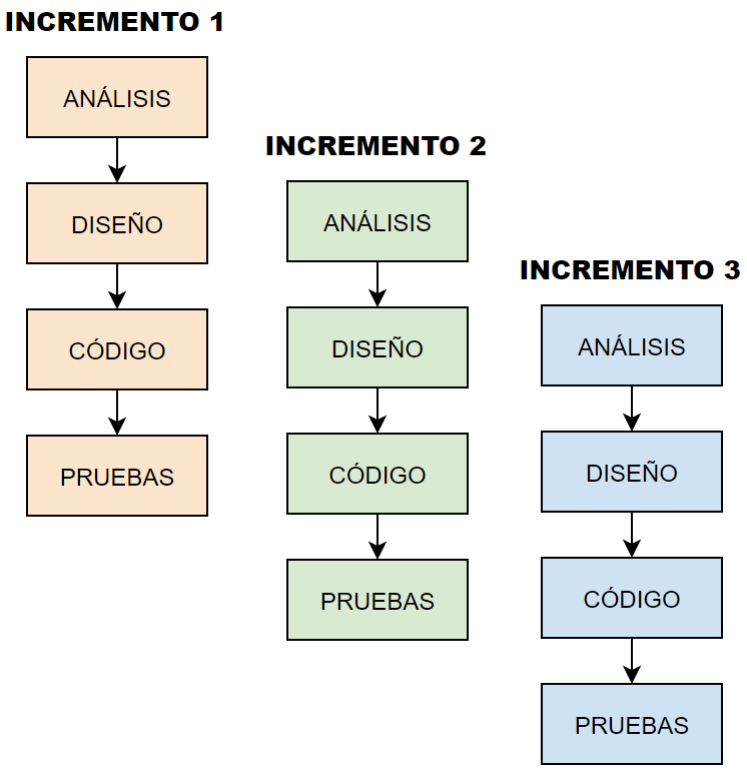

Figura 3.2: Ciclo de vida incremental del proyecto

- **I1-Requisitos mínimos (Prototipo 1):** analizar, diseñar e implementar los requisitos básicos que ha solicitado el cliente; es decir, obtener el árbol de llamadas. Para ello, el prototipo dispone de las siguientes funcionalidades:
  - Obtener el árbol de llamadas, de manera que los datos recopilados puedan ser utilizados por otros módulos del proyecto. Esto es importante para poder realizar diferentes métodos de exportación y poner añadir métricas a cada método del árbol.
  - Filtrar los datos obtenidos (p. ej.: por clase).
  - Exportar esos datos para visualizarlo en formato CSV u otros.
- **I2-Visualización (Prototipo 2):** una vez implementado el primer prototipo, el siguiente prototipo dispone una visualización más gráfica de los resultados obtenidos. De esta manera, se transforma el árbol de llamadas en un grafo de llamadas. Esta representación dispone de ventajas que se han enumerado en el apartado 3.1. Para realizar este prototipo, es necesario realizar un estudio de mercado sobre las diferentes herramientas disponibles y seleccionar la más adecuada. Se valora positivamente que esa interfaz gráfica sea interactiva.

- **I3-Métricas (Prototipo 3)**: cuando se obtenga una versión estable que permita visualizar los datos del código, se procederá a aplicar diferentes filtros. Estos filtros serán solicitados por el cliente, y se podrá extender teniendo en cuenta la gestión del tiempo del proyecto. Entre los filtros se encuentran:

  - Obtener la complejidad ciclomática.
  - Datos de las clases y métodos (p. ej.: paquete, número de líneas...)
  - Jerarquía de niveles de los métodos.

  En esta fase también se definirán pruebas más exhaustivas, que evalúen la robustez de la aplicación. Como se ha definido en los requisitos, es de suma importancia que el programa soporte proyectos con un gran número de clases y líneas de código. Una vez implementados todos los filtros y realizadas todas las pruebas, se decidirá si se quieren añadir más métricas o no. Se tiene en cuenta la posibilidad de incorporar métricas de SonarQube al *framework*.

A cada incremento se le ha asignado una fecha de finalización estimada. Para realizar estas estimaciones, se han tenido en cuenta las fechas de los exámenes del alumno y el resto de tareas del proyecto. La fecha fin estimada en cada fase han sido las siguientes:

- **I1-Requisitos mínimos:** 29/03/2021.
- **I2-Visualización:** 26/04/2021.
- **I3-Métricas** 31/05/2021.

En el caso de que se terminase con éxito todas las fases; es decir, satisfaciendo la calidad mínima establecida y de disponer de tiempo para continuar con el proyecto, se ha definido una funcionalidad más ambiciosa sin la necesidad de que sea finalizada. Una vez disponible el último prototipo, añadir una funcionalidad extra que permita refactorizar código en tiempo real. Es decir, en el caso de mover un método de una clase A a una nueva clase B, la herramienta debería de comprobar posibles errores de compilación y recalcular las métricas y árboles de llamada del proyecto. Una vez obtenido el diseño deseado, debería de concatenar todo el código y dar la posibilidad de exportarlo.

3.2.4. Descomposición de tareas

La Estructura de Descomposición de Trabajo (EDT) del proyecto se ha creado teniendo en cuenta el ciclo de vida incremental del proyecto. A través del mismo, se pretende dividir las diferentes tareas del proyecto en componentes más pequeños, con el objetivo de que sean más fáciles de entender y gestionar. Como se indicará más adelante en este documento, el proyecto se ha planificado teniendo en cuenta unos riesgos. En el caso de no identificar y definir bien estos riesgos, conllevaría a tener que redefinir el alcance del proyecto.

En la Figura 3.3 se encuentra el EDT. En ella, se pueden identificar 8 paquetes principales, que dividen las tareas principales del proyecto. Estos paquetes disponen de unas fechas de finalización estimadas, que se presentará más adelante en el diagrama de Gantt (véase el apartado 3.2.7) y también se explicarán las dependencias que pueden tener entre ellos (véase el apartado 3.2.6).

3.2.5. Paquetes del proyecto

En este apartado se muestra la descripción de los paquetes de cada una de las fases, ya que cada paquete se podría dividir en paquetes todavía más pequeños.

- **Administración del campo (AC):**
  - *Estudio de campo (EC):* en la fase inicial del proyecto, se debe realizar un estudio corto sobre el tema a tratar en el proyecto. Los temas a tratar son: *refactoring*, métricas y calidad del software. Para ello, se repasarán los conceptos sobre el *refactoring*, para saber que funcionalidades debería de tener el sistema y conocer el estado de los temas en la actualidad. El estudio será corto ya que se conocen con anterioridad los temas.
  - *Estudio de las herramientas (EH):*
    - *Alternativas de grafo de llamadas (AL):* estudio de las alternativas que se disponen para implementar en el proyecto la funcionalidad *grafo de llamadas*. Se deberá de realizar en la fase inicial del proyecto. También incluye el estudio de las herramientas de visualización de grafos y árboles. Es muy importante realizar un buen estudio, ya que en el caso de que exista una herramienta que realice las funciones que se necesitan, se reduciría mucho el tiempo del Prototipo 1 (P1).

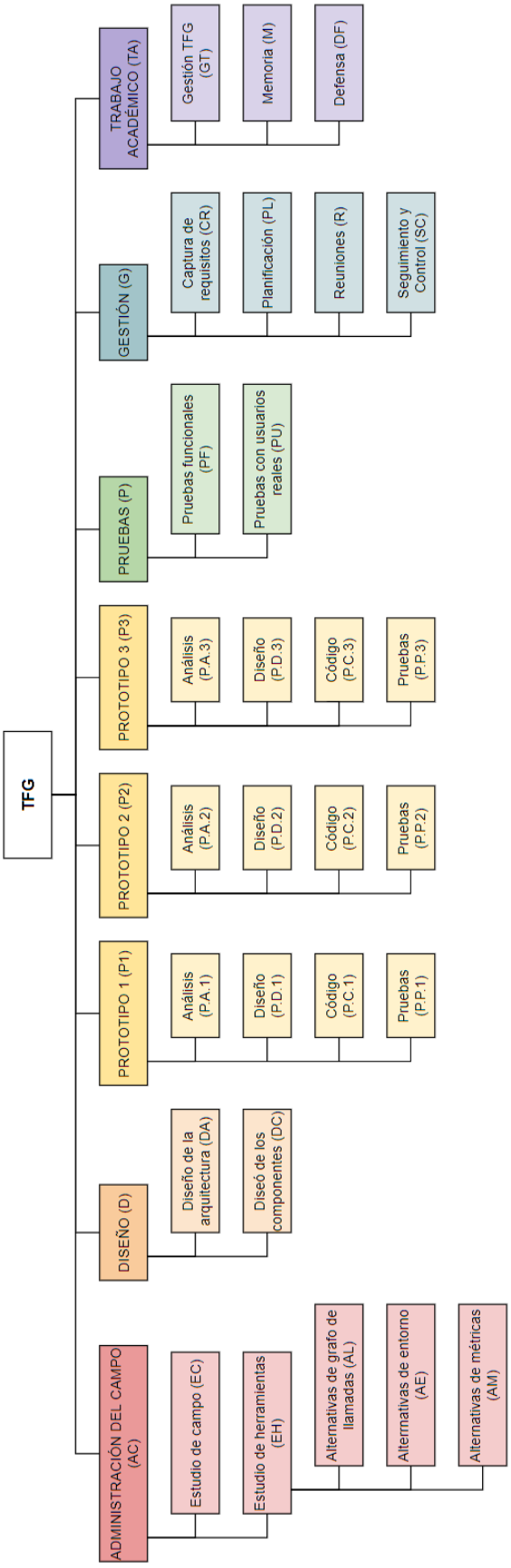

Figura 3.3: Diagrama EDT

- *Alternativas de métricas (AM):* estudio sobre las herramientas que son necesarias a tener en cuenta para refactorizar, y seleccionar cuáles se pueden añadir al sistema. Se deberá de realizar en la fase inicial del proyecto aunque estas métricas servirán como filtros para el árbol de llamadas (fase más avanzada del proyecto). En el caso de disponer de tiempo, se estudiarán herramientas de terceros que aporten esta información.
- *Alternativas de entornos (AE):* estudiar las diferentes alternativas en las que se pueda desplegar la herramienta: web, aplicación, *plug-in*... Es necesario tener en cuenta que debe ser un sistema centralizado, por lo que será necesario tener en cuenta la compatibilidad de los módulos.

- **Diseño (DI):**
  - *Diseño de la arquitectura (DA):* diseño necesario para la implementación de la lógica de negocio y las interfaces gráficas.
  - *Diseño de los componentes (DC):* diseño de las dependencias entre los componentes a implementar. Su objetivo es detectar interdependencias entre los mismos.
- **Prototipo (PX):** El proyecto dispone 3 prototipos distintos. Estos se definen en apartado 3.2.3, pero disponen los 3 la misma estructura.
  - *Análisis (P.A):* analizar los objetivos que se quieren implementar en este prototipo, y estudiar la viabilidad del mismo.
  - *Diseño (P.D):* realizar el diseño del prototipo, siguiendo las indicaciones previas realizadas en DA y DC.
  - *Código (P.C):* desarrollo de las funcionalidades del proyecto: gráfico de llamadas, análisis de métricas, interfaces gráficas...
  - *Pruebas (P.P):* pruebas que aseguren que lo implementado en el prototipo es correcto, y comprobar que no genera problemas con el resto de componentes desarrollados.
- **Pruebas (PU):**
  - *Pruebas funcionales (PF):* en la fase final del proyecto, se deberán de implementar y diseñar pruebas que aseguren las funcionalidades y el mantenimiento del programa. Estas pruebas deberán satisfacer todos los requisitos definidos. Por ello, se deberán de realizar pruebas del análisis del árbol de llamadas,

comprobar que todas las representaciones gráficas son correctas y comprobar si funcionan correctamente los filtros.

- *Pruebas con usuarios reales (PR):* tarea de la fase final del proyecto. Se realizará una reunión con la directora del proyecto, para comprobar que todos los requisitos definidos previamente se satisfacen. En el caso de que se cumplan los requisitos, lo idóneo sería realizar más pruebas con personal de la empresa, recopilando todo el *feedback* que puedan satisfacer. En el caso de que los cambios sean factibles, se implementarán, alargando las tareas de *Diseño* e *Implementación* del proyecto. También se comprobará la usabilidad del programa, realizando pruebas sobre las funcionalidades implementadas para comprobar que pueden ser manejables, cumpliendo los estándares de la Interacción Persona-Computador.

- **Gestión (G):**

  - *Captura de requisitos (CR):* al comienzo del proyecto, será necesario definir los requisitos del proyecto. Estos requisitos tienen que estar bien definidos para que no tengan ambigüedades y también validados por el responsable de la empresa.
  - *Planificación (PL):* planificación a realizar para mantener el control del proyecto. Para ello, es necesario realizar la gestión de diferentes apartados del proyecto: calidad, tiempo, alcance, interesados, comunicaciones y riesgos.
  - *Seguimiento y control (SC):* seguimiento y control de todos los puntos definidos en la planificación. Esta tarea perdurará en todo el proyecto, ya que es necesario mantener el control durante su ciclo de vida.
  - *Reuniones (R):* control de las reuniones a realizar durante el proyecto. Es necesario redactar todas las actas realizadas de manera adecuada, y planificar las siguientes reuniones para reducir su duración. Gracias a ello, se realizarán las reuniones justas y necesarias, y se evita la sobrecarga de trabajo a los integrantes del proyecto. Las reuniones son de suma importancia en este proyecto, ya que como el alcance no está definido al completo, se podrá realizar un seguimiento y control del mismo, acotando o extendiendo el mismo durante el desarrollo del proyecto. La frecuencia será de 2 semanas en las fases de implementación y de 1 semana en el resto de fases.

- **Trabajo académico (TA):**

- *Memoria (M):* redactar la memoria del proyecto. Será necesario redactar la misma durante el desarrollo del proyecto, para que quede anotado todos los conceptos con el máximo detalle. De esta manera, se evita el riesgo de no saber bien cómo documentar algún apartado del proyecto.
- *Defensa (DF):* preparación de la defensa del proyecto. Para ello, se preparará la presentación, con el objetivo de plasmar todo el trabajo realizado durante el proyecto. Además, también será necesario realizar varias pruebas previas a la presentación oficial.
- *Gestión TFG (GT):* gestionar los estándares de calidad del trabajo, con la información accesible a través de la plataforma de la EHU/UPV. Para ello, se revisará las rúbricas para la corrección del proyecto, sus normativas y se estudiarán algunos proyectos de años anteriores. En la fase final del proyecto, será necesario gestionar todos los tramites para la presentación del trabajo a través de la plataforma de ADDI.

3.2.6. Dependencias entre tareas

Las tareas del proyecto tienen unas dependencias entre sí, y resulta importante identificar esas dependencias para realizar una buena planificación. Es importante determinar aquellas tareas que puedan crear un problema en el desarrollo del proyecto; es decir, identificar el camino crítico. Al realizar este análisis, no se ha encontrado ninguna dependencia grave entre las tareas. A continuación, se presentan los paquetes que dependen de otros, por lo que su finalización implica el comienzo de otro/s paquete/s. Esta explicación se apoya en la Figura 3.4.

El proyecto comienza con el paquete **G.PL**, que indica el comienzo del proyecto ya que se trata de la planificación. Es primordial realizar en primer lugar la planificación, para poder acotar el alcance, gestionar bien el tiempo del proyecto, gestionar las comunicaciones y la calidad del proyecto. Algunas tareas de la planificación se tendrán que realizar cuando se tengan algunos conceptos sobre el proyecto, como los requisitos.

Una vez realizada una primera planificación, se continúa con la adquisición de conocimientos sobre los TFG de la UPV/EHU (**TA.GT)**, la captura de requisitos para definirlos en la memoria y todo ello con la ayuda de reuniones iterativas. Estas reuniones se realizan durante todo el proyecto. De esta manera, la planificación inicial se complementa y se da un comienzo del proyecto controlado y organizado.

En paralelo, se obtienen los conocimientos necesarios para poder diseñar e implementar el proyecto. En primer lugar, se realiza un estudio breve sobre los temas principales que envuelven el proyecto (**AC.EC**). Después, se estudian las diferentes alternativas para poder desplegar la implementación realizada (**AC.EH.AE**). Una vez adquiridos los conocimientos básicos, se buscan las alternativas para las funcionalidades primordiales del proyecto: gráfico de llamadas (**AC.EH.AL**) y métricas (**AC.EH.AM**). Sin embargo, no es necesario realizar un estudio exhaustivo de las métricas, ya que se realizará en una fase más avanzada del proyecto (finalización de P2.P.P2).

Una vez seleccionadas las diferentes alternativas de las herramientas, se realiza un diseño completo del programa (**D.DA** y **D.DC**) y se comienzan a implementar los diferentes prototipos. Cada prototipo finaliza con su paquete de pruebas (**P1.P.P1**, **P2.P.P2** y **P3.P.P3**) y da comienzo con su fase de análisis (**P1.P.A1**, **P2.P.A2** y **P3.P.A3**).

En el momento que se da por finalizado el prototipo P3, se comienza con la fase de pruebas globales (**P.PF** y **P.PU**). Cuando se terminan las pruebas, y todos los conceptos quedan reflejados en la memoria (**TA.M**), se puede preparar la defensa del proyecto. Con la finalización de este paquete, se da por concluido el proyecto.

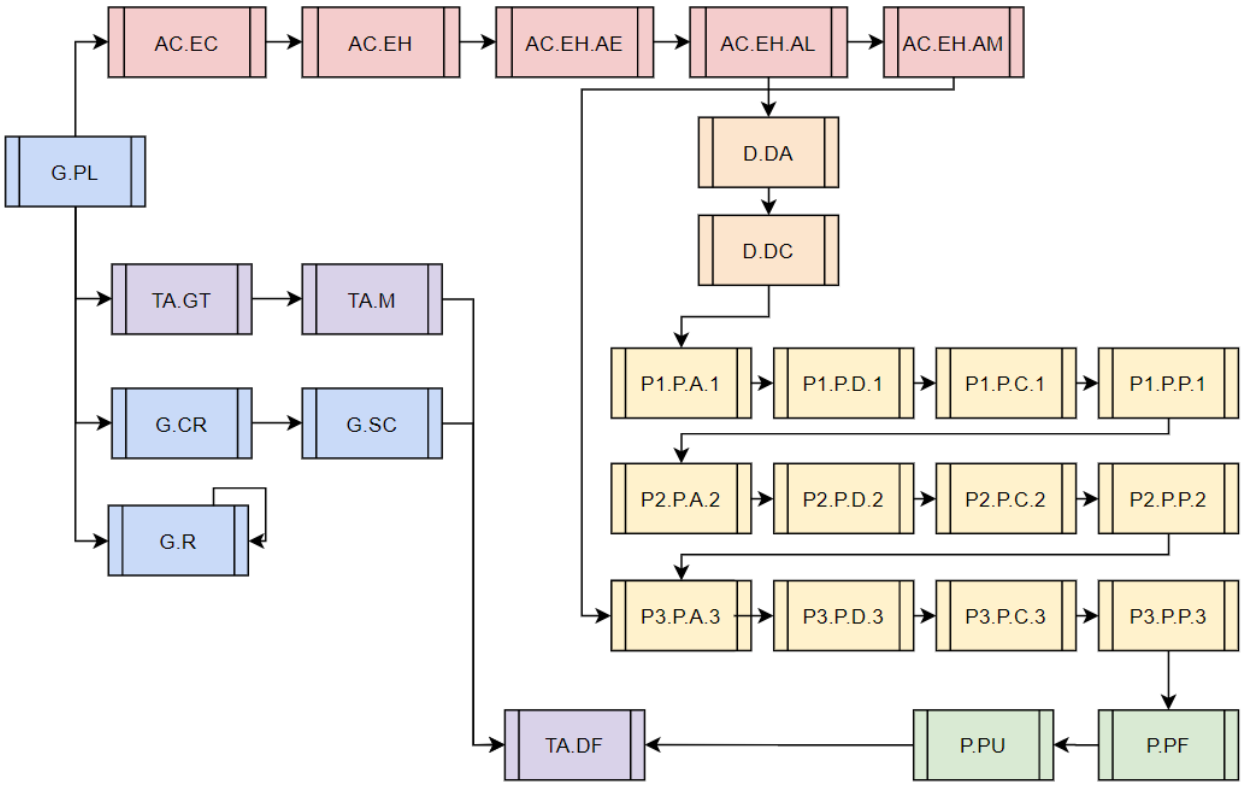

Figura 3.4: Dependencias entre tareas

3.2.7. Diagrama de Gantt

En este apartado se presenta el diagrama de Gantt, que servirá para tener una idea general de la duración de cada tarea del proyecto. El proyecto dio comienzo el día **18 de enero de 2021**, con una reunión con los interesados del proyecto. En esa reunión se definió el alcance y la duración estimada del proyecto.

En el diagrama se presentan todas las fases y paquetes presentados en el anterior apartado. Para la planificación, se ha tenido en consideración el trabajo académico de otras asignaturas, por lo que alguna fase es más larga que otra en las semanas más críticas del curso. También se han añadido los hitos más importantes del proyecto.

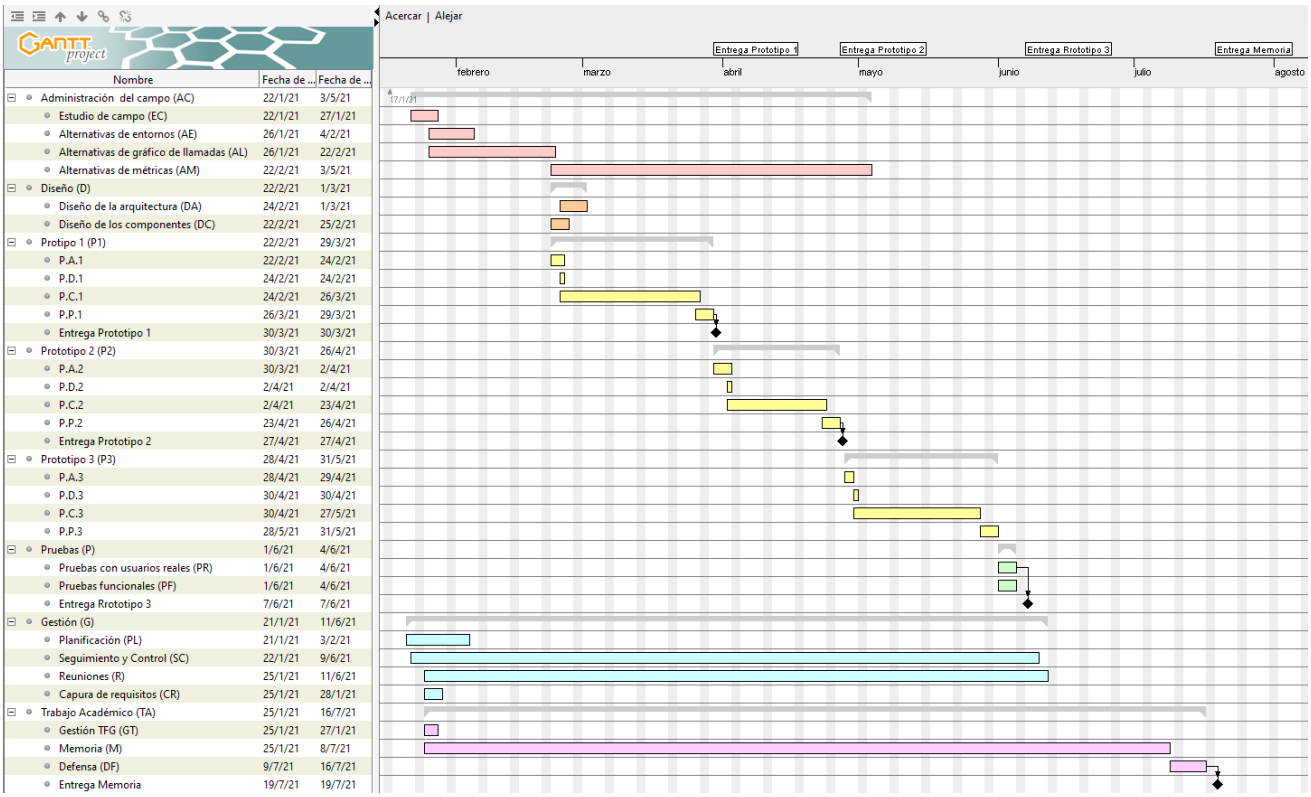

Figura 3.5: Diagrama de Gantt del proyecto

3.3. Gestión del Tiempo

Teniendo en cuenta el alcance definido, se gestiona el tiempo con el objetivo de cumplir con las fechas e hitos establecidos.

3.3.1. Tiempo estimado a cada tarea

PAQUETE DE TRABAJO	TAREA	DEDICACIÓN ESTIMADA
ADMINISTRACIÓN DEL CAMPO (AC)	Estudio de campo (EC)	4h
	Estudio de las herramientas (EH)	*45h*
	Alternativas de gráfico de llamadas (AL)	30h
	Alternativas de entornos (AE)	5h
	Alternativas de métricas (AM)	10h
	SUBTOTAL	**49h**
DISEÑO (D)	Diseño de la arquitectura (DA)	5h
	Diseño de los componentes (DC)	6h
	SUBTOTAL	**11h**
PROTOTIPO (P1)	Análisis (P.A.1)	10h
	Diseño (P.D.1)	8h
	Código (P.C.1)	70h
	Pruebas (P.P.1)	8h
	SUBTOTAL	**96**
PROTOTIPO (P2)	Análisis (P.A.2)	8h
	Diseño (P.D.2)	2h
	Código (P.C.2)	13h
	Pruebas (P.P.2)	2h
	SUBTOTAL	**25**
PROTOTIPO (P3)	Análisis (P.A.3)	4h
	Diseño (P.D.3)	1h
	Código (P.C.3)	11h
	Pruebas (P.P.3)	2h
	SUBTOTAL	**18**
PRUEBAS (P)	Pruebas funcionales (PF)	20h
	Pruebas con usuarios reales (PR)	3h
	SUBTOTAL	**23h**
GESTIÓN (G)	Captura de requisitos (CR)	5h
	Planificación (PL)	20h
	Seguimiento y Control (SC)	4h
	Reuniones	15h
	SUBTOTAL	**44h**
TRABAJO ACADÉMICO (TA)	Memoria (M)	70h
	Defensa (DF)	7h
	Gestión TFG (GT)	3h
	SUBTOTAL	**80h**
HORAS TOTALES		**346h**

Tabla 3.1: Horas estimadas para cada tarea del proyecto

3.4. Gestión de Riesgos

Uno de los aspectos más importantes del proyecto es la gestión de los riesgos, a causa de la situación actual del COVID-19 y por la incertidumbre que acarrea el proyecto. Para ello, con esta gestión se pretende disminuir la probabilidad y/o impacto de los riesgos.

R1-Compatibilidad del proyecto con el curso académico:

- *Descripción:* el proyecto da comienzo con el inicio del segundo cuatrimestre del cuarto curso del grado. Gran parte del desarrollo del proyecto se realizará junto a las asignaturas de la carrera. Durante los periodos de gran carga de trabajo de las asignaturas; como los exámenes o entregas de trabajos, pueden ocurrir aplazamientos de algunas tareas del proyecto.
- *Prevención*: realizar la planificación teniendo en consideración las fechas claves de las asignaturas del grado.
- *Plan de acción*: si conlleva una gran desviación, realizar una nueva planificación.

R2-Captura de requisitos

- *Descripción*: la especificación de los requisitos es de suma importancia, ya que define el alcance del proyecto, por lo que la planificación se realiza basándose en ese alcance. En el caso de que los requisitos estén mal definidos, cambiará el alcance del proyecto, por lo que a su vez también se tendrá que cambiar su planificación.
- *Prevención*: es importante que se definan de manera correcta y que estén verificados por el cliente.
- *Plan de acción*: se realizará una plantilla que plasme todos los requisitos proporcionados por el cliente. De esta manera, quedará constancia de los requisitos iniciales. En el caso de que se definan requisitos durante el proyecto, también se deberán de documentar.

R3-Uso de tecnologías desconocidas

- *Descripción*: al hacer uso de nuevas tecnologías que no se han usado con anterioridad, es necesario tener en cuenta alguna herramienta/tecnología puede modificar las estimaciones de tiempo del proyecto.

- *Prevención*: realizar un estudio previo de las herramientas a utilizar, y solicitar ayuda si se tiene la opción.

- *Plan de acción*: comunicar de la situación al grupo para obtener ayuda. Si supone un desvío crítico, realizar una nueva planificación.

R4-Riesgos individuales

- *Descripción*: es necesario tener en cuenta los riesgos que puede tener una persona (enfermedades, lesiones, problemas externos...), además agravado por la situación de la pandemia (el alumno padece asma, por lo que es un riesgo a considerar). Además, el alumno está pendiente de una operación, la cual tiene pospuesta porque se necesita de rehabilitación, lo cual imposibilitaría realizar este TFG.

- *Prevención*: cuidado de salud.

- *Plan de acción*: se asumen los riesgos de problemas personales. En el caso de padecer alguno, será necesario comunicarlo directamente la directora del proyecto y la representante de la empresa (es decir, a los integrantes del proyecto). Si las consecuencias son críticas, posponer la entrega del proyecto.

R5-Cambio de herramienta

- *Descripción*: una gran parte del tiempo del proyecto se dedicará a la búsqueda de información para la futura implementación. Una vez el proyecto avance en la fase de implementación, cabe la posibilidad de que se cambie las herramientas seleccionadas para la implementación del proyecto. Esto puede darse debido al estudio más profundo de las tecnologías, descubrimiento de tecnologías que se ciñen más a los requisitos...

- *Prevención*: realizar un buen análisis previo estudiando el funcionamiento de cada herramienta, y una buena captura de requisitos.

- *Plan de acción*: para mitigar este riesgo, será necesario realizar un estudio previo, comprobando el funcionamiento de cada herramienta. En el caso de no solucionar las dependencias, será necesario notificar la situación a los integrantes del proyecto, para poder encontrar una solución.

R6-Integración de los módulos

- *Descripción*: a pesar de que en el proyecto se quieran implementar muchas funcionalidades, todas ellas tienen que estar unificadas en un mismo programa. Para ello, se implementarán todos los módulos diferentes (JAR, API, fragmentos de código...) que tendrán que ser unificados. Esta integración puede ser muy costosa, ya que pueden ocurrir acoplamientos entre programas/librerías.
- *Prevención*: realizar un estudio previo, comprobando la integración de cada herramienta. Buscar diferentes ejemplos donde se integra cada una de las herramientas.
- *Plan de acción*: en el caso de no solucionar las dependencias, será necesario notificar la situación a los integrantes del proyecto, para poder encontrar una solución. En el caso de que no se encuentra solución, se tendrá que utilizar una nueva herramienta, lo cual conllevaría otro nuevo estudio de las alternativas. Este estudio se tendrá que definir en una segunda planificación.

R7-Versiones de las herramientas

- *Descripción*: al hacer uso de diferentes herramientas; como pueden ser librerías, aplicaciones, repositorios... puede ocurrir una colisión entre ellas o una actualización en el transcurso del proyecto.
- *Prevención*: realizar un estudio previo, comprobando la versión de cada herramienta.
- *Plan de acción*: en el caso de que actualicen las herramientas en el desarrollo del proyecto, será necesario notificar la situación a los integrantes del proyecto, para poder encontrar una solución.

R8-Planificación incorrecta:

- *Descripción*: el programa tendrá que disponer de muchas funcionales, las cuales se pueden incrementar durante el desarrollo del proyecto, con la definición de nuevos requisitos. Dichas nuevas funcionalidades modificarían el alcance del proyecto, por lo que conllevaría su correspondiente cambio en la planificación.
- *Prevención*: se realizarán reuniones con mucha frecuencia (1-2 semanas), de manera que se controlen las desviaciones y problemas que ocurran.

- *Plan de acción*: en el caso de tener que realizar una nueva planificación, será necesario redactarla en la memoria dejando constancia de la realizada anteriormente. Se tendrá en cuenta el alcance definido y el tiempo transcurrido en el proyecto, el cual no se podrá desviar mucho de las horas de trabajo totales estimadas.

R9-Pérdida del sistema de información

- *Descripción*: el proyecto está pensado que se desarrolle en diferentes escenarios: universidad, biblioteca y residencia del alumno. Por ello, será necesario que esa información esté en todo momento sincronizado, para no evitar retrasos en el proyecto. Además, esta característica aumenta la probabilidad de que la información se pierda, debido a errores humanos o del propio sistema.

- *Prevención*: realizar copias de seguridad del sistema de información. Mantener el sistema de información distribuido en diferentes dispositivos puede mitigar el riesgo en gran medida, pero no garantiza la seguridad por completo.

- *Plan de acción*: comunicar al resto de los interesados de la situación y realizar un plan de recuperación. En el caso de que la documentación y las actas se realizan de manera correcta, se reducirá el impacto de este riesgo, ya que se podrán consultar las decisiones tomadas previamente bien detalladas.

3.4.1. Escalabilidad, probabilidad e impacto

Los riesgos definidos en los anteriores apartados, pueden anidarse entre sí. Por ello, es necesario definir la gravedad de cada uno y el impacto que tendría en el proyecto. Esto se ha definido la Tabla 3.2, que contiene las siguientes apartados:

- *Riesgo:* identificador del Riesgo ”R”, que se han definido en el apartado 3.4.

- *Escala:* escala/impacto que tendría en el proyecto.

- *Probabilidad:* probabilidad de que el riesgo ocurra.

- *Tiempo:* estimación del tiempo añadido al proyecto si el riesgo ocurre.

- *Calidad:* ámbito del proyecto que sufriría el impacto.

* Riesgo y Escala se califican con los calificadores: “Muy Alta”, “Alta”, “Media”, “Baja”, “Muy Baja”, “Nulo”.

RIESGO	ESCALA	PROB.	+/- IMPACTO SOBRE LOS OBJETIVOS	
			TIEMPO	CALIDAD
R1	Media	Muy Alta	1-3 semanas	Ningún cambio en la funcionalidad
R2	Muy Alta	Media	1-4 semanas	Impacto significativo en el desarrollo
R3	Alta	Media	1-2 meses	Impacto significativo en el desarrollo
R4	Muy Alta	Muy Baja	Indefinido	Impacto mayor en el proyecto
R5	Alta	Baja	1 mes	Impacto significativo en el desarrollo
R6	Media	Media	1-3 semanas	Impacto en fase final
R7	Baja	Muy Baja	1 semana	Impacto menor en la implementación
R8	Media	Baja	1-3 semanas	Impacto medio en el desarrollo
R9	Muy alta	Media	1-4 semanas	Impacto medio en el desarrollo

Tabla 3.2: Probabilidad e impacto de los riesgos

3.5. Gestión de la Calidad

En este apartado se describe de la calidad tanto del proyecto como la de sus entregables. El objetivo es minimizar las variaciones futuras para que se cumplan los requisitos de los interesados (véase el apartado 3.7). También se evalúa la satisfacción del cliente y la mejora continua de la gestión de proyecto. Por ello, como resultado se obtiene un proyecto que cumple con todos los requisitos definidos.

Se han establecido los siguientes criterios de calidad y se ha dictaminado el método de calificación de los mismos:

- **Escalabilidad de los datos:** el programa deberá de gestionar y almacenar un gran número de datos, ya que los programas que se quieren analizar son de grandes dimensiones. Estos datos se describen en los requisitos funcionales (véase el apartado 3.2.2).

  - *Mala:* se pierden datos en la ejecución del programa.
  - *Buena:* se conservan los datos en la ejecución del programa.
  - *Excelente:* se conservan los datos en la ejecución del programa, y además se dispone de mecanismos de control.

- **Modificabilidad o Mantenibilidad:** El sistema debe ser modificable para cubrir, en un futuro, todas las necesidades de la empresa. Para ello, deberá tener un diseño acorde a los estándares de la Programación Orientada a Objetos. Para ello, será necesario diferenciar el programa en diferentes capas que tendrán que estar intercomunicados. De esta manera, se obtiene un aislamiento de los problemas de compilación y ayuda a la implementación de pruebas.

  - *Mala:* diseño pobre, carente de polimorfismos y estructuras abstractas.
  - *Buena:* diseño correcto, pero sería necesario realizar alguna modificación para poder añadir nuevos componentes.
  - *Excelente:* diseño que se puede adaptar a casi cualquier escenario, mejorando así el mantenimiento del mismo.

- **Usabilidad:** la interfaz deberá de ser simple, clara e intuitiva. Deberá de poseer un número de botones/opciones acordes a los requisitos definidos, para que los futuros usuarios no tengan problemas durante su uso. Además deberá de satisfacer los estándares de Interacción Persona-Computador. En el caso de disponer de tiempo, se realizará de una guía extensa para el uso de la misma, y en el caso de carecer del mismo, se indicarán apuntes en modo de ayuda de las opciones del programa con una guía más simplificada.

  - *Mala:* un usuario con la guía de uso no es capaz de utilizar la herramienta.
  - *Buena:* un usuario puede entender la herramienta haciendo uso de una guía.
  - *Excelente:* un usuario puede entender la herramienta sin el uso de una guía, y se ve capacitado de enseñar la herramienta a otro usuario.

- **Entregables:** los entregables deberán cumplir con los estándares básicos de un documento formal: fecha, objetivos, resultados, editores... Además, deberá quedar constancia y bien definida la fecha de entrega, para que no acarree retrasos en el proyecto.

  - *Mala:* las actas no disponen de toda la información de la reunión, creando conflictos con los conceptos ya realizados. Los documentos carecen de estructura y formato.
  - *Buena:* las actas recogen todos los acontecimientos de las reuniones y los documentos siguen los estándares básicos de calidad.

- *Excelente:* las actas además de recoger el estado de las reuniones, también sirven para crear una traza del estado del proyecto. De esta manera, con la lectura de las actas se puede saber el desarrollo completo del proyecto con todos sus detalles. Por otra parte, los documentos a entregar disponen de elementos extra que aportan más calidad a los mismos, como puede ser el uso de imágenes explicativas u otros recursos.

3.6. Gestión de Comunicaciones e Información

3.6.1. Sistema de información

Es de suma importancia mantener la información segura y disponible en todo momento. Para ello, se ha creado un sistema de información para poder evitar el riesgo R9 que se ha presentado en el apartado 3.4. El sistema de información es distribuido en dos dispositivos: ordenador de mesa y portátil. En los dos dispositivos se dispone del siguiente sistema de ficheros, que tiene como título **TFG**:

- **Recursos:** contiene los recursos multimedia que se han utilizado en el proyecto, como pueden ser el diagrama de Gantt o el EDT.
- **Repositorio:** repositorio que contiene la lógica de negocio del proyecto. Para evitar la perdida de información y disponer del sistema distribuido, el repositorio está alojado en *GitHub* de manera privada. De esta manera, se dispone de un control de versiones y se puede compartir el progreso de una manera sencilla.
- **Recovery:** contiene diferentes versiones del proyecto completo. El directorio dispone de ficheros comprimidos de LaTeX, que se irán almacenando de manera periódica cuando se avance en la redacción de la memoria. En el caso de que se quiera revisar algún apartado que se haya borrado o modificado, se podrá exportar el archivo comprimido a la plataforma de LaTeX (*Overleaf*[3]).
- **Otros:** contiene otros recursos que no se incluyen en las otras carpetas, como los ficheros necesarios para el diseño (.vpp), el diagrama de Gant...

Esta carpeta se sincroniza a través de *Google Drive*[4], por lo que en caso de ser necesario se puede compartir los contenidos de manera sencilla. Por otro lado, el documento del TFG

[3]Overleaf: https://es.overleaf.com/
[4]Google Drive: https://drive.google.com/drive/my-drive

se realiza en la plataforma *Overleaf*, por lo que también permite un control de versiones y brinda la posibilidad de compartir el proyecto a los interesados del proyecto.

3.6.2. Sistema de comunicación

La comunicación en este proyecto es de suma importancia, ya que es necesario tener un control con diferentes partes: alumno, universidad y empresa (véase el apartado 3.7). Por consiguiente, se han dictado una serie de pautas y herramientas para preservar este control. Además, se tiene como objetivo recolectar todas las comunicaciones realizadas para poder tener constancia de las mismas.

Herramientas/Vías de comunicación

- **Correo electrónico:** principal herramienta de comunicación. Se utiliza como método de comunicación de dudas, concretar fechas de reuniones y para el seguimiento y control del proyecto. Aunque el mensaje esté dirigido a una única persona, se adjunta al resto de los integrantes para que el grupo sea consciente del desarrollo del proyecto. También se utilizará el correo electrónico para comunicarse con personas que no estén dentro del grupo del proyecto.

- **Teléfono:** herramienta para dudas ocasionales. Generalmente se utiliza para hablar con la empresa y universidad: secretaría de la UPV/EHU, personal de la empresa de INDABA u otros.

- **Reuniones telemáticas:** como se define en el apartado 3.4, como consecuencia de la pandemia las reuniones tendrán que ser de manera telemática. Estas reuniones se realizan de manera periódica, debido al tipo de ciclo de vida del incremental del proyecto (véase el apartado 3.3). Por ello, el objetivo es que las reuniones se realizan con una preiodicidad de 1-2 semanas, para el correcto desarrollo del proyecto. Las herramientas para el mismo será un canal fijo de *Blackboard Collaborate*[5], y se utilizará Google Meet[6] en caso de problemas. Para poder tener una trazabilidad y control de las reuniones, todas las reuniones serán resumidas en su correspondiente Acta, que serán redactadas por el alumno (Véase el Anexo A).

- **Reuniones presenciales:** en el caso de que la pandemia concluya, se tiene programado realizar reuniones presenciales en una fase avanzada del proyecto. Estas

[5]BBC: https://www.blackboard.com/
[6]Google Meet: https://meet.google.com/

reuniones serían de un carácter más especial, como el cierre del proyecto o las pruebas del programa realizado con usuarios reales de la empresa. En el caso de que no se puedan llevar a cabo, se realizarán de manera telemática.

Reuniones

Todas las reuniones que se realicen se verán reflejadas en las actas. Todas estas actas seguirán un mismo esquema (Véase A) y se guardarán de manera segura. Las actas tienen 2 objetivos:

- Se tiene documentada la traza del proyecto. Esto puede ayudar a la hora de consultar alguna información y para la documentación final.
- La directora del proyecto por parte de la empresa podrá solicitar al finalizar el proyecto las actas, para adjuntar las mismas a la hora de presentar el proyecto (para justificar las horas).

En estas actas, se definen muchos puntos de gran ayuda: Orden del día, tareas asignadas... Uno de los apartados es el *Tipo de reunión*, que aporta una categoría a cada una de las reuniones:

- **Seguimiento y control:** reuniones que se harán de manera periódica para que las directoras conozcan el estado del proyecto.
- **Demostración de funcionalidades:** el alumno realizará una demostración del módulo implementado.
- **Intercambio de información:** reuniones para poner en común con cocimientos que otros integrantes no tengan.
- **Toma de decisiones:** cuando se presenten ciertas conclusiones, fin de una fase o acontecimientos inesperados, será necesario que las directoras del proyecto tomen alguna decisión.

3.7. Gestión de los Interesados

Es necesario identificar a los integrantes del proyecto, para analizar y documentar la información relevante relativa a sus intereses, interdependencias, influencia e impacto. En este proyecto, las personas interesadas son:

- **Alumno:** Iñaki García Noya
- **Directora (INDABA CONSULTORES S.L.):** Beatriz Pérez Lamancha
- **Co-Directora (UPV/EHU):** Maider Azanza Sesé

La **directora por parte de la UPV/EHU**; Maider Azanza, tiene como objetivo liderar al equipo responsable de alcanzar los objetivos del proyecto. En este caso, debe liderar al alumno a conseguir los objetivos definidos, entre ellos la gestión del proyecto, el control de calidad académica y la supervisión de la memoria. Para ello, se realizarán preguntas de índole académicas, para la mejora de la implementación, desarrollo y gestión del proyecto. También revisará la memoria cuando esté terminada.

La **directora por parte de INDABA**; Beatriz Pérez, tiene como objetivo liderar al equipo responsable de alcanzar los objetivos del proyecto. En este caso, debe liderar al alumno a conseguir los objetivos definidos. También, define los requisitos que tiene que cumplir el proyecto, ya que será la usuaria principal del programa resultante del TFG.

En un futuro se pueden añadir más integrantes al proyecto, ya que la herramienta tiene como objetivo ser utilizada por los empleados de la empresa. Por ende, esos empleados serán los **usuarios**, los cuales también podrán indicar algún requisito extra, o modificar alguna de las funcionalidades del programa.

4. CAPÍTULO

Estudio y análisis de las alternativas

Este capítulo hace referencia al análisis realizado durante el paquete de *Administración del campo (AC)*, después de terminar la *Planificación (PL)*. Este estudio de alternativas tiene como objetivo obtener una herramienta que cumpla los siguientes requisitos y no cumpla las siguientes exclusiones (véase el apartado 3.2.2): IR-01, IR-02, IR-03, IR-04, IR-05, IR-06, IR-07, IR-08, IR-09, IR-10, IR-11, IR-12, IR-13, IR-16, IE-02 e IE-05.

Para ello, se realizó un estudio y comparativa sobre los diferentes componentes necesarios para obtener un árbol de llamadas (4.1). Después, se identifican los posibles errores que pueden tener las herramientas a la hora de realizar el análisis, para tener en consideración esos fallos a la hora de elegir la herramienta. También, se presentan todas las herramientas que se analizaron y por cada herramienta se da una descripción y se identifican sus puntos positivos y negativos (4.3 y 4.4). Para finalizar, se presentan los argumentos para descartar cada una de las herramientas y también para probar otras, con un razonamiento final sobre la herramienta seleccionada (4.3.8 y 4.4).

4.1. Alternativas de análisis de código

A la hora de analizar el código de un proyecto, se disponen diferentes tipos de análisis. En el mercado existen herramientas; que, a pesar de disponer de las mismas funcionalidades, utilizan un enfoque diferente a la hora de realizar el análisis. Antes de seleccionar la herramienta que analice el gráfico de llamadas de un programa, se adquirieron los conceptos necesarios para tener criterio a la hora de elegir la herramienta adecuada. En primer lugar, se analizaron los diferentes análisis que dependen del estado del programa (4.1.1) y

después los análisis que dependen de la entrada que analizan (4.1.2).

4.1.1. Estático vs Dinámico

En este apartado se define el análisis estático y dinámico, junto a sus ventajas y desventajas. Para finalizar, se realiza una comparativa sobre los dos tipos de análisis [Ghahrai, 2017].

Análisis estático

El análisis estático es aquel que se realiza cuando el código no está en ejecución, lo que permite encontrar errores antes de ejecutar la aplicación. Para ello, recorre el código fuente en busca de código no-compilable. El ejemplo más básico es el compilador de un IDE, ya que busca errores sintácticos e incluso semánticos. También, se utiliza para revisar el código con el objetivo de garantizar que se utilicen ciertas normas y convenciones de codificación. A este proceso se le denomina revisión de código o *code review*.

VENTAJAS	DESVENTAJAS
Puede identificar puntos débiles del código en el sitio exacto	Si se lleva a cabo de forma manual, requiere mucho tiempo
Al trabajar con código fuente, favorece la comprensión de los desarrolladores	Las herramientas automatizadas producen falsos positivos y falsos negativos.
Permite una mayor rapidez en las modificaciones	No encuentra vulnerabilidades introducidas en el entorno de ejecución
Los puntos débiles se detectan durante el desarrollo, lo que reduce el tiempo de corrección	
Se detectan defectos únicos que no pueden o apenas pueden detectarse mediante pruebas dinámicas: código inalcanzable, uso de variables, funciones no invocadas o violaciones de valores límite	

Tabla 4.1: Ventajas y desventajas de análisis estático

Análisis dinámico

El análisis dinámico es aquel que se basa en la ejecución del sistema, a menudo utilizando otras herramientas. La práctica de análisis dinámico más común es la ejecución de pruebas unitarias contra el código para encontrar cualquier error.

VENTAJAS	DESVENTAJAS
Identifica vulnerabilidades en un entorno de ejecución	Las herramientas automatizadas producen falsos positivos y falsos negativos
Permite el análisis de programas que no se dispone del código fuente	Las herramientas automatizadas son tan buenas como las reglas que utilizan para escanear
Identifica vulnerabilidades que podrían haber sido falsos negativos en el análisis de código estático	Es más difícil rastrear la vulnerabilidad hasta la ubicación exacta en el código, por lo que se tarda más en solucionar el problema

Tabla 4.2: Ventajas y desventajas de análisis dinámico

4.1.2. Código fuente vs *bytecode*

En este apartado se define el código fuente y *bytecode*, junto a sus ventajas y desventajas. Para finalizar, se realiza una comparativa sobre los dos tipos de código [Between.Com, 2018].

Código fuente

El código fuente se refiere al código de alto nivel o código ensamblador que escribe un programador. El código fuente es fácil de leer y modificar. Lo escribe el programador utilizando cualquier lenguaje de alto nivel o lenguaje intermedio que sea legible para el ser humano (p. ej.: Java). Para escribirlo, se puede hacer uso de editores de texto o de IDEs (*Integrated Development Environments*).

A continuación, se presenta un ejemplo de un método Java en código fuente:

Código 4.1: Ejemplo código fuente de Java

```
public static void main(String[] args) {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

Código compilado o *bytecode*

Al convertir el lenguaje de programación de código fuente a código máquina, algunos lenguajes de programación convierten el código fuente en un código intermedio conocido como *bytecode*. Java es uno de los principales lenguajes de programación que utiliza el

bytecode. Si se convierte el código fuente del anterior apartado, el resultado en *bytecode* sería el siguiente fragmento de código (ejemplo obtenido de [Anouti, 2018]):

Código 4.2: Ejemplo código compilado de Java

```
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: (0x0009) ACC_PUBLIC, ACC_STATIC
Code:
stack=2, locals=4, args_size=1
0: iconst_1
1: istore_1
2: iconst_2
3: istore_2
4: iload_1
5: iload_2
6: iadd
7: istore_3
8: return
```

El proceso de conversión del código fuente a *bytecode* es el siguiente.

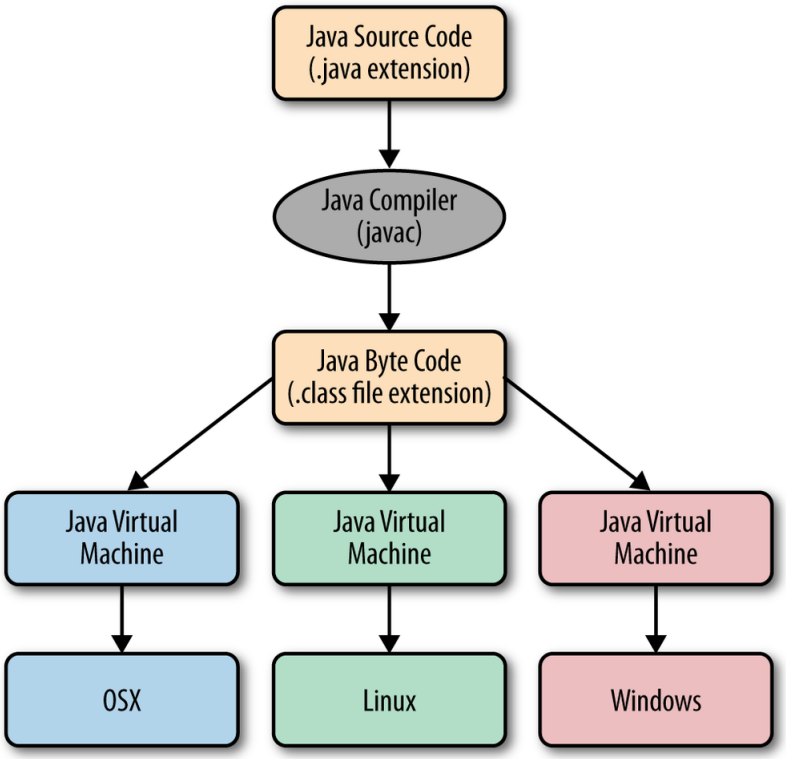

Figura 4.1: Ejecución de un programa Java [Oreilly, 2021]

Cuando se ejecuta el programa Java, el compilador convierte el código fuente en un *bytecode* Java. Luego, la JVM (*Java Virtual Machine*) convierte el código de *bytes* en código máquina, para que después el código máquina sea ejecutado por el sistema operativo. En un entorno Java, existe una máquina virtual llamada **JVM** [Lindholm et al., 2014], que tiene como objetivo ejecutar programas Java. JVM es una máquina de computación abstracta independiente del *hardware* y del sistema operativo, con la capacidad de proteger a los usuarios de programas maliciosos. La máquina virtual no tiene el conocimiento del lenguaje de programación Java, solo conoce el formato binario, el formato del fichero *class*. El fichero *class* contiene instrucciones de la máquina virtual Java (o *bytecode*) y una tabla de símbolos, así como información auxiliar. Define con precisión la representación de una clase o interfaz, incluyendo detalles como el orden de los *bytes*, que podrían darse por supuestos en un formato de fichero específico para objetos.

JVM solo entiende un grupo concreto de instrucciones, que son las siguientes:

- Carga y almacenamiento.
- Aritméticas.
- Conversión de tipos.
- Creación de objetos y su manipulación.
- Gestión de la pila (*push* y *pop*).
- Saltos.
- Llamada a métodos y salida de estos.
- Generación de excepciones.

Las máquinas virtuales Java disponen de una estructura concreta para poder completar el proceso completo para compilar el código [JVM, 2020]:

- *ClassLoader:* es el componente de la arquitectura JVM que carga las clases en memoria. Realiza 3 funciones: inicializar, cargar y enlazar.
- *Memory Area:*
  - *Method Area:* almacena estructuras de las clases.

- *Heap:* asignación de los objetos durante el tiempo de ejecución.
- *Stacks:* almacena los resultados temporales y las variables locales.
- *PC Registers:* almacena la dirección de la instrucción que se está ejecutando.
- *Native Method Stacks:* incluye todos los métodos nativos que son necesarios en cualquier aplicación.

- *Execution Engine:*
  - *Interpreter:* lee el flujo de *bytecode* y luego ejecuta las instrucciones.
  - *JIT Compiler:* Mejora el rendimiento. El JIT compila al mismo tiempo partes *bytecode* con funcionalidad similar y reduce el tiempo de compilación.
  - *Garbage Collector*.
- *Native Method interface:* facilita la comunicación entre diferentes aplicaciones escritas en diferentes lenguajes.

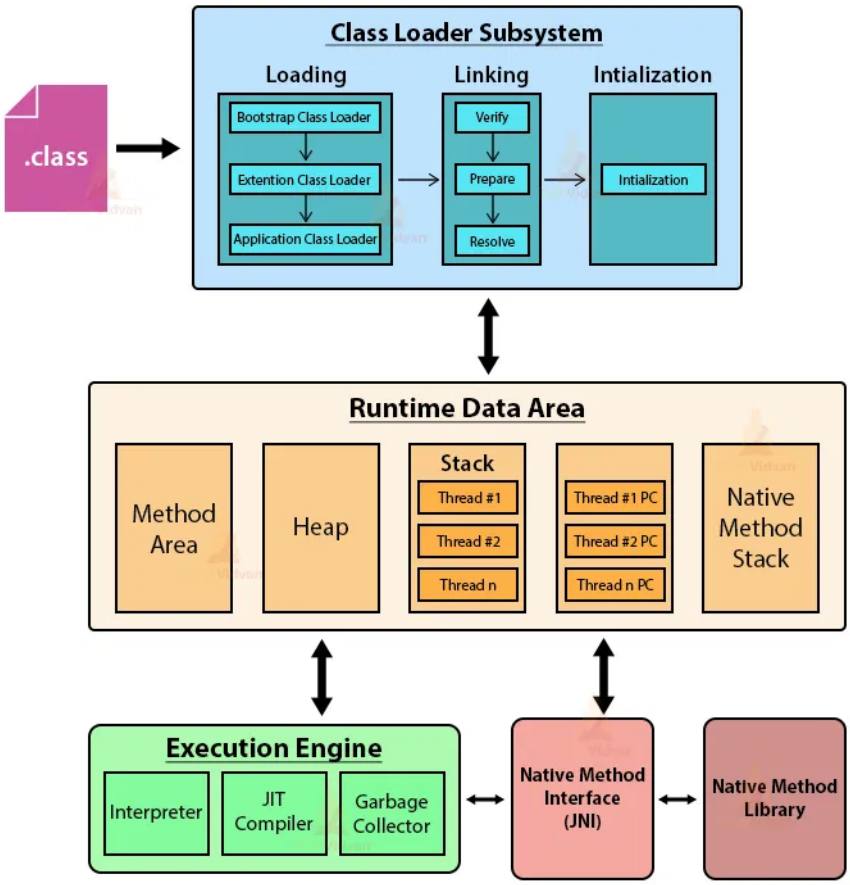

Figura 4.2: Estructura de JVM (*Java Virtual Machine*) [Javatutorial.net, 2017]

Comparación

Para finalizar, en la siguiente tabla se agrupan las diferencias principales entre los dos tipos de código, resumiendo toda la información presentada anteriormente:

	Código fuente	***Bytecode***
Comprensibilidad	Legible por una persona	Legible por una máquina virtual
Generación	Persona/programador	Compilador
Formato	Texto plano (generalmente en inglés), puede contener comentarios	Códigos numéricos, constantes y referencias que codifican el resultado del análisis sintáctico y semántico
Ejecución	No se puede ejecutar directamente en la máquina	Ejecutado por la máquina virtual
Velocidad de ejecución	Menor que la de *bytecode*	Mayor que la de código fuente
Rendimiento	Menor que el *bytecode*	Mayor que la de código fuente, ya que se comunica directamente con el lenguaje máquina
Portabilidad	Menor	Mayor

Tabla 4.3: Comparación entre *bytecode* y código fuente

Toda esta información se tuvo en cuenta a la hora de realizar la elección de las herramientas de la aplicación a desarrollar. Ninguna de las dos opciones es mejor que otra, simplemente cada una tiene sus ventajas y desventajas, y es necesario encontrar la herramienta que cumpla con los requisitos de la empresa.

4.2. Gráfico de llamadas

Durante años, muchos desarrolladores han creado diversas herramientas para obtener el gráfico de llamadas de un código. Por una parte, esto se debe a que existen diferentes análisis como se ha presentado en los apartados anteriores. Pero por otra parte, existen diferentes mecanismos para modelar las estructuras y el flujo de datos. Un gráfico de llamadas es preciso si contiene exactamente aquellos métodos y aristas de llamada que podrían ser ejecutados durante una ejecución real del programa [Jász. et al., 2019]. Aun así, se puede observar que muchas herramientas no obtienen el mismo número de nodos ni de aristas.

En el caso de los lenguajes orientados a objetos, muchas veces la llamada de un código depende del comportamiento del entorno de ejecución del programa, por lo que en el caso

de utilizar análisis estático tiene que realizar suposiciones sobre los métodos que pueden ser llamados. Para ello, existen diferentes tipos de algoritmos que disponen de muchos años de desarrollo y pruebas, para obtener las llamadas de los métodos. Estos son algunos ejemplos de algoritmos que se utilizan para la construcción de gráficos de llamadas de lenguajes orientados a objetos:

- *Class HierClass Hierarchy Analysis* (CHA).
- *Rapid Type Analysis* (RTA).
- *Hybrid Type Analysis* (XTA).
- *Variable Type Analysis* (VTA).

Incluso, existen otros factores que influyen en la creación de la estructura del gráfico de llamadas. A continuación, se resumen los elementos del lenguaje que se analizan de manera diferente, por lo que generan diferentes gráficos de llamadas. No se incluyen las diferencias de todas las herramientas, solo las más importantes. Es importante tener en cuenta esto a la hora de seleccionar la herramienta, ya que se requiere que el análisis esté libre de errores, en la medida de lo posible.

4.2.1. Formatos diferentes

Muchas de las herramientas pueden obtener el mismo número de nodos y aristas, pero pueden ser diferentes entre sí, ya que pueden disponer de diferentes formatos para nombrar los diferentes componentes. Esto ocurre ya que no existe un estándar para calificar los distintos elementos en el gráfico de llamadas. Por ejemplo en Java, los métodos se pueden distinguir por nombres totalmente cualificados que incluyen el nombre del paquete, el nombre de la clase nombre de la clase, el nombre del método y la lista de los tipos de los parámetros. Por otro lado, también existen diferentes formatos a la hora de nombrar las clases y métodos anónimos, así como las anotaciones de expresiones *lambda*. En el caso de que se utilicen diferentes formatos, será necesario modificar dicho formato durante la ejecución del programa.

En la Figura 4.3, se puede observar como analizando el mismo código como parámetro de entrada el análisis puede mostrar resultados diferentes. En esencia, son el mismo resultado pero siguiendo un formato diferente.

El código que se analiza es el siguiente:

Código 4.3: Ejemplo que se puede representar con diferentes formatos de gráfos de llamadas

```
class Test {
    Integer i;
    public Test(){}
    public Test(String s) {}
    {
        i = new Integer(89);
    }
}
```

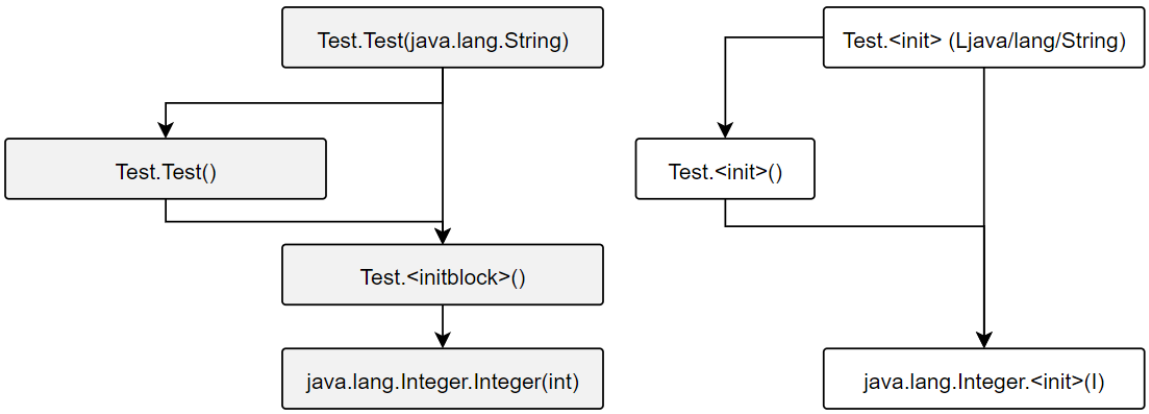

Figura 4.3: Diferentes formatos para representar llamadas en un gráfico

4.2.2. Métodos de inicialización

La mayoría de herramientas representan los métodos constructores de las clases, pero todas ellos detectan y conectan los constructores por defecto generados incluso sin la instanciación de un objeto. En el caso de los gráficos de llamada basados en AST (se profundiza en detalle en el apartado 4.3.3), los bloques inicializadores y los constructores tienen diferentes nodos en el gráfico de llamadas. Las herramientas que analizan *bytecode* representan una referencia a la clase externa como un parámetro adicional en la lista de parámetros, mientras que las herramientas basadas en el código fuente pasan por alto este parámetro, ya que no está presente en el código real.

Por otra parte, todas las herramientas representan los bloques inicializadores estáticos; sin embargo, cada uno con diferentes detalles y llamadas.

4.2.3. Poliformismo

El polimorfismo [Oracle, 2021] en Java se da cuando una clase puede disponer de diferentes estados, siendo uno de los aspectos más importante de un lenguaje orientado a objetos. Las subclases de una clase pueden definir sus propios comportamientos únicos y, sin embargo, compartir algunas de las funcionalidades de la clase madre. Esto puede generar problemas en el gráfico de llamadas, ya que los analizadores de código estático pueden ser incapaces de decidir si una referencia a un objeto es de su tipo declarado o de otro subtipo declarado. Por consiguiente, muchas herramientas solo enlazan el método padre en el gráfico.

Este problema se puede solucionar utilizando alguno de los algoritmos enumerados antes. Por ello, es importante utilizar un algoritmo que solucione este problema, para evitar errores en proyectos con una alta complejidad en el diseño de clases.

4.2.4. Elementos de código fuente anónimos

Las clases y métodos anónimos [Oracle, 2021] permiten tener un código más conciso, ya que declaran e instancian una clase al mismo tiempo. Son como las clases locales, salvo que no tienen nombre. Los métodos y clases anónimos de métodos pueden generar problemas en su formato, ya que no existe ningún estándar para definirlos. Por ello, es importante comprobar primero que la herramienta representa estos elementos y en el caso de que los represente tener en cuenta el formato que usa. También, es importante conocer cuando valor se le quiere dar a estos fragmentos de código, ya que pueden no ser muy representativos a la hora de realizar un análisis del código. Es decir, su existencia o su inexistencia pueden no variar el análisis realizado.

4.2.5. Java 8

Java 8 [Oracle, 2021] introdujo muchos conceptos novedosos en el lenguaje orientado a objetos, como las interfaces funcionales (clases que contiene un solo método abstracto). Las expresiones *lambda* y las referencias a métodos, que también son nuevas características de Java 8, pueden usarse para hacer referencia a interfaces funcionales. Estas no se pueden considerar como métodos, por lo que a la hora de representarlos como nodos en el gráfico es una tarea engorrosa. Por ello, algunas herramientas en lugar de tratarlo como nodos, las representan con la interfaz que implementan.

Es importante tener esto en consideración, ya que Java 8 fue lanzado en el año 2014, por

lo que muchos proyectos que estén constantemente actualizados probablemente utilizan las nuevas características.

4.2.6. Llamadas a métodos dinámicos

El mecanismo de manejo de métodos de Java hacen posible determinar el fin de la invocación de un método dinámico durante el tiempo de ejecución. Esto hace que las herramientas de análisis estático, generen nodos adicionales en el gráfico de llamadas. Por otro lado, las herramientas de análisis dinámico sí que controlan bien este comportamiento.

4.2.7. Gestión de las llamadas a librerías

Incluir las llamadas a las clases que contienen las librerías del programa no solo puede hacer que el gráfico sea más extenso, sino que también puede consumir muchos recursos del *hardware*. Sin embargo la exclusión de clases de la librería puede causar inexactitudes cuando se implementan interfaces de librerías o se heredan de las clases de la biblioteca. Por consiguiente, el análisis de las clases de las librerías puede incluir métodos privados e inaccesibles.

A raíz del auge en el mercado de programas o librerías *open-source*, muchos programas disponen de dependencias alojadas en sistemas de código centralizado como *Maven* o *npm*. Muchos de los incidentes que sufren los programas provienen una de estas librerías que tienen fallos. Además, la inclusión de código arbitrario de un repositorio en línea conlleva implicaciones de confianza y seguridad [Hejderup et al., 2018]. ¿Cómo puede un desarrollador asegurar que el código que importa no tiene fallos? ¿Cómo se puede saber cuando un problema de seguridad descubierto en una dependencia transitiva requiere una actualización? ¿Cómo se puede evaluar el impacto directo o transitivo del mantenimiento de una librería?.

Como la fuente de los fallos es desconocido, es difícil de evaluar la gravedad, el impacto y la propagación de los fallos en las redes de dependencia. Aunque los verificadores de dependencias se están adaptando como contramedida solo proporcionan información indicativa. Otra de las soluciones es crear gráficos de llamadas de redes de dependencias, cuyos nodos representan librerías o versiones de librerías.

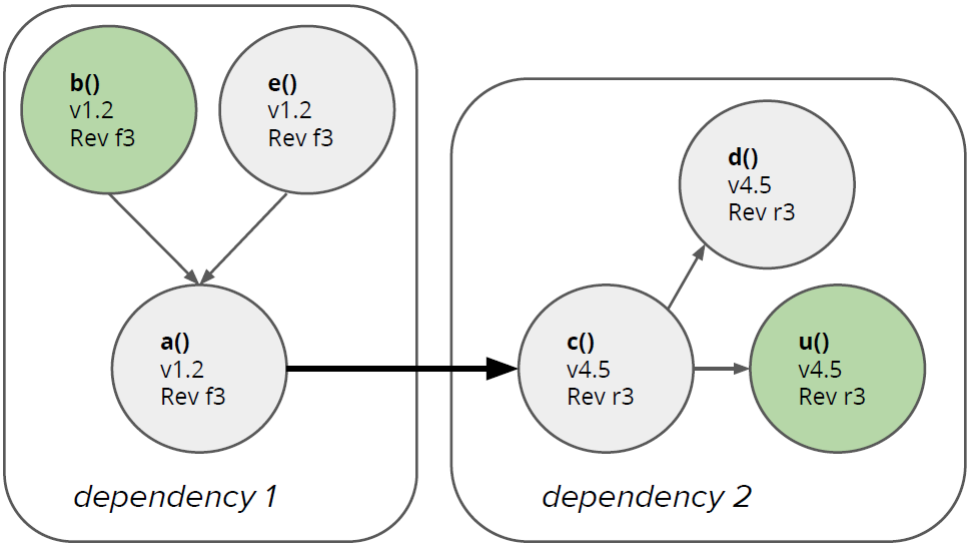

Figura 4.4: Gráficos de llamadas de redes de dependencias [Hejderup et al., 2018]

También es importante recalcar que estas librerías no siempre se encuentran dentro del código, por lo que muchas veces no se pueden analizar. Por ejemplo si se está analizando un JAR; es decir *bytecode*, si en el JAR no están almacenadas las librerías no se podrá obtener información de las mismas.

4.3. Alternativas de gráficos de llamadas

Tomando en consideración la información presentada en este capítulo, se realizó un estudio sobre las diferentes alternativas de las herramientas para el análisis de gráfico de llamadas sobre código Java. Para realizar esta búsqueda, la directora Beatriz Pérez especificó un requisito primordial: la herramienta tenía que ser *open source*. Esto se debe a que en el caso de que la herramienta presentase algún error, se pudiese modificar fácilmente sin tener que buscar una nueva herramienta en el desarrollo. Además, era preferible que la herramienta estuviese lo más actualizada posible, ya que eso indica que una herramienta es usada por otros desarrolladores. También, se tuvo en consideración si la herramienta dispone de la posibilidad de acoplarse al IDE Eclipse, ya que la empresa INDABA hace uso de ese IDE en la mayoría de sus proyectos.

A destacar, que la mayoría de IDEs proporcionan maneras de encontrar para un método sus llamadas (denominado *callers*) y también por quién es llamado (denominado *callees*). Sin embargo, no proporcionan el gráfico de llamadas entero. También disponen de herramientas externas que permiten generar el gráfico de manera manual, siguiendo las

indicaciones que introduce el usuario. En la siguiente Figura se puede ver un ejemplo[1].

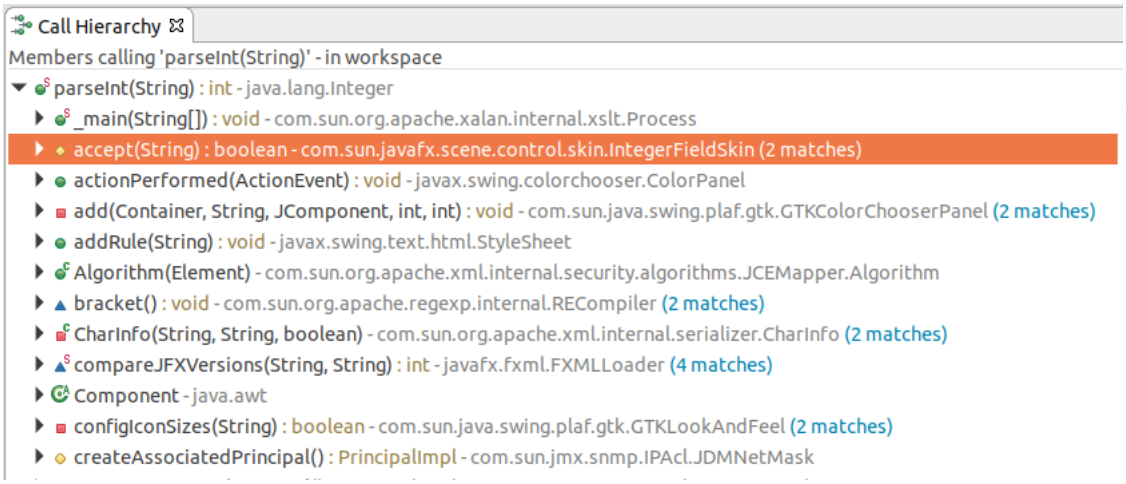

Figura 4.5: Ejemplo de jerarquía de métodos de Eclipse

En un principio, la búsqueda de información no fue muy sencilla, a raíz de la escasez de herramientas que tengan la funcionalidad de crear gráficos de llamadas. Además, cuando se encontraba una herramienta, no se disponía de información sobre su rendimiento ni su índice de errores. Para obtener esa información, se tendría que realizar una implementación de prueba con las herramientas más interesantes. Todo esto, podría conllevar retrasos como indica el riesgo R3 (véase el apartado 3.4). Durante la captura de información, se encontró un informe redactado por la Universidad de Szeged (Hungría), que realizó un estudio sobre las diferentes herramientas del mercado [Jász. et al., 2019] y otro realizado anteriormente por alguno de sus autores [Pengo and Ságodi, 2019]. Gracias a este estudio, se obtuvo información valiosa que se utilizó para seleccionar la herramienta adecuada. Como ya se ha dicho, no se dispone de mucha información sobre estas herramientas, por lo que estas pueden tener ventajas y desventajas que no se indican en este análisis.

Como muchas de las herramientas están alojadas en GitHub, se utilizó como criterio la popularidad del repositorio. Este criterio es un criterio más a tener en cuenta, pero en ningún momento se ha utilizado como criterio principal. Esto se debe a que algunas de las herramientas analizadas, han sido desarrolladas durante años sin que estuviesen alojadas en su respectivo repositorio. GitHub proporciona esta información ya que permite guardar y apoyar los repositorios con diferentes mecanismos:

- *Watch:* permite guardar el repositorio para revisarlo más tarde.

[1]Imagen: https://xenovation.com/

- *Star:* el usuario quiere resaltar o destacar ese repositorio.
- *Fork:* acción que consiste continuar con la implementación del proyecto en un repositorio propio. Esto puede ser para solucionar ciertos fallos o para implementar funcionalidades que no se disponen, entre otras.

4.3.1. Soot

- **Versión:** 4.2.1
- **Mantenimiento:** mantenimiento activo en GitHub. 11 *watch*, 1900 *star* y 586 *fork*.[2]
- ***Input:*** *bytecode*.
- **Creador(es):** Sable Research Group de McGill University.
- **Requisitos:**
  - Java 8 o superior.
  - Maven, versión mínima no especificada (en el caso de que se utilice el módulo de Maven).
  - Gradle, versión mínima no especificada (en el caso de que se utilice el módulo de Gradle).
  - SBT, versión mínima no especificada (en el caso de que se utilice el módulo de SBT).

Framework utilizado para analizar, instrumentar, optimizar y visualizar aplicaciones Java y Android. Soporta el análisis hasta Java 9 y funciona con los binarios compilados. Dispone de muchas funcionalidades (análisis punto-a-punto, cadenas de defensa/uso...) entre ellas el generador de gráfico de llamadas. Para obtenerlo, dispone de muchos algoritmos.

Soot transforma los programas en una representación intermedia (*Intermediate Representation (IR)*), para que pueda ser analizada *a posteriori*. Para ello, se dispone de 4 representaciones para analizar y transformar *bytecode* Java:

- *Baf:* simplifica el *bytecode* para que sea fácil de manipular.

[2]Soot: https://soot-oss.github.io/soot/

- *Jimple:* representación intermedia tipada de 3 direcciones adecuada para sistemas optimizados.
- *Shimple:* variación SSA de Jimple.
- *Grimp:* versión agregada de Jimple adecuada para la des-compilación y la inspección del código.

Jimple es el principal IR de Soot y la mayoría de los análisis se implementan en el nivel de *Jimple*. También se dispone de la opción de añadir representaciones intermedias propias.

VENTAJAS	DESVENTAJAS
Proyecto creado en 1999, por lo que dispone de mucha experiencia	Limitado por la versión de Java 9.
Mantenimiento al día. El último *commit* de GitHub es reciente y dispone de interacción con otros programadores	Necesario realizar cambios para obtener los datos del gráfico de llamadas, ya que trabaja con representaciones intermedias.
El código se encuentra disponible en GitHub, por lo que si fuese necesario realizar una modificación se puede realizar sin problemas	
Documentación muy extensa	
Multilenguaje (Java y Android)	

4.3.2. OpenStaticAnalyzer (OSA)

- **Versión:** 4.0
- **Mantenimiento:** último *commit* de GitHub el 13/07/2018. 15 estrellas, 12 *watch* y 11 *fork*.[3]
- ***Input:*** código fuente.
- **Creador(es):** departamento de software, Universidad de Szeged, Hungría.
- **Requisitos:**
  - Java JDK 1.8 o superior.
  - Microsoft Visual C++ 2015 *Redistributable Package*.
  - Maven 2.2.1 o superior (en el caso de que se utilice el módulo de Maven).

[3]OSA: https://openstaticanalyzer.github.io/

- Librería GNU C en el caso de utilizar Linux.

Framework open-source de análisis estático multilenguaje (Java, JavaScript, Python y C#). Calcula métricas a nivel de componente, archivo, paquete, clase y método:

- Métricas de código fuente.
- Métricas de duplicado.
- Métricas de violación de las reglas de codificación.

Además del análisis recursivo del código fuente basado en directorios, OSA también es capaz de agrupar el sistema (Maven o Ant) del proyecto examinado. Esto puede hacer el análisis más preciso, ya que los archivos generados serán analizados. Por otro lado, es una plataforma independiente de la línea de comandos, dispone de una integración transparente durante el proceso de compilación y también dispone de filtros.

VENTAJAS	DESVENTAJAS
Multilenguaje (Java, JavaScript, Python y C#)	El repositorio no tiene mucha popularidad.
Documentación muy extensa y con definición de ayuda a los posibles errores.	Tiene muchas dependencias, lo cual puede originar problemas en el futuro.

4.3.3. Eclipse JDT

- **Versión:** 4.2.
- **Mantenimiento:** mantenimiento activo, última versión 4.2 con fecha 11/06/2021.[4]
- ***Input:*** código fuente.
- **Creador(es):** Eclipse Foundation.
- **Requisitos:** Java.

Java Development Tools (JDT) [Eclipse, 2020] es uno de los componentes principales de Eclipse SDK. Proporciona un compilador para Java, un modelo completo para componentes Java y el árbol abstracto AST (*Abstract Syntax Tree*) [Vogel et al., 2020]. Para la generación del gráfico de llamadas, se hace uso del módulo AST.

[4]Eclipse JDT: https://projects.eclipse.org/projects/eclipse.jdt

AST es una representación detallada del código fuente de Java en forma de árbol. Para ello, se hace uso de una API para poder crear, leer y eliminar código fuente. Cada elemento de Java se representa como una subclase de *ASTNode* y cada elemento dispone de información especifica sobre el objeto que representa.

VENTAJAS	DESVENTAJAS
Como el creador es Eclipse, el proyecto dispone de muchos *sponsors* y apoyo de la comunidad.	La generación de AST es muy lenta y requiere de muchos recursos.
Documentación muy extensa sobre AST, y muchos proyectos utilizan esta implementación.	Al tratarse de una herramienta, no es un *framework*/programa, por lo que requiere del estudio de la herramienta y su posterior implementación.

4.3.4. SPOON

- **Versión:** 8.2.0.
- **Mantenimiento:** mantenimiento activo en GitHub. 1200 estrellas, 64 *watch* y 245 *fork*.[5]
- ***Input:*** código fuente.
- **Creador(es):** Pawlak, Renaud and Monperrus, Martin and Petitprez, Nicolas and Noguera, Carlos and Seinturier, Lionel.
- **Requisitos:**
  - Java.
  - Maven, versión mínima no especificada (en el caso de que se utilice el módulo de Maven).
  - Gradle, versión mínima no especificada (en el caso de que se utilice el módulo de Gradle).

Herramienta de código abierto para el análisis y la transformación de Java. Soporta hasta Java 9, aunque conceptos de alto nivel no son soportados por la herramienta. Por lo tanto, se trata de una infraestructura (*framework*) accesible para programadores que quieran realizar una aplicación. Para realizar el análisis itera los directorios del código fuente y

[5] SPOON: https://github.com/INRIA/spoon

construye un modelo que conforma al metamodelo de AST para poder realizar las transformaciones. Estas son algunas de las características:

- El metamodelo está lo más cerca posible de los conceptos del lenguaje.
- La API de análisis y transformación es intuitiva y regular.
- Los operadores de transformación están diseñados para advertir lo más rápidamente posible sobre los programas no válidos. Esto se hace con la comprobación estática de tipos o con comprobaciones dinámicas cuando se utilizan los operadores.
- El proyecto es miembro del consorcio de código abierto OW2.[6]

Al ser una infraestructura, se puede ejecutar de diferentes maneras:

- Desde Java.
- Línea de comandos.
- Desde Maven.
- Desde Gradle.

VENTAJAS	DESVENTAJAS
Documentación extensa	Al tratarse de una herramienta, no es un *framework*/programa, por lo que requiere del estudio de la herramienta y su posterior implementación
Su versión 8.2.0 indica que es un proyecto avanzado, añadiendo funcionalidades y corrigiendo errores de manera incremental	
Documentación muy extensa, con foro de errores y ejemplos de ejecución (código de ejemplo, videos...)	

[6]OW2: https://www.ow2.org/

4.3.5. WALA

- **Versión:** 1.3.4.
- **Mantenimiento:** mantenimiento activo en GitHub. 434 estrellas, 32 *watch* y 182 *fork*. [7]
- ***Input:*** *bytecode*.
- **Creador(es):** IBM.
- **Requisitos:**
  - Java 8 o superior.
  - Gradle, versión mínima no especificada (en el caso de que se utilice el módulo de Gradle).
  - Eclipse 3.7 o superior.

Watson Libraries for Analysis (WALA) es una herramienta para el análisis estático y dinámico de Java *bytecode* (compatible con elementos sintácticos hasta Java 8) y JavaScript. Originalmente, fue desarrollado por el grupo de investigación T.J. Watson's Research Center de IBM, pero ahora se encuentra disponible como *open-source*. Al igual que Soot (véase el apartado 4.3.1), también cuenta con generación de gráficos de llamada con la ayuda de diferentes algoritmos. El generador tiene que ser parametrizado con los *entry-point*, para que pueda comenzar a construir desde ese punto. Para generar el gráfico de llamada de manera visual, dispone de la integración de DOT (Graphviz [8]), un lenguaje descriptivo para describir grafos.

Dispone de las siguientes características:

- Sistema de tipos Java y análisis de la jerarquía de clases.
- Análisis de flujo de datos interprocedimental (solucionador RHS).
- Análisis de punteros y construcción de gráficos de llamadas.
- Lenguaje de transferencia de registros basado en SSA (*Static Single Assignment form*).

[7] WALA: https://github.com/wala/WALA
[8] Graphviz: https://graphviz.org/

- *Framework* para el flujo de datos iterativo.
- Utilidades de análisis general y estructuras de datos.
- Biblioteca de instrumentación de *bytecode* (Shrike[9], herramienta que permite leer, modificar y escribir *bytecode* Java)

VENTAJAS	DESVENTAJAS
Multilenguaje (Java y JavaScript)	
Dispone las herramientas para realizar un *plug-in* integrado en Eclipse	
Documentación muy extensa	
Viene integrado con representación visual del gráfico de llamadas	

4.3.6. JCG

- **Versión:** no contiene versión.
- **Mantenimiento:** último *commit* de GitHub el 24/10/2018. 488 estrellas, 42 *watch* y 187 *fork*.[10]
- ***Input:*** *bytecode*.
- **Creador(es):** Georgios Gousios.
- **Requisitos:**
  - Maven, versión mínima no especificada.
  - Java.

Java Call Graph (JCG) es una utilidad basada en **Apache BCEL** (Apache Commons, 2019) [Apache, 2020], para construir gráficos de llamadas estáticas y dinámicas. *Byte Code Engineering Library* (BCEL) pretende ofrecer a los usuarios una forma cómoda de analizar, crear y manipular archivos binarios java. Las clases son representadas en objetos, que contienen toda la información simbólica de la clase que se ha ofrecido: métodos, campos e instrucciones de código en *byte*s particularmente. Los objetos pueden leerse en

[9] Shrike: https://github.com/wala/WALA/wiki/Shrike
[10] JCG: https://github.com/gousiosg/java-callgraph

un archivo, transformados por un programa y que posteriormente se escriba en otro archivo. Además, BCEL contiene un verificador de *bytecode* llamado *Justice*, el cual aporta mejor información de los errores del código de lo que aporta los mensajes de JVM.

JCG es un proyecto *open-source* pequeño, ya que solo tiene un contribuyente. Soporta análisis hasta de estructuras Java 8, y requiere un JAR como *input*. Por otra parte, destaca en una característica especial que le permite analizar código inalcanzable (conocido como *unreachable code*) así como librerías, pero no incluye las llamadas de segmentos de código que nunca se ejecutan.

VENTAJAS	DESVENTAJAS
Analiza código inalcanzable	No está actualmente en mantenimiento
Análisis estático y dinámico	Solo analiza estructuras complejas de Java 8
Proyecto sencillo, ya que solo aporta la funcionalidad de gráfico de llamada. Esto ayuda a su rápida comprensión	No incluye las llamadas de segmentos de código que nunca se ejecutan

4.3.7. Call Graph IntelliJ IDEA

- **Versión:** 2021.1.3
- **Mantenimiento:** último *commit* de GitHub el 26/04/2018. 28 estrellas, 3 *watch* y 9 *fork*.[11]
- ***Input:*** código fuente.
- **Creador(es):** Chentai Kao
- **Requisitos:** IntelliJ IDEA 2021.1 o superior (no se especifica que sea la mínima, pero es la única que se puede descargar desde la web oficial).

Plug-in para visualizar el gráfico de llamadas a funciones de cualquier código integrado en el IDE IntelliJ IDEA. Permite analizar proyectos enteros, por módulos o bien por una dirección del directorio. Dispone de dos métodos de representación gráfica: ajuste a la mejor relación y ajuste a la ventana gráfica. Aunque no se especifica en detalle, utiliza la tecnología de IntelliJ para obtener el gráfico de llamadas. Por otra parte, permite aplicar filtros a la parte visual, como el tipo de función (*public*, *protected* o *private*) o el nombre de la clase.

[11] CallGraph IntelliJ: https://github.com/Chentai-Kao/call-graph-plugin

VENTAJAS	DESVENTAJAS
La representación gráfica dispone de filtros, que permiten un mejor análisis.	El proyecto no dispone de mucha actividad.
Integrado en un IDE, lo que evita de tener que utilizar una herramienta fuera del entorno de trabajo.	Documentación casi inexistente.
	Como está integrado en el IDE, no se puede modificar ni tampoco se puede obtener información.
	Solo disponible en el IDE IntelliJ IDEA

4.3.8. Análisis final

Teniendo en cuenta toda la información redactada en los anteriores apartados, se probaron diferentes herramientas. En un principio, era preferible que la herramienta se pudiese añadir a Eclipse como un *plug-in*, integrando una herramienta ya creada o implementando una nueva con la ayuda de una de los *framwork*s presentados. Se utilizaron los *plug-in* de Eclipse, ya que la mayoría de proyectos de la empresa se implementan con el uso de este IDE, por lo que ayuda al usuario de disponer la herramienta en su entorno de trabajo. El análisis de estos *plug-in* no se incluyen en el anterior apartado, ya que no se realizó un estudio previo, el objetivo era comprobar su funcionamiento de inmediato, y en el caso de que el resultado fuese satisfactorio realizar un estudio *a posteriori*.

Es necesario aclarar que muchas de las herramientas que se han probado no han funcionado. Por consiguiente, no se ha dedicado más de 1 hora y media para probar una herramienta que no funciona, ya que si no se alargaría mucho el análisis de las herramientas. Este tiempo máximo solo se ha alcanzado con las herramientas más prometedoras. Los *plug-in* que se probaron fueron los siguientes:

- ***CallGraph Viewer*:** (versión 0.9.7) genera un gráfico satisfactorio e interactivo con el usuario. Además, dispone de la posibilidad de generar un diagrama de secuencia. Descartado porque no se dispone de ninguna información sobre su funcionalidad, no dispone de filtros y es un poco lento en su ejecución (además de consumir recursos *hardware*).

- ***Java Dependency Viewer*:** (versión 1.0.10) no funciona.

- **Atlas:** (versión 2.0.0) las opciones más avanzadas pertenecen a una versión de pago.

Después se buscaron otras herramientas en otros IDEs, por lo que se probó la herramienta de IntelliJ: Call Graph IntelliJ IDEA (4.3.7). El gran defecto de esta herramienta es su rendimiento. Realizando pruebas con proyectos que no disponían de un gran tamaño, se ha comprobado que consume muchos recursos *hardware* por lo que fue descartado de inmediato. Hay que destacar que si careciese de ese defecto es una herramienta válida, ya que los filtros son muy útiles y los gráficos son interactivos.

En este punto, después de realizar una reunión, se tomó la decisión de no utilizar *plug-in*s por las siguientes razones:

- **Documentación:** generalmente no se dispone de información sobre su implementación, por lo que ante un error se complicaría mucho su solución.
- **Rendimiento:** el rendimiento de todos los *plug-in* probados no son muy buenos a raíz de que están dentro de otro programa (IDE).
- **Sistemas cerrados:** los *plug-in* son programas ”cerrados” por lo que no se puede obtener información de los mismos para que sea procesado por otro programa (ya sea externo o propio).

Teniendo en cuenta esta información, se descartaron los *plug-in* y se comenzó a probar las herramientas analizadas previamente. En primer lugar, se probaron las herramientas que más prometían: **SPOON** (4.3.4), **WALA** (4.3.5) y **Soot** (4.3.1). Las 3 disponen de la posibilidad de utilizar los datos con Graphviz, y WALA dispone de esta opción de manera nativa. Esta herramienta genera la información estructural en forma de diagramas de gráficos y redes abstractas, sin embargo no son interactivos por lo que no sería lo óptimo para utilizar. Esto se debe a que en proyectos muy grandes, los nodos del gráfico serían minúsculos, por lo que el usuario no podría realizar el análisis. Además, era necesario realizar un estudio de la otra librería externa para poder implementar sus características en el programa. Por ello, solamente se ha utilizado para visualizar los datos que exportaban las diferentes herramientas. A destacar, que WALA no se ha podido ejecutar por lo que se descartó inmediatamente y SPOON no se ha podido ejecutar en algunos proyectos.

Después, para disponer de otra perspectiva y también porque era otra las herramientas más interesantes, se probó **JCG** (4.3.6). La herramienta se ejecuta a través de línea de comandos, y no dispone de representación gráfica. Después de probar la herramienta con diferentes proyectos, los resultados eran satisfactorios, por lo que era la principal candidata para utilizar. Para finalizar con las pruebas, se probó **OSA** 4.3.2, pero tampoco se pudo

ejecutar. Por consiguiente, el alumno se puso en contacto por correo electrónico con los autores de las herramientas de SPOON, WALA y OSA, pero nunca se obtuvo respuesta.

Por consiguiente, como no se podría posponer más el análisis se seleccionó la herramienta **JCG** para uso. A pesar de esa circunstancia, también se seleccionó por los siguientes argumentos:

- ***Bytecode***: el *bytecode* dispone de ciertas ventajas que benefician a este proyecto. Esto se debe a que el código fuente siempre se puede convertir a *bytecode*, por lo que si la empresa dispone de versiones de JAR antiguos pero no del código fuente, se puede efectuar el análisis. También, su velocidad de ejecución es mayor, por lo que dispone de una mayor probabilidad de cumplir el requisito IR-11.

- **Librerías:** analiza las librerías que contiene un proyecto. Se sabe de antemano, que la empresa dispone de proyectos que tienen muchas dependencias, por lo que puede ser de utilidad en ciertos proyectos. Como se puede seleccionar qué paquetes del JAR se quieren analizar, se puede excluir el análisis de las librerías.

- **Análisis estático y dinámico:** aunque no se ha utilizado su implementación, como se ha analizado previamente el análisis dinámico puede ser beneficioso para ciertos proyectos. Por ello, que disponga de la posibilidad de utilizar este tipo de análisis es una ventaja a tener en cuenta.

- **Número de métodos:** analiza la mayoría de métodos y llamadas de diferentes de proyectos, por lo que sus análisis son muy completos [Jász. et al., 2019].

- **Apache BCEL:** gracias al uso de la librería BCEL, la herramienta es robusta y dispone de un buen rendimiento, además de que es una librería en continuo crecimiento. Incluso, permite el análisis de código inalcanzable, lo cual es una característica única entre las demás herramientas.

- **Popularidad:** uno de los puntos negativos más importante es la fecha del último *commit* (2018). Esto hace que posibles errores no se hayan solucionado, aunque el primer *commit* data del 27/05/2011. Sin embargo, la popularidad del *plug-in* y las continuas referencias al mismo desde artículos y foros, hace que esta desventaja quede en segundo plano. Además, el análisis es realizado por BCEL y esta sí que está en constante mantenimiento, por lo que los posibles errores del análisis sí que se solucionan continuamente.

- **Proyecto pequeño:** muchas de las herramientas analizadas, disponían de funciones extra que no se iban a utilizar durante el análisis. Eso hace que el código sea más extenso, por lo que dificulta su comprensión. JCG está disponible en GitHub y dispone de una estructura sencilla lo que agiliza su comprensión.

Es necesario volver a destacar, que Soot, SPOON y WALA eran herramientas válidas, no funcionaban correctamente, se precisaba de implementación extra para obtener el gráfico de llamadas, y no disponen de las ventajas de JCG.

4.4. Alternativas de métricas

Como se documenta en el Capítulo 8, el alcance del proyecto cambió drásticamente, por lo que las funcionalidades del programa cambiaron por completo. En primer lugar, se decidió que no era necesario disponer de una representación gráfica del árbol de llamadas. Esto fue una decisión de la directora del proyecto, ya que la representación gráfica era algo positivo para el proyecto, pero se querían priorizar las pruebas exhaustivas del análisis y las métricas.

Para obtener las métricas, también se tuvieron desviaciones que se explican en ese mismo capítulo. Finalmente, se optó por utilizar un análisis estático del código fuente. En el siguiente enlace se encuentran recopiladas muchas de las herramientas estáticas que se utilizan para el análisis estático (también existe otro repositorio de análisis dinámico: enlace). Como se indica en la segunda planificación, es necesario utilizar una herramienta que analice código fuente y no *bytecode*. Por ende, tiene que ser una herramienta que se complemente con JCG, que analiza *bytecode*. Eso implica que a partir de ahora, cuando se analice un proyecto se necesitan los dos tipos de código como parámetros de entrada.

Uno de los requisitos de la nueva herramienta para analizar las métricas es que analice código fuente. También, debe cumplir los requisitos que se indican al principio de este capítulo. Existen muchas herramientas que analizan las métricas, ya que son una de las principales herramientas para realizar *refactoring*. En el repositorio citado antes, se comparan un total de 21 herramientas diferentes que analizan código Java. Entre ellas, se pueden encontrar algunas de las herramientas analizadas en el anterior apartado: SPOON y Soot.

La integración de las herramientas se pueden diferenciar utilizando las siguientes categorías [Analysistools.dev, 2021]:

- **Línea de comandos:** pueden llamarse directamente desde la terminal del SO, y representan la información ya sea en texto plano en la propia terminal o en ficheros externos. Es una opción válida ya que JCG podría analizar los informes realizados por la herramienta. Sin embargo, esto disminuiría mucho el rendimiento del programa, ya que sería necesario realizar 3 análisis: árbol de llamadas, cálculo de métricas y obtener los datos de los ficheros generados. Por otro lado, si la herramienta es *open-source*, se puede añadir como un módulo extra a JCG.

- **Servicio web:** herramientas que proporcionan un sitio web para obtener los informes y generalmente se integran con sistemas de terceros (p. ej.: *GitHub Actions*, *TravisCI* o *CircleCI*). Queda descartada esta opción porque se disponen de mejores opciones.

- **IDE *plug-in*:** existen muchos *plug-in* que se pueden añadir a un IDE, algunos de ellos se han presentado durante este capítulo. Estos *plug-in* otorgan sugerencias en tiempo real durante la fase de implementación. Muchas otras disponen de funcionalidades que requieren una ejecución, ya que realizan cálculos más complejos como las métricas. A pesar de ser una opción válida, se descartaron ya que la directora quiere disponer de la información de las métricas a la vez que se lee el gráfico de llamadas. Por ende, no se pueden tener los dos análisis por separado, ya que complicaría mucho el análisis realizado por un usuario.

Teniendo en cuenta los requisitos de la empresa y las diferentes categorías de las herramientas, se realizó un estudio de las diferentes herramientas. Este análisis no ha sido tan exhaustivo como el realizado sobre las herramientas de gráfico de llamadas, ya que estas pueden tener muchas dificultades y además, es la funcionalidad principal del programa. Por ello, se realizó un análisis más superficial sobre las diferentes herramientas, analizando principalmente su popularidad, número de métricas, rendimiento y mantenimiento.

Finalmente, la herramienta que cumple a los requisitos definidos se llama **CK**. Se han realizado diferentes análisis para comprobar sus resultados y ha sido aprobada por la directora del proyecto. Antes de seleccionar esta herramienta, se habían probado 2 herramientas que disponían más popularidad que esta, pero la cantidad de métricas que esta analiza y las características que se presentarán a continuación fueron determinantes para utilizarla.

4.4.1. CK

- **Versión:** 0.6.5.
- **Mantenimiento:** mantenimiento activo en GitHub. 212 estrellas, 17 *watch* y 81 *fork*.[12]
- ***Input:*** código fuente.
- **Creador(es):** Maurício Aniche.
- **Requisitos:**
  - Java 11, pero dispone de la posibilidad de utilizar otras versiones.
  - Maven, versión mínima no especificada (en el caso de que se utilice el módulo de Maven).

CK calcula las métricas de código a nivel de clase y de método en proyectos Java mediante un análisis estático. Se puede usar ejecutándolo por línea de comandos mediante un JAR o también integrándolo como una dependencia con Maven. Para realizar el análisis hace uso de Eclipse JDT (véase el apartado 4.3.3), que como se ha analizado anteriormente es una librería muy popular que dispone de muchas ventajas y que está en continuo crecimiento.

Actualmente, contiene un amplio conjunto de métricas que se exportan a dos ficheros. Esta cantidad de métricas es lo mejor que tiene esta herramienta. Además, en el repositorio se indica que se están añadiendo nuevas métricas de manera continuada. A continuación, se presentan las métricas que son analizadas por CK [Harrison et al., 1997] [Honglei et al., 2009]. Estas métricas se dividen en dos ficheros con formato CSV, siendo cada métrica una columna de dicho fichero.

- ***class.csv:*** contiene las métricas de cada clase. Las métricas que contiene son las siguientes:
  - **File:** fichero de código fuente que se ha analizado.
  - **Class:** clase que se ha analizado.
  - **Type:** tipo de la clase.

[12] CK: https://github.com/mauricioaniche/ck

- **CBO:** (*Coupling Between Objects*) número de acoplamientos entre dos clases. Cuando una clase (*claseA*) llama a los métodos de otra clase (*claseB*), entonces la primera clase está acoplada con la segunda (*claseA* acoplada con *claseB*). Cuanto más pequeño sea el índice CBO, menor será el efecto de los cambios que se tendrá en otras clases, que significa que es más independiente la clase. Por consiguiente, implica que se tienen que menos actualizaciones cuando sufra alguna modificación. Cuanto mayor sea el CBO, menor reusabilidad tendrá la clase.
- **WMC:** mide la complejidad de una clase individual. Si todos los métodos se consideran igualmente complejos, entonces WMC es el número de métodos definidos en cada clase.
- **DIT:** (*Depth Inheritance Tree*) mide el nivel máximo de la jerarquía de herencia de una clase; la raíz del árbol de herencia no hereda de ninguna clase y está en el nivel cero del árbol de herencia. Recuento de los niveles de los niveles en una herencia jerarquía de herencia. El objetivo es medir la complejidad, la complejidad del diseño y el potencial de reuso ya que cuanto más bajo esté una clase, mayor será el número que tendrá que heredar.
- **LCOM:** (*Lack of Cohesion of Methods*) mide el grado en que los métodos hacen referencia a los datos de instancia de las clases. Un valor alto implica falta de cohesión; es decir, baja similitud y la clase puede ser una composición de objetos no relacionados. No es una métrica a tener en cuenta para decisiones, ya que existe una versión mejorada llamada LCOM-HS, que no se encuentra en el análisis.
- **NOSI:** (*Number of static invocations*) cuenta el número de invocaciones a métodos estáticos. Solo puede contar las que pueden ser resueltas por el JDT.
- **TCC:** (*Tight Class Cohesion*) mide la cohesión de una clase con un rango de valores de 0 a 1. TCC mide la cohesión de una clase a través de conexiones directas entre métodos visibles, dos métodos o sus árboles de invocación acceden a la misma variable de clase.
- **LCC:** (*Loose Class Cohesion*) similar a TCC, pero incluye además el número de conexiones indirectas entre las clases visibles para el cálculo de la cohesión. Así, la restricción LCC $\geq$ TCC se mantiene siempre.
- **Contadores de métodos:** diferentes contadores de diferentes tipos de métodos:

  - *totalMethodsQty:* número total de métodos.
  - *staticMethodsQty:* número total de métodos estáticos.
  - *publicMethodsQty:* número total de métodos públicos.
  - *privateMethodsQty:* número total de métodos privados.
  - *protectedMethodsQty:* número total de métodos *protected*.
  - *visibleMethodsQty:* número total de métodos visibles.
  - *abstractMethodsQty:* número total de métodos abstractos.
  - *finalMethodsQty:* número total de métodos *final*.
  - *synchronizedMethodsQty:* número total de métodos *synchronized*.
- **Contadores de campos:** diferentes contadores de diferentes tipos de campos de la clase:
  - *totalFieldsQty:* número total de campos.
  - *staticFieldsQty:* número total de campos estáticos.
  - *publicFieldsQty:* número total de campos públicos.
  - *privateFieldsQty:* número total de campos privados.
  - *protectedFieldsQty:* número total de campos *protected*.
  - *finalFieldsQty:* número total de campos *final*.
  - *synchronizedFieldsQty:* número total de campos *synchronized*.
- **LOC:** (*Lines Of Code*) contador de las líneas de código, ignorando las líneas vacías y los comentarios.
- **Contadores de instrucciones:** diferentes contadores de diferentes tipos de instrucciones de la clase:
  - *returnQty:* número total de *return*.
  - *loopQty:* número total de bucles.
  - *comparisonsQty:* número total de comparaciones.
  - *tryCatchQty:* número total de *try/catch*.
  - *parenthesizedExpsQty:* número total de expresiones con paréntesis.
  - *stringLiteralsQty:* número total de *string literal*.
  - *numbersQty:* número total de números.
  - *assignmentsQty:* número total de asignaciones.
  - *mathOperationsQty:* número total de operaciones matemáticas.
  - *variablesQty:* número total de variables.

- *lambdasQty:* número total de instrucciones *lambda*.
- *logStatementsQty:* número total de instrucciones *log*.

- **Otros contadores:**
  - *maxNestedBlocksQty:* número total de bloques anidados.
  - *anonymousClassesQty:* número total de clases anónimas.
  - *innerClassesQty:* número total de clases internas.
  - *uniqueWordsQty:* número total de palabras únicas.
  - *modifiers:* número total de modificadores.

- ***method.csv:*** contiene todos los métodos de todas las clases, con sus métricas adicionales. Las métricas que tiene son las siguientes:
  - **File:** fichero de código fuente que se ha analizado.
  - **Class:** clase que se ha analizado.
  - **Method:** nombre del método.
  - **Constructor:** "TRUE" si es un método constructor y "FALSE" si no lo es.
  - **CBO:** (*Coupling Between Objects*) número de acoplamientos entre dos clases (Explicado en detalle en anterior punto).
  - **WMC:** mide la complejidad de un método. Para calcular, se utiliza la siguiente fórmula con la ayuda de la estructura en árbol del método.

$$WMC = Arcos - Nodos + 2$$

  - **RFC**: (Response for a Class) mide la comunicación (o el envío de mensajes) dentro de las clases. Cuanto más grande sea la RFC, más compleja es la clase y, por lo tanto, las pruebas y el mantenimiento serán más difíciles.
  - **LOC:** (*Lines Of Code*) contador de las líneas del método, ignorando las líneas vacías y los comentarios.
  - **Contadores de instrucciones:** diferentes contadores de diferentes tipos de instrucciones de la clase:
    - *returnsQty:* número total de *return*.
    - *variablesQty:* número total de variables.
    - *parametersQty:* número total de parámetros.

  - *methodsInvokedQty:* número total de métodos que invoca.
  - *methodsInvokedLocalQty:* número total de métodos que invoca de manera local.
  - *methodsInvokedIndirectLocalQty:* número total de métodos que invoca de manera local indirectamente.
  - *loopQty:* número total de bucles.
  - *comparisonsQty:* número total de comparaciones.
  - *tryCatchQty:* número total de *try/catch*.
  - *parenthesizedExpsQty:* número total de expresiones con paréntesis.
  - *stringLiteralsQty:* número total de *String literal*.
  - *numbersQty:* número total de números.
  - *assignmentsQty:* número total de asignaciones.
  - *mathOperationsQty:* número total de operaciones matemáticas.
  - *lambdasQty:* número total de instrucciones *lambda*.
  - *logStatementsQty:* número total de instrucciones *log*.
- **Otros contadores:**
  - *maxNestedBlocksQty:* número total de bloques anidados.
  - *anonymousClassesQty:* número total de clases anónimas.
  - *innerClassesQty:* número total de clases internas.
  - *uniqueWordsQty:* número total de palabras únicas.
  - *modifiers:* número total de modificadores.
  - *hasJavaDoc:* "TRUE" si tiene JavaDoc especificado y "FALSE" si no lo tiene.

La herramienta también analiza más métricas sobre las variables y los campos de la clase, pero se ha tenido que eliminar del programa final (véase el apartado 7.2.2).

5. CAPÍTULO

Diseño

Después de realizar la planificación (*Planificación (PL)* y *Captura de requisitos (CR)*, la captura de los conocimientos y selección de herramientas (*Administración del campo (AC)*), se realiza un diseño general de la aplicación (*Diseño (D)*). En este capítulo, se redacta el diseño final después de integrar las dos herramientas seleccionadas: JCG (árbol de llamadas) y CK (métricas) (véase el Capítulo 4). Para ello, se enumeran los patrones de diseño que se han utilizado en el diseño (5.1), justificando así el diseño empleado en el programa. Posteriormente, el diseño final se especifica con el uso de diagramas de clases (5.2), se explican todas las clases agrupándolas en sus respectivos módulos. Los diagramas de secuencias se ubican en el Anexo C.

A pesar de que parte de este diseño no sea propio, ya que provienen de las diferentes herramientas utilizadas, se explica su diseño e implementación para entender el funcionamiento total del programa. Aun así, el funcionamiento de estas herramientas es muy complejo, por lo que tampoco se especifican en total detalle. La implementación se detalla en el Capítulo 6), capítulo en el que se explica toda la implementación y las decisiones tomadas durante el desarrollo. En dicho capítulo también se indica qué clases son nuevas, cuáles se han modificado y se realiza una comparación de las líneas de código.

5.1. Patrones de diseño

5.1.1. *Visitor* (Visitante)

Visitor[1], es un patrón de diseño de la categoría de comportamiento que permite separar algoritmos de los objetos sobre los que operan. Estas son las principales aplicaciones que se han hecho con este patrón:

- Permite realizar una operación sobre todos los elementos de una estructura de datos compleja. El patrón permite ejecutar una operación sobre un grupo de objetos con diferentes clases, haciendo que un objeto visitante implemente diferentes variantes de una misma operación que correspondan a todas las clases objetivas. De esta manera, se pueden visitar las clases y métodos de los proyectos que se quieren analizar, que suponen una estructura de datos compleja.

- Se aísla la lógica de negocio de comportamientos auxiliares. De esta manera, las clases principales disponen de menos funciones, ya que se utilizan nuevas clases auxiliares por cada funcionalidad que se quiera añadir. Gracias a ello, también se reduce la tarea de *refactoring* ya que las funcionalidades están divididas en diferentes clases. Si en un futuro, se quiere modificar el análisis sobre métodos o clases, se aísla el código que se quiere analizar.

- Aísla un comportamiento que solo tiene sentido en algunas clases de una jerarquía de clases, pero no en otras. Para ello, se extrae el comportamiento y se aísla en una clase separada y se implementan solo aquellos métodos visitantes que acepten objetos de clases relevantes. Por consiguiente, el resto de clases se quedan vacías. También permite almacenar información en un objeto visitante, permitiendo así pasar objetos de métodos y de clases que se analizan.

5.1.2. *Simple Factory* (Fábrica simple)

El patrón *Simple Factory*[2] describe una clase que tiene un método de creación con un gran condicional que, basándose en los parámetros del método, elige la clase de producto que tiene que instanciar. Se ha utilizado para implementar la parte que permite exportar

[1] Visitor: refactoring.guru/es/design-patterns/visitor
[2] Factory: refactoring.guru/es/design-patterns/factory-comparison

información en diferentes ficheros. Estas son las principales aplicaciones que se han hecho con este patrón:

- Cuando no se conocen las dependencias y los tipos de antemano, permite su futura implementación. Permite que en un futuro se puedan añadir más métodos de exportación, con tan solo implementar los métodos.
- Ahorro de recursos de *hardware* gracias a la reutilización de objetos existentes en lugar de reconstruirlos cada vez.

5.1.3. *Singleton* (Instancia única)

El *Singleton*[3] es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia. Estas son las principales aplicaciones que se han hecho con este patrón:

- Cuando solo se quiere tener una única instancia para todos los objetos de la jerarquía de clases. Con el patrón, se deshabilita a la creación del objeto de clase. En este caso, se ha utilizado para la clase principal del programa (JCallGraph), ya que se invoca desde otras clases durante la ejecución del programa.
- Permite un control más estricto sobre las variables globales.

5.2. Diagrama de clases

El programa se estructura en 3 componentes principales: ***callgraph*** (5.2.2), **com.github.mauricioaniche.ck** (5.2.3) y **org.apache.bcel** (5.2.1). Las herramientas que han integrado utilizan 2 librerías para poder interpretar los datos que contienen los ficheros Java. En la siguiente Figura, se ve reflejado de forma resumida esta arquitectura:

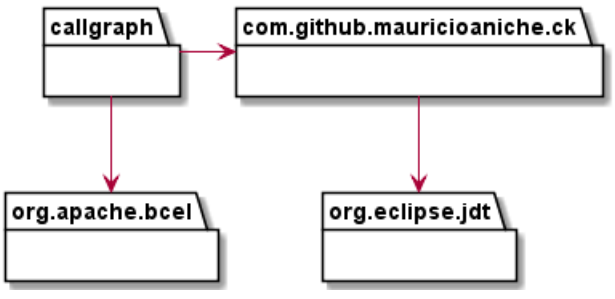

Figura 5.1: Estructura simplificada del programa

[3]Singleton: refactoring.guru/es/design-patterns/singleton

En la Figura 5.2 se representan las clases y librerías del programa, para poder entender mejor el entorno del programa. Esta figura dispone de clases separadas en diferentes categorías, para indicar qué clases han sido modificadas y cuáles no. Esto se profundiza más en el apartado 6.3. Las categorías son las siguientes:

- **Propio:** clase que se ha creado desde 0 para implementar las funcionalidades del programa.
- **Original:** clase que tenía la herramienta original, la cual prácticamente no se ha modificado.
- **Hibrido:** clase original de la herramienta, pero que se le han aplicado cambios significativos.

Después se encuentra la Figura 5.3, que representa el diagrama de clases del módulo *callgraph*, módulo más importante del programa y es en el que más cambios se han realizado. En la última Figura, no se representan clases de los módulos de BCEL y de CK, ya que sería un diagrama mucho más extenso. Después de las figuras, se explican todas las clases de cada uno de los módulos, incluso de la librería BCEL para entender bien el funcionamiento del programa.

5.2.1. BCEL

BCEL ([Apache, 2020]) es una librería de Apache que se ha detallado en profundidad en el apartado 4.3.6. Claro está, que no se hace uso de todas las clases de esta extensa librería. A continuación, se detallan las clases principales que se han utilizado para que la herramienta JCG pueda funcionar. Como es obvio, al ser una librería no se ha modificado nada del código, por lo que esta explicación tiene como objetivo definir la funcionalidad de algunas de las clases que se utilizan en el resto del programa. Estas clases son las que permiten obtener la información de los ficheros *bytecode*:

- **JavaClass:**[4] presenta las estructuras, constantes, campos, métodos y comandos de un fichero *.class* Java. El objetivo de esta clase es representar las estructuras del fichero Java en un formato accesible para un desarrollador, lo cual permite interactuar con las clases de los ficheros *bytecode*. Mediante esta interacción, se pueden utilizar los atributos mencionados para realizar diferentes análisis, como métricas o las llamadas de los métodos.

[4] JavaClass: commons.apache.org/proper/commons-bcel/apidocs/org/apache/bcel/classfile/JavaClass.html

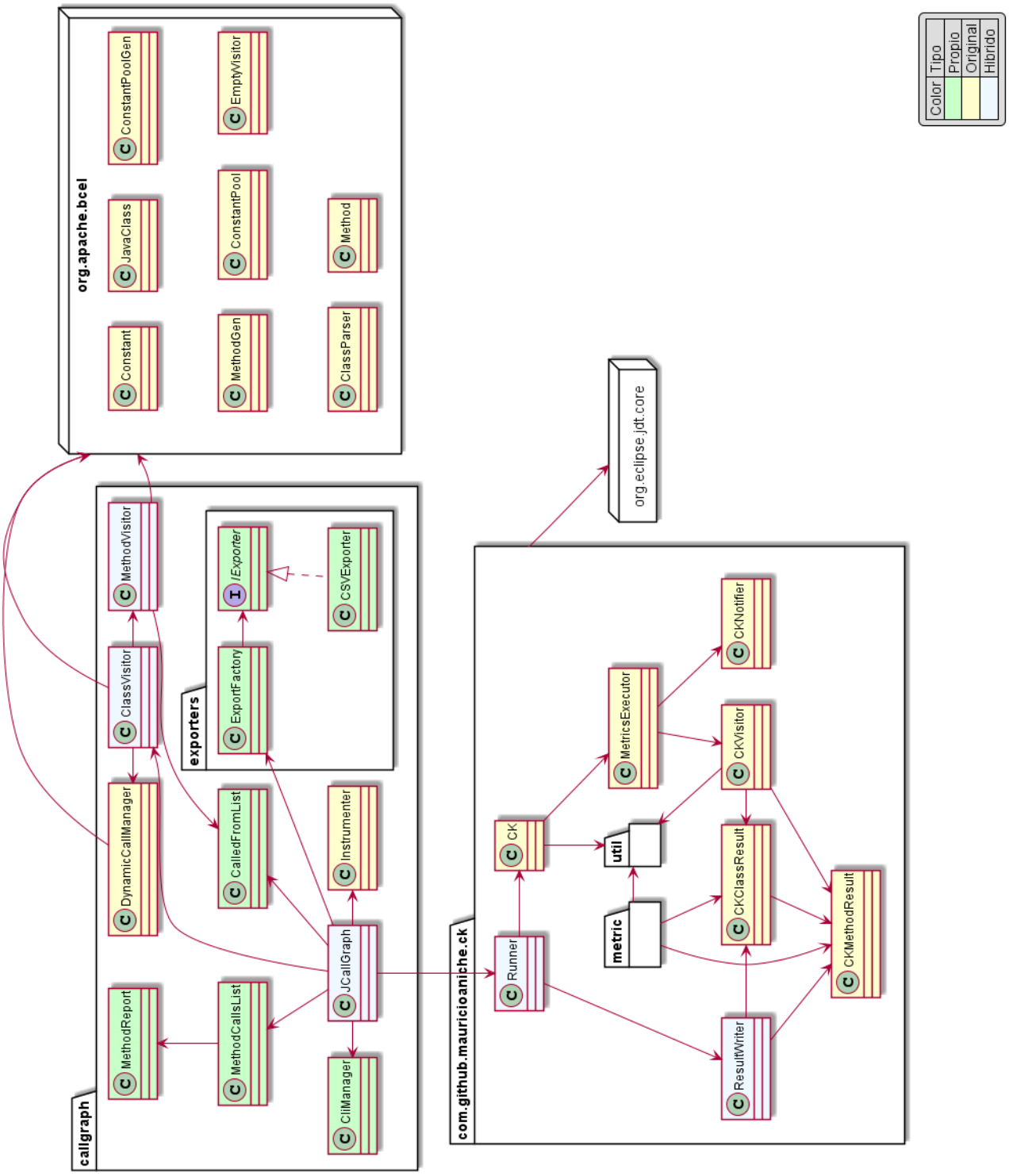

Figura 5.2: Diagrama de clases del programa

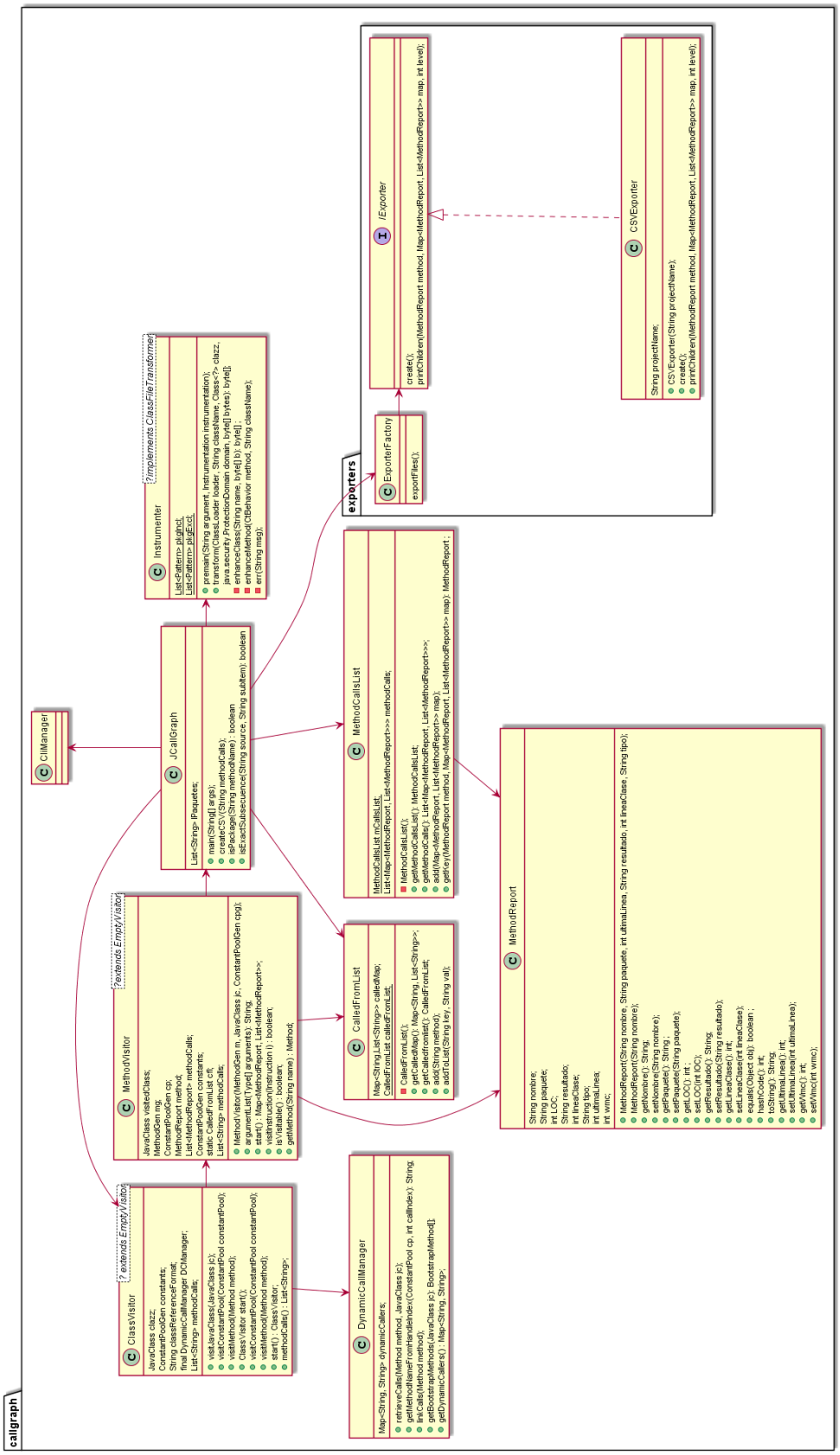

Figura 5.3: Diagrama de clases del módulo *callgraph*

- **ClassParser:**[5] analiza un archivo Java *.class* y devuelve una instancia de JavaClass.
- **Constant:**[6] superclase abstracta para representar las diferentes constantes que tiene un fichero *.class* Java.
- **ConstantPoolGen:**[7] permite construir un grupo (*pool*) de constantes. El desarrollador puede añadir constantes mediante diferentes métodos y estos mismos devuelven un índice indicando su lugar en el grupo de constantes. De esta manera, cuando se analiza un fichero *.class* se dispone de un grupo de constantes para construir el resto de estructuras.
- **ConstantPool:**[8] representa el grupo de constantes (*constant pool*). Puede contener valores nulos, ya que es generada por la JVM y solo admite entradas constantes (8B). Por consiguiente, se diferencia de ConstantPoolGen porque es creada de manera automática por JVM.
- **MethodGen:**[9] plantilla para la construcción de un método. Esto se hace definiendo los controladores de excepciones, añadiendo excepciones, variables locales y atributos. Se ha utilizado para crear los objetos que conforman el árbol de llamadas (MethodReport, véase el apartado 5.2.2).
- **Method:**[10] representa la estructura de información de un método. Un método dispone de los indicadores de accesos (*access flags*), nombre, firma (seguridad de Java) y atributos. No se utiliza este objeto para guardar los datos cuando se construye el árbol de llamadas, sin embargo es parte de la clase de JavaClass (entre otros).
- **EmptyVisitor:**[11] implementación de la interfaz Visitor, que proporciona cuerpos de métodos para que puedan ser sobrescritos por sus subclases. Se ha utilizado para poder visitar las clases y métodos del proyecto que se está analizando.

5.2.2. *Callgraph*

El núcleo principal del programa es la herramienta JCG, ya que interactúa directamente con el usuario. Esto se debe a que gestiona los parámetros de entrada (*input*) y los

[5] ClassParser: commons.apache.org/proper/commons-bcel/apidocs/org/apache/bcel/classfile/ClassParser.html
[6] Constant: commons.apache.org/proper/commons-bcel/apidocs/org/apache/bcel/classfile/Constant.html
[7] ConstantPoolGen: commons.apache.org/proper/commons-bcel/apidocs/org/apache/bcel/generic/ConstantPoolGen.html
[8] ConstantPool: commons.apache.org/proper/commons-bcel/apidocs/org/apache/bcel/classfile/ConstantPool.html
[9] MethodGen: commons.apache.org/proper/commons-bcel/apidocs/org/apache/bcel/generic/MethodGen.html
[10] Method: commons.apache.org/proper/commons-bcel/apidocs/org/apache/bcel/classfile/Method.html
[11] EmptyVisitor: commons.apache.org/proper/commons-bcel/apidocs/org/apache/bcel/generic/EmptyVisitor.html

parámetros de salida (*output*). El programa original no dispone de un control de errores y solo admite parámetros siguiendo cierto formato específico. Se ha modificado esta función para que sea más interactivo con el usuario y evitar análisis no deseados. Por otra parte, realiza el análisis del *bytecode* Java con la ayuda de la librería de BCEL (véase el apartado 5.2.1). Para ello, dispone de diferentes clases para poder ejecutar y obtener esos datos, para posteriormente poder realizar el análisis del árbol de llamadas. Otra de las carencias de la herramienta es que no dispone de ningún método para exportar los datos, ya que realiza la representación de los datos mediante la terminal. Esto se ha solventado para que el programa cumpla con el requisito IR-08.

A continuación se presentan las clases de este módulo. Estas clases sí que han tenido que ser modificadas, y se reflejan esas modificaciones en el Capítulo 6).

- **CliManager:** como ya se ha comentado, la herramienta no disponía de mecanismos para gestionar los parámetros de entrada. Esta clase asegura que los parámetros de entrada siguen un formato específico, imprime al usuario el listado de opciones que puede utilizar y se utiliza para saber que opciones ha utilizado el usuario en otras clases. Por consiguiente, se utiliza el patrón Singleton para poder mantener los valores de las variables en el resto de clases.
- **JCallGraph:** clase principal de la aplicación, que se encarga de invocar aquellos métodos que ejecutan las funcionales base del programa. Como es la aplicación principal, sigue el esquema de un patrón Singleton, ya que dispone de información que necesitan el resto de clases. Entre todas las funcionalidades, solo realiza la búsqueda de los ficheros *.class* del JAR. Por consiguiente, se utiliza como gestor para el resto de clases. Las funciones que realiza son las siguientes:
  - Envía los parámetros de entrada (*input*) a CliManager para que sean verificados.
  - Itera sobre los ficheros que contiene el JAR, identificando aquellos que sea *.class* y descartando el resto de ficheros.
  - Llama a los métodos de ClassVisitor para construir el árbol de llamadas. El árbol de llamadas se almacena en una instancia de MethoCallsList.
  - Llama a Runner del módulo de CK, para calcular las métricas de los ficheros que se han analizado.
  - Llama a ExportFactory para crear los ficheros que contienen los datos del análisis realizado.

  - Informa al usuario a través de la terminal sobre el estado de la ejecución del programa.

- **ClassVisitor:** invoca los métodos visitantes por cada método que se encuentre dentro de una clase. Para ello, dispone de las clases de BCEL: JavaClass, ConstantPoolGen, ConstantPool, EmptyVisitor, MethodGen y Method. Para poder construir el árbol de llamadas. utiliza la clase DynamicCallManager para obtener las llamadas de un método.

- **MethodVisitor:** clase que funciona como visitante de los métodos. Es invocada por el visitante de clases (ClassVisitor). Recopila toda la información del método que analiza (MethodGen, ConstantPoolGen), pero también almacena la lista de métodos que visita, para poder realizar el árbol de llamadas.

- **MethodReport:** almacena toda la información necesaria para poder imprimir los datos de los métodos del árbol de llamadas. Dispone de todos los atributos identificativos de un método:

  - Nombre.
  - Paquete del proyecto.
  - Líneas del método (*Lines Of Code (LOC)*).
  - Línea en la que se encuentra dentro de la clase.
  - Resultado del método.
  - Complejidad ciclomática.

- **CalledFromList:** los métodos también tiene que disponer de los métodos de las otras clases que son llamados. Es decir, si un método ”A” de la ”claseA” es llamado por otro método ”B” de la ”claseB”, es necesario imprimir que el ”A” es llamado por ”B”. Esta lista no está dentro de la clase MethodReport, por motivos que se explican en el Capítulo 6.

- **Instrumenter:** dispone de funcionalidades que no se han utilizado en este programa. Entre esas funcionalidades, se encuentra el preprocesado que se realiza en el programa original, y también las llamadas a métodos para poder realizar el análisis dinámico. Como ya se ha argumentado previamente, se ha decidido optar par análisis estático y descartar el análisis dinámico.

- **ExportFactory:** clase principal para exportar el análisis del programa, que sigue el patrón *Single Factory* (véase el apartado 5.1.2), siendo esta clase la creadora de los productos. Dispone de todos los casos posibles para exportar los datos del programa. Para ello, utiliza la información que tiene almacenada CliManager.

- **IExporter:** interfaz que contiene las cabeceras de los productos, las cuales permiten exportar los ficheros que contienen el árbol de llamadas del programa. Esta interfaz representa los métodos de los productos, y es implementado por los productos concretos. Esta versión del proyecto, solo incluye un producto concreto (CSVExporter), ya que por ahora no se ha solicitado introducir más tipos de productos.

5.2.3. CK

La herramienta CK apenas dispone de modificaciones, solamente se ha modificado la entrada de los datos a la herramienta y ciertas configuraciones para mejorar el rendimiento (Véase los Capítulos 6 y 7). Al igual que BCEL, se explica el funcionamiento de esta herramienta en el análisis de la misma (véase el apartado 4.4) y solo se explican las clases más importantes para entender el funcionamiento del programa.

CK no utiliza las estructuras que proporciona BCEL sobre el *bytecode*, sino que utiliza la librería JDT de Eclipse para construir un árbol conforme a un modelo AST. Al igual que BCEL, JDT proporciona mecanismos al desarrollador para poder interactuar con el código Java, en este caso con el código fuente. A continuación se definen las clases importantes de este módulo:

- **Runner:** clase principal de la herramienta. A diferencia de BCEL, sí que dispone de un control sobre los parámetros de entrada que recibe. En primer lugar, transforma esos datos de entrada a objetos para que se pueda realizar un análisis acorde a esos parámetros. Aparte del control de parámetros, su función principal es recorrer los ficheros Java para invocar las clases pertinentes para el análisis de métricas (CK).

- **CK:** calcula las métricas de un fichero Java, construyendo un árbol siguiendo un modelo AST. Para ello, utiliza la interfaz CKNotifier, que es la encargada de visitar el fichero.

- **MetricExecutor:** se encarga de ejecutar diferentes clases para poder calcular las métricas. Para ello, implementa la interfaz FileASTRequestor, que permite gestionar los árboles AST.

- **CKNotifier:** interfaz que dispone de las cabeceras para avisar de los resultados de un análisis de métricas.
- **CKClassResult:** contiene el resultado del análisis realizado sobre una clase.
- **CKMethodResult:** contiene el resultado del análisis realizado sobre un método.
- **ResultWriter:** una vez finalizado el análisis, imprime en un CSV el resultado del análisis. Este resultado del análisis, solo dispone de las métricas calculadas y no tiene relación con el árbol de llamadas. Por ende, no tiene relación con los ficheros que se exportan con la clase ExporterFactory.

6. CAPÍTULO

Desarrollo

Después de terminar con el diseño inicial del programa (*Diseño (D)*), se continuó con el desarrollo de la aplicación. Este desarrollo, como se redacta en la primera planificación (Véase el Capítulo 3) consistía en realizar 3 prototipos siguiendo un ciclo de vida de manera incremental. Sin embargo, como se indica en la segunda planificación, esto fue modificado a raíz de ciertas desviaciones (véase el apartado 8.1).

En este capítulo se detalla la implementación final del programa, argumentado todas las decisiones tomadas en las diferentes fases del proyecto. El capítulo se apoya en ciertos fragmentos de código que se consideran relevantes para entender el funcionamiento del programa. Por ende, no se explica toda la implementación del programa. Para entender la diferencia entre la implementación que disponían las herramientas de base y las modificaciones que se han realizado, se realiza una comparativa de las líneas de clase (6.3). Esto no se puede tomar tampoco como un criterio, ya que a muchas clases incluso se le han eliminado fragmentos de código, solamente sirve para mostrar la comparativa y que se pueda apreciar dónde se han realizado los cambios. Se puede consultar el código en un repositorio público de GitHub.

Para explicar mejor la implementación, se ha separado cada funcionalidad en un apartado. Estos apartados están ordenados de manera que el lector entienda bien el funcionamiento del proyecto. Estas explicaciones se complementan con los diagramas presentados en el Capítulo 5.

6.1. Funcionamiento del programa

A continuación se realiza una breve descripción sobre cómo se ejecuta el programa. Esto se detalla en profundidad en la guía de uso del programa que se ha entregado a la empresa, véase el Anexo B. En ese documento se indican en detalle los métodos de uso con ejemplos, las dependencias del programa, resultados de ejemplo y análisis de los resultados.

En un principio, el programa se planteó para que se ejecutase utilizando dos métodos distintos:

- **IDE:** opción que va a utilizar la empresa. Consiste en crear un fichero en un IDE que pase los parámetros de entrada a la clase principal del programa. Una de las opciones más utilizadas, es la de crear JUnit (fichero de pruebas unitarias), y crear una prueba por cada análisis que se quiere realizar. De esta manera, se tienen todos los análisis previos y cada uno se puede ejecutar por separada. Esta opción es la que más se asemeja a la interfaz gráfica, por lo que es la opción que utiliza la empresa, ya que no se tiene que salir del IDE en el cual se está trabajando.

- **Línea de comandos:** idea inicial para usar el programa. Se precisa de un JAR del programa, y se introducen los parámetros mediante la línea de comandos. Esta opción está implementada, pero no se tiene pensado utilizar por el momento.

Sin importar cuál de los dos métodos de ejecución se utilice, el programa necesita los siguientes parámetros:

- **Nombre del JAR:** el programa necesita el JAR del proyecto que se tiene que analizar, este JAR puede crearse a través de un MANIFEST o un *software* de construcción de proyectos como Maven o Graddle. En el caso de que el JAR no se sitúe en la raíz del proyecto principal, será necesario especificar la dirección absoluta.

- **Paquete(s) que se quiere(n) incluir:** el programa analiza todo el código que está almacenado dentro del JAR, incluso las dependencias del código analizar si es que las tuviese. Por ello, es necesario especificar qué paquetes son necesarios analizar. Para incluir varios paquetes se tienen que separar por comas (p. ej.: ”paquete1, paquete2”). En el caso de que se disponga una estructura Maven o similar, siendo la raíz *src/main/java*, es necesario especificar los paquetes a partir de este punto.

- **Paquete(s) que se quiere(n) excluir:** igual que el parámetro anterior, pero excluyendo los paquetes que no se quieren analizar. También se puede especificar que se excluyan todos los paquetes externos introduciendo ”*,”.

- **Código fuente del proyecto:** es necesario además del JAR, incluir la dirección absoluta de la carpeta en la que se encuentra el proyecto que se quiere analizar.

6.2. Implementación

6.2.1. Inicio de la aplicación

La clase CliManager interactúa directamente con el usuario, ya que recibe los parámetros de entrada del programa a través de la clase JCallGraph. En primer lugar, verifica que el formato de los parámetros se adecuan con los definidos en el programa, y en el caso de que sean correctos se ejecuta el programa. Para ello, se utiliza la clase CommandLine de la librería de Apache. Se definen los parámetros que se quieran en esta clase, y se encarga de verificar cada uno de los que sean obligatorios y los iguala a atributos de su clase. De esta manera, cuando se termina la verificación de los argumentos, se utiliza la instancia de CliManager durante la ejecución para conocer las opciones que se han seleccionado. El siguiente fragmento de código sirve para añadir argumentos de entrada a CommandLine y después se indica como se guardan los datos de los parámetros en variables de la clase CliManager:

Código 6.1: Introducir argumentos a CommandLine

```
Options options = new Options();
options.addOption("jar", true, "Direccion absoluta del Jar");
options.addOption("s", true, "Dirección absoluta del codigo fuente");
options.addOption("h", "help", false, "Imprime el mensaje de ayuda");
```

Código 6.2: Guardar parámetros de entrada introducidos por el usuario

```
jar = cmdLine.getOptionValue("jar");
if (jar == null) {
    throw new org.apache.commons.cli.ParseException("La direccion absoluta del
    JAR es requerida");
}
```

Si falta alguno de los argumentos o los formatos son incorrectos, se para la ejecución del programa y se muestra el error al usuario.

6.2.2. Iterar sobre el *bytecode*

Para iterar sobre el *bytecode*, en primer lugar se convierte el *bytecode* a tipo Stream, para que sea más fácil su análisis. Para ello, se utiliza una función que convierte un tipo enumerado en Stream (`Stream<T> enumerationAsStream()`). Para obtener el tipo enumerado, se utiliza la clase JarFile, que permite transformar todas las entradas a tipo enumerado. En resumen, se dispone de un fichero *bytecode* que se convierte a tipo enumerado porque es su única transformación posible, para que posteriormente se convierta en tipo *Stream* ya que es más fácil de iterar.

Código 6.3: Método *enumerationAsStream()*

```
public static <T> Stream<T> enumerationAsStream(Enumeration<T> e) {
    return StreamSupport.stream(Spliterators.spliteratorUnknownSize(new
    Iterator<T>() {
        public T next() {
            return e.nextElement();
        }
        public boolean hasNext() {
            return e.hasMoreElements();
        }
    }, Spliterator.ORDERED), false);
}
```

Se itera por cada entrada que contenga el JAR. En primer lugar, se identifica si la entrada es un directorio que está dentro del JAR. Por otra parte, como un JAR puede contener cualquier tipo de archivo, es necesario filtrar estos ficheros para analizar solo los archivos *.class*. Después, se comprueba que el fichero pase los filtros que se han especificado mediante los parámetros de entrada. Para esa comprobación, en primer lugar se comprueba que el formato del fichero sea el correcto. Como se ha comentado en el apartado 4.2.1, los ficheros de *bytecode* pueden tener un formato diferente al original. Una vez se verifique su buen formato, se comprueba que esté incluido en la lista de paquetes a incluir y no esté en la lista de paquetes a excluir.

Código 6.4: Método *isPackage()*

```
public static boolean isPackage(String name) {
```

```
    if (name.contains("$")) {
        return false;
    } else if (lInclude.isEmpty() && lExclude.isEmpty()) {
        return true;
    } else if (lInclude.size()==0) {
        return true;
    } else if (lExclude.get(0).equals("*")) {
        return isExactSubsecuence(name, lInclude.get(0));
    } else {
        for (String include : lInclude) {
            if (isExactSubsecuence(name, include)) {
                boolean exit = false;
                for (String exclude : lExclude) {
                    if (isExactSubsecuence(name, exclude)) {
                        exit = true;
                        break;
                    }
                }
                if (!exit) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

En el caso de que se hayan pasado todos estos condicionales con éxito, se transforma el fichero de tipo Stream a JavaClass para poder realizar el análisis, con la ayuda de la clase ClassParser. Gracias a esta clase se pueden interactuar con todas las características del fichero Java con mayor facilidad. Por ello, se pasa como parámetro esta clase a ClassVisitor, para que pueda visitar la clase y todos su métodos.

Código 6.5: Iterar los ficheros *bytecode*

```
try (JarFile jar = new JarFile(f)) {
    Stream<JarEntry> entries = enumerationAsStream(jar.entries());
    entries.forEach(e -> {
        if (!e.isDirectory() && e.getName().endsWith(".class")) {
            if (isPackage(e.getName())) {
                ClassParser cp = new ClassParser(args[0], e.getName());
```

```
                HashMap<MethodReport, List<MethodReport>> map =
                getClassVisitor.apply(cp).start().methodCalls();
                if (map != null) {
                    methodCallsList.add(map);
                }
            }
        }
    });
}
```

Para obtener el visitante ClassVisitor, se utiliza una función auxiliar que realiza la transformación de ClassParser a ClassVisitor:

Código 6.6: Método *getClassVisitor()*

```
Function<ClassParser, ClassVisitor> getClassVisitor = (ClassParser cp) -> {
    try {
        return new ClassVisitor(cp.parse());
    } catch (IOException e) {
        throw new UncheckedIOException(e);
    }
};
```

6.2.3. Visitar clases

Siguiendo la estructura del patrón *Visitor*, se utiliza un visitante para obtener los datos de la case Java. La clase ClassVisitor obtiene una instancia JavaClass, con todas las características del fichero Java. Cuando se visita una clase del fichero *.class*, se obtienen todos los métodos de JavaClass y se visita cada uno de los métodos. Cuando se itera sobre los métodos de la clase, ClassVisitor llama a DynamicCallManager que realiza esta búsqueda de las llamadas. Esta tarea es posible gracias a muchas de las clases de la librería BCEL ya comentadas, como ConstantPool, Method, JavaClass... Esta implementación es bastante compleja, por lo que si fuese necesario su implementación se hubiese necesitado mucho más tiempo para su estudio e implementación.

Al finalizar, se consigue que cada clase contenga todos sus llamadores (*callers*), para poder construir el árbol de llamadas.

Código 6.7: Método *visitJavaClass()*

```
public void visitJavaClass(JavaClass jc) {
```

```
    jc.getConstantPool().accept(this);
    Method[] methods = jc.getMethods();
    for (int i = 0; i < methods.length; i++) {
        Method method = methods[i];
        DCManager.retrieveCalls(method, jc);
        DCManager.linkCalls(method);
        method.accept(this);
    }
}
```

Una vez se tiene construida la clase entera con todas sus características para poder realizar el análisis, se procede a visitar los métodos que contiene cada método de la clase, construyendo así el árbol de llamadas de la clase que se visita.

6.2.4. Visitar métodos de un método

En primer lugar, la clase ClassVisitor dispone de un método que se ejecuta por cada método que contiene la clase. Cuando itera sobre todos los métodos, visita cada uno de ellos realizando una transformación de un método a MethodGen, para que posteriormente se pueda pasar como parámetro a la clase MethodVisitor. MethodGen es una plantilla que permite obtener todas las características de un método. Finalmente, se invoca la case MethodVisitor con los siguientes parámetros:

- Instancia de MethodGen creada en base a un método (Method).
- La clase java que se está visitando (JavaClass).
- Las constantes que tiene la clase. Este último parámetro no se encontraba en la implementación de la herramienta JCG. Sin embargo, es necesario para crear instancias de MethodReport, como se explica más adelante en este apartado.

Código 6.8: Método *visitMethod()*

```
public void visitMethod(Method method) {
    MethodGen mg = new MethodGen(method, clazz.getClassName(), constants);
    MethodVisitor visitor = new MethodVisitor(mg, clazz,constants);
    if (visitor.isVisitable()) {
        methodCalls.putAll(visitor.start());
    }
}
```

Por cada método, se visitan todos los métodos que contiene el método. Para poder almacenar y construir el árbol de llamadas, se introduces los datos en la siguientes estructuras:

- `HashMap<MethodReport, List<MethodReport>> map`: este mapa está formado por un par que tiene como clave (*key*) un objeto MethodReport y como valor (*value*) una lista de MethodReport. Es decir, la clave contiene la instancia del método que se está visitando y en el valor se almacena la lista de métodos que llama ese método. Todo ello guardado en un mapa, que se devolverá a la clase ClassVisitor para que se este tenga el árbol de llamadas de la clase. Por ende, la clase ClassVisitor dispone de una lista (mapa) de cada uno de sus métodos, y por cada uno de los métodos dispone de las llamadas a los métodos que realiza.
- `MethodReport`: nueva clase que no disponía previamente JCG. La herramienta original, no dispone de la funcionalidad de exportar el árbol de llamadas en ficheros, y tampoco realiza operaciones extra a parte del análisis del árbol de llamadas. Por eso, no necesita almacenar los datos en objetos, por lo que se almacena toda la información en listas de tipo String. Este comportamiento se ha modificado, creando un objeto que permite añadir muchos datos necesarios para poder exportar el árbol de llamadas y las métricas calculadas. Como en una fase más avanzada de la ejecución del programa es necesario calcular las métricas, se introducirán los valores calculados en estos objetos.

Código 6.9: Inicializar un MethodReport

```
method = new MethodReport(mg.getName(), mg.getClassName(), mg.getReturnType().
    toString());
```

Una vez creado el método que se está visitando, se procede a analizar las llamadas que dispone en su código. Para ello, las llamadas pueden ser de distintos tipos de la clase Instruction, que heredan de la superclase InvokeInstruction. Por lo cual, en realidad en este caso se visitan tipos de instrucciones y no métodos. Como se ha redactado en el apartado 4.1.2, la máquina virtual de Java JVM dispone de muchos tipos de instrucciones. Estos son los diferentes tipos de instrucciones que se visitan, para obtener las llamadas del método [Oracle, 2021]:

- `INVOKEVIRTUAL`: invoca el método de una instancia basado en la clase. Proporciona el nombre y el descriptor del método, así como una referencia simbólica a la clase en la que se encuentra el método.

- `INVOKEINTERFACE`: invoca el método de una interfaz. Proporciona el nombre y el descriptor del método de interfaz, así como una referencia simbólica a la interfaz en la que se encuentra el método de interfaz.
- `INVOKESPECIAL`: invoca el método instancia, el cual proporciona un manejo especial para invocaciones de métodos de superclase, privados y de inicialización de instancia. Proporciona el nombre y el descriptor del método, así como una referencia simbólica a la clase o interfaz en la que se encuentra el método.
- `INVOKESTATIC`: invoca un método estático. Proporciona el nombre y el descriptor del método, así como una referencia simbólica a la clase o interfaz en la que se encuentra el método.
- `INVOKEDYNAMIC`: invoca un método dinámico. Proporciona una referencia simbólica a un especificador de sitio de llamada.

Como son de tipo Instruction, no se puede realizar de manera abstracta, por lo que se crea un método por cada tipo de instrucción. La herramienta original de JCG tiene esta implementación ya que a cada método le asigna el tipo. Como la empresa no le interesa esta característica, no se imprime este atributo pero sí que se ha conservado para el futuro.

Cada uno de los métodos, se ejecuta cuando se encuentra una de las instrucciones enumeradas. Cuando se ejecuta dicho método, se realizan dos comprobaciones:

- El método tiene que estar en la lista de paquetes que se quiere analizar, los cuales se han definido por los parámetros de entrada. Para ello, se utiliza el método que dispone la clase JCallGraph (véase el código 6.4).
- El método tiene que ser de la clase que se está analizando. La empresa solo quiere obtener el árbol de llamadas de la misma clase, sin representar las llamadas que se realizan a métodos de otras clases. Sin embargo, como se explica más adelante, sí que se quiere saber por qué otros métodos es llamado.

Una vez superados los filtros, se realiza la transformación de las clases que se disponen a la clase MethodReport, para que se pueda introducir el valor en la lista del mapa. Para obtener los atributos de MethodReport, es necesario disponer de una instancia de MethodGen sobre la instrucción que se llama. Sin embargo, antes no se disponía de esa opción, porque como se ha explicado los datos se presentaban en forma de String. Para

poder construir la instancia de MethodGen, es necesario disponer de las constantes de la clase, lo cual justifica el paso del parámetro ConstantPoolGen que se ha mencionado previamente.

En último lugar, es necesario calcular los métodos que llaman al método actual, para poder formar la lista de los métodos por los que es llamado. Para ello, el propio método no puede calcularlo, ya que tendría que visitar el resto de métodos que se están analizando. En su lugar, cuando se visita un método y este llama a otro, se introduce a sí mismo en como valor el mapa de CalledFromList. Esta clase gestiona un mapa que lo forma un par de claves. La clave (*key*) es el nombre del método que es llamado, y el valor (*value*) es la lista de nombres de métodos que es llamado. No es necesario disponer de información sobre estos métodos, por lo que se guarda únicamente los String. Esta clase sigue la estructura de un patrón *Singleton* y utiliza un HashMap, características que permiten el acceso desde cualquier clase y el acceso a la clave en un tiempo lineal.

Aquí un ejemplo de un método que visita una instrucción, en este ejemplo concretamente de tipo `INVOKEVIRTUAL`:

Código 6.10: Método *visitINVOKEVIRTUAL()*

```
@Override
public void visitINVOKEVIRTUAL(INVOKEVIRTUAL i) {
    if (JCallGraph.isPackage(i.getReferenceType(cp).toString())) {
        if (i.getClassName(cp).equals(this.visitedClass.getClassName())) {
            Method methodAux = this.getMethod(i.getMethodName(cp));
            if (methodAux != null) {
                MethodGen mgAUx = new MethodGen(methodAux, visitedClass.getClassName(), constants)
                MethodReport m = new MethodReport(mgAUx.getName(), mgAUx.getClassName(),
                        mgAUx.getReturnType().toString());
                if (JCallGraph.isExactSubsecuence(i.getReferenceType(cp).toString(),
                        this.visitedClass.getClassName())) {
                    methodCalls.add(m);

                }
            }
        } else {
            cfl = CalledFromList.getCalledfromlist();
            cfl.addToList(i.getReferenceType(cp).toString() + i.getMethodName(cp),
                    method.getPaquete() + method.getNombre());
        }
    }
}
```

Después de visitar todas las instrucciones, se devuelve el mapa que contiene las llamadas del método a ClassVisitor, para que pueda construir su árbol de llamadas.

Código 6.11: Método *start()*

```
public Map<MethodReport, List<MethodReport>> start() {
    method = new MethodReport(mg.getName(), mg.getClassName(), mg.getReturnType().toString());
    if (JCallGraph.isPackage(mg.getClassName()) && JCallGraph.isPackage(visitedClass.getClassName
     ())) {
        if (mg.isAbstract() || mg.isNative())
            return (HashMap<MethodReport, List<MethodReport>>) Collections.<MethodReport, List<
     MethodReport>>emptyMap();
        for (InstructionHandle ih = mg.getInstructionList().getStart(); ih != null; ih = ih.
     getNext()) {
            Instruction i = ih.getInstruction();
            if (!visitInstruction(i))
                i.accept(this);
        }
    }
    HashMap<MethodReport, List<MethodReport>> map = new HashMap<>();
    map.put(method, methodCalls);
    return map;
}
```

6.2.5. Calcular las métricas

Después de terminar la fase que calcula el árbol de llamadas, es necesario calcular las métricas del programa. En este momento de la ejecución del programa, se dispone de las siguientes estructuras:

- `List<HashMap<MethodReport, List<MethodReport>>> methodCalls`: la clase MethodCallsList gestiona una lista de mapas. En cada posición de la lista, se encuentra un mapa que representa el árbol de llamadas de una clase que se ha analizado. Esta estructura se ha generado en base a los resultados obtenidos por la clase ClassVisitor, la cual visita las clases y métodos.

- `Map<String,List<String>> calledMap`: la clase CalledFromList gestiona un mapa que contiene la lista de métodos que llaman a un método.

El módulo CK, se encarga de introducir las diferentes métricas en los objetos que están almacenados en la lista de MethodCallsList. Estos objetos en este momento de la ejecución, disponen de valores nulos en las métricas.

En primer lugar, la clase JCallGraph ejecuta el método principal de Runner, clase principal del módulo CK. Este método precisa de parámetros que permiten realizar un análisis personalizado. Estos valores son algunos que se han recibido en el inicio del programa

por el usuario, y que han sido validados por la clase CliManager. La clase Runner se encarga de invocar a la clase CK para iterar sobre todos los ficheros *.java*. Esta clase, crea un visitante llamado CKNotifier y se calculan todas las métricas que se han especificado. Cada métrica, se calcula utilizando los atributos que dispone el árbol creado conforme a un modelo AST.

Código 6.12: Métricas con código fuente

```
new CK(useJars, maxAtOnce, variablesAndFields).calculate(path, new CKNotifier()
     {
    @Override
    public void notify(CKClassResult result) {
        try {
            writer.printResult(result);
            JCallGraph.addCKMetrics(result);
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
```

La clase Runner, instancia la clase encargada de generar los ficheros de las métricas extra: ResultWriter. Esto se explica en el siguiente apartado.

6.2.6. Exportar métricas auxiliares

Como ya se ha citado en el apartado 4.4, muchas métricas que se calculan con la herramienta CK no se muestran junto al árbol de llamadas. Esto se debe a que por el momento a la empresa no le interesa, ya que añadiría demasiada información. Aún así, ya que el cálculo del resto de métricas no supone un tiempo de ejecución significativo ni disminuye considerablemente el rendimiento, también se muestran en ficheros a parte en la carpeta auxiliar *metrics*. Esto ocurre ya que la creación del árbol AST es la tarea que más recursos consume. Esta tarea se realiza después de calcular todas las métricas y es gestionada por la clase ResultWriter.

Esta clase itera sobre todas las clases y sobre todas las métodos de cada clase, para generar como resultado final dos ficheros CSV. Uno de los ficheros contiene las métricas de todas las clases analizadas y el otro contiene las métricas de todos los métodos analizados. Los dos ficheros aportan información extra, como los nombres, tipos. Esta implementación hace que sea complicado encontrar una clase o método en concreto, incluso usando los

buscadores de campos que tienen Excel u otros programas. Sin embargo, como solo se va a utilizar como información auxiliar en ocasiones puntuales, no se ha cambiado esta funcionalidad. Una vez que se han exportado todos los ficheros, la herramienta CK ha terminado con su trabajo, y se puede continuar con la ejecución en la clase JCallGraph.

6.2.7. Exportar métricas y árbol de llamadas

El programa genera diferentes archivos que el usuario puede analizar una vez que la aplicación finaliza el análisis. Como requisito de la empresa, el formato principal del archivo tenía que ser CSV, ya que era el formato que estaban utilizando antes de realizar este proyecto. Sin embargo, la directora del proyecto indicó como requisito, que en un futuro se tuviese la posibilidad de expandir a diferentes formatos. Para ello, el programa utiliza un patrón *Factory* (5.1.2) para poder crear con un mismo método diferentes tipos de generación de archivos. Gracias a ello, en el caso de que en un futuro se quisiera crear un nuevo método de exportación con otro formato (o otra implementación diferente de un formato), se puede añadir una nueva clase que implemente los métodos de la interfaz *IExporter*.

Para ello, JCallGraph invoca la factoría ExporterFactory para poder crear los diferentes objetos concretos. Esta clase, tiene un condicional con todos los objetos concretos posibles. En función del tipo de nombre de objeto que recibe como parámetro, instancia un producto concreto en específico. Sin embargo, el método constructor devuelve la interfaz, no el objeto concreto creado. De esta manera, se puede abstraer la llamada al objeto concreto.

Código 6.13: Método *createFiles()*

```
public IExporter createFiles(CliManager cli) {
    String exportType = cli.getType();
    if (exportType == null) {
        return null;
    }
    switch (exportType) {
        case "CSVExporter":
            return new CSVExporter(projectName);
        default:
            return null;
    }
}
```

La interfaz IExporter tiene dos métodos, para poder imprimir el árbol de llamadas:

- *create()*: método principal que realiza diferentes funciones:

  - Comprueba que la carpeta *treecall* está creada para poder introducir fichero en ellas. En caso contrario, la crea.
  - Crea una carpeta con el nombre del JAR que se está analizando, dentro de la carpeta *treecall*. En caso de que exista se borra y se crea otra vez. En esta nueva carpeta se introducen los nuevos ficheros que se exportan.
  - Recorre la lista de mapas que contiene el árbol de llamadas de cada una de clases, iterando sobre el árbol de cada clase. Por cada método, se imprimen los siguientes parámetros:
    - *LOC*: número de líneas del método.
    - *Nombre*: nombre completo del método.
    - *Nivel*: nivel del método.
    - *WMC*: complejidad ciclomática del método.
    - *Resultado*: tipo de objeto que devuelve el método.
    - *Linea en clase*: línea en la cual se ubica el método dentro de la clase.
    - *Llamado por*: lista de métodos por lo que es llamada el método en cuestion. El nombre del método de la lista viene compuesto por la clase en la cual se ubica.

    Por cada clase, imprime todos los métodos de nivel 0. Si un método llama o otros métodos; es decir de nivel 1 o más, se llama al método `printChildren()`.

- *printChildren()*: se encaga de imprimir todos los métodos que sean de superiores al nivel 0. Se ha implementado de manera recursiva, ya que aporta claridad al programa, tiene mejor rendimiento y es mejor para solucionar estructuras de árbol.

Código 6.14: Fragmento del método *create()*

```
for (MethodReport method : entry.getValue()) {
    if (map.get(method) != null) {
        JCallGraph.visitedMethods.clear();
        this.printChildren(method, new HashMap<>(map), 1);
    }
}
```

Código 6.15: Llamada recursiva del Método *printChildren()*

```
if (method != null && map != null && !map.isEmpty() && map.containsKey(method))
     {
    for (MethodReport aux : map.get(method)) {
        if (aux != null) {
            printChildren(aux, map, level + 1);
        }
    }
}
```

Cada uno de los productos concretos imprime estos los valores listados antes siguiendo su formato en especício. Por ejemplo, la clase CSVExporter utiliza la clase CSVPrinter de la librería *Apache commons*. A continuación, se muestra un ejemplo de como crea las cabeceras del fichero e imprime los valores en la tabla CSV.

Código 6.16: Ejemplo para imprimir los datos en un CSV

```
csvPrinter = new CSVPrinter(writer, CSVFormat.DEFAULT.withHeader("LOC", "Nombre", "Nivel",  "WMC",
     "Resultado", "Linea en clase", "Llamado por"));
csvPrinter.printRecord(entry.getKey().getLOC(), entry.getKey().getNombre(), 0, entry.getKey().
    getWmc(), entry.getKey().getResultado(), entry.getKey().getLineaClase(), calledFrom);
```

Desde JCallGraph, solo es necesario instanciar la factooría con el nombre del objeto que se quiere. Después, se ejecuta el método `create()` para imprimir los ficheros.

Código 6.17: Método *exportFiles()*

```
private static void exportFiles(String[] args) throws IOException {
    ExportFactory exportFactory = new ExportFactory();
    IExporter exporter = exportFactory.createFiles("CSVExporter", args[0]);
    exporter.create();
}
```

6.3. Comparación entre la herramientas originales y finales

En este apartado se muestra en dos tablas una comparación entre las líneas de código originales frente a las finales. El objetivo de esta apartado es exhibir las clases que se han modificado y qué clases se han añadido para cumplir con los requisitos del programa.

Clase	Propia	LOC totales antes	LOC totales finales	Diferencia (%)
CalledFromList	✓	-	40	-
ClassVisitor	×	99	120	+21,21 %
CliManager	✓	-	83	-
DynamicCallManager	×	125	133	+6,4 %
ExportFactory	✓			-
Instrumenter	×	182	182	-
JCallGraph	×	110	242	+120 %
MethodCallsList	✓	-	41	-
MethodReport	✓	-	112	-
MethodVisitor	×	117	217	+85,47 %
IExporter	✓	-	14	-
CSVExporter	✓	-	114	-

Tabla 6.1: Comparación de líneas de código del módulo *callgraph*

Clase	Propia	LOC totales antes	LOC totales finales	Diferencia (%)
CK	×	127	129	+1,5 %
CKClassResult	×	474	-	-
CKMethodResult	×	344	-	-
CKNotifier	×	7	-	-
CKVisitor	×	1289	-	-
MetricsExecutor	×	50	-	-
ResultWriter	×	228	-	-
Runner	×	53	57	+7,55 %

Tabla 6.2: Comparación de líneas de código del módulo *ck*

El contenido de estas tablas se complementan con las siguientes conclusiones:

- Se han creado un total de **7** clases nuevas.
- Se han modificado un total de **7** clases de los programas originales.
- La cantidad de líneas implementadas demuestran que gran parte del tiempo del proyecto se ha utilizado para la comprensión de los programas, así como la mejora de rendimiento y las pruebas para verificar el correcto funcionamiento.

7. CAPÍTULO

Pruebas

Durante el desarrollo del programa, se fueron realizando pruebas para comprobar su correcto funcionamiento. La fase pruebas se realizó después de finalizar la fase de desarrollo del programa (*Implementación (I)*). La finalidad de estas pruebas es la de ejecutar el programa y comprobar su funcionamiento de manera manual, ya que no suponía mucho esfuerzo su comprobación. Una vez implementado todas las funcionalidades, se tenía planeado realizar una serie de pruebas exhaustivas sobre el programa (*Pruebas funcionales (PF)*) y también con los usuarios de la empresa (*Pruebas con usuarios reales (PU)*). El objetivo de estas pruebas finales fueron: verificar que en diferentes escenarios el programa funciona correctamente, identificar errores y detectar posibles errores futuros. Además de esto, se tenía que verificar que el programa cumplía con ciertos requisitos no funcionales definidos en la planificación: IR-09, IR-10, IR-11, IR-12, IR-13, IR-14 e IR-15.

El capítulo comienza explicando las pruebas que se realizaron durante la implementación del proyecto 7.1. Después, se explican las tareas realizadas en la fase de pruebas: pruebas funcionales 7.2 y pruebas con usuarios de la empresa 7.3. El capítulo se cierra con unas conclusiones que se tuvieron después de realizar las pruebas.

7.1. Pruebas durante el desarrollo

Las pruebas realizadas durante el desarrollo, al no ser unas pruebas muy exhaustivas no podían certificar el funcionamiento correcto del programa. Para ello, se realizó una búsqueda sobre diferentes proyectos Java que se disponen en repositorios públicos, para poder realizar pruebas de compatibilidad del programa. A la hora de realizar esta búsque-

da, solo se tenían en cuenta 3 requisitos:

- El proyecto tenía que ser implementado en Java, o por lo menos que tuviese un gran porcentaje de código en Java. GitHub ofrece este tipo de información, indicando por cada lenguaje de programación qué porcentaje acapara en el repositorio.

- Extenso en número de clases y número de líneas, para realizar pruebas de rendimiento. El objetivo es que sean cifras cercanas a las definidas en los requisitos IR-09 e IR-10.

- Generación de JAR de manera rápida, sin tener que crear un MANIFEST o modificar el proyecto de otra manera. Por consiguiente, se valoraba positivamente los proyectos con el JAR ya creado o que utilizase un gestor de proyectos como Maven o Gradle.

La tabla 7.1 dispone de los siguientes campos para describir cada uno de los escenarios de pruebas utilizados:

- **ID Escenario:** nombre que identifica al proyecto.

- **Propio:** indica si el código del proyecto es propio o se ha obtenido de un repositorio público.

- **N.º ficheros:** los ficheros que analiza la herramienta. Estos ficheros mayoritariamente son Java, pero también incluyen otro tipo de ficheros. Se ha integrado una extensión a GitHub para conocer con exactitud estos datos.

- **LOC:** número de líneas totales analizadas.

- **Descripción:** breve descripción para entender el escenario.

ID Escenario	Propio	N.º ficheros	LOC	Descripción
ProyectoAnalizar	✓	5	37	Primer proyecto de prueba con diseño sencillo
ProyectoAnalizar Extendido	✓	9	104	Proyecto de prueba con estructuras más complejas
Arthas	×	23118	105900	Herramienta de diagnóstico de Alibaba
jOOQ	×	105907	1662758	Herramienta para escribir SQL en Java
SonarReport	✓	2294	1662758	Proyecto realizado en las prácticas de la empresa
TFG	✓	60	5281	El programa que se documenta en esta memoria
CK	×	103	6987	Herramienta CK, análisis de métricas

Tabla 7.1: Escenarios de prueba utilizados en el desarrollo y en la fase de pruebas

Los proyectos *ProyectoAnalizar*, *ProyectoAnalizarExtendido* y *TFG* fueron los proyectos que se utilizaron durante la fase de desarrollo para probar las modificaciones realizadas. Son proyectos que se conoce bien su diseño, por lo que comprobar el resultado de las pruebas era fácil y rápido.

El resto de proyectos junto con estos 3 citados, fueron probados al principio de la fase de pruebas final y en la fase más avanzada de la implementación. Para probar si un proyecto realizaba el análisis de manera correcta, se buscaba de manera aleatoria un método que tuviese muchas llamadas a otros métodos. Se comprobaba que el análisis era el correcto de manera manual. Tras probar todos los proyectos, se pudieron obtener las siguientes conclusiones.

- **Fallos:** durante la ejecución de todos los proyectos no se tuvieron problemas, salvo con el proyecto *SonarReport*. Este proyecto dispone de muchas dependencias, ya que utiliza muchas librerías de terceros. Al analizar este proyecto sin realizar restricciones de análisis de paquetes, se mostraba un error *NullException* ya que visitaba librerías que no se podían visitar. Por consiguiente, para arreglar este error se añadió un control de errores extra al realizar la búsqueda.

- **Problemas para encontrar una buena muestra:** para poder verificar los requisitos IR-09, IR-10 e IR-11, no se han encontrado proyectos adecuados para probar.

Aunque se analicen proyectos grandes como *jOOQ* o *Arthas*, esas líneas de código no son generalmente Java, por lo que no se ha podido verificar el cumplimiento de esos requisitos (solución en el apartado 7.2).

- **Rendimiento:** algunos proyectos presentan un elevado tiempo de ejecución lo cual preocupa, ya que como se ha dicho en el anterior punto no es una muestra válida. Por consiguiente, se tuvo que buscar de una solución para mejorar el rendimiento y reducir el tiempo de ejecución del programa. Para ello, es necesario realizar las pruebas sobre una muestra válida, para poder verificar el requisito IR-11 (solución en el apartado 7.2).

- **Versiones de Java:** tras realizar las pruebas, se han realizado diferentes configuraciones de versiones de Java. Tras probar con todos los proyectos, se ha verificado que la versión mínima que soporta el proyecto es Java 11, ya que esta restricción proviene de JCG. Sin embargo, en la documentación de la herramienta se indica la posibilidad de cambiar esta versión, así que la herramienta no está limitada a esa versión.

- **Versiones de dependencias:** para que la empresa pudiese hacer también las pruebas sobre el programa, se ha realizado un estudio previo sobre las versiones mínimas necesarias para ejecutar el programa. Para ello, se han realizado diferentes pruebas con diferentes versiones, ejecutando en cada escenario todos los proyectos mencionados. El resultado de este análisis es que las únicas dependencias entre versiones que presenta el programa son:

  - Java 11 o superior.
  - Maven 3.6.3 o superior, para poder compilar el proyecto.

En definitiva, las pruebas han sido buenas porque ha permitido obtener bastantes conclusiones sobre el estado del proyecto, pero no lo suficiente para verificar el cumplimiento de los requisitos mencionados. Por ello, se tuvieron que plantear diferentes soluciones, las cuales están redactadas en el siguiente apartado.

7.2. Pruebas funcionales (PF)

Al no poder verificar los requisitos no funcionales con los proyectos propios ni con los proyectos públicos se tuvo que buscar una solución. Esto como no estuvo planificado, generó un desvío en las horas estimadas de la fase de pruebas. En primer lugar, se planteó

la posibilidad de crear proyectos propios de manera manual; es decir, copiar y pegar fragmentos de código muchas veces hasta crear un proyecto grande. Esta idea fue descartada por las siguientes razones:

- Al no tener coherencia el código, no se podría verificar el correcto rendimiento del programa, ya que no se estarían probando las diferentes estructuras Java. Se necesitaría de mucho tiempo para crear un proyecto que tuviese muchas de las estructuras avanzadas de Java.
- No se podrían generar diferentes tipos de árboles de llamadas ni de métricas, porque todo seguiría la misma estructura.
- Para solucionar estos dos problemas se podría generar código más complejo, pero conllevaría una gran carga de trabajo.
- Posibilidad de errores para compilar el programa.

Para dar solución a este problema, se realizó una búsqueda de herramientas que pudiesen facilitar esta tarea. Finalmente, solamente se encontró una herramienta que hiciese esta tarea, y es necesario puntualizar que no fue fácil su búsqueda. Esto se ve reflejado en el desvío de las horas del paquete. La herramienta que se ha utilizado se denomina **Java Bullshifier**

7.2.1. *Java Bullshifier*

- **Versión:** 3.0.0
- **Mantenimiento:** mantenimiento activo en GitHub. 36 *watch*, 349 *star* y 24 *fork*.[1]
- **Creador:** OverOps.
- **Requisitos:**
  - Java 8 o superior.
  - Gradle, versión mínima no especificada.
  - Groovy, versión mínima no especificada.

[1] Java Bullshifier: https://github.com/takipi/java-bullshifier

Herramienta de línea de comandos que permite generar proyectos Java personalizados, a través de diferentes parámetros de entrada. También, dispone de la posibilidad de ser ejecutado en Docker con facilidad. Se utiliza para probar capacidades de monitorización sobre bases de código grandes, con transacciones que van a miles de llamadas de profundidad sobre miles de clases. Entre sus características principales se encuentran:

- Generar proyectos masivos con muchas líneas de código y monitorización (*logging*).
- Lanzar excepciones causadas por el estado de las variables aleatorias.
- Capacidad para generar llamadas de métodos profundas.
- Generación automática de JAR.

En primer lugar se ejecuta el programa indicando los parámetros para personalizar el proyecto que se quiera crear. El programa dispone de varias plantillas por defecto, de las cuales a continuación se presenta una de ellas:

Código 7.1: Plantilla de ejemplo para crear un proyecto con *Java Bullshifier*

```
./gradlew run -Pskip-logic-code \
    -Pname=$bullshifier_name \
    -Poutput-directory=$bullshifier_name \
    -Psubprojects=1 \
    -Pio-cpu-intensive-matrix-size=0 \
    -Pmethods-per-class=1 \
    -Plog-info-per-method=1 \
    -Plog-warn-per-method=0 \
    -Plog-error-per-method=0 \
    -Pbridge-switch-size=4 \
    -Pswitcher-max-routes=200 \
    -Pentry-points=50 \
    -Pclasses=1000 \
     $seedParameter
```

Una vez generado el proyecto, es necesario generar el JAR que contiene todos los ficheros del proyecto. Para ello, se ejecuta el siguiente comando:

Código 7.2: Generar JAR con *Java Bullshifier*

```
gradle fatJar
```

El siguiente fragmento de código corresponde a un método generado aleatoriamente por la herramienta. Como se puede ver, el método carece de sentido, pero sí que dispone de la lógica que tiene que tener un método para que sea compilado.

Código 7.3: Método generado por *Java Bullshifier*

```
public static void method006(Context context) throws Exception
{
    int methodId = 6;
    Integer entryPointNum = Config.get().entryPointIndex.get();     Config.get().updateContext(
     context, entryPointNum, 1, 6);

    // Start of fake locals generator
    Map<Object, Object> root = new HashMap();
    int mapValFfynbisktsa = 282;

    String mapKeyGagqkijwuse = "StrVwgwhwwpnkq";

    root.put("mapValFfynbisktsa","mapKeyGagqkijwuse" );
    int mapValLmxarnyfhix = 112;

    int mapKeyXwwlwtkpeyp = 81;

    root.put("mapValLmxarnyfhix","mapKeyXwwlwtkpeyp" );
    // End of fake locals generator

    String currentTime = dateFormat.format(new Date());
    long currentTimeMillis = System.currentTimeMillis();

    if (Config.get().shouldWriteLogInfo(context))
    {
        logger.info("Class000001.method006 called at {} (millis: {}) (stack-depth: {}) (prev
     method fail rate is: {})",
                currentTime, System.currentTimeMillis(), context.framesDepth, context.
     lastSpotPrecentage);
    }

    if (Config.get().shouldWriteLogWarn(context))
    {
        logger.warn("Class000001.method006 called at {} (millis: {}) (stack-depth: {}) (prev
     method fail rate is: {})",
                currentTime, System.currentTimeMillis(), context.framesDepth, context.
     lastSpotPrecentage);
    }
    // [..]
}
```

7.2.2. Resultados de pruebas exhaustivas

Con la ayuda de esta herramienta se crearon proyectos con diferentes configuraciones, para poder los diferentes requisitos en diferentes escenarios (aproximadamente cada método de una clase está formado por 80 líneas de código).

ID Escenario	N.º clases	N.º métodos por clase	N.º líneas por clase (aprox.[2])	N.º líneas totales (aprox.)	Tiempo (min:s.ms)
Mediano	2	1000	90000	180000	00:10.890
ProyectoGrande	200	1100	100000	20000000	¡ERROR!
ProyectoGrande2	20	1100	100000	2000000	17:56.098
ProyectoExtenso	2000	10	1000	2000000	05:46.981

Tabla 7.2: Tiempos de ejecución sobre proyectos generados por *Java Bullshifier*

Al finalizar las pruebas, se observa que los tiempos de respuesta no son nada satisfactorios. *ProyectoGrande2* estaba muy cerca del límite de los 20 minutos marcado por el requisito IR-11. Los valores de los tiempos variaban entre diferentes ejecuciones, ya que dependen del rendimiento del ordenador. Sin embargo, el proyecto *ProyectoGrande* mostraba un error ya que el programa superaba el máximo de memoria permitido, al superar aproximadamente los 20 minutos de ejecución. A raíz de esto, se estimó que el número máximo de líneas de código que se podían analizar era de **2.500.000**. Para obtener este valor estimado, se analizaron muchos proyectos con diferentes configuraciones.

Código 7.4: Error de memoria Java

```
java.lang.OutOfMemoryError: Java heap space
```

Ya que los resultados no fueron satisfactorios, era necesario buscar soluciones. Al realizar un análisis, se pudo comprobar que había un cuello de botella en el análisis de métricas, mientras que el árbol de llamadas se realizaba en un tiempo casi despreciable. Por consiguiente, el alumno se puso en contacto mediante correo electrónico con el creador de la herramienta CK. En esta ocasión, sí que se tuvo respuesta (a diferencia de los programas del apartado 4.3), y el desarrollador de la herramienta proporcionó posibles soluciones:

- **Paralelizar el programa:** la primera propuesta fue paralelizar el programa mediante hilos (*threads*). Esta opción fue descartada de inmediato, ya que complicaba mucho la implementación y además no se tenían garantías de que fuese la solución correcta.

- **Reducir el número de métricas:** la herramienta CK calcula muchas métricas, de las cuales la mayoría no se van a utilizar. Por ello, el alumno le comentó la posibilidad al desarrollador de reducir el número de métricas. Sin embargo, el desarrollador explicó que lo que de verdad consume recursos es la generación del árbol AST, por lo que reducir el cálculo de métricas no cambiaría mucho el rendimiento del programa.

- **Cálculos precisos:** el programa CK tiene habilitada por defecto una opción que realiza más comprobaciones sobre diferentes estructuras para obtener cálculos más precisos. Esta fue otra de las soluciones planteadas por el desarrollador del programa. Tras realizar varias pruebas activando y desactivando la opción, se obtuvieron las siguientes conclusiones:

  - Estos cálculos precisos afectaban solo a algunas métricas, entre las cuales no se encontraba ninguna de las métricas importantes para la empresa.
  - El tiempo de ejecución no se reducía demasiado. En alguna ocasiones, la diferencia era de un par de segundos, mientras que en la mayoría no existía casi diferencia.

  Por ende, esta opción tampoco fue valida, ya que los tiempos de ejecución no se reducían.

- **Variables y campos:** presentadas las anteriores soluciones, el desarrollador del programa no proporcionó ninguna solución más. Por lo tanto, el alumno analizó el código del programa en profundidad, con el objetivo de encontrar alguna solución. Después de realizar una búsqueda en profundidad, se observó que el programa también calculaba métricas sobre variables y campos de una clase. Esta opción no era de interés para la empresa, por lo que se podía descartar su uso. Al desactivar esta opción, el rendimiento del programa mejoró considerablemente. Acto seguido, se verificó con el desarrollador del programa que al desactivar esta opción no generaba cambios en el resto de métricas. El desarrollador verificó este comportamiento, por lo que la solución fue la correcta.

Después de desactivar la opción, los tiempos de respuestas finales son los siguientes:

ID Escenario	Tiempo antes (min:s.ms)	Tiempo después (min:s.ms)	Diferencia (min:s)
Mediano	00:10.890	00.05.234	- 0.5
ProyectoGrande	¡ERROR!	05:59.054	- 5.59
ProyectoGrande2	17:56.098	00:47.399	- 17.09
ProyectoExtenso	05:46.981	00:35.892	- 5.11

Tabla 7.3: Tiempos de ejecución después de desactivar análisis de variables y campos

En definitiva, el programa final puede analizar proyectos enormes gracias a desactivar la opción de análisis de campos y variables que tenía por defecto la herramienta CK. También, se ha eliminado la restricción de 2.5000.000 de líneas de código, aunque lo más probable es que exista un límite pero mucho más grande. Realizando diferentes pruebas, no se ha alcanzado dicho límite.

7.3. Pruebas con usuarios reales (PU)

Durante la fase de *Pruebas funcionales (PF)*, en paralelo la empresa realizaba pruebas sobre el programa final, indicando todos los errores que se encontraban y propuesta de cambios para mejorar el programa. Concretamente, las pruebas fueron realizadas por Beatriz Pérez. Para realizar las pruebas, se creó una guía de uso para que entendiese bien el funcionamiento de la herramienta (Véase el Anexo B). En la Tabla 7.5 se resumen todos los errores y en la Tabla 7.4 comentarios realizados por Beatriz Pérez, para mejorar el programa.

Las tablas, también indican el coste de cada una de ellas respecto al tiempo necesario para realizar el cambio. Los valores posibles son: *Bajo*, *Medio* y *Alto*.

Comentario	Solución	Coste
Dentro de la carpeta de los ficheros generados, crear una carpeta por cada proyecto analizado	Se crea una nueva carpeta con el nombre que tiene el JAR que se analiza	*Bajo*
La métrica LOC tiene que ser la primera columna del CSV que se exporta	Modificar las cabeceras del fichero en CSVExporter	*Bajo*
Cambiar el nombre de la carpeta *csv*, donde se exportan todos los ficheros del análisis	Se ha modificado el nombre de la carpeta por *treecall*, nombre proporcionado por la directora	*Bajo*
Incluir la carpeta *metrics* dentro de la carpeta generada en *treecall*. De esta manera, se pueden consultar las métricas generales por cada análisis realizado	Se ha modificado la dirección donde se creaba la carpeta de *metrics*	*Bajo*

Tabla 7.4: Comentarios realizados por la empresa para mejorar el programa

Salida esperada	Salida real	Solución	Coste
Analizar proyectos enteros sin tener que indicar ningún paquete al programa	El programa funciona pero no analiza ningún fichero del JAR	Se ha modificado el comportamiento del parámetro de entrada que indicaba qué proyectos se tienen que analizar. Ahora, sin introducir ningún parámetro se analiza todo el proyecto	*Bajo*
Los nombres de los métodos son únicos en el fichero de salida	Algunos métodos disponen de texto extra en su nombre. Por ejemplo, “metodoA/1” y “metodoA/2”	Cambiado el formato en el que se escriben algunos métodos, a raíz del análisis sobre bytecode. Como se ha comentado previamente, el análisis de bytecode puede utilizar formatos diferentes para el mismo método	*Alto*

Tabla 7.5: Errores identificados por la empresa

Después de solucionar estos errores y mejorar el programa con los comentarios, se ha conseguido un programa que está más adecuado con la idea inicial de la empresa.

7.4. Conclusiones de las pruebas

Gracias a la fase de pruebas el programa ha mejorado en los siguientes aspectos:

- **Robustez ante fallos:** el programa ya no presenta fallos después de haber probado el mismo con muchos programas de diferentes arquitecturas, implementación y diseño. Sin embargo, no se puede garantizar al 100 % su robustez ante fallos.
- **Rendimiento:** se ha mejorado considerablemente el rendimiento del programa, siendo ahora los tiempos de ejecución del programa muy bajos.
- **Interacción persona-computador:** se ha mejorado el *feedback* con el usuario que utiliza el programa.

Para conseguir estas mejoras, se ha necesitado más tiempo de lo estimado, a raíz del escaso rendimiento que tenía el programa y la dificultad de probar los requisitos no funcionales definidos. En la siguiente tabla, se resume las observaciones realizadas sobre los requisitos no-funcionales que se han enumerado al comienzo de este capítulo.

Cada requisito se ha calificado con una nota: REALIZADO si se ha podido verificar el requisito y NO REALIZADO en caso contrario.

IR	Calificación	Comentario
IR-09	REALIZADO	Se han realizado pruebas con proyectos que tienen más de 100.000 líneas de código en una clase (p. ej.: *ProyectoGrande*)
IR-10	REALIZADO	Se han realizado pruebas con proyectos que tienen más de 2000 clases (p. ej.: *ProyectoExtenso*)
IR-11	REALIZADO	Después de ejecutar muchos proyectos, los cuales muchos de ellos superan los requisitos establecidos, ninguno ha superado el máximo de 20 minutos. Sin embargo, en el caso de que se ejecuten proyectos excesivamente grandes, siempre se puede limitar el análisis con los filtros que se pasan como parámetro de entrada
IR-12	REALIZADO	La herramienta JCG y CK analizan proyectos Java superiores a la versión 1.8, por lo que 1.6 inclusive
IR-13	NO REALIZADO	No se han realizado pruebas con personal de la empresa que no tuviese información previa sobre el proyecto. A pesar de que Beatriz Pérez sí que pertenece a la empresa, tenía conocimiento previo sobre el proyecto
IR-14	REALIZADO	Se ha obtenido *feedback* de Beatriz Pérez, el cual ha sido positivo
IR-15	NO REALIZADO	No se han especificado casos de prueba sobre las métricas, ya que en un principio las métricas se iban a calcular con una implementación propia. La herramienta CK, dispone de métricas propias y al estar en constante mantenimiento, no es necesario realizar un control sobre sus resultados

8. CAPÍTULO

Seguimiento y control del proyecto

Las tareas de *Seguimiento y control (SC)* se han realizado durante todo el ciclo de vida del proyecto. En este capítulo, se documenta la segunda planificación que se realizó a raíz de las desviaciones del proyecto (8.1). Después, se analiza el control realizado sobre diferentes ámbitos del proyecto, los cuales se planificaron de antemano en la primera planificación: alcance (8.2), riesgos (8.3), calidad (8.4) y tiempo (8.5).

8.1. Segunda planificación

El día **27/05/2021**, se realizó una reunión para modificar la planificación inicial del proyecto, para poder cerrar el mismo de una manera adecuada. Como ya se ha reiterado previamente en la memoria, el proyecto disponía gran ambigüedad respecto a las tecnologías y herramientas utilizadas. Esto generó que ciertos riesgos no se cumpliesen generando así una desviación en el tiempo del proyecto. Estas desviaciones se comentan en este capítulo. Para esta planificación, se ha recortado el alcance del proyecto, y teniendo en cuenta el nuevo alcance se ha realizado una nueva planificación. En esta nueva planificación, se detallan las partes modificadas de la primera planificación y se redacta la planificación restante del proyecto. Todo ello con mucha más información que se disponía en un principio, por lo que se espera un mejor resultado que la anterior.

8.1.1. Gestión del Alcance

Como en un principio era desconocido el alcance del proyecto, se creó un alcance ambicioso con la mayor cantidad de funcionalidades que deseaba el cliente (directora del

proyecto). Por ende, se ha recortado el alcance principal para poder acabar el proyecto con el objetivo de no tener una gran desviación en las horas planificadas. A tener en cuenta, que a la hora de realizar esta planificación, se dispone del siguiente prototipo: una herramienta que obtiene el árbol de llamadas de un proyecto, junto a un filtrado por métricas (véase el apartado 3.2.3).

Como se puede observar, este prototipo es una mezcla entre el Prototipo 1 y 3, excluyendo el Prototipo 2. Esto se debe a que durante el desarrollo del proyecto, la directora del proyecto decidió que tenían más prioridad las métricas que la visualización con diferentes métodos. La visualización gráfica a pesar de estar redactado como un requisito, nunca fue prioritario porque era necesario disponer de una herramienta funcional y bien probada. Por ello, se decidió suprimir esa funcionalidad del proyecto. También, se ha modificado parte del Prototipo 3, ya que se decidió añadir las métricas especificadas por la directora del proyecto, sin la necesidad de obtener información de una herramienta de terceros (por ejemplo, SonarQube como se especificó en la primera planificación).

Niveles de árbol de llamadas: la herramienta seleccionada JCG (véase en el apartado 4.3.8) fue una buena elección ya que realiza bien su cometido. Sin embargo, la directora especificó que quería incluir la jerarquía de niveles de los métodos. Estos niveles indican cuantas llamadas se han realizado con anterioridad para llegar a ese método. Por ejemplo, si el método A llama al método B, el método A es de nivel 0 mientras que el método B es de nivel 1. Para obtener esta información, se tuvieron que realizar ciertas modificaciones en el programa, siendo estas modificaciones difíciles de aplicar ya que era complicado comprender el funcionamiento del programa. Además, cuando el programa tiene que exportar la información de estos niveles en orden a un fichero, supuso una tarea complicada que no se tenía prevista. Esta tarea fue problemática, ya que se buscó la mayor eficiencia posible, por lo que se realizó de manera iterativa. Por consiguiente, la suma de la complejidad, el nuevo requisito de métodos por niveles y la exportación de los datos a ficheros, conllevaron más tiempo de lo estimado.

Problema en el análisis de métricas: una de las mayores desviaciones que sufrió el proyecto se produjo por el análisis de las métricas. En la primera fase del proyecto, cuando se realizó un estudio breve de las métricas (paquete *Alternativas de métricas (AM)*), se estudió la posibilidad de utilizar la herramienta JCG (véase en el apartado 4.3) para analizar las métricas durante el análisis. De esta manera se analizan las llamadas del gráfico de llamadas y las métricas a la vez, reduciendo así el tiempo de computación. Sin embargo, a la hora de implementar esto en la fase de desarrollo, se descubrió que con el análisis de

bytecode es imposible de analizar ciertas métricas (entre ellas, la complejidad ciclomática que es uno de los requisitos del proyecto). Esto se debe a que el *bytecode* dispone de más de 200 instrucciones, pero ninguna de ellas puede determinar el inicio de un método, ni tampoco los condicionales. Estas carencias imposibilitan la obtención de métricas que precisan de esa información. Esta información no se comprobó en la fase de análisis de las alternativas, por lo que este supuesto no fue correcto. Para solucionar este error, se tuvo que realizar una nueva búsqueda de alternativas (véase el apartado 4.4). Por ende, se tuvo que descartar esa posibilidad y se buscaron alternativas. En el apartado 4.4 se ve reflejado dicho análisis.

Requisitos

Como se ha recortado el alcance, los requisitos también se han modificado. Esto supone que se han eliminado ciertos requisitos porque no se dispone del tiempo para completarlos, y se han añadido nuevos requisitos a medida que se han obtenido más conocimientos. Se han eliminado los siguientes requisitos:

- **IR-04:** El sistema permitirá crear y modificar el diseño de la aplicación que recibe como parámetro de forma gráfica, y comprobará su viabilidad.
- **IR-05:** El sistema mostrará los datos que ha obtenido como resultado de manera gráfica, mediante distintos tipos de gráficos.
- **IR-07:** El sistema permitirá filtrar los gráficos con diferentes filtros.

Por otro lado, se ha añadido el siguiente requisito no-funcional. Este requisito no es definido por la empresa, sino que tiene que ser definido por la desviación que se ha definido en el anterior apartado.

- **IR-21:** El sistema recibe código no-compilado como parámetro de entrada y analiza las métricas de los métodos.

Descomposición de tareas

A raíz del recorte del alcance, el EDT también se ve afectado. Este cambio es pequeño, ya que solo se tuvieron que eliminar los 2 últimos prototipos.

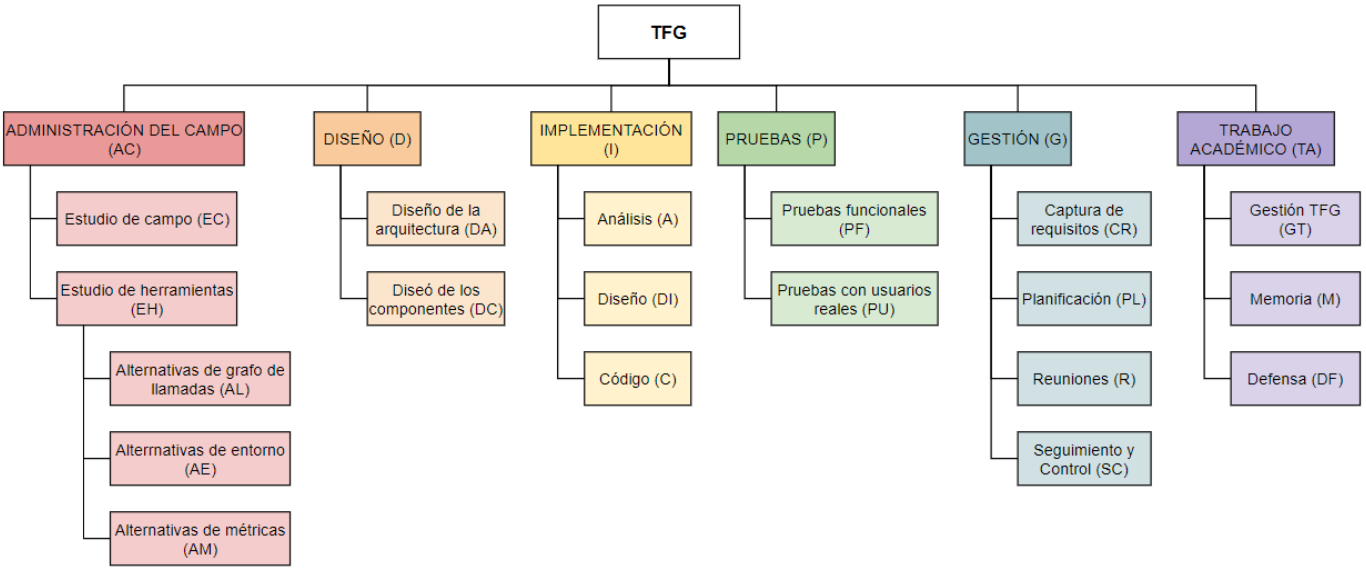

Figura 8.1: Diagrama EDT

Las diferencias de los paquetes respecto al primer EDT son las siguientes:

- **Administración del campo (AC):**
  - *Alternativas de métricas (AM)*: el objetivo del paquete es modificado ya que se tiene que realizar un análisis con un enfoque diferente al realizado en la primera fase. En lugar de utilizar la herramienta que realiza el árbol de llamadas (JCG), ahora es necesario utilizar una herramienta que utiliza código fuente para analizar las métricas.
- **Implementación (I):**
  - *Análisis (A)*: analizar las alternativas que se han encontrado en *AC*, para implementarlas en el proyecto. Como se ha acotado el alcance, solo es necesario analizar las siguientes funcionalidades: grafo de llamadas y métricas.
  - *Diseño (DI)*: diseñar el diagrama de clases para implementar las funcionalidades del nuevo alcance.
  - *Código (C)*: implementar en Java las funcionalidades del nuevo alcance.

Diagrama de Gantt

El diagrama de Gantt sí que ha sufrido grandes cambios. El diagrama que se ha realizado en esta nueva planificación comienza el día que se realiza esta segunda planificación.

Por consiguiente, no tiene en consideración las desviaciones que ha sufrido la primera planificación. Estos cambios se redactan en el apartado 8.2.

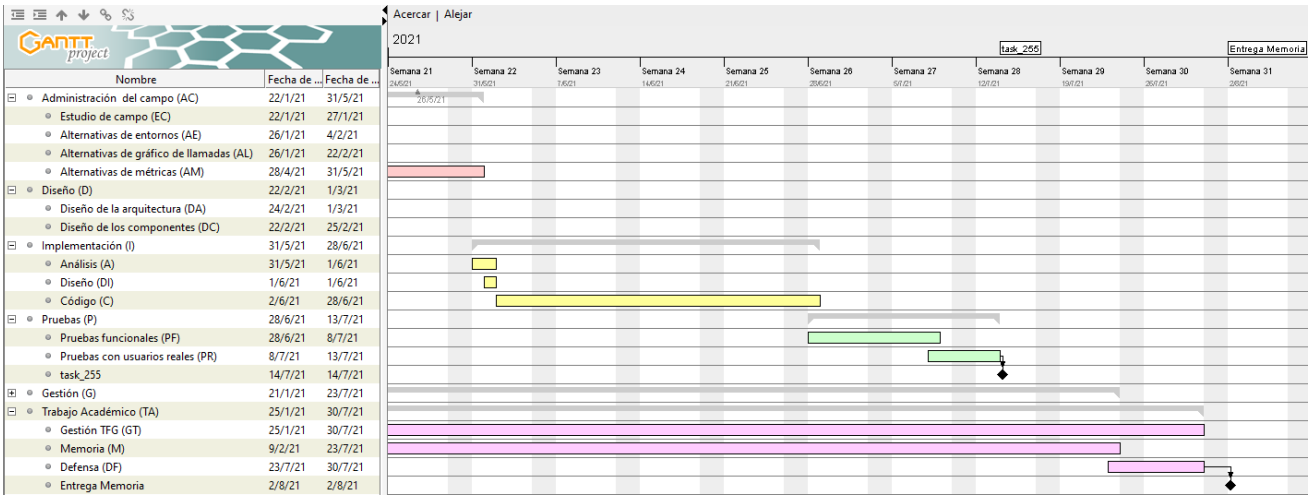

Figura 8.2: Diagrama de Gantt del proyecto (Segunda planificación)

8.1.2. Gestión del Tiempo

A raíz de todas las sorpresas e imprecisiones, otro de los aspectos que más cambios tiene es la gestión del tiempo. A día 27 de mayo, no se han cumplido los tiempos establecidos, por lo que también se ha realizado una nueva planificación del tiempo que restaba del proyecto. Esta planificación tiene en cuenta el estado del proyecto, que es el siguiente:

- El primer prototipo está terminado y probado. Aun así, se tiene que finalizar la tarea de añadir más métricas, probar esas métricas y probar todo el proyecto en su conjunto. A raíz de la experiencia adquirida, se tiene consciencia que las pruebas son muy costosas, por lo que es importante tener ese aspecto en cuenta para la planificación.

Teniendo todo esto en cuenta, se han establecido las siguientes fechas para la finalización de las tareas restantes.

- **27/05/202 - 22/06/2021:** finalizar la tarea de las métricas, y comprobar su correcto funcionamiento. Realizar primera prueba con la empresa.
- **23/06/2021 - 29/06/2021:** arreglar los posibles problemas restantes, modificar las posibles correcciones que presente la empresa y probar su correcto funcionamiento.

- **30/06/2021 - 13/07/2021:** realizar pruebas exhaustivas, cumpliendo todos los requisitos presentados por la empresa. Continuar redactando la memoria del proyecto de manera paralela.

- **14/07/2021 - 27/07/2021:** última demostración a la empresa y finalización de la parte de implementación.

A la hora de realizar la planificación del tiempo restante, el proyecto ya ha consumido **260h** de las 346h estimadas. Por ello, se tiene en cuenta el tiempo transcurrido y se planifica el tiempo a futuro, pero no se tiene en cuenta las desviaciones que las tareas cumplidas hayan sufrido. Por consiguiente, el tiempo final estimado podía ser incrementado.

BLOQUE DE TRABAJO	TAREA	DEDICACIONES
ADMINISTRACIÓN DEL CAMPO (AC)	*Estudio de las herramientas (EH)*	*4h*
	Alternativas de métricas (AM)	4h
	SUBTOTAL	**4h**
IMPLEMENTACIÓN (I)	Análisis (A)	2h
	Diseño (DI)	2h
	Código (C)	20h
	SUBTOTAL	**24**
PRUEBAS (P)	Pruebas funcionales (PF)	12h
	Pruebas con usuarios reales (PR)	3h
	SUBTOTAL	**15h**
GESTIÓN (G)	Reuniones	3h
	SUBTOTAL	**3h**
TRABAJO ACADÉMICO (TA)	Memoria (M)	60h
	Defensa (DF)	6h
	SUBTOTAL	**66h**
HORAS TOTALES		**112h**

Tabla 8.1: Segunda planificación de horas estimadas para cada tarea del proyecto

Esta nueva planificación estima un total de **372h**, lo que supone un incremento de 126 horas (+7.5 %≈) respecto a la primera planificación. Esto no supone una desviación muy grande sobre la planificación inicial, ya que desde un principio se planificó todo teniendo diferentes riesgos en cuenta.

8.2. Control del alcance

Como está redactado en el Capítulo 3, desde un principio se tenía en cuenta que el alcance podía ser modificado. Como ya se ha explicado en el apartado 8.1 el alcance de la primera planificación fue modificado, pero incluso entre el resultado final y esta nueva planificación también se encuentran diferencias. Estas desviaciones han hecho que el tiempo estimado para finalizar el proyecto fuese incrementado, como se ha analizado en apartado 8.5.

Respecto a la segunda planificación, en la fase de pruebas también ocurrieron desviaciones para poder probar bien la herramienta (véase el Capítulo 7). Esto se debe a que no se podían probar bien las funcionalidades y se tuvo que realizar una búsqueda de un programa de terceros para poder crear proyectos que permitiesen realizar unas pruebas válidas. El aprendizaje de uso de la herramienta y el análisis de los resultados obtenidos de los proyectos creados por la herramienta, originaron desviaciones en el tiempo y alcance del proyecto.

A continuación se citan las tareas que han sido modificadas respecto a la segunda planificación.

- **Pruebas (P)**
  - *Pruebas funcionales (PF)*: para verificar los requisitos de la empresa, se ha buscado una herramienta que permite realizar muestras válidas para realizar pruebas en ellas. Para utilizar este programa, en primer lugar se tiene instalar y después su funcionamiento, para poder realizar mejores muestras.
  - *Pruebas con usuarios reales (PU)*: no se han podido realizar pruebas con el personal de la empresa, pero sí que se han podido hacer con la directora del proyecto.
- **Trabajo académico:**
  - *Memoria (M)*: búsqueda de información para explicar la introducción y los antecedentes del proyecto. Para ello, ha sido necesario leer varios artículos.

8.3. Gestión de los riesgos

En la primera planificación del proyecto, se había realizado un plan de acción por cada uno de los riesgos que podían ocurrir. Gracias a esa planificación previa, cuando el proyecto

ha tenido una incidencia se ha podido estimar su duración y seguir un plan de acción para mitigar su impacto en el proyecto. Durante las diferentes fases del proyecto, han ocurrido alguno de estos riesgos.

En este apartado, se enumeran que riesgos se han transformado finalmente en incidencias del proyecto, cuál ha sido su impacto real y qué se ha hecho para solucionarlo.

8.3.1. R1-Compatibilidad del proyecto con el curso académico

Este riesgo se tuvo presente durante todo el proyecto, ya que su probabilidad era bastante elevada. Durante los meses comprendidos entre enero y mayo, no supuso ningún problema el trabajo académico y no se tuvieron desviaciones considerables. Sin embargo, en junio el proyecto estuvo parado durante una semana porque era necesario terminar un trabajo de una asignatura del grado. Este problema se comunicó al grupo y se reforzó con trabajo extra las siguientes semanas.

Este trabajo era necesario terminarlo cuanto antes, ya que si no se calificaba como aprobado no se podía aprobar una asignatura, imposibilitando así la finalización del grado en el mes de septiembre.

8.3.2. R2-Captura de requisitos

La captura de requisitos se ha realizado en dos fases. Primero, se realizó una captura de requisitos en la fase inicial del proyecto, cuando no se había empezado con la búsqueda de información. Posteriormente, cuando ya se había seleccionado la herramienta de JCG, se realizó otra reunión para la captura de requisitos. La captura de requisitos fue buena, ya que se han tenido que añadir pocas funcionalidades extra durante el desarrollo del proyecto.

Sin embargo, uno de los requisitos que dio la empresa no se entendió del todo bien. El alumno, había entendido mal un requisito definido por la directora. El requisito en cuestión es el IR-02, del cual no se entendió bien el concepto de “los métodos por niveles”. Sin embargo, esto no tuvo mucha repercusión en el proyecto, ya que gracias a las reuniones periódicas, la directora del proyecto observó que este requisito no se había entendido bien. Esto supuso una desviación de 1 semana del proyecto. Con esta incidencia el alumno ha aprendido a definir mejor los requisitos, y se ha podido comprobar la eficacia que tienen las reuniones periódicas.

8.3.3. R3-Uso de tecnologías desconocidas

El programa final utiliza dos herramientas totalmente diferentes entre sí. Por consiguiente, se ha tenido que estudiar cada una de las herramientas por separado. El estudio de la herramienta JCG conllevó más tiempo de lo estimado, ya que hace uso de la librería BCEL es muy compleja. Además, era necesario realizar cambios sobre la herramienta original, por lo que era de suma importancia entender bien el funcionamiento de la misma. Esto, sumado a que no se podía ver la implementación de muchas tareas que realiza el programa ya que el código es de la librería, originó un pequeño desvió respecto a la planificación inicial. Este desvió no ha repercutido mucho en el desarrollo del proyecto ya que ha sido casi inexistente (menos de 1 semana).

Este riesgo era uno de los que se sabía que podían ocurrir, pero finalmente no ha generado grandes desvíos. Además, se ha disminuido bastante su impacto gracias a la ayuda de las directoras del proyecto, ya que han aportado su experiencia para comprender mejor las herramientas.

8.3.4. R5-Cambio de herramienta

Una de las mayores incidencias del proyecto ha sido ocasionada por no comprender el perfecto funcionamiento de una tecnología. A pesar de realizar un estudio en profundidad de las herramientas, no se estudió en la suficiente profundidad el funcionamiento de la máquina virtual Java (JVM). Como está redactado en el apartado 8.1, la idea original era utilizar la herramienta JCG para calcular las métricas a su vez que se generaba el árbol de llamadas. Sin embargo, esto no es posible ya que el *bytecode* no almacena la información suficiente para realizar estos cálculos.

Como consecuencia, se tuvo que buscar una herramienta extra para realizar el cálculo de las métricas (CK). La suma del tiempo empleado para implementar las métricas propias (que finalmente no se ha utilizado), la búsqueda de una nueva herramienta y la implementación junto a esta nueva herramienta, acapararon un total de 4 semanas. Esta incidencia deja una duda al alumno, ya que sí que se realizó un estudio en profundidad antes de utilizar las herramientas, y no se utilizó más tiempo porque si no se alargaba mucho la fase de adquirir los conocimientos. Beneficiándose de esta situación, el alumno ha adquirido experiencia para futuros proyectos, ya que se tendrán en consideración estas incidencias. Por ello, en el futuro es importante definir bien los planes de acción, porque este tipo de incidencias siempre pueden ocurrir.

8.3.5. R7-Versiones de las herramientas

Cuando se integró la herramienta CK, el proyecto mostraba un error a la hora de compilar el programa. Este error, solo se daba en un ordenador auxiliar que se utilizaba para las pruebas, pero no se daba en el ordenador principal donde se desarrollaba el programa. La comprobación de las versiones, se dejó para una fase más avanzaba, ya que no imposibilitaba el desarrollo del proyecto.

Para que la empresa no tuviese problemas a la hora de utilizar la herramienta, se realizó un estudio sobre la compatibilidad de las diferentes tecnologías y herramientas que utiliza el programa. Esta tarea se llevó a cabo cuando se redactó la guía de uso de la herramienta. Para probar las versiones, era necesario descargar muchas veces la misma herramienta, pero con una versión diferente, tarea que en algunas tecnologías/herramientas era tediosa ya que se necesitaba borrar la versión anterior. Una vez realizada las comprobaciones, se identificaron las dependencias que sufría el programa y se identificó la tecnología que originaba el error anteriormente comentado (en este caso, el error era la versión 9 del JDK de Java). Esta incidencia solo supuso un retraso de un par de días.

8.3.6. R8-Planificación incorrecta

Durante toda la memoria, se ha repetido constantemente que la primera planificación fue muy ambiciosa desde su origen, ya que el alcance planteado estaba definido teniendo en cuenta muchos supuestos. Como se había previsto, el alcance era muy difícil de llevar a cabo, lo que conllevó a realizar una segunda planificación en una fase avanzada del proyecto.

La redacción de esta nueva planificación, conllevó el retraso de 3 días en el proyecto. Sin embargo, como se ha podido comprobar gracias a la buena identificación de los riesgos y al cumplimiento de los planes de acción, no ha supuesto una incidencia muy grave en el proyecto. Además, la empresa también sabía que era un proyecto muy ambicioso y están satisfechos con el resultado.

8.4. Control de la calidad

En la primera planificación también se gestionó la calidad del proyecto. Durante todo el proyecto, se han tenido en cuenta los criterios recopilados en ese apartado, para poder mantener la calidad del proyecto en el futuro. A continuación, se realiza un análisis sobre los criterios definidos:

Criterio de calidad	Evaluación	Comentario(s)
Escalabilidad de los datos	*Excelente*	Se exportan todos los resultados del análisis a diferentes ficheros
Modificabilidad/ Mantenibilidad	*Excelente*	El programa está diferenciado en dos módulos que separan las dos herramientas utilizadas. Cada clase añadida ha seguido las indicaciones para crear un software mantenible, siguiendo varios patrones de diseño
Usabilidad	*Buena*	No se ha utilizado una interfaz gráfica, pero sí que se tiene una interacción con el usuario mediante la línea de comandos. Se ha creado una guía de uso y la terminal proporciona ayuda al usuario mostrando los comandos que puede utilizar
Entregables	*Excelente*	Las actas, la memoria y la guía de uso están bien documentadas. Las actas siguen una plantilla y cumplen los requisitos para dejar constancia de una reunión, lo que permite tener una trazabilidad del proyecto

La tabla refleja que se han seguido los criterios de calidad que se definieron, teniendo como resultado un proyecto de una alta calidad según los criterios definidos.

8.5. Control del tiempo

Como consecuencia de la primera planificación y los incidentes que ha tenido el proyecto, no se han cumplido con los tiempos estimados ni tan poco las fechas estimadas de las diferentes fases del proyecto. Tampoco se han cumplido las estimaciones realizadas en la segunda planificación. En primer lugar, se realiza una comparativa entre las desviaciones entre las fases, y después entre los paquetes que forman el proyecto.

8.5.1. Desviaciones en las fases

No se puede realizar una comparación sobre las fases de la primera planificación porque esas fases fueron planificadas para los 3 prototipos. Sin embargo, en la segunda planificación sí que se definieron unas fechas límites las cuales sí que se cumplieron. Esto se debe a que el tiempo que era necesario planificar era reducido, y además si no se cumplían con las estimaciones no sería posible terminar el proyecto a tiempo.

8.5.2. Desviaciones en los paquetes

En este apartado, se presentan las desviaciones que han tenido los paquetes y las tareas del proyecto. Para explicar las desviaciones se muestra en la Tabla 8.2 la comparativa entre las horas estimadas y las horas finales, y se acompaña esta tabla con dos gráficos para ilustrar mejor los resultados.

Paquete de trabajo	Duración estimada	Duración real	Diferencia
Administración del campo (AC)	**49**	**57**	+8
Estudio de campo (EC)	4	8	+4
Alternativas de gráfico de llamadas (AL)	30	35	+5
Alternativas de métricas (AM)	10	13	+3
Alternativas de entornos (AE)	5	1	−4
Diseño (DI)	**11**	**8**	−3
Diseño de la arquitectura (DA)	5	4	−1
Diseño de los componentes (DC)	6	4	−2
Implementación (I)	**127**	**147**	+10
Análisis (A)	22	20	−2
Diseño (DI)	11	8	−3
Código (C)	94	109	+15
Pruebas (P)	**35**	**44**	+9
Pruebas funcionales (PF)	32	40	+8
Pruebas con usuarios reales (PR)	3	4	+1
Gestión (G)	**44**	**52**	+8
Captura de requisitos (CR)	5	5	0
Planificación (PL)	20	25	+5
Reuniones (R)	15	18	+3
Seguimiento y control (SC)	4	4	0
Trabajo académico (TA)	**80**	**121**	+41
Gestión TFG (GT)	3	2	−1
Memoria (M)	70	114	+44
Defensa (DF)	7	5	−2
Proyecto	**346**	**419**	+73

Tabla 8.2: Comparativa de las horas estimadas y las horas reales de las tareas

En la tabla, el paquete *Implementación (I)* se representa sumando la suma de las horas planificadas para los 3 prototipos de la primera planificación.

Se puede observar, que el proyecto se ha desviado aproximadamente un **21** % respecto a la primera planificación y un **12.63** % respecto a la segunda. Como se lleva explicando

en todo el capítulo, esta desviación se debe al cambio del alcance y las incidencias del proyecto. En futuros proyectos se espera realiza mejor las planificaciones de los tiempos, a medida que se adquieran conocimientos sobre los diferentes aspectos que conciernen a los proyectos informáticos.

A continuación se encuentran dos gráficas (8.3 y 8.4), que resumen las desviaciones de las horas y de las tareas. De esta manera, se puede visualizar mejor la diferencia entre la planificación inicial con el resultado final.

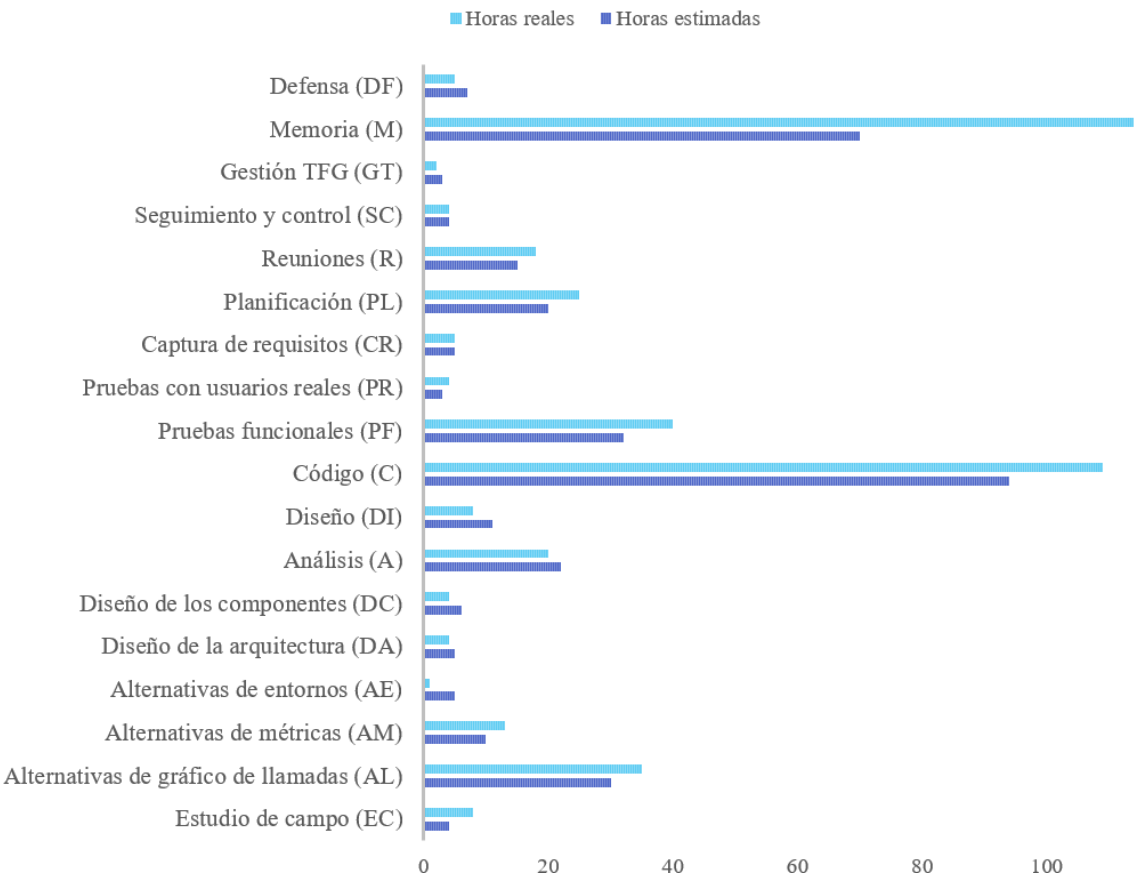

Figura 8.3: Gráfico de las horas estimadas y las horas reales de tareas

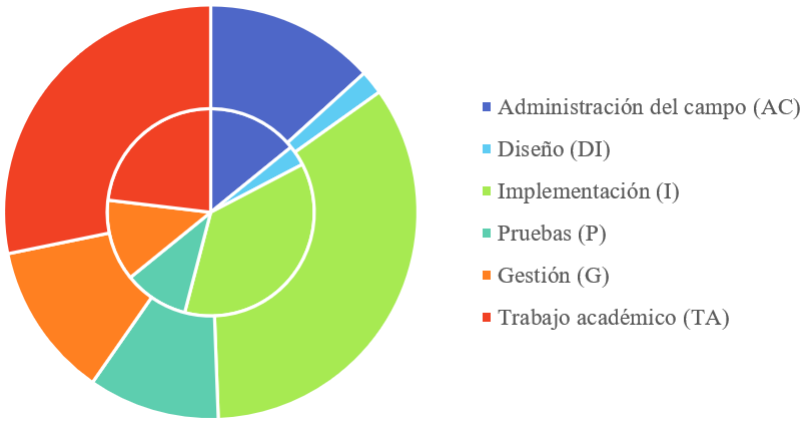

Figura 8.4: Gráfico de las horas estimadas y las horas reales de paquetes

9. CAPÍTULO

Conclusiones y líneas futuras

En este último capítulo, para empezar se enumeran las conclusiones que se han obtenido al finalizar el TFG, tanto a nivel técnico como personal (9.1). Junto a las conclusiones, también se enumeran las posibles mejoras que podría tener el proyecto (9.2). Estas mejoras se han identificado desarrollando el proyecto, pero no se han podido aplicar por la falta de tiempo. Para finalizar, se indican las posibles líneas de futuro del proyecto que se podría hacer o bien en otro TFG o bien por los desarrolladores de la empresa (9.3).

9.1. Conclusiones

Este TFG fue planteado con el objetivo principal de obtener una herramienta que generase un árbol de llamadas junto a unas métricas, para poder realizar un análisis sobre proyectos de la empresa. Este objetivo se ha cumplido con creces, a pesar de las incidencias e imprevistos que ha tenido el proyecto.

Conclusiones técnicas

En general, la valoración técnica del proyecto es positiva. Se han desarrollado los requisitos principales de la empresa, aunque no se han podido realizar los requisitos más secundarios. Los objetivos técnicos que se han alcanzado durante el TFG son los siguientes:

- **Herramienta que genera un árbol de llamadas:** se ha implementado un programa que genera un árbol de llamadas a partir de un proyecto Java, haciendo uso de

una herramienta que está bien valorada por la comunidad. El árbol permite al usuario conocer el diseño de las clases de un programa, ya que sigue las llamadas de los métodos que lo compone. El usuario puede utilizar este árbol para identificar posibles cambios necesarios en el diseño, detectar cuellos de botella de librerías o incluso detectar cuellos de botella en algún método o clase.

- **Herramienta que genera métricas:** se ha implementado un programa que calcula las métricas a partir de un proyecto Java, haciendo uso de una herramienta que está en constante mantenimiento y tiene el apoyo de la comunidad. Las métricas aportan información extra al usuario, sobre los métodos y clase de un proyecto. El usuario puede utilizar esta información para identificar problemas en la implementación, diseño o para identificar un método o clase del árbol de llamadas. Además, se proporcionan métricas extra que no se había pedido como requisito, pero que se valora positivamente para utilizar en ocasiones puntuales.

- **Análisis de herramientas:** para seleccionar todas las tecnologías o herramientas que se han utilizado en el TFG; ya sea de manera indirecta o directa, se ha realizado un estudio previo de las mismas. Incluso, se ha realizado un estudio sobre las bases de las herramientas para entender mejor su funcionamiento. Gracias a este estudio, se puede justificar con argumentos de peso la elección de cada una de ellas, y se reduce la posibilidad de tener incidencias con su uso.

- **Interoperabilidad de herramientas:** se ha conseguido juntar varias herramientas junto a sus librerías. Además, se ha realizado un estudio de la compatibilidad de las versiones de cada uno de ellas, para evitar problemas a la de usar el programa final.

- **Aplicación Java:** el producto final es un JAR que contiene todas las librerías del programa. Este programa dispone de un control de parámetros y puede proporcionar ayudas al usuario a través de la línea de comandos. El JAR gestiona las dependencias y las versiones de esas dependencias a través de Maven, *software* que permite la gestión y construcción de proyectos Java.

9.1.1. Conclusiones personales

Durante las diferentes fases del proyecto, se han aprendido unas lecciones que se usaran en el futuro para mejorar tanto como trabajador. La evaluación personal después de terminar este TFG ha sido positiva. Los objetivos personales que se han alcanzado durante el TFG son los siguientes:

- **Conceptos avanzados de Java:** Java es un lenguaje de programación que se aprende en el primer curso del grado. Sin embargo, en escasas ocasiones se ha profundizado en ella. Para poder completar este proyecto, se han tenido que adquirir los conceptos base de Java, como funciona la máquina virtual (JVM) o cuál es su estructura. También se ha entendido la diferencia entre los ficheros compilados y los ficheros de código fuente.

- **Tipos de análisis de código:** se ha comprendido las diferencias que suponen el análisis estático, dinámico, de código fuente y *bytecode*. Para ello, se ha realizado un estudio previo de cada tipo de análisis, destacando las ventajas y desventajas de cada uno. Finalmente, se han usado los conocimientos para entender el funcionamiento de las herramientas y librerías utilizadas en la implementación. También se ha tenido que modificar parte de las herramientas, que no podría ser posible si no se entendiesen los conceptos en los que se basan.

- **Gestión y desarrollo de un proyecto real:** durante la carrera se han hecho una multitud de proyectos y muchos de ellos han sido en equipo. Este TFG es diferente a esos proyectos ya que se tenían dos directoras controlando en todo momento el estado del proyecto, pero la gestión y desarrollo del mismo es de manera individual. Para poder finalizar con éxito el TFG, se ha tenido que realizar una buena planificación inicial, y todavía más importante un buen seguimiento y control del trabajo realizado. Además, se han tenido que superar muchas incidencias, las cuales no suelen tener los proyectos de la carrera, pero que siempre tienen los proyectos reales.

- **Mejora de trabajo en grupo:** se han adquirido muchas competencias para mejorar el trabajo en equipo. Por una parte, después de realizar tal cantidad de reuniones se ha mejorado en la redacción de las actas, la preparación previa a la reunión y también en transmitir de la mejor manera los aspectos importantes a los asistentes a la reunión. Por otra parte, también se han adquirido competencias con la comunicación con otras personas, a través de correos electrónicos o a la hora de discutir sobre un tema que concierne al proyecto.

- **Calidad del *software*:** durante la fase inicial del proyecto y para poder redactar esta memoria, se han adquirido muchos conocimientos respecto a la calidad del *software*, como puede ser la aplicabilidad del *refactoring*, la importancia del mantenimiento del *software* o el estado del arte de la calidad. Algunos de estos conceptos ya se

conocían de antemano, pero tras realizar un estudio más en profundidad se han expandido los conocimientos. Por otra parte, también se han obtenido conceptos para realizar un buen diseño, como separar un proyecto en módulos o un nuevo patrón de diseño como es el *Visitor*.

9.2. Posibles mejoras

El TFG ha finalizado con el cumplimiento de los requisitos prioritarios de la empresa. Sin embargo, no se han podido lograr todos los objetivo debido a la gran envergadura que suponía el proyecto. Aun así, los requisitos que se han conseguido realizar pueden mejorarse para conseguir un programa mejor. A continuación, se enumeran las posibles mejoras que harían que el programa fuese de mayor calidad:

- **Análisis con código fuente:** sería interesante cambiar la herramienta JCG, por otra que junto a CK calculase el árbol de llamadas con código fuente. De esta manera, no se tendría que analizar dos veces el programa. Aun así, tampoco se puede asegurar que esta opción sea mejor, ya que el *bytecode* permite un análisis más rápido frente al código fuente. Sin embargo, sería interesante disponer de las dos versiones del programa, para realizar pruebas de rendimiento y poder seleccionar la mejor solución.

- **Más personalización:** introducir más opciones de personalización al usuario, para que se pueda realizar un análisis más concreto.

- **Añadir formatos de ficheros:** gracias al buen diseño del programa, es fácil añadir más formatos de ficheros los cuales contiene la información del análisis. No se han añadido más formatos, ya que la empresa no los necesita en este momento. Aun así, existen formatos que en función del escenario podrían ser beneficiosos, como un documento *txt* o DOT (representación de gráficos no-interactivos).

- **Paralelizar el programa:** esta idea fue propuesta por el autor de la herramienta CK, como se ha indicado en el apartado 7.2.2. En el caso de que no se cambie la implementación actual, se podría paralelizar por una parte el árbol de llamadas y por la otra el cálculo de métricas. En el caso de que se cambie de implementación como indica el primer punto de este apartado, se podría paralelizar todo el programa. El problema es que para paralelizar, también se debería realizar un estudio previo para comprobar que en el caso de aplicar los cambios, se consiga disminuir el tiempo de

ejecución del programa. Esto, puede suponer mucho esfuerzo y trabajo, e incluso es probable que se llegue a la conclusión que no sea una solución factible.

9.3. Líneas futuras

En este apartado, se detallan las líneas futuras que podría seguir en su futuro desarrollo. Estas propuestas se diferencian del apartado anterior, ya que se tendrían que aplicar en un futuro y su desarrollo sería más complejo. Además, estas líneas futuras añadirían más funcionalidades al programa, enriqueciendo así el proyecto.

- **Interfaz gráfica:** requisito que no se ha podido realizar. Una pequeña parte del tiempo del proyecto se utilizó para analizar las diferentes posibilidades para visualizar los datos de manera gráfica. Incluso, se realizó una reunión con un integrante de un grupo de investigación de la UPV/EHU, para buscar alternativas. En esa reunión, se llegó a la conclusión que sería una buena solución utilizar la librería D3[1] de JavaScript, por lo que los resultados se mostraría a través de una página web. Sin embargo, no se disponía del tiempo necesario para implementar esta funcionalidad la cual sería muy compleja, ya que la empresa requería de filtros para tener gráficos de llamadas personalizados y tenían que ser interactivos. En el caso de integrar esta funcionalidad, también se había planteado otra funcionalidad extra que consiste en que si se modifica el diseño del gráfico de llamadas que muestre la herramienta, se compruebe de manera automática que ese cambio sea posible (es decir, que se pueda compilar el código con el nuevo diseño). Al finalizar este proyecto, se ha comprobado que esta tarea es de una dificultad mucho mayor, pero aportaría un incremento de calidad soberbio al programa.
- ***Plug-in* para un IDE:** uno de los requisitos de la empresa era que el programa final funcionase como un *plug-in*. Sin embargo, esto se descartó porque no era los beneficios que da esta solución frente al tiempo que había que emplear para desarrollarlo no era rentable. Esto se debe a que en primer lugar, habría que hacer un estudio de cómo crear un *plug-in* y luego desarrollar el código. Además, el *plug-in* podría quedar obsoleta en un futuro, en el caso de que la empresa quisiera cambiar de IDE o se haya cambiado la integración de un *plug-in* en el IDE que está integrado. Aun así, en el caso de que se hiciese sería una buena solución ya que el usuario podría analizar un proyecto dentro del propio IDE, lo cual aceleraría el proceso de análisis.

[1] D3: https://d3js.org/

Anexos

A. ANEXO

Actas de reuniones

Este anexo recopila todas las actas de las reuniones que se han realizado durante el proyecto. El objetivo de estas actas es agrupar los puntos más importantes de las reuniones. A raíz de ellas, se puede tener una trazabilidad de todo el proyecto. Estas actas se han realizado con los interesados del proyecto, y son redactadas por el alumno.

Todas las actas siguen una plantilla, donde se recopila en primer lugar información básica (p. ej.: hora inicio, asistentes...) y después se separan los puntos más importantes en diferentes apartados. Los puntos más importantes son:

- **Orden del día:** información anterior a la reunión, indicando el objetivo de la misma.
- **Resumen de la reunión:** se resume todas las fases de la reunión, sin indicar las decisiones tomadas.
- **Estado del proyecto:** sitúa el proyecto dentro de uno de los paquetes que se ha definido en la planificación.
- **Decisiones:** decisiones que se han tomado para el futuro del proyecto, y se enumeran las tareas que tienen que realizar los interesados de la reunión.

PLANTILLA DE ACTAS

Fecha:
Inicio:
Fin:
Lugar: Reunión telemática
Tipo de Reunión:
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1.

Resumen de la reunión

Estado del proyecto

Decisiones

- **Decisiones adoptadas:**
  -
- **Asignación de tareas a realizar:**
  -
- **Próxima reunión:**

Acta de reunión TFG

Fecha: 22/01/2021
Inicio: 15:30
Fin: 16:30
Lugar: Reunión telemática
Tipo de Reunión: Inicio del proyecto
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Iniciar el proyecto.
2. Definir los requisitos del proyecto.
3. Definir el alcance del proyecto.
4. Definir la gestión del tiempo del proyecto.

Resumen de la reunión

La reunión comenzó la comenzó Beatriz, indicando como cliente cuales eran los requisitos principales del proyecto. Estos requisitos han sido apuntados por el alumno, para su posterior análisis. Después, sobre la gestión se han propuesto diferentes medidas para que el proyecto tenga un desarrollo controlado, haciendo incapié en la trazabilidad de las horas y las reuniones a realizar. Durante la reunión, se demuestra que el alcance es dificil de acotar, por lo que será necesario hacer un estudio del mismo.

Estado del proyecto

El proyecto se encuentra en su fase inicial. Solo se dispone de la idea inicial del proyecto, la cual se decidió un mes atrás entre los integrantes del proyecto.

Decisiones

- **Decisiones adoptadas:**
  - El ciclo de vida del proyecto será incremental.
  - Se realizará un Excel/Hoja de calculo en el cual se recogerán todas las horas del proyecto por parte de Iñaki, para que el resto de integrantes tengan constancia del desarrollo del proyecto.

- El *framework* a crear deberá de ser en Java, sobre la plataforma de Eclipse.
- Las métricas que tendrá que soportar el *framework* será extensa, incluyendo métricas de diferentes niveles y también; preferiblemente, métricas de terceros (ejemplo, Sonarqube).
- Para la creación de la herramienta del árbol de llamadas se podrá: copiar un algoritmo ya creado, utilizar la API de una herramienta de terceros o implementar una propia.
- Las clases/métodos que se tienen que analizar tendrán una cantidad de lineas inmensa, por lo que es necesario estudiar su acoplamiento.
- No se dispone de una restricción sobre las versiones de Eclipse o Java, pero preferible que la de Java sea superior a la 1.6.

- **Asignación de tareas a realizar:**
  - Iñaki deberá de leer un articulo, que sintetiza la situación actual de *refactoring*. Dicho articulo es estregado por Maider.
  - Búsqueda de información sobre la viabilidad del análisis de las métricas (Iñaki).
  - Búsqueda de información sobre la viabilidad del árbol de llamadas de un código (Iñaki).
  - Comenzar con la gestión del proyecto (Iñaki).
  - Realizar calendario de horas (Iñaki).
- **Próxima reunión:** 29/01/2021 a las 15:30.

Acta de reunión TFG

Fecha: 29/01/2021
Inicio: 15:30
Fin: 16:00
Lugar: Reunión telemática
Tipo de Reunión: Seguimiento y control
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Puesta en común de la información por parte de Iñaki.
2. Analizar la información presentada e indicar el camino a seguir en el proyecto.
3. Presentar la gestión del proyecto (Iñaki).

Resumen de la reunión

La reunión ha comenzado con una presentación por parte de Iñaki sobre el *paper* que ha leído y después, ha proseguido indicando las herramientas de que ha utilizado para la búsqueda de información. Por una parte, el árbol de llamadas es donde más se ha focalizado, y se han indicado las siguientes herramientas: GraphCall de Eclipse, *plug-in* de IntelliJ IDEA, repositorio de GitHub con ejecución de árbol de llamadas por terminal y alguna otra sin explorar en demasía. Por otra parte, en cuanto a las métricas, se indica que se ha instalado SonarLint. Sin embargo, Beatriz ha indicado que puede ser que no sea la solución, aunque queda pendiente de investigación.

Después de exponer toda la información, se ha decidido seguir por esa línea de investigación, pero a su vez, se ha remarcado por parte de las directoras del proyecto que se realice la planificación/gestión para la próxima reunión.

Para finalizar, Iñaki ha presentado el calendario de horas en modo Hoja de Cálculo, y ha sido aprobado por las directoras. Por ende, se ha compartido dicho formulario con el resto de integrantes del grupo.

Estado del proyecto

Decisiones

- **Decisiones adoptadas:**

- Se ha validado la Hoja de Cálculo de las horas del TFG.
- Continuar con la línea de investigación (Iñaki).

- **Asignación de tareas a realizar:**
  - Buscar herramientas utilizadas por parte de la empresa que pueden ayudar al proyecto (Beatriz).
  - Realizar la planificación completa para la próxima reunión.
- **Próxima reunión:** 10/02/2021

Acta de reunión TFG

Fecha: 10/02/2021
Inicio: 17:00
Fin: 16:00
Lugar: Reunión telemática
Tipo de Reunión: Demostración de funcionalidades y toma de decisiones
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Verificación de la planificación redactada (Iñaki).
2. Presentación de las herramientas encontradas (Iñaki)
3. Especificar los requisitos del proyecto con el cliente.

Resumen de la reunión

La reunión ha comenzado con la presentación realizada por la parte de Iñaki. Para ello, se ha compartido el proyecto mediante un enlace, para que el resto de integrantes puedan consultarlo en cualquier momento. Después, se ha presentado toda la información recopilada por parte de Iñaki. Se ha concluido que no se debe depender de un *plug-in* propietario, ya que puede dar problemas en el futuro. Por consiguiente, se ha decidido que Iñaki prosiga con la línea de investigación.

Estado del proyecto

Fase de investigación de las herramientas que existen para las funcionalidades del proyecto. Planificación a punto de rematar, ya que queda pendiente la Gestión del Alcance y revisar todas las partes redactadas.

Decisiones

- **Decisiones adoptadas:**
  - La investigación de las herramientas no puede superar las 75h.
  - En el caso de no encontrar ninguna solución acorde al requisito de que se despliegue en Eclipse, se tendrá que encontrar en una solución. Se ha presentado la posibilidad de utilizar IntellijIDEA.

    - Crear una matriz para poder comparar todas las opciones investigadas, para que en el futuro se pueda revisar dicha información (Iñaki).

- **Asignación de tareas a realizar:**
    - Enviar una tarea la cual tiene una planificación bien redactada, y poder tomar ideas de la misma (Maider).
    - Enviar horarios disponibles para la próxima reunión (Maider).
    - Crear la matriz de las herramientas y acabar la planificación. Enviar el borrador de la planificación a Maider (Iñaki).
- **Próxima reunión:** Sin especificar.

Acta de reunión TFG

Fecha: 19/02/2021
Inicio: 16:00
Fin: 17:00
Lugar: Reunión telemática
Tipo de Reunión: Toma de decisiones y demostración de funcionalidades
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Presentación de las herramientas encontradas (Iñaki)

Resumen de la reunión

La reunión ha comenzado con Iñaki presentado el estado de la línea de investigación del árbol de llamadas. Se ha presentado que el resto de herramientas no son factibles por varios motivos: *software* propietario, código cerrado, *bugs*, falta de documentación... Por ello, se ha tomado la decisión de utilizar el repositorio de GitHub, analizar el código y su calidad. En el caso de que sea satisfactoria, será la línea a seguir. Después, se ha decidido realizar una reunión con Raúl, para obtener información sobre visualización de estructuras de tipo árbol o grafo. Para finalizar, por parte de Beatriz se ha solicitado que para la próxima reunión se implemente una primera versión del *framework*, que permita la visualización en CSV.

Estado del proyecto

Finalizando la fase inicial, ya que se dispone de uno de los módulos del *framework*. Se ha comenzado con la tarea de DA (Diseño de la arquitectura) y IL (Implementación de lógica de negocios).

Decisiones

- **Decisiones adoptadas:**
  - Escribir correo al proyecto OpenStaticAnalyzer para obtener más información.
  - No se puede realizar un análisis de un proyecto que esté en un repositorio en la nube (GitHub...), ya que los proyectos que se analicen serán privados.

    - Se realizará una reunión con Raúl para obtener más información sobre la visualización.
    - Iñaki realizará una primera prueba para visualizar los datos en Excel.

- **Asignación de tareas a realizar:**
    - Concretar una reunión con Raúl.
    - Programar visualización en Excel (Iñaki)
- **Próxima reunión:** Sin especificar

Acta de reunión TFG

Fecha: 01/03/2021
Inicio: 15:00
Fin: 16:15
Lugar: Reunión telemática
Tipo de Reunión: Intercambio de información y toma de decisiones
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha
Raúl Medeiros

Orden del día

1. Presentación de los diferentes métodos de representaciones gráficas (Raúl).
2. Decidir la alternativa a utilizar en el proyecto.

Resumen de la reunión

Beatriz ha comentado a Raúl el alcance y el estado actual del proyecto, así como los intereses de la empresa respecto a la herramienta a realizar. Después de obtener esos conocimientos, Raúl ha mostrado la herramienta de 3DES.js. Para ello, ha mostrado dos herramientas implementadas por el grupo de investigación, y ha realizado una pequeña demo. Posteriormente, ha explicado que la herramienta no es muy compleja, y muestra disponibilidad en ayudar para el intercambio de conocimientos de la misma. Para finalizar, se han definido las características que son necesarias que analice el primer prototipo del proyecto.

Estado del proyecto

Finalizando paquete de Alternativas de entornos (AE), fase inicial de la planificación.

Decisiones

- **Decisiones adoptadas:**
  - En el caso de que se realice la representación gráfica, se utilizará 3DES, si es posible conectar el módulo al sistema.
  - Se han definido las diferentes fases del proyecto (4 fases).

    - Organizar reunión para realizar una demo a Beatriz (Iñaki).

- **Asignación de tareas a realizar:**
    - Compartir diferentes enlaces con información referente a las tecnologías comentadas (Raúl).
- **Próxima reunión:** 5/01/2021 a las 15:30.

Acta de reunión TFG

Fecha: 05/03/2021
Inicio: 15:30
Fin: 16:30
Lugar: Reunión telemática
Tipo de Reunión: Seguimiento y control
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Seguimiento y control del proyecto.

Resumen de la reunión

Iñaki ha comenzado la reunión presentando lo que ha realizado desde la última reunión, demostrando las funcionalidades que ha implementado. Después, se ha comentado la posibilidad de realizar filtrados sobre los análisis creados. También se ha realizar un estudio sobre la calidad del código que se ha descargado, utilizando SonarCloud. No se ha podido completar la tarea ya que ha dado un error. Para finalizar, se ha realizado un estudio sobre el diagrama de clases del sistema, identificando varios patrones (*Visitor*).

Estado del proyecto

Desarrollo del primer prototipo del proyecto.

Decisiones

- **Decisiones adoptadas:**
  - Definir los requisitos (Iñaki).
- **Asignación de tareas a realizar:**
  - Realizar diagrama de Casos de Uso para la definición de los requisitos (Iñaki).
  - Redactar documento con los requisitos del proyecto (Iñaki).
  - Revisar la calidad del código con SonarCloud (Iñaki).

- **Próxima reunión:** Sin especificar.

Acta de reunión TFG

Fecha: 16/03/2021
Inicio: 15:00
Fin: 16:00
Lugar: Reunión telemática
Tipo de Reunión: Toma de decisiones
Asistentes:
Iñaki García Noya
Maider Azanza Sese

Orden del día

1. Planificación del proyecto.
2. Revisión de corrección realizada por Maider.

Resumen de la reunión

Previo a la reunión, Iñaki envío a Maider una primera versión de la planificación del proyecto. Esta misma fue corregida por Maider, y comienza la reunión revisando los Casos de Uso. Después, se comenta la posibilidad de aislar el programa de Eclipse, ya que no es necesario y conllevaría muchas horas del proyecto. Sobre la planificación, se comenta que es necesario realizar varios prototipos del programa, para realizar una mejor gestión del tiempo, y se definen los diferentes prototipos. A raíz de esto, se define el ciclo de vida del proyecto, los requisitos del programa, interesados, y el desarrollo general del proyecto.

Estado del proyecto

Planificación del desarrollo.

Decisiones

- **Decisiones adoptadas:**
  - Realizar 3 prototipos del programa.
  - En la siguiente reunión, especificar requisitos con Beatriz.
  - Beatriz, en el ámbito de Interesados del proyecto, no actúa como cliente.
  - Juntar los riesgos con el plan de acción.

    - Quitar el Requisito R1.
    - Especificar en más detalle los requisitos R4 y R9.
- **Asignación de tareas a realizar:**
    - Modificar la planificación (Iñaki).
- **Próxima reunión:** 23/03/2021 a las 3:30.

Acta de reunión TFG

Fecha: 23/03/2021
Inicio: 9:30
Fin: 11:30
Lugar: Reunión telemática
Tipo de Reunión:
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Captura de requisitos.
2. Desarrollo de primer prototipo.

Resumen de la reunión

La reunión comienza con la captura de requisitos. En la anterior reunión, se reunieron Maider e Iñaki para aclarar algunos requisitos que se querían preguntar a Beatriz. Después, Beatriz también ha especificado requisitos extra, tanto funcionales como no funcionales. Después de la captura de requisitos, Beatriz ha reiterado que es importante realizar el primer prototipo. También, ha indicado que es necesario utilizar Sonarqube para analizar la calidad de la herramienta que se va a utilizar. En el caso de que no paso los requisitos mínimos de calidad, se tendrá que descartar esa opción. Se finaliza la reunión puntualizando que cuando se terminen de definir los requisitos, tienen que ser verificados tanto por Maider como por Beatriz.

Estado del proyecto

La planificación está casi terminada y se ha comenzado a desarrollar el código del proyecto.

Decisiones

- **Decisiones adoptadas:**
  - Se han definido los siguientes requisitos:
    - El gráfico de llamadas no tiene que ser interactivo.
    - No es necesario realizar pruebas sobre todos los sistemas de gestión de proyectos (Maven, Gradle...).

    - Las clases que el sistema tiene que analizar tendrán un tamaño máximo de 100.000 líneas.
    - Las clases por proyecto que el sistema tiene que analizar serán un máximo de 2000.
    - El sistema no podrá realizar un análisis superior a 20 minutos, y deberá mostrar un *feedback* al usuario.
    - El sistema deberá de disponer de unos casos de prueba que aseguren el correcto funcionamiento. Estos casos de prueba se realizarán sobre las métricas.
    - El sistema podrá exportar los análisis en diferentes formatos (por ejemplo, CSV).
    - El sistema dispone de los siguientes datos para poder filtrar: número de líneas, LOC, jerarquía por niveles, paquete, nombre del método, línea en la clase y argumento de entrada.
    - No es necesario analizar los tipos enumerados.
    -
  - Realizar un análisis de la calidad del código con Sonarqube. Si no se supera la calidad mínima, no se podrá utilizar.

- **Asignación de tareas a realizar:**
  - Redactar los requisitos y enviar a los integrantes del grupo (Iñaki).
  - Revisar los requisitos redactados por Iñaki (Maider y Beatriz).
  - Analizar la cálida del código con Sonarqube (Iñaki).
- **Próxima reunión:** 13/04/2021 a las 9:30.

Acta de reunión TFG

Fecha: 13/04/2021
Inicio: 9:30
Fin: 10:20
Lugar: Reunión telemática
Tipo de Reunión: Demostración de funcionalidades e intercambio de información
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Iñaki demuestra el desarrollo realizado sobre el primer prototipo.
2. Explicación por parte de Beatriz sobre algunos conceptos para realizar el análisis.

Resumen de la reunión

El motivo principal de la reunión es que Iñaki no comprende la jerarquía de niveles del análisis. Para ello, Beatriz explica su objetivo y su funcionamiento. La explicación se ayuda de ejemplos reales realizados previamente por Beatriz. Iñaki muestra el desarrollo actual del proyecto, que contiene el árbol de llamadas de una clase, pero con llamadas a librerías y a otras clases. Beatriz indica que ese no es el comportamiento esperado, ya que no interesan las llamadas a las librerías y además, solo se quieren las llamadas de una misma clase.

Estado del proyecto

Desarrollo del código del primer prototipo.

Decisiones

- **Decisiones adoptadas:**
  - Solo se requieren las llamadas de los métodos de la misma clase, por lo que es necesario excluir las llamadas externas.
  - El nivel 0 de la jerarquía de niveles se atribuye a aquel método que no es invocado por nadie.
- **Asignación de tareas a realizar:**

    - Seguir con la implementación del código del proyecto (Iñaki).

- **Próxima reunión:** Sin especificar.

Acta de reunión TFG

Fecha: 23/04/2021
Inicio: 15:00
Fin: 15:30
Lugar: Reunión telemática
Tipo de Reunión: Intercambio de información y toma de decisiones
Asistentes:
Iñaki García Noya
Beatriz Pérez Lamancha

Orden del día

1. Reunión convocada por Iñaki para preguntar dudas a Beatriz.

Resumen de la reunión

Como la reunión ha sido convocada por Iñaki, la dinámica consiste en realizar una consulta y Beatriz la responde. En primer lugar, se deja claro que es necesario realizar un análisis por cada clase, generando así un fichero por clase. Después, también se indica que sí que se quieren las llamadas externas, pero solo las de retorno. Es decir, por cada método se quiere saber por quién es llamado. También, se han definido los filtros por clase y por paquete.

Estado del proyecto

Desarrollo del código del primer prototipo.

Decisiones

- **Decisiones adoptadas:**
  - Se tiene que generar un fichero por clase.
  - Filtros por clase y paquete.
  - Llamadas desde otras clases por cada método.
- **Asignación de tareas a realizar:**
  - Continuar con el desarrollo del primer prototipo (Iñaki).
- **Próxima reunión:** Sin especificar.

Acta de reunión TFG

Fecha: 04/05/2021
Inicio: 9:30
Fin: 10:30
Lugar: Reunión telemática
Tipo de Reunión: Demostración de funcionalidades
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Demostración del desarrollo del primer prototipo.

Resumen de la reunión

Iñaki muestra el trabajo realizado en las últimas semanas. En la demostración, se indica que se ha realizado la jerarquía de niveles. La jerarquía es verificada por Beatriz, pero indica que los métodos que no son llamados por nadie son de nivel 0, lo cual no se indica en la implementación actual. Después de la demostración, Beatriz indica que es necesaria una guía de uso en la cual se tienen que documentar los parámetros de entrada. También indica que es necesario gestionar la escalabilidad del programa, para que no se tengan problemas en el futuro. Por otra parte, se ha decidido que se tiene que comenzar con la implementación que calcule el LOC y la complejidad ciclomática. Se ha finalizado la reunión, comentando los requisitos definidos previamente. Se tienen que modificar algunos de ellos y volver a verificar.

Estado del proyecto

Desarrollo del código del primer prototipo.

Decisiones

- **Decisiones adoptadas:**
  - Realizar una guía de uso.
  - Modificar algunos requisitos, para eliminar la ambigüedad de algunos de ellos.
  - Es necesario controlar la escalabilidad del programa, realizando buenas pruebas.

- **Asignación de tareas a realizar:**
  - Realizar la guía de uso en una fase más avanzada del proyecto (Iñaki).
  - Redactar de nuevo los requisitos y definir algunos extra (Iñaki).
  - Verificar los requisitos redactados por Iñaki (Maider y Beatriz).
- **Próxima reunión:** 14/05/2021 a las 9:30.

Acta de reunión TFG

Fecha: 14/05/2021
Inicio: 9:30
Fin: 10:30
Lugar: Reunión telemática
Tipo de Reunión: Demostración de funcionalidades y toma de decisiones
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Realizar una demostración sobre lo desarrollado.
2. Discutir sobre los casos de uso y requisitos.

Resumen de la reunión

Iñaki ha explicado los inconvenientes que ha tenido para implementar el LOC y la complejidad ciclomática. Ha implementado dichas funcionalidades utilizando un BufferedReader, opción que fue seleccionada después de un estudio previo. Ha añadido que tiene varias dificultades, como contar las llaves o determinar el final del fichero. Por ende, la tarea no está finalizada. Después, se ha realizado un debate sobre los últimos requisitos añadidos, para quitar la ambigüedad de los nuevos. Para finalizar, se ha planteado la idea de que los prototipos no se podrán completar como tal, lo cual conllevará a modificaciones en los mismos.

Estado del proyecto

Fase avanzada del desarrollo del código, sin seguir con los prototipos establecidos. Aún no se han realizado pruebas.

Decisiones

- **Decisiones adoptadas:**
  - Se continúa con la implementación con BufferedReader.
  - Se ha descartado introducir diagramas de casos de uso en la memoria.
  - Definir los requisitos mejor en su apartado, y resumirlos en el alcance (objetivos).

    - Cambiar fechas en la planificación.

- **Asignación de tareas a realizar:**
    - Continuar con el desarrollo del proyecto (Iñaki).
    - Cambiar los últimos requisitos (Iñaki).
- **Próxima reunión:** 26/05/2021 a las 9:30.

Acta de reunión TFG

Fecha: 26/05/2021
Inicio: 9:30
Fin: 11:00
Lugar: Reunión telemática
Tipo de Reunión: Toma de decisiones
Asistentes:
Iñaki García Noya
Maider Azanza Sese

Orden del día

1. Aclarar dudas sobre la memoria (Beatriz).

Resumen de la reunión

El objetivo de esta reunión es la de preparar una reunión con Beatriz, para finalizar con la memoria. En primer lugar, se han repasado todos los requisitos y exclusiones, ya es necesario que estén bien definidos en la memoria. Se han separado por categorías los requisitos funcionales, gracias a las aportaciones de Maider. Se ha decidido que es necesario realizar una nueva planificación, y dejar como está la primera planificación. En la nueva planificación, se recorta el alcance y se plantea una nueva gestión del tiempo.

Estado del proyecto

Fase avanzada del desarrollo del código, sin seguir con los prototipos establecidos. Aún no se han realizado pruebas. Primera planificación revisada y comienzo de una segunda planificación.

Decisiones

- **Decisiones adoptadas:**
  - Preguntar exclusiones a Beatriz.
  - Los requisitos textuales son suficientes, no es necesario introducir casos de uso.
  - Los objetivos tienen que ser medibles.
  - Se han separado los requisitos funcionales en las siguientes categorías:
    - Visualización.
    - Filtrado.

    - Exportación.
    - Análisis.
  - El árbol de llamadas se tiene que definir en la introducción.
  - Quitar los requisitos de la información.
  - El requisito IR-09 se convierte en una exclusión.
  - El requisito IR-12 se mide con el tiempo, exactamente se verifica cuando no supera los 15 minutos.
  - El requisito IR-13 se verifica preguntando a la empresa directamente.
  - El requisito IR-14 se verifica realizando casos de prueba en los módulos importantes.
  - Planificación inicial no se modifica, y se realiza una nueva planificación.
  - La nueva planificación tiene un alcance más acortado.
- **Asignación de tareas a realizar:**
  - Realizar la nueva planificación (Iñaki).
  - Preparar los requisitos para la siguiente reunión (Iñaki).
- **Próxima reunión:** 27/05/2021 a las 9:30.

Acta de reunión TFG

Fecha: 27/05/2021
Inicio: 9:30
Fin: 10:45
Lugar: Reunión telemática
Tipo de Reunión: Demostración de funcionalidades
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Demostración del desarrollo para el cálculo de las métricas.

Resumen de la reunión

La dinámica de la reunión ha sido la siguiente: Iñaki realizada una pregunta y Beatriz la respondía. Cuando se respondía, Maider verificaba la respuesta. En primer lugar, se ha comenzado con los requisitos discutidos por Iñaki y Maider en la anterior reunión. Después, se ha decidido definitivamente no realizar los 3 prototipos. Para ello, se ha estimado que la planificación del proyecto tiene que estar terminada en el mes de junio, mientras que la memoria se termina en el mes de julio. Para finalizar, Iñaki ha hecho una demostración sobre el código implementado para el cálculo de las métricas. Indica que sigue teniendo problemas con el BufferedReader y también que ha tenido problemas para sincronizar el HashMap. Se ha decidido dar una última oportunidad a esta implementación, en el caso de que no funcione para la siguiente reunión, se descartará.

Estado del proyecto

Fase avanzada del desarrollo del código, sin seguir con los prototipos establecidos. Aún no se han realizado pruebas. Comienzo de una segunda planificación.

Decisiones

- **Decisiones adoptadas:**
  - Las exclusiones no se quitan de la memoria.
  - Eliminar el requisito IR-10.
  - Beatriz ha aceptado realizar una nueva planificación.
  - Se han eliminado los 3 prototipos y se han aunado en uno.

    - El día 10 de junio se tiene que realizar una prueba con ejemplos reales de Beatriz.

- **Asignación de tareas a realizar:**
    - Continuar con el desarrollo del proyecto (Iñaki).
    - Redactar los cambios en la memoria (Iñaki).
    - Realizar la nueva planificación (Iñaki).
- **Próxima reunión:** Sin especificar.

Acta de reunión TFG

Fecha: 14/06/2021
Inicio: 11:00
Fin: 11:30
Lugar: Reunión telemática
Tipo de Reunión: Demostración de funcionalidades
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Demostración de la nueva herramienta para el cálculo de pruebas.

Resumen de la reunión

Desde la anterior reunión, Iñaki ha intentado encontrar una solución al cálculo de las métricas. La opción del BufferedReader, además de tener fallos no era uy eficiente con proyectos de gran tamaño. Por ello, se tuvo que buscar una nueva solución. Iñaki explica que ha integrado en el programa un nuevo *plug in* que realiza cálculos de métricas. El *plug in* se muestra al resto de integrantes y es verificado por Beatriz. Por consiguiente, se desecha la anterior implementación y se continúa con la nueva herramienta. Iñaki finaliza la reunión indicando que también se ha avanzado mucho con la segunda planificación.

Estado del proyecto

Finalizando desarrollo del código, pero todavía no se han realizado pruebas. Fase avanzada de la segunda planificación.

Decisiones

- **Decisiones adoptadas:**
  - Se ha verificado el nuevo *plug in*, por lo que integrará en el programa principal.
- **Asignación de tareas a realizar:**
  - Integrar la nueva herramienta en el programa principal (Iñaki).
  - Finalizar la nueva planificación (Iñaki).
- **Próxima reunión:** Sin especificar.

Acta de reunión TFG

Fecha: 22/06/2021
Inicio: 11:00
Fin: 12:00
Lugar: Reunión telemática
Tipo de Reunión: Demostración de funcionalidades y toma de decisiones
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Demostración de la integración de la nueva herramienta para el cálculo de pruebas.
2. Analizar la nueva planificación.

Resumen de la reunión

Iñaki indica que ha terminado de integrar las nuevas métricas. Se realiza una demostración sobre el comportamiento del programa en su totalidad. Beatriz índica algunos cambios a realizar en el programa, como quitar los métodos constructores. También indica que el proyecto se subirá en el repositorio oficial de la empresa, una vez terminado el proyecto. Este repositorio tendrá que tener la guía de uso adjuntado y tendrá un fichero *README* resumiendo su funcionamiento.

Estado del proyecto

Finalizado el desarrollo del programa, pendiente de posibles modificaciones en la fase de pruebas. Se ha terminado de definir la segunda planificación.

Decisiones

- **Decisiones adoptadas:**
  - El proyecto se subirá al repositorio oficial de la empresa al finalizar el proyecto.
  - El repositorio contará con un fichero introductorio.
  - Eliminar las constructoras del análisis.
  - El 29 de Junio Beatriz comenzará con las pruebas.
- **Asignación de tareas a realizar:**

- Finalizar el desarrollo del código y realizar las pruebas (Iñaki).
- Subir el proyecto al repositorio oficial de la empresa al terminar el proyecto (Iñaki y Beatriz).
- Revisar la nueva planificación (Maider).

- **Próxima reunión:** 29/06/2021 a las 11:00.

Acta de reunión TFG

Fecha: 29/06/2021
Inicio: 11:00
Fin: 11:30
Lugar: Reunión telemática
Tipo de Reunión: Seguimiento y control
Asistentes:
Iñaki García Noya
Maider Azanza Sese
Beatriz Pérez Lamancha

Orden del día

1. Problemas con las pruebas exhaustivas del programa.

Resumen de la reunión

Iñaki comienza la reunión comentando los problemas que ha tenido con las pruebas. Una de ellos no se pudo solucionar, y tenía relación con el último *plug in* que se ha añadido. A raíz de ello, Iñaki se puso en contacto con el autor del repositorio y se logró una solución. Ahora el programa es mucho más eficiente y funciona correctamente, aunque es necesario realizar más pruebas. Después, se ha reiterado que es necesario realizar la guía de uso, para que Beatriz pueda realizar las pruebas. Se termina la reunión ordenando a Iñaki que realice un estudio sobre las dependencias que tiene el programa, para que Beatriz pueda realizar las pruebas.

Estado del proyecto

Fase de pruebas exhaustivas del código.

Decisiones

- **Decisiones adoptadas:**
  - Continuar con la última herramienta que se ha integrado (CK).
- **Asignación de tareas a realizar:**
  - Crear la guía de uso para que Beatriz pueda realizar las pruebas (Iñaki).
  - Finalizar el desarrollo del código y realizar las pruebas (Iñaki).
- **Próxima reunión:** 07/07/2021 a las 11:00.

Acta de reunión TFG

Fecha: 07/07/2021
Inicio: 11:00
Fin: 12:00
Lugar: Reunión telemática
Tipo de Reunión: Intercambio de información
Asistentes:
Iñaki García Noya
Beatriz Pérez Lamancha

Orden del día

1. Realizar pruebas por parte de Beatriz, con la ayuda de Iñaki.

Resumen de la reunión

Reunión para realizar las pruebas en el ordenador de Beatriz. Antes de ello, Iñaki comenta que se han realizado varias modificaciones en el código. Se han modificado los parámetros de entrada, se ha disminuido el tiempo de ejecución y se ha modificado el diseño para que sea más extensible. Para finalizar, indica que gran parte del tiempo desde la última reunión se ha dedicado en la memoria del proyecto. Después, se comienza con la prueba. Para ello, se sigue la guía de uso entregada por parte de Iñaki. Se descarga u¡el programa principal y un proyecto de prueba. La prueba es un éxito, aunque se han tenido problemas cuando se han importado las dependencias del proyecto. Para finalizar, Beatriz ha indicado que se cambie el nombre de una carpeta y también, que se creen nuevas carpetas cada vez que se realiza un análisis.

Estado del proyecto

Fase de pruebas y fase avanzada de la memoria.

Decisiones

- **Decisiones adoptadas:**
  - Cambiar el nombre de la carpeta *metrics*.
  - Crear una nueva carpeta por cada análisis que se realice.
- **Asignación de tareas a realizar:**
  - Realizar los cambios en el código (Iñaki).
  - Realizar pruebas sobre el programa (Iñaki y Beatriz).

    - Mandar un correo a Beatriz con información adicional (Iñaki).

- **Próxima reunión:** Sin especificar.

B. ANEXO

Guía de uso

En la primera planificación, la empresa definió el requisito IR-20 que indica que es necesario realizar un manual para comprender el funcionamiento del programa paso por paso. Este manual será utilizado por trabajadores de la empresa que no han trabajado en este proyecto, por lo que es necesario explicar todos los puntos en detalle, ya que no tienen ningún tipo de información sobre el funcionamiento del mismo.

Esta guía de uso es la segunda versión, ya que la anterior quedó obsoleta al realizar la fase de pruebas exhaustivas. Como se indica en el Capítulo 7, para realizar las pruebas con la empresa se tuvo que redactar en primer lugar esta guía de uso. Después de que la empresa realizase las pruebas, realizaron varias sugerencias de mejora y además también se encontraron varios errores. Esta guía de uso está actualizado después de recopilar dicha información.

San Sebastián / Donostia

Guía de uso herramienta CallGraph

INDABA CONSULTORES S.L.

Iñaki García Noya

15/07/2021

Índice

1 Introducción

En este documento se detalla la guía de uso para utilizar la herramienta que obtiene el gráfico de llamadas de un proyecto Java. Para ello, en primer lugar es necesario descargar el proyecto principal y descomprimirlo: https://github.com/iakigarci/TFG/tree/command-cli.

Es importante aclarar, que durante este documento, se define como "programa principal" el programa de esta guía, ya que este mismo tiene como función analizar otros proyectos, de esta manera diferenciando los dos términos.

1.1 Descripción del programa

El programa tiene como objetivo, ayudar en la toma de decisiones para la mejora de la calidad de un proyecto Java. Para ello, analiza el proyecto Java obteniendo el gráfico de llamadas y diferentes métricas, que favorecen las modificaciones para el *refactoring* de código. El gráfico de llamadas se compone de las llamadas que un método realiza. En este caso, solo se guardan las llamadas que realiza a la misma clase que se está analizando. Una vez formado el gráfico, se obtienen una serie de métricas que aportan información sobre la calidad de la clase y métodos, y todo ello se exporta a un fichero para su lectura y análisis.

El programa precisa de un proyecto que se quiere analizar, y el mismo se necesita en dos formatos (Véase el apartado 3.1):

- Código compilado o *bytecode*.
- Código fuente.

2 Dependencias

El proyecto necesita de unos programas externos para que funcione. Esto se debe a que el proyecto está desarrollado en Maven, haciendo uso de muchas dependencias. Para entender el uso de las dependencias, es necesario definir dos aspectos del programa, ya que se separa en dos fases:

- **Compilar:** el programa principal genera un JAR, que contiene todo lo necesario para que el programa funcione y ayuda a su fácil despliegue. Por ello, en primer lugar se compila el programa principal y luego se puede ejecutar, aunque también existe la opción de utilizar el programa sin compilar (véase el apartado 3.2). El programa principal solo se tiene que compilar cuando se realiza algún cambio en el mismo generando así un nuevo JAR, pero en caso contrario no es necesario realizar esto.
- **Ejecutar:** como se explica en el Capítulo 3, el programa se puede utilizar con diferentes métodos. A la hora de ejecutar, se analiza un programa Java ya creado previamente.

Para probar los posibles problemas de acoplamiento, se han realizado diferentes pruebas en diferentes dispositivos (Linux y Windows). Aún así, no se han podido realizar las pruebas suficientes para asegurar las versiones mínimas para que los dependencias funcionen ni tampoco se han podido probar todas las combinaciones con versiones superiores.

- **Java:** es importante tener instalado el JDK de Java. Se han probado muchas versiones pero solo funciona la versión **11.0.10**+. También se han probado las versiones superiores y funcionan correctamente. Es necesario disponer de Java para compilar el código, y también para su uso.
- **Maven:** el programa sigue una estructura Maven, con todas las dependencias que esta gestiona. Para crear el proyecto principal, se han utilizado las últimas versiones de las dependencias, por lo que se requiere una versión avanzada de Maven. En este caso, para desarrollar el proyecto se ha utilizado la versión **3.6.3**, por lo que es necesario disponer de una versión superior a la 3.6.0. Esto solo es necesario para poder compilar el programa, pero no es necesario disponer de Maven para utilizarlo. En el caso de que se quiera utilizar otra versión de Maven, sería necesario modificar el fichero *pom.xml* del programa.
- **Eclipse:** como se detalla más adelante, el programa se puede utilizar vía linea de comandos o con un IDE. En esta guía se utiliza Eclipse ya que ha sido un requisitos, pero tiene que funcionar con el resto de IDEs). Se ha desarrollado utilizando la versión **2020-12 (4.18.0)**.

Dependencia	Versión utilizada	Versión mínima (funcional)	Compilar	Ejecutar
JDK Java	11.0.10	11.0.10	✓	✓
Maven	3.6.3	3.6.0	✓	×
Eclipse	2020-12 (4.18.0)	-	×	✓

3 Uso

Para usar el programa se pueden usar dos métodos:

- Línea de comandos utilizando un JAR (Véase apartado 3.3).
- IDE, sin tener que compilar el proyecto principal previamente (Véase apartado 3.2).

El método más sencillo es utilizar un IDE, ya que no es necesario compilar el código, por lo que se reduce el tiempo y además, es más intuitivo de utilizar.

3.1 Parámetros de entrada

El método principal del programa es el método `main(String[] args)` de la clase *callGraph.JCallGraph*. Por ello, o bien mediante una clase Java o mediante los parámetros del JAR, es necesario enviar los parámetros. Los parámetros son los siguientes:

- **[1] Nombre del JAR:** el programa necesita el JAR del proyecto que se tiene que analizar, este JAR puede crearse a través de un MANIFEST o un *software* de construcción de proyectos como Maven o Graddle. En el caso de que el JAR no se situe en la raíz del proyecto principal, será necesario especificar la dirección absoluta.
- **[2] Paquete(s) que se quiere(n) incluir:** el programa analiza todo el código que está almacenado dentro del JAR, incluso las dependencias del código analizar si es que las tuviese. Por ello, es necesario especificar qué paquetes son necesarios analizar. Para incluir varios paquetes se tienen que separar por comas (p. ej.: "paquete1, paquete2"). En el caso de que se disponga una estructura Maven o similar, siendo la raíz *src/main/java*, es necesario especificar los paquetes a partir de este punto.
- **[3] Paquete(s) que se quiere(n) excluir:** igual que el parámetro anterior, pero excluyendo los paquetes que no se quieren analizar. También se puede especificar que se excluyan todos los paquetes externos introduciendo "*,".
- **[4] Código fuente del proyecto:** es necesario además del JAR, incluir la dirección absoluta de la carpeta en la que se encuentra el proyecto que se quiere analizar.

3.2 IDE

Para explicar el uso de IDE, se va a explicar utilizando Eclipse. Para realizar una prueba, se puede utiliza un proyecto creado específicamente para este programa llamado *ProyectoAnalizarExtendido*, que es un proyecto Java sencillo que incluye una gran parte de la casuística de la estructura Java:

https://github.com/iakigarci/ProyectoAnalizarExtendido.

3.2.1 Importar proyecto

En primer lugar, se importa el proyecto principal (enlace) al IDE, haciendo uso de la opción de ***Import*** → ***General*** → ***Existing Projects into Workspace***.

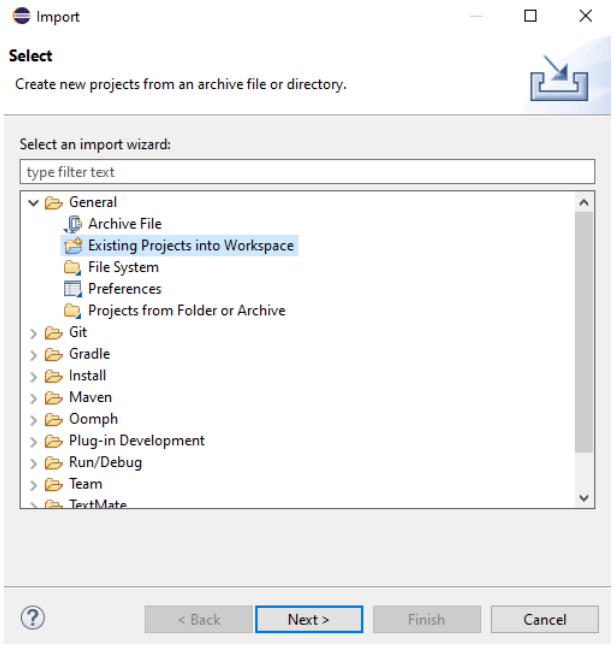

Figure 1: Importar el proyecto al IDE

3.2.2 Paso de parámetros

Una vez importado el proyecto, el objetivo es enviar los parámetros de entrada al método `main(String[] args)` de la clase *callGraph.JCallGraph*, especificando que otro proyecto se quiere analizar. Para ello, se tiene que crear una clase Java e invocar ese método con dichos parámetros. Se recomienda utilizar JUnit, ya que permite realizar esta tarea de manera más sencilla. En este caso, el objetivo de JUnit no es comprobar su correcto funcionamiento, sino comprobar que funciona. Se puede utilizar la carpeta *src/test/java* para realizar este cometido.

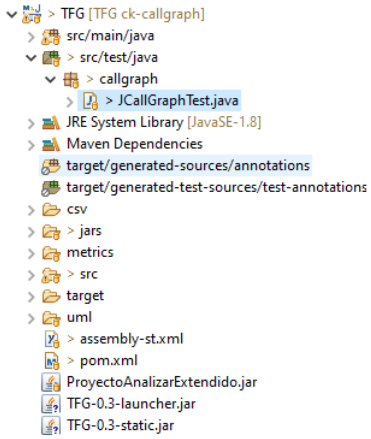

Figure 2

3.2.3 Ejecución

A continuación se muestra un ejemplo de como analizar el proyecto *ProyectoAnalizarExtendido* con un JUnit, indicando que analice todo el proyecto, ya que el paquete *"paquete"* es la raíz del proyecto. Se puede observar, que para el paso de parámetros de una dirección a través de un fichero java en lugar de indicar la carpeta con "\" es necesario utilizar "\\".

```
@Test
void testMain4() {
    String[] s = new String[4];
    s[0] = "ProyectoAnalizarExtendido.jar";
    s[1] = "paquete";
    s[2] = "*,";
    s[3] = "D:\\UNIVERSIDAD\\TFG\\Repositorio\\ProyectoAnalizarExtendido-master\\src";
    JCallGraph.main(s);
}
```

Figure 3

Para ejecutar este JUnit, se puede ejecutar haciendo **Click derecho** sobre el propio método y ejecutarlo con *Run As* → *JUnit Test*. Esto permite disponer de un fichero JUnit con muchos proyectos definidos, y analizar cada uno por separado. Una vez ejecutado, se puede ver el proceso del análisis en el terminal del IDE.

```
Problems  Javadoc  Declaration  Console
<terminated> JCallGraphTest.testMain4 [JUnit] C:\Users\GARCI\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_15.0.1.v20201027-0507\jre\bin\javaw.exe (Jul 2, 2021, 12:02:10 PM – 12:02:12 PM)
[METRICAS]:paquete.ClaseEnum
[METRICAS]:paquete.InstanciaAbstracta
[METRICAS]:paquete.InstanciaInterfaz
[METRICAS]:paquete.Interfaz
[METRICAS]:paquete.Main
[METRICAS]:paquete.Segundo
[METRICAS]:paquete.Tercero
[METRICAS]: Tiempo transcurrido -> 0:00:00.993

[IMPRIMIR]: D:\UNIVERSIDAD\TFG\Repositorio\TFG\TFG\csv/paquete.InstanciaInterfaz.csv
[IMPRIMIR]: D:\UNIVERSIDAD\TFG\Repositorio\TFG\TFG\csv/paquete.Tercero.csv
[IMPRIMIR]: D:\UNIVERSIDAD\TFG\Repositorio\TFG\TFG\csv/paquete.ClaseAbstracta.csv
[IMPRIMIR]: D:\UNIVERSIDAD\TFG\Repositorio\TFG\TFG\csv/paquete.Main.csv
[IMPRIMIR]: D:\UNIVERSIDAD\TFG\Repositorio\TFG\TFG\csv/paquete.InstanciaAbstracta.csv
[IMPRIMIR]: D:\UNIVERSIDAD\TFG\Repositorio\TFG\TFG\csv/paquete.ClaseEnum.csv
[IMPRIMIR]: D:\UNIVERSIDAD\TFG\Repositorio\TFG\TFG\csv/paquete.Segundo.csv
[FINAL]: Tiempo transcurrido -> 0:00:01.003
```

Figure 4

El programa imprime mucha información por el terminal, y cada fase del análisis dispone de un *flag* al principio del mensaje. También se indica que cada fase se ha finalizado con la ayuda de un contador, que se imprime al finalizar cada fase.

- *[CALLGRAPH:]* se están analizando las llamadas de los métodos. Hace uso del JAR del proyecto (primer parámetro de entrada).
- *[METRICAS:]* analizando todas las métricas de las clases y de los métodos. Hace uso del código fuente del proyecto (cuarto parámetro de entrada). Es la fase que más duración conlleva del análisis.
- *[IMPRIMIR:]* imprimiendo los resultados en los diferentes ficheros.

3.2.4 Resultado

Cuando termina la ejecución, se crean dos nuevas carpetas. Se detallan los ficheros que incluyen en el Capítulo 4.

3.3 Línea de comandos

*Pendiente de redactar

4 Resultado

Al terminar la ejecución, se crean dos carpetas.

- *csv:* se incluyen todos los reportes de las llamadas de los métodos, por cada clase que se ha analizado.
- *metrics:* incluye las métricas de los métodos y de las clases, en dos CSV diferentes.

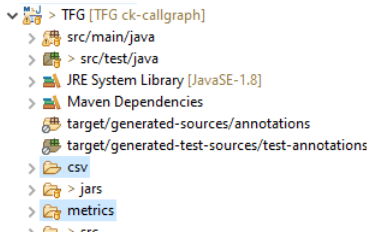

Figure 5

4.1 Gráfico de llamadas - *csv*

Se genera un fichero por cada uno de las clases que se ha analizado. Estos ficheros en el estado actual del proyecto solo se exportan en CSV, pero en un futuro existe la posibilidad de aumentar a más métodos de exportación. Cada fichero tiene los siguientes campos, separados en columnas:

- **Nombre:** nombre del método.
- **Nivel:** nivel del método, empezando por 0. El árbol se crea en el caso de que la siguiente fila sea de nivel 1, en caso contrario no llama a otro método.
- **LOC:** número de lineas que tiene el método, incluyendo la cabecera del método y el cierre del mismo, pero excluyendo los comentarios y líneas vacías.
- **WMC:** mide la complejidad de un método. Para calcular, se utiliza la siguiente formula con la ayuda de la estructura en árbol del método.

$$WMC = Arcos - Nodos + 2$$

- **Resultado:** resultado del método (p. ej.: *void*).
- **Linea en clase:** número de linea en el cual el método se sitúa dentro de la clase.
- **Llamado por:** métodos que es llamado desde otras clases. Estos métodos están separados por comas, y tienen la clase como identificador.

4.2 Métricas - *metrics*

En la carpeta *metrics*, se ubican dos ficheros, que no varían de nombre entre diferentes ejecuciones:

- ***class.csv:*** contiene las métricas de cada clase. Las métricas que contiene son las siguientes:
  - **File:** fichero de código fuente que se ha analizado.
  - **Class:** clase que se ha analizado.
  - **Type:** tipo de la clase.
  - **CBO:** (*Coupling Between Objects*) número de acoplamientos entre dos clases. Cuando una clase (*claseA*) llama a los métodos de otra clase (*claseB*), entonces la primera clase está acoplada con la segunda (*claseA* acoplada con *claseB*). Cuanto más pequeño sea el índice CBO, menor será el efecto de los cambios que se tendrá en otras clases, que significa que es más independiente la clase. Por consiguiente, implica que se tienen que menos actualizaciones cuando sufra alguna modificación. Cuanto mayor sea el CBO, menor reusabilidad tendrá la clase.
  - **WMC:** mide la complejidad de una clase individual. Si todos los métodos se consideran se consideran igualmente complejos, entonces WMC es el número de métodos definidos en cada clase.
  - **DIT:** (*Depth Inheritance Tree*) mide el nivel máximo de la jerarquía de herencia de una clase; la raíz del árbol de herencia no hereda de ninguna clase y está en el nivel cero del árbol de herencia. Recuento de los niveles de los niveles en una herencia jerarquía de herencia. El objetivo es medir la complejidad, la complejidad del diseño y el potencial de reuso ya que cuanto más bajo esté una clase, mayor será el número que tendrá que heredar.
  - **LCOM:** (*Lack of Cohesion of Methods*) mide el grado en que los métodos hacen referencia a los datos de instancia de las clases. Un valor alto implica falta de cohesión, es decir, baja similitud y la clase puede ser una composición de objetos no relacionados. No es una métrica a tener en cuenta para decisiones, ya que existe una versión mejorada llamada LCOM-HS, que no se encuentra en el análisis.
  - **NOSI:** (*Number of static invocations*) cuenta el número de invocaciones a métodos estáticos. Sólo puede contar las que pueden ser resueltas por el JDT.
  - **TCC:** (*Tight Class Cohesion*) mide la cohesión de una clase con un rango de valores de 0 a 1. TCC mide la cohesión de una clase a través de conexiones directas entre métodos visibles, dos métodos o sus árboles de invocación acceden a la misma variable de clase.
  - **LCC:** (*Loose Class Cohesion*) similar a TCC, pero incluye además el número de conexiones indirectas entre las clases visibles para el cálculo de la cohesión. Así, la restricción LCC ¿= TCC se mantiene siempre.
  - **Contadores de métodos:** diferentes contadores de diferentes tipos de métodos:
    - *totalMethodsQty:* número total de métodos.
    - *staticMethodsQty:* número total de métodos estáticos.
    - *publicMethodsQty:* número total de métodos públicos.
    - *privateMethodsQty:* número total de métodos privados.
    - *protectedMethodsQty:* número total de métodos *protected*.
    - *visibleMethodsQty:* número total de métodos visibles.
    - *abstractMethodsQty:* número total de métodos abstractos.
    - *finalMethodsQty:* número total de métodos *final*.

    - *synchronizedMethodsQty:* número total de métodos *synchronized*.
  - **Contadores de campos:** diferentes contadores de diferentes tipos de campos de la clase:
    - *totalFieldsQty:* número total de campos.
    - *staticFieldsQty:* número total de campos estáticos.
    - *publicFieldsQty:* número total de campos públicos.
    - *privateFieldsQty:* número total de campos privados.
    - *protectedFieldsQty:* número total de campos *protected*.
    - *finalFieldsQty:* número total de campos *final*.
    - *synchronizedFieldsQty:* número total de campos *synchronized*.
  - **LOC:** (*Lines Of Code*) contador de las líneas de código, ignorando las líneas vacías y los comentarios.
  - **Contadores de instrucciones:** diferentes contadores de diferentes tipos de instrucciones de la clase:
    - *returnQty:* número total de *return*.
    - *loopQty:* número total de bucles.
    - *comparisonsQty:* número total de comparaciones.
    - *tryCatchQty:* número total de *try/catch*.
    - *parenthesizedExpsQty:* número total de expresiones con paréntesis.
    - *stringLiteralsQty:* número total de *string literal*.
    - *numbersQty:* número total de números.
    - *assignmentsQty:* número total de asignaciones.
    - *mathOperationsQty:* número total de operaciones matemáticas.
    - *variablesQty:* número total de variables.
    - *lambdasQty:* número total de instrucciones *lambda*.
    - *logStatementsQty:* número total de instrucciones *log*.
  - **Otros contadores:**
    - *maxNestedBlocksQty:* número total de bloques anidados.
    - *anonymousClassesQty:* número total de clases anónimas.
    - *innerClassesQty:* número total de clases internas.
    - *uniqueWordsQty:* número total de palabras únicas.
    - *modifiers:* número total de modificadores.

- ***method.csv:*** contiene todos los métodos de todas las clases, con sus métricas adicionales. Las métricas que tiene son las siguientes:
  - **File:** fichero de código fuente que se ha analizado.
  - **Class:** clase que se ha analizado.
  - **Method:** nombre del método.
  - **Constructor:** "TRUE" si es un método constructor y "FALSE" si no lo es.
  - **CBO:** (*Coupling Between Objects*) número de acoplamientos entre dos clases (Explicado en detalle en anterior punto).
  - **WMC:** mide la complejidad de un método. Para calcular, se utiliza la siguiente formula con la ayuda de la estructura en árbol del método.

$$WMC = Arcos - Nodos + 2$$

- **RFC**: (Response for a Class) mide la comunicación (o el envío de mensajes) dentro de las clases. Cuanto más grande sea la RFC, más compleja más compleja es la clase y, por lo tanto, las pruebas y el mantenimiento serán más difíciles.
- **LOC:** (*Lines Of Code*) contador de las líneas del método, ignorando las líneas vacías y los comentarios.
- **Contadores de instrucciones:** diferentes contadores de diferentes tipos de instrucciones de la clase:
  * *returnsQty:* número total de *return*.
  * *variablesQty:* número total de variables.
  * *parametersQty:* número total de parámetros.
  * *methodsInvokedQty:* número total de métodos que invoca.
  * *methodsInvokedLocalQty:* número total de métodos que invoca de manera local.
  * *methodsInvokedIndirectLocalQty:* número total de métodos que invoca de manera local indirectamente.
  * *loopQty:* número total de bucles.
  * *comparisonsQty:* número total de comparaciones.
  * *tryCatchQty:* número total de *try/catch*.
  * *parenthesizedExpsQty:* número total de expresiones con paréntesis.
  * *stringLiteralsQty:* número total de *String literal*.
  * *numbersQty:* número total de números.
  * *assignmentsQty:* número total de asignaciones.
  * *mathOperationsQty:* número total de operaciones matemáticas.
  * *lambdasQty:* número total de instrucciones *lambda*.
  * *logStatementsQty:* número total de instrucciones *log*.
- **Otros contadores:**
  * *maxNestedBlocksQty:* número total de bloques anidados.
  * *anonymousClassesQty:* número total de clases anónimas.
  * *innerClassesQty:* número total de clases internas.
  * *uniqueWordsQty:* número total de palabras únicas.
  * *modifiers:* número total de modificadores.
  * *hasJavaDoc:* "TRUE" si tiene JavaDoc especificado y "FALSE" si no lo tiene.

4.3 Ejemplo

En este caso, al ejecutar el proyecto *ProyectoAnalizarExtendido*, esto son los resultados obtenidos. Se muestra en la primera tabla el fichero *paquete.Segundo.csv*, y en las dos tablas restantes *class.csv* y *methods.csv* (estas dos últimas se representan recortadas ya que son muy extensas).

Nombre	Nivel	LOC	WMC	Resultado	Linea en clase	Llamado por
<init>	0	0	0	void	0	[paquete.Mainf1]
getList	0	3	1	java.util.ArrayList	14	[paquete.Mainf1]
toString	0	4	1	java.lang.String	19	

Table 1: Fichero *paquete.Segundo.csv*

class	type	cbo	wmc	dit	rfc	lcom	tcc	lcc
paquete.ClaseAbstracta	class	1	2	1	2	1	0.0	0.0
paquete.ClaseEnum	enum	0	0	1	0	0	-1.0	-1.0
paquete.InstanciaAbstracta	class	2	2	2	2	1	0.0	0.0
paquete.InstanciaInterfaz	class	2	2	1	2	1	0.0	0.0
paquete.Interfaz	interface	0	1	1	0	0	NaN	NaN
paquete.Main	class	3	4	1	6	6	0.0	0.0
paquete.Segundo	class	0	3	1	3	1	0.0	0.0
paquete.Tercero	class	1	5	1	5	10	0.0	0.0

Table 2: Fichero *class.csv* (recortado)

class	method	constructor	line	cbo	wmc	rfc	loc
paquete.ClaseAbstracta	noOverride/0	false	5	1	1	2	6
paquete.ClaseAbstracta	siOverride/0	false	12	0	1	0	1
paquete.InstanciaAbstracta	siOverride/0	false	10	2	1	2	5
paquete.InstanciaAbstracta	InstanciaAbstracta/0	true	5	0	1	0	2
paquete.InstanciaInterfaz	InstanciaInterfaz/0	true	5	0	1	0	2
paquete.InstanciaInterfaz	fInterfaz/0	false	10	1	1	2	5
paquete.Interfaz	fInterfaz/0	false	4	0	1	0	1
paquete.Main	main/1[java.lang.String[]]	false	13	2	1	3	6
paquete.Main	f2/0	false	28	0	1	1	3
paquete.Main	f1/0	false	20	2	1	4	7
paquete.Main	Main/0	true	8	1	1	2	4
paquete.Segundo	toString/0	false	19	0	1	2	4
paquete.Segundo	Segundo/0	true	9	0	1	1	3
paquete.Segundo	getList/0	false	14	0	1	0	3
paquete.Tercero	f8/0	false	24	0	1	1	3
paquete.Tercero	f7/0	false	19	1	1	2	4
paquete.Tercero	f6/0	false	14	1	1	2	4
paquete.Tercero	f5/0	false	9	1	1	2	4
paquete.Tercero	Tercero/0	true	5	1	1	1	3

Table 3: Fichero *methods.csv* (recortado)

C. ANEXO

Diagramas de secuencia

En este anexo se encuentran los diagramas de secuencia asociados a las diferentes funcionalidades del programa. Estos diagramas de secuencia se complementan con los diagramas de clases que se encuentran en el Capítulo 5. El objetivo de estos diagramas es dar a entender la interacción entre los diferentes objetos del programa, para entender mejor la ejecución del mismo. Se explican las funcionalidades principales.

- **Parámetros de entrada:** Figura C.1.
- **Funcionalidades principales del programa:** Figura C.2.
- **Visitar clases del análisis:** Figura C.3.
- **Visitar métodos del análisis:** Figura C.4.
- **Cálculo de métricas:** Figura C.5.
- **Exportar datos:** Figura C.6.

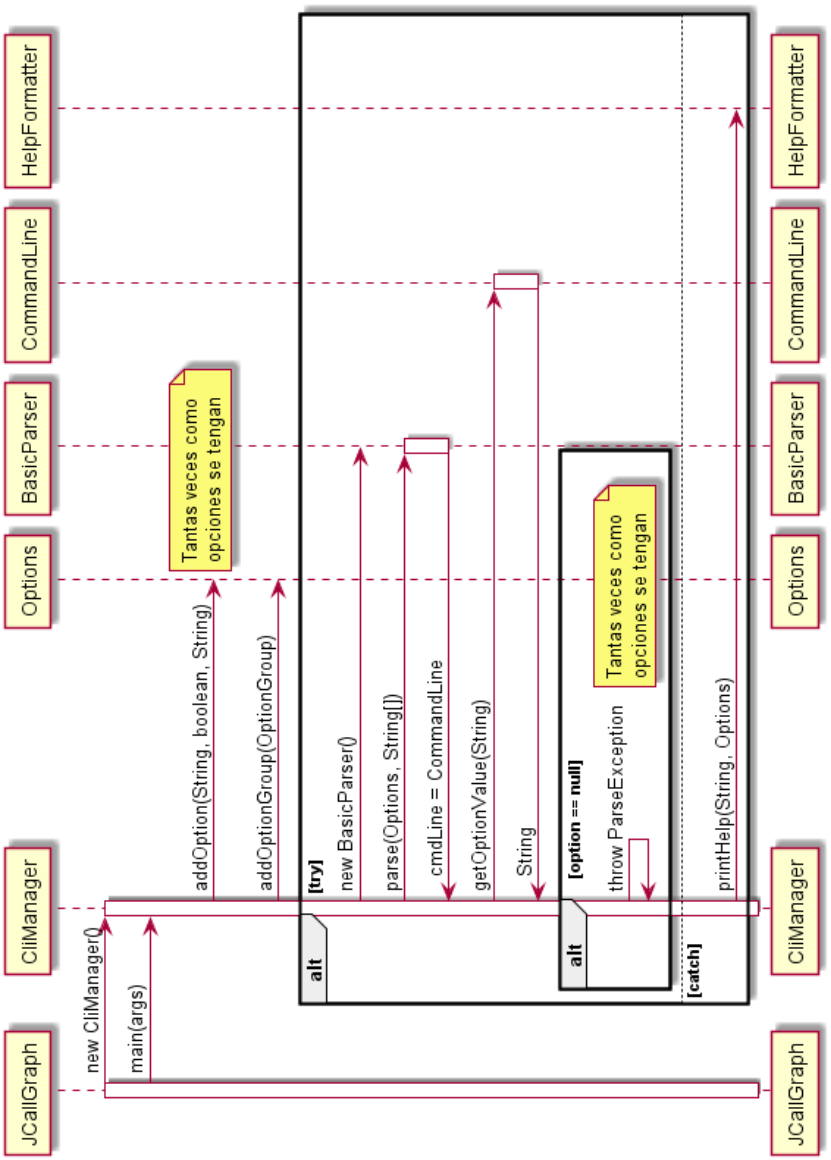

Figura C.1: Diagrama de secuencia de gestión de parámetros de entrada

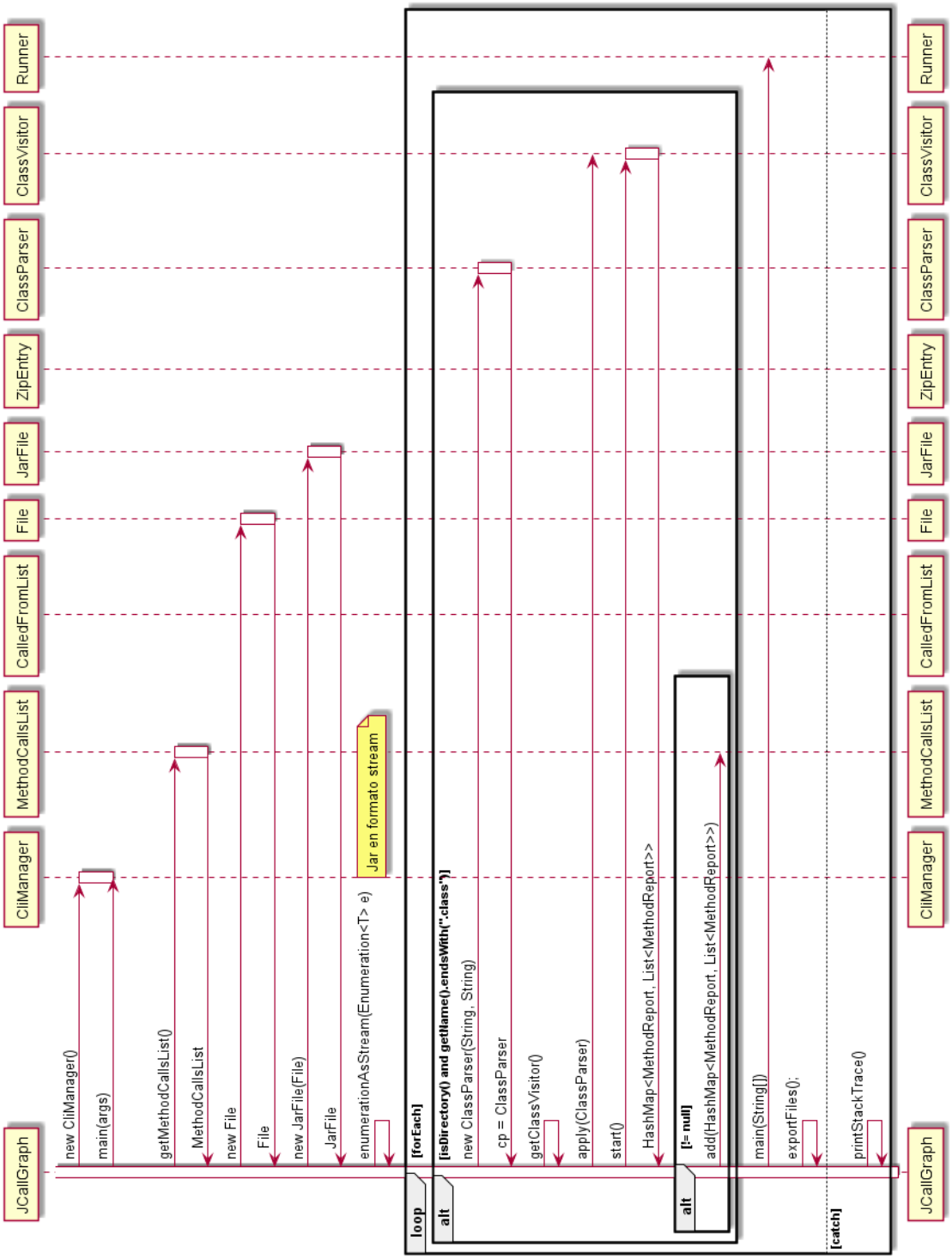

Figura C.2: Diagrama de secuencia de clase principal del programa

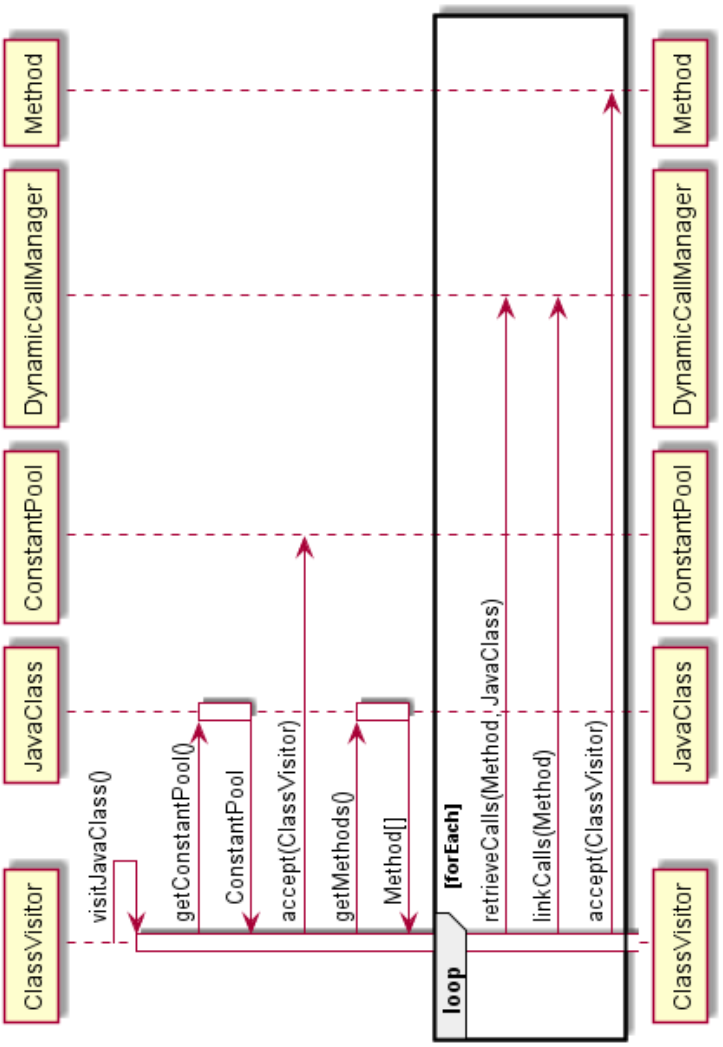

Figura C.3: Diagrama de secuencia de visita a las clases en un análisis

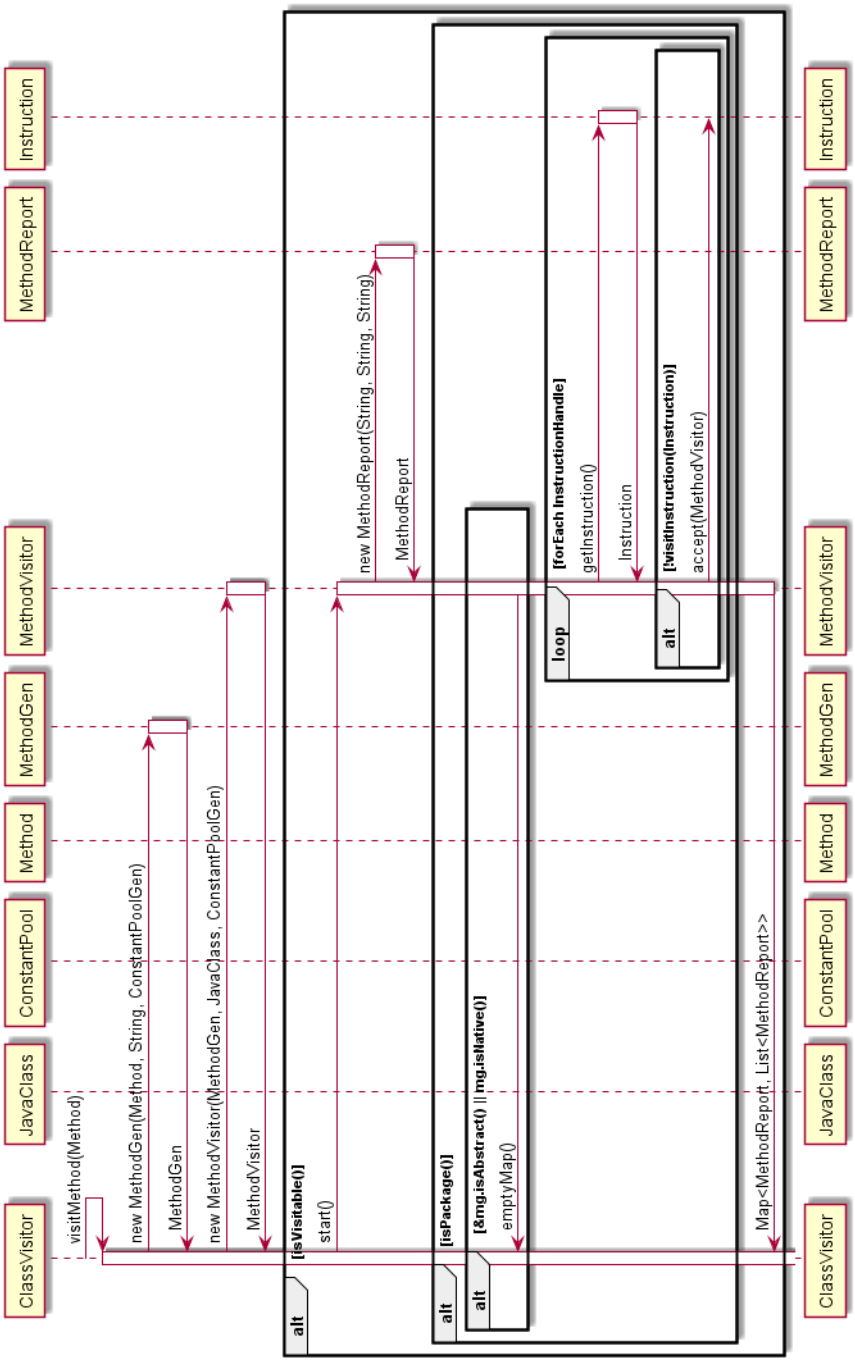

Figura C.4: Diagrama de secuencia de visita de los métodos en un análisis

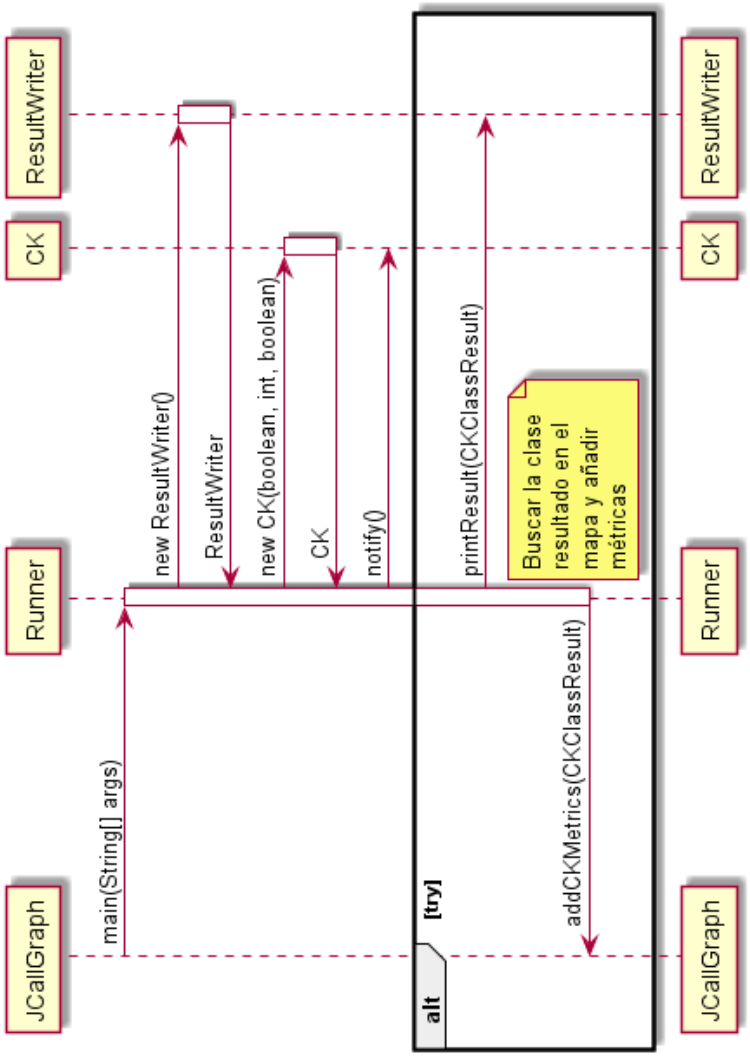

Figura C.5: Diagrama de secuencia de cálculo de métricas

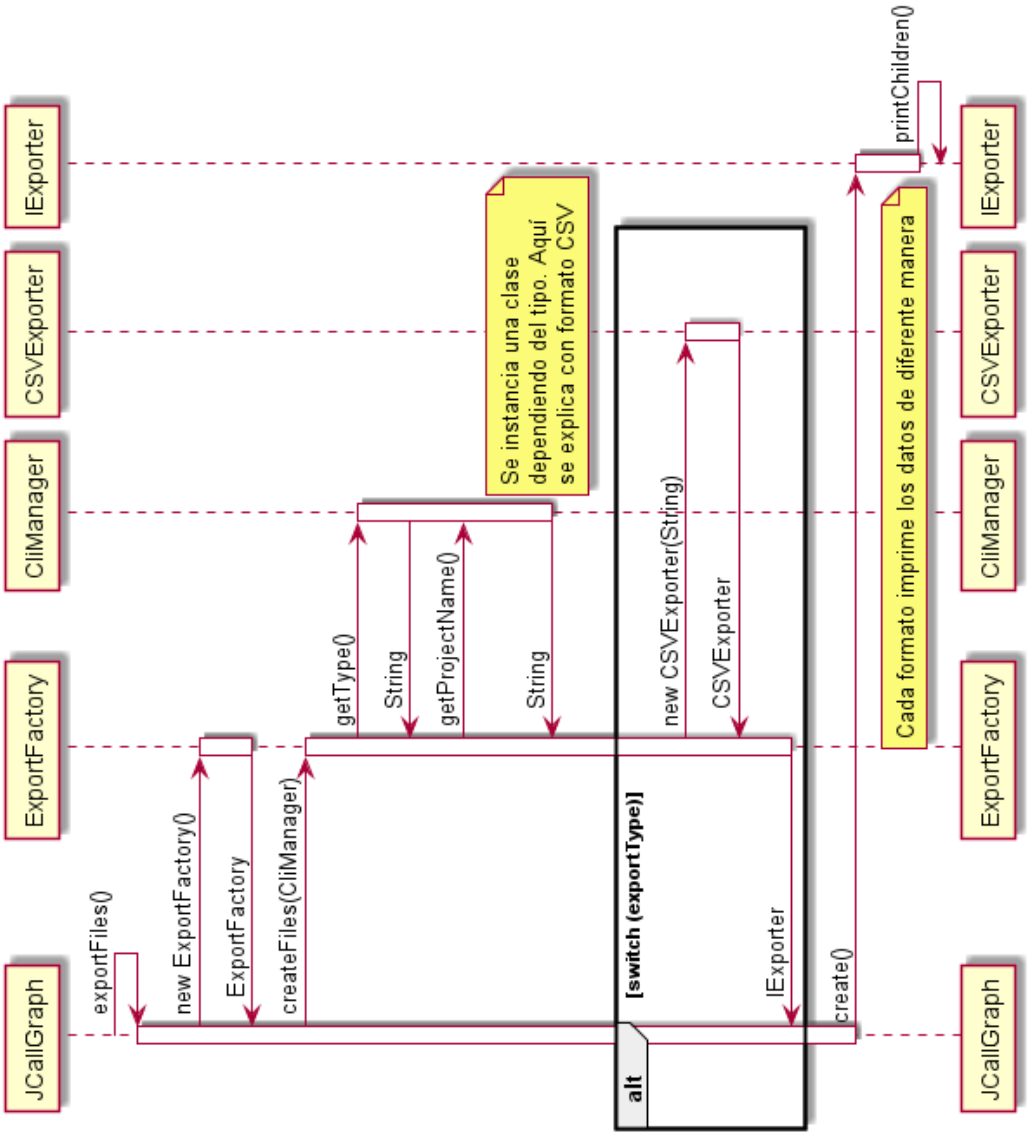

Figura C.6: Diagrama de secuencia de exportación de datos en ficheros

Bibliografía

[JVM, 2020] (2020). JVM - Java Virtual Machine Working and Architecture.

[Analysistools.dev, 2021] Analysistools.dev (2021). Picking the right static analysis tool for your use-case - static analysis tools, linters, code quality.

[Anouti, 2018] Anouti, M. (2018). Introduction to Java Bytecode - DZone Java.

[Antal et al., 2021] Antal, G., Tóth, Z., Hegedűs, P., and Ferenc, R. (2021). Enhanced bug prediction in javascript programs with hybrid call-graph based invocation metrics. volume 9.

[Apache, 2020] Apache (2020). Apache Commons BCEL™ – Home.

[Between.Com, 2018] Between.Com, D. (2018). Difference between source code and bytecode.

[Bourque and R.E. Fairley, 2014] Bourque, P. and R.E. Fairley, e. (2014). Guide to the software engineering body of knowledge, version 3.0. pages 177–180. IEEE Computer Society.

[Christodorescu and Jha, 2003] Christodorescu, M. and Jha, S. (2003). Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, page 12, USA. USENIX Association.

[Donzelli, 2006] Donzelli, P. (2006). A decision support system for software project management. *IEEE Software*, 3:67.

[Eclipse, 2020] Eclipse (2020). Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model.

[Feng et al., 2014] Feng, Y., Anand, S., Dillig, I., and Aiken, A. (2014). Apposcopy: semantics-based detection of Android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587, Hong Kong China. ACM.

[Gao et al., 2004] Gao, D., Reiter, M. K., and Song, D. (2004). Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, page 318–329, New York, NY, USA. Association for Computing Machinery.

[Ghahrai, 2017] Ghahrai, A. (2017). Static Analysis vs Dynamic Analysis in Software Testing.

[Harrison et al., 1997] Harrison, R., Counsell, S., and Nithi, R. (1997). An overview of object-oriented design metrics. In *Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering*, pages 230–235, London, UK. IEEE Comput. Soc.

[Hejderup et al., 2018] Hejderup, J., van Deursen, A., and Gousios, G. (2018). Software ecosystem call graph for dependency management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER '18, page 101–104, New York, NY, USA. Association for Computing Machinery.

[Honglei et al., 2009] Honglei, T., Wei, S., and Yanan, Z. (2009). The Research on Software Metrics and Software Complexity Metrics. In *2009 International Forum on Computer Science-Technology and Applications*, volume 1, pages 131–136.

[Javatutorial.net, 2017] Javatutorial.net (2017). JVM Explained | Java Tutorial Network.

[Jász. et al., 2019] Jász., J., Siket., I., Pengő., E., Ságodi., Z., and Ferenc., R. (2019). Systematic comparison of six open-source java call graph construction tools. In *Proceedings of the 14th International Conference on Software Technologies - ICSOFT,*, pages 117–128. INSTICC, SciTePress.

[Keim et al., 2008] Keim, D., Andrienko, G., Fekete, J.-D., Görg, C., Kohlhammer, J., and Melançon, G. (2008). Visual analytics: Definition, process, and challenges. In Kerren, A., Stasko, J. T., Fekete, J.-D., and North, C., editors, *Information Visualization: Human-Centered Issues and Perspectives*, pages 154–175. Springer Berlin Heidelberg.

[Lacerda et al., 2020] Lacerda, Guilherme, Petrillo, Fabio, Pimenta, Marcelo, Guéhéneuc, and Yann-Gaël (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. page 44.

[Lindholm et al., 2014] Lindholm, Tim, Yellin, Frank, Bracha, Gilad, Buckley, and Alex (2014). *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition.

[Mark, 1981] Mark, W. (1981). Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 439–449. IEEE Press.

[Oracle, 2021] Oracle (2021). The java language specification, java se 16 edition.

[Oreilly, 2021] Oreilly (2021). 4. Java Tools - Client-Server Web Apps with JavaScript and Java [Book]. ISBN: 9781449369330.

[Pengo and Ságodi, 2019] Pengo, E. and Ságodi, Z. (2019). A preparation guide for java call graph comparison. *Acta Cybern.*, 24:131–155.

[Pressman, 2009] Pressman, R. (2009). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., USA, 7 edition.

[Reddivari et al., 2013] Reddivari, S., Rad, S., Bhowmik, T., Cain, N., and Niu, N. (2013). Visual requirements analytics: a framework and case study. 19(3):257–279.

[Turhan et al., 2008] Turhan, B., koçak, G., and Bener, A. (2008). Software defect prediction using call graph based ranking (cgbr) framework. pages 191–198.

[Vogel et al., 2020] Vogel, L., Scholz, S., and Pfaff, F. (2020). Eclipse JDT - Abstract Syntax Tree (AST) and the Java Model.