Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Herramienta de Métricas de Calidad de Diseño Software en Aplicaciones para iPhone

Autor: Andy Ying Ye
Tutor(a): Ángel Herranz Nieva

Madrid, Junio 2024

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Herramienta de Métricas de Calidad de Diseño Software en Aplicaciones para iPhone

Junio 2024

Autor: Andy Ying Ye
Tutor: Ángel Herranz Nieva
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Agradecimientos a los tutores Jorge David Ortiz Fuentes y Ángel Herranz Nieva por proporcionar el esqueleto del código inicial y por guiarnos, instruirnos y resolver todas nuestras dudas durante este TFG. Finalmente, agradecer a mis compañeros Josué Rivero Morales y María Esther Colmenar Lamas por su compromiso y arduo trabajo durante este periodo.

Resumen

En este TFG se ha creado una herramienta para medir la calidad del diseño software de aplicaciones hechas en Swift mediante el uso de métricas de acoplamiento. El proyecto se ha realizado por un equipo compuesto por 3 personas, bajo la supervisión de 2 tutores. Aunque este TFG documentará solo una parte de la herramienta, todos los miembros han contribuido en todas las etapas del desarrollo.

La herramienta ha sido implementada utilizando el lenguaje Rust. La decisión del lenguaje se hizo pensando en la eficiencia y el posible uso que se le podría dar a la herramienta con otros lenguajes diferentes a Swift.

La herramienta funcionará de la siguiente manera, se le pasará un árbol sintáctico abstracto (ast) generado por el compilador de Swift, el cual se analizará creando tokens y, posteriormente, nodos. Con la información de estos nodos se creará una estructura intermedia en formato `json`. Tras esto, se analizará dicha estructura intermedia para poder realizar las métricas y concluir si un código es muy dependiente entre sus elementos.

Este TFG documentará el segundo paso del proceso, en el cual se generarán las estructuras intermedias mediante la información obtenida de los nodos creados en la primera parte. Estas estructuras serán la entrada del último paso en el que se calcularán las métricas de acoplamiento del código Swift inicial.

El trabajo realizado ha implicado una curva de aprendizaje significativa con respecto al lenguaje Rust, lenguaje que no se conocía en un inicio. Además, se ha realizado un gran esfuerzo a la gestión del repositorio Github en el que se encuentra la herramienta, así como a la implementación de test unitarios y de integración para garantizar la funcionalidad de todos los elementos de la herramienta.

En este enlace se encuentra el código fuente de la herramienta implementada: https://github.com/jdortiz/ast2md/tree/develop

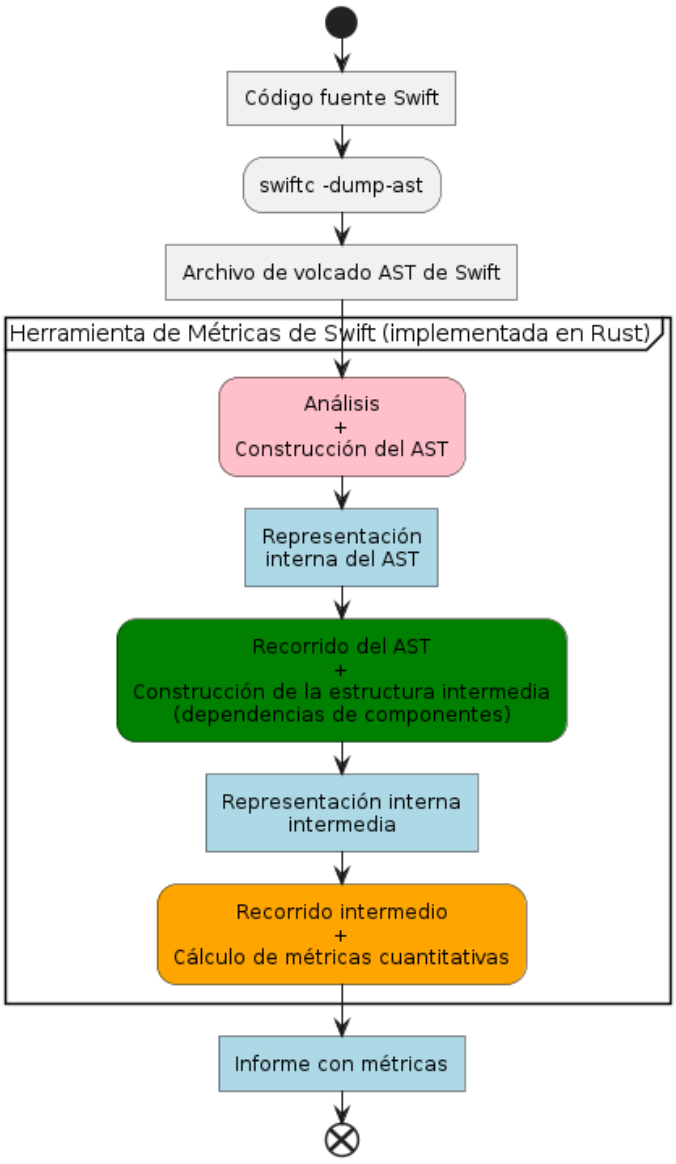

Figura 1: Diagrama de Actividad de la herramienta

Abstract

In this Bachelor's Thesis (TFG), a tool has been developed to measure the software design quality of applications developed in Swift by using coupling metrics. The project has been undertaken by a team comprising 3 individuals, under the supervision of 2 mentors. Although this TFG will document only a portion of the tool, all members have contributed across all stages of its development.

The tool has been implemented using the Rust programming language. The choice of language was made considering the efficiency and the potential usage of the tool with languages other than Swift.

The tool will operate as follows: it will be provided with an abstract syntax tree (ast) generated by the Swift compiler, which will be analyzed by creating tokens and subsequently nodes. Using the information from these nodes, an intermediate structure in `json` format will be created. This intermediate structure will then be analyzed to perform metrics and determine if a piece of code has high inter-element dependency.

This TFG will document the second step of the process, in which intermediate structures will be generated using the information obtained from the nodes created in the first part. These structures will serve as the input for the final step, where the coupling metrics of the initial Swift code will be calculated.

The work undertaken has entailed a significant learning curve regarding the Rust language, which was unfamiliar to the team at the project's outset. Furthermore, substantial effort has been dedicated to managing the GitHub repository housing the tool, as well as implementing unit and integration tests to ensure the functionality of all tool components.

Here is the source code for the implemented tool:

https://github.com/jdortiz/ast2md/tree/develop

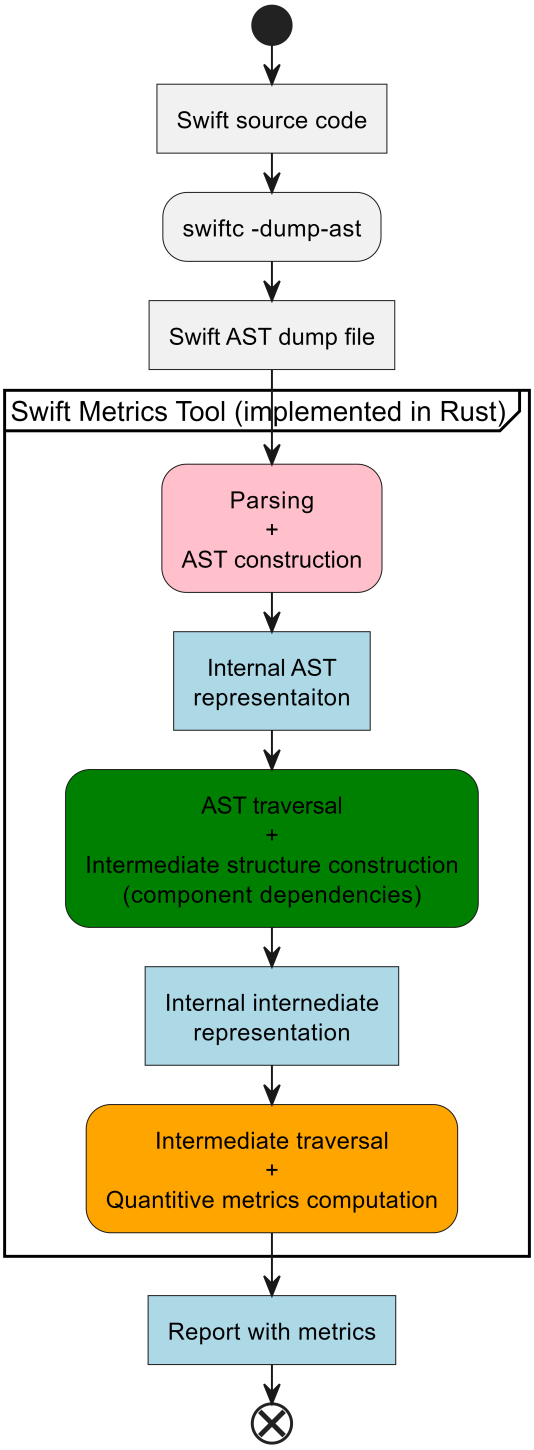

Figure 2: Tool Activity Diagram

Tabla de contenidos

Capítulo 1

Introducción

1.1. Repositorio de la herramienta

En este enlace se encuentra todo el código fuente de la herramienta:

https://github.com/jdortiz/ast2md/tree/develop

1.2. Motivación

Actualmente el mercado de dispositivos con iOS, el sistema operativo de dispositivos desarrollado por Apple, ha experimentado un crecimiento significativo y una creciente popularidad entre los usuarios. En el desarrollo de aplicaciones para estos dispositivos, el lenguaje de programación Swift ha emergido como el lenguaje predominante. Introducido por Apple en 2014, Swift ha ganado rápidamente la aceptación de la comunidad de desarrolladores debido a su facilidad de aprendizaje, seguridad y rendimiento optimizado. Estas características han hecho de Swift una opción atractiva para desarrolladores de todo tipo.

Apple

Swift

El lenguaje Swift, como la mayoría de lenguajes de programación, es una herramienta fundamental para asegurar que el código fuente esté bien estructurado y libre de errores sintácticos. Sin embargo, el compilador no tiene la capacidad de

determinar la calidad del código. La calidad del código es un aspecto fundamental que incluye factores como la mantenibilidad, la eficiencia y la modularidad, características importantes en un entorno profesional que pueden ahorrar mucho trabajo. En multitud de ocasiones, a pesar de las capacidades del compilador, necesitaremos información adicional para evaluar aspectos cualitativos del código. En este contexto, el compilador juega un papel crucial en nuestro proyecto, ya que es el que generará los árboles de sintaxis abstractos (ast: abstract syntax tree) [1] una representación del código fuente. Este ast es la base sobre la cual se llevará a cabo nuestro análisis para desarrollar una herramienta que tiene como propósito medir y mejorar la calidad del código.

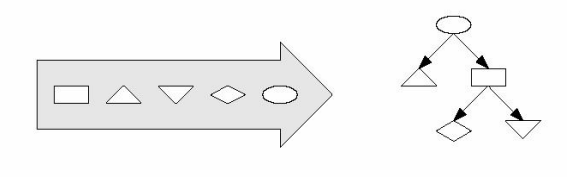

Figura 1.2: ast

Para dar más de contexto, este TFG se ha realizado en un contexto de un doctorado sobre patrones arquitectónicos de aplicaciones móviles, es decir, un proyecto de investigación, llevado a cabo por Jorge David Ortiz Fuentes bajo la supervisión de Ángel Herranz Nieva. Durante la investigación se dio con el problema de la ausencia de herramientas de análisis de códigos Swift, es por ello que se decidió hacer una propuesta de un TFG que ayudara a realizar la fase inicial de una herramienta de análisis como la buscada.

1.3. Objetivo

El objetivo principal de este Trabajo de Fin de Grado es medir la calidad del diseño software de las aplicaciones. Aunque nuestro enfoque inicial serán aquellas implementadas en Swift, la metodología y la herramienta desarrollada podría adaptarse fácilmente a otros lenguajes de programación.

Para lograr este objetivo, hemos optado por implementar la herramienta en el lenguaje Rust [2], un lenguaje de programación emergente que ha ganado popularidad por su enfoque en la seguridad y el rendimiento, comparable al de lenguajes como C y C++. Además, Rust ofrece garantías de seguridad de memoria y concurrencia que son esenciales para el desarrollo de software robusto y eficiente.

1.4. Descripción

La herramienta que proponemos proporcionará métricas de calidad del código fuente basadas en acoplamiento, una métrica que mide la dependencia entre los elementos del código [3]. Las diferentes métricas aplicadas pueden ofrecer

distintos resultados y perspectivas sobre la calidad del código. La herramienta analizará el ast generado por el compilador de Swift tras compilar el código fuente pasado como entrada y producirá estructuras intermedias que permitan aplicar estas métricas de manera efectiva. Así, se espera que los desarrolladores puedan obtener una visión más clara y detallada sobre la calidad de su código y realizar las mejoras necesarias. A continuación se proporciona un esquema general del funcionamiento de la herramienta 1.3, al cual se volverá en el capítulo de 3:

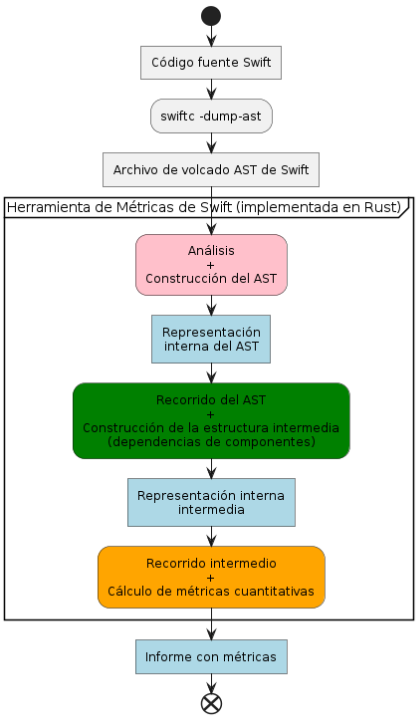

Figura 1.3: Diagrama de Actividad

Como se puede ver en el esquema, el primer paso de la ejecución de la herramienta será el parsing (análisis de texto para determinar su estructura lógica) del ast generado por el compilador de Swift. Este proceso generará una representación del ast, convirtiéndolo en nodos específicos que describiremos más adelante en la sección de 4. En el segundo paso, se generarán estructuras intermedias a partir de la representación interna del ast obtenida en el paso anterior. Finalmente, en el último paso, se aplicarán métricas sobre las estructuras intermedias generadas, obteniendo un informe que evalúe la calidad del código fuente proporcionado como entrada.

1.5. Colaboración y desarrollo en equipo

Este trabajo de fin de grado ha sido realizado en colaboración con dos compañeros más, debido al gran tamaño de la herramienta, lo que ha implicado un trabajo conjunto y coordinado. Esto se ha conseguido mediante la realización de reuniones semanales con todo el equipo, incluyendo a los tutores, donde se comentaban dudas y metas a conseguir durante esa semana. Además, hemos contado con el apoyo de un tutor académico, Ángel Herranz Nieva, y de un tutor externo, Jorge David Ortiz Fuentes, cuya contribución ha sido esencial para el desarrollo del proyecto.

1.6. Alcance del proyecto

Ya que el proyecto se ha realizado en equipo, se ha tratado de separar la documentación de la herramienta en tres partes, basados en los pasos visibles en el diagrama 1.3, los cuales se han descrito anteriormente. Por ello, en este TFG se documentará únicamente la parte de la generación de estructuras intermedias que se le pasarán a la parte del cálculo de métricas. Teniendo en cuenta que todas las partes están relacionadas, en ocasiones, se hará referencia y se explicarán términos de otras partes que se saldrían fuera del alcance de este TFG, ya que estos son imprescindibles para el entendimiento de esta parte.

1.7. Estado actual de la herramienta

Decir que, por un lado, la herramienta no está terminada al completo y lo que hemos hecho en este TFG es una primera versión de esa herramienta, la cual se seguirá desarrollando fuera del contexto de este trabajo; y, por otro, que toda la documentación localizada en el repositorio está en inglés, al estar pensado para el uso de todo tipo de usuario, desde comentarios hasta el nombre de variables dentro del propio código.

1.8. Estructura del documento

Durante el siguiente capítulo se dará un breve contexto del estado del arte en el contexto de las herramientas de análisis de código Swift. Posteriormente, se describirá el diseño de la herramienta. Tras eso, se explicará la implementación de la herramienta en detalle, viendo la estructura del proyecto, la metodología seguida y la ejecución de la herramienta. Finalizado el capítulo de implementación, se describirán y se mostrarán las pruebas realizadas para verificar el funcionamiento de la herramienta. Finalmente, encontraremos una conclusión del proyecto, con las dificultades encontradas y el impacto en ODS, y los pasos futuros a seguir. Al final se encuentra la bibliografía empleada.

Capítulo 2

Estado del arte

Actualmente, disponemos de numerosas herramientas para analizar un código de programación, similares a lo que propone nuestro TFG, tanto en otros lenguajes como en Swift. En esta sección describiremos algunas herramientas que abordan diferentes aspectos de la calidad del código, como la detección de errores o la identificación de código redundante. Aunque existen diversas herramientas para todo tipo de lenguajes, nos centraremos únicamente en aquellas que son útiles en el desarrollo de código en Swift.

En primer lugar, mencionaremos SonarQube [4], una potente herramienta que nos permite analizar un código, proporcionándonos información sobre vulnerabilidades de este o sobre la calidad del código, lo que nos ayudará a obtener métricas para optimizar el código. El uso de esta herramienta está enfocada a la revisión de código de forma automatizada, lo cual, nos garantiza el tener un código estandarizado y de calidad en todo momento. Además, la herramienta realiza pruebas sobre el código, lo que facilita la integración continua al detectar errores de forma temprana. Finalmente, destacar la cantidad de lenguajes que soporta la herramienta, siendo de 20 lenguajes. Las desventajas de esta herramienta son la complejidad de configuración y el coste computacional.

En segundo lugar tenemos Lizard [5], una herramienta de análisis de código que evalúa la complejidad de un código y se puede usar con Visual Studio Code. Para ello, detecta funciones largas, realiza un conteo de líneas, detecta código duplicado y puede realizar análisis de la complejidad ciclomática del código, que mide el numero de recorridos que se pueden seguir durante una ejecución. Aun así, Lizard ofrece una interfaz muy limitada y funcionalidades muy básicas, lo que reduce la utilidad de la herramienta en ciertas situaciones.

Otro ejemplo podría ser SwiftLint [6], una herramienta la cual nos permitirá generar informes sobre métricas de código como las mencionadas anteriormente. Se puede integrar en el proceso de desarrollo con otras herramientas como Xcode para verificar el cumplimiento de las normas de estilo y calidad, ayudando a mantener un código limpio y eficiente.

Asimismo, tenemos Arc [7], herramienta de desarrollo diseñada por Cubewise para trabajar con IBM Planning Analytics, una plataforma de planificación

y análisis que ayuda a las empresas a modelar, analizar y planificar sus datos. Arc nos proporciona, entre otras funcionalidades, una interfaz moderna, automatizaciones y métricas detalladas sobre el uso de memoria, el tiempo de procesamiento o indicadores clave de rendimiento (KPIs).

Además, una de las herramientas de análisis de métricas de código Swift, disponibles en Github, es swift-code-metrics [8], una herramienta que calcula varios tipos de métricas de un código Swift dado, entre las que se encuentran las métricas de acoplamiento.

Por último, SwiftMetrics [9] es una biblioteca del propio lenguaje Swift, la cual nos proporciona métricas para monitorear el rendimiento y la salud de un programa en tiempo de ejecución. Esta herramienta es especialmente útil para identificar cuellos de botella y optimizar el rendimiento del software en entornos de producción.

A pesar de la disponibilidad de numerosas herramientas de análisis de código, estas, en ocasiones, resultan ser demasiado complejas, afectando su rendimiento. Con este proyecto, se resolveremos el problema de los altos tiempos de ejecución, esto se debe que, por un lado, una de las principales características de Rust es la capacidad de ejecutar programas eficientemente, algo que no pueden ofrecer otros lenguajes de programación; y, por otro, el poco costo computacional que tiene la herramienta. Además, otra desventaja de las herramientas descritas anteriormente es la curva de aprendizaje que se requieren para usarlas, algo que no es un inconveniente en nuestro caso, ya que únicamente necesitaremos ejecutar un pipeline por línea de comandos. Otra característica de nuestra herramienta (aunque que esto no ha sido implementado aún) es la portabilidad, es decir, la capacidad de ejecutar la herramienta en múltiples sistemas operativos y lenguajes, aunque esto también lo implementan algunas de las herramientas anteriores.

Capítulo 3

Diseño

Este TFG abarcará todo lo relacionado con la generación de estructuras intermedias. Tras haber analizado el árbol sintáctico, generaremos estas estructuras, a las cuales se les añadirán propiedades que dependerán de la implementación de las métricas. Estamos ante un trabajo que no se entiende sin los trabajos de los demás compañeros, ya que cada parte es dependiente de la otra.

3.1. Metodología

3.1.1. Git Flow

La metodología seguida para la gestión del repositorio ha sido **Git Flow** [10], un modelo de creación de ramas en Git en el que tenemos ramas de principales y ramas función.

Las ramas principales estarán formadas por, *main* y *develop*, que contendrán la información publicada oficialmente y toda la implementación de las funciones integradas, respectivamente.

Por otro lado, se creará una nueva rama de función, *feature*, cada vez que se quiera implementar una nueva funcionalidad, utilizando *develop* como rama primaria. Tras implementar la funcionalidad se fusionará con *develop*, por lo que esta última rama pasará a contener la implementación de la rama función. Este proceso se realiza cada vez que se quiera añadir una funcionalidad nueva.

En el momento que se quiera publicar una nueva actualización del código en *main*, se deberá de crear una rama *release* a partir de *develop*. En esta rama se añadirá documentación y tareas relacionadas con la publicación, aunque, no se podrá añadir ninguna funcionalidad durante este proceso. Finalmente, cuando *release* está listo, se fusionará con *main* etiquetando la rama como una una nueva versión.

Por último, tenemos las ramas de corrección, las cuales se crean a partir de *main* para corregir ”bugs” críticos detectados. Es la única rama que se podrá crear a

partir de *main*. Cuando finalice la corrección se deberá de fusionar con *develop* (y release) y *main*, etiquetando *main* con una nueva versión.

Concluyendo, Git Flow nos ha permitido organizar el trabajo de forma estructurada, sin causar conflictos en el repositorio, y realizar este TFG de manera más eficiente.

3.1.2. Test-Driven Development

En cuanto a la metodología de los test hemos seguido **Test-Driven Development** (desarrollo dirigido por pruebas) [11], o *TDD*, una metodología iterativa de programación de pruebas, normalmente unitarias, en las que se crearán pruebas para cada aspecto o característica del código que se quiera probar, especificando el resultado esperado y comparándolo con el obtenido. Una vez superadas las pruebas se refactorizará el código para adaptarlo a las nuevas funcionalidades y mejorar la legibilidad.

Esta metodología nos ha facilitado la implementación, la depuración y la detección de errores en el código, acelerando el desarrollo de forma significativa.

3.1.3. Trabajo semanal

Asimismo, el modo de trabajo empleado para la realización de este trabajo de fin de grado ha consistido en reuniones semanales donde se discutía lo realizado en la semana anterior y pasos a seguir en la misma semana.

Durante cada semana se han empleado las metodologías descritas anteriormente. En el inicio del periodo se creaba una nueva rama *feature*, y durante la semana se implementaba la funcionalidad acordada en la reunión. Si se había conseguido implementar, pasando las pruebas unitarias, se creaba una *pull request* que era revisada por los tutores antes de fusionar la rama *feature* con *develop*.

3.1.4. Revisiones

Finalmente, durante el desarrollo, se han realizado code reviews semanales, revisando los comentarios hechos por los tutores tras realizar una *pull request* y depurando errores no detectados por las pruebas o causados, precisamente, por un error en la definición de las pruebas.

3.2. Arquitectura

En este punto se describirá la arquitectura de la herramienta. Volveremos al diagrama 1.3 expuesto varias veces durante esta memoria y lo estructuraremos por componentes y componentes internos, mostrando las interacciones entre estos.

3.2.1. Componentes

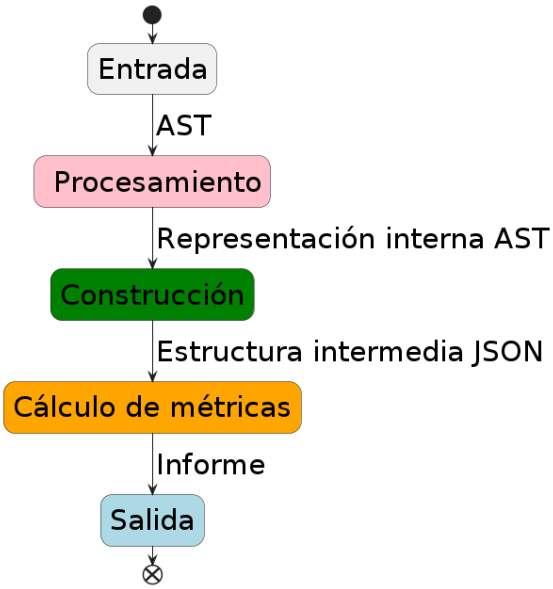

Figura 3.1: Diagrama de Actividad por Componentes

En el diagrama 3.1 se puede observar que se ha abstraído la información del diagrama 1.3 separándola en componentes. En este punto se dará una descripción para cada componente, especificando qué módulos las forman. Estos componentes son: Entrada, Procesamiento, Construcción, Cálculo de métricas y Salida.

Componente de entrada

Este componente se encargará de recibir y procesar las entradas, que se corresponden con el código fuente en Swift inicial. Tras el procesamiento se creará el ast, enviándolo al siguiente componente. Estará formado por el componente interno de linea de comandos.

Componentes internos de entrada

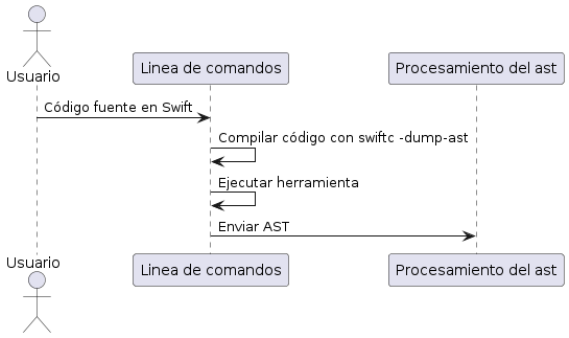

Figura 3.2: Diagrama de secuencia de componentes de entrada

En el diagrama de secuencia 3.2 podemos ver las interacciones del componente entrada con los demás componentes, ampliándolo al nivel de los componentes internos y describiendo las acciones que realizan esos componentes durante la ejecución de la herramienta. El proceso será el siguiente, el usuario proporcionara un código Swift a la línea de comandos, donde se compilará y se ejecutará la herramienta, pasando el ast generado durante la compilación al siguiente componente.

Componente de procesamiento

Este componente se encargará de procesar el ast recibido, creando Tokens para leer y Nodos que serán la salida que se le pasará al siguiente componente. Estará formado por el componente interno de procesamiento del ast y el de creación de nodos.

Componentes internos de procesamiento

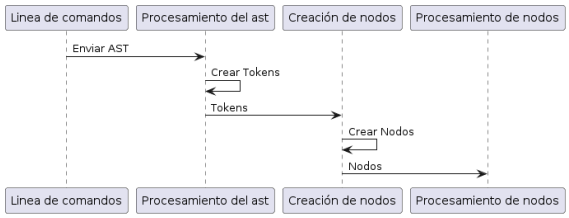

Figura 3.3: Diagrama de secuencia de componentes de procesamiento

En este caso, el diagrama 3.3 describirá el procesamiento del ast. En primer lugar se generarán ”Tokens”, que se le pasarán al componente de creación de nodos, que creará ”Nodos” que se enviarán para su procesamiento. Los términos ”Tokens” y ”Nodos” se describirán en el siguiente capítulo.

Componente de Construcción

Este componente se encargará de recibir y procesar la representación interna del ast, y, además, será el encargado de crear las estructuras intermedias que reco-

gerán la información necesaria para calcular las métricas. Estará formado por el componente interno de procesamiento de nodos y el de creación de estructuras.

Componentes internos de construcción

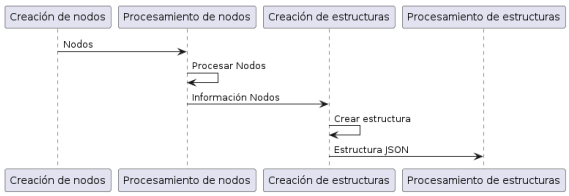

Figura 3.4: Diagrama de secuencia de componentes de construcción

Aquí, como indica el diagrama 3.4 se procesarán los nodos recibidos, pasando la información al componente de creación de estructuras, el cual tras crear estas estructuras, se las pasará al siguiente componente para el cálculo de las métricas. La creación de estructuras se hará de forma distinta dependiendo de la información recibida al procesar el nodo.

Componente de Cálculo de métricas

Este componente se encargará de recibir y procesar las estructuras intermedias, calculando las métricas de acoplamiento empleando estas estructuras. Estará formado por el componente interno de procesamiento de estructuras y el del cálculo de métricas.

Componentes internos de cálculo de métricas

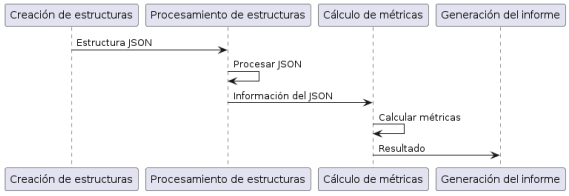

Figura 3.5: Diagrama de secuencia de componentes de métricas

Tras recibir la estructura `json`, se deberá de procesar en el componente interno de procesamiento de estructuras, para obtener la información y así poder calcular las métricas en el siguiente componente. El resultado de las métricas será transmitido al último componente de salida. Esto se puede observar en el diagrama 3.5.

Componente de salida

Este componente se encargará de recibir y mostrar el resultado del cálculo de las métricas. Estará formado por el componente interno de generación del informe y el componente interno de presentación.

Componentes internos de salida

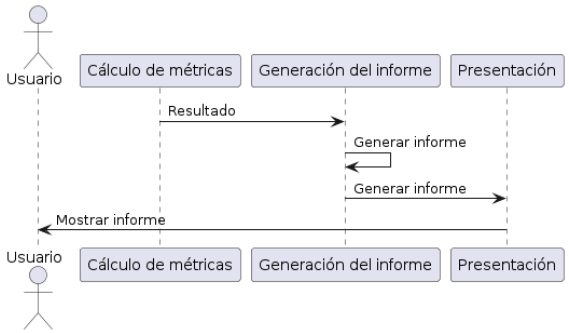

Figura 3.6: Diagrama de secuencia de componentes de salida

Finalmente, tras obtener el resultado de las métricas se generará el informe, que será enviado al componente interno de presentación, el cual se encargará de mostrar el informe al usuario, completando así la ejecución, como se observa en el diagrama de secuencia 3.6.

3.2.2. Tipos

Ya que el objetivo de esta memoria es documentar solo la parte de la creación de estructuras intermedias, solo se mostrarán los tipos relacionados con esta parte.

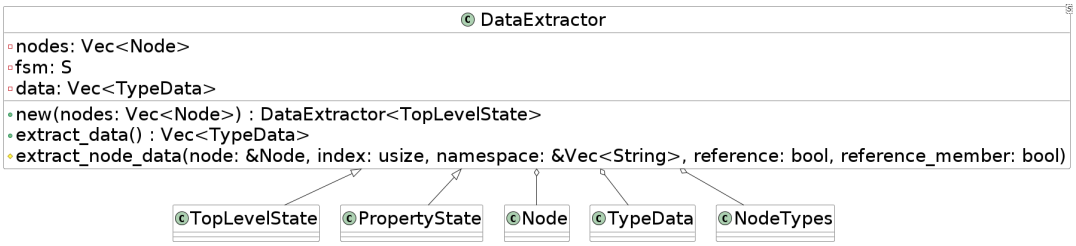

Figura 3.7: Diagrama de clases de `node_types.rs`

La información descrita en el diagrama 3.7 se corresponde a la clase que se encargará de leer los nodos recibidos al procesar el ast. Tendremos una función *`extract_data`* que leerá los nodos y una función *`extract_node_data`* que será llamada por la primera y la cual tiene la función de extraer la información del nodo. La función *`extract_node_data`* llamará a la función *`process`* de NodeTypes, donde se creará adecuadamente la estructura. A continuación se describirán las clases *`Node`*, *`TypeData`* y *`Node_Types`*

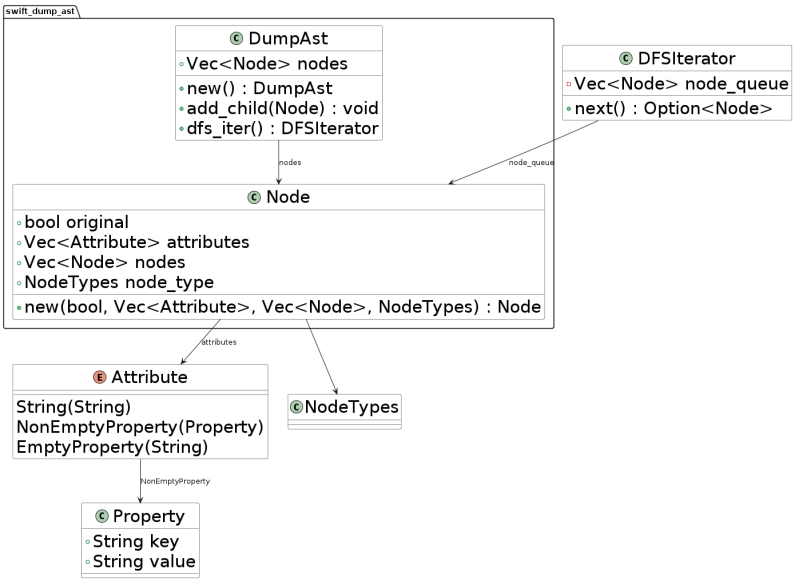

Figura 3.8: Diagrama de clases de Nodo

El diagrama 3.8 muestra la información contenida en un nodo. Un nodo contendrá atributos, un vector de nodos ”hijos” y el tipo de nodo. Además, la clase *Node* implementa un iterador, esto quiere decir que a *DataExtractor* se le pasará realmente una cola con múltiples nodos, el padre y todos sus hijos.

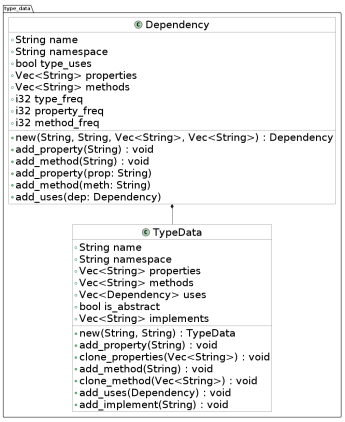

Figura 3.9: Diagrama de clases de TypeData

Por otro lado, el diagrama 3.9 nos muestra las propiedades y métodos que tendrá la clase *TypeData*, aunque esto se describirá más detalladamente en el siguiente capítulo, ya que estas estructuras son esenciales para la implementación de este TFG.

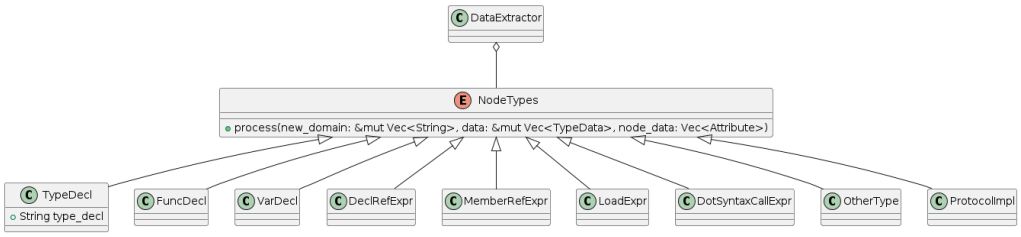

Figura 3.10: Diagrama de clases de `node_types.rs`

Finalmente, tenemos el diagrama 3.10 que nos muestra una enumeración de tipos de nodos, los cuales tendrán una implementación distinta entre ellos. Aunque esta parte también se describirá también en el siguiente capítulo.

Capítulo 4

Implementación

En primer lugar, recordar que este TFG cubrirá la parte de la generación de estructuras intermedias que se hará después de analizar el árbol sintáctico, y el cual se le pasará a la parte del cálculo de métricas.

4.1. Estructura del proyecto

Para la realización de este proyecto se ha decidido estructurar el código de la siguiente manera (esto se ha realizado usando el comando `tree` en una terminal `linux`):

```
ast2md
|-- ast2md-cli
|-- ast2mdlib
|-- Cargo.lock
|-- Cargo.toml
|-- couplinglib
|-- HACKING.md
|-- LICENSE.txt
|-- metcomp
|-- README.md
|-- target
|-- type_data
`-- ViewController.ast
```

Esta TFG hace uso de 3 carpetas principales, `ast2md-cli`, `ast2mdlib` y `type_data`, a continuación se explicará más en detalle la funcionalidad y el contenido de cada una de ellas. Además, se pueden ver otros archivos, `README.md`, `LICENSE.txt`, `Cargo.toml`, que configura la compilación, archivos y carpetas auto generadas (`.github`, `.gitignore`, `Cargo.lock`, `target`), y carpetas pertenecientes a la parte del cálculo de métricas, estas son `metcomp` y `couplinglib`.

```
ast2md-cli
|-- Cargo.toml
|-- src
```

```
    |-- cli_args.rs
    |-- generated.rs
    |-- main.rs
    `-- swift_dump_ast.pest
`-- tests
    |-- input
        |-- ClassesTypesUsed.ast
        |-- ClassesTypesUsedReverse.ast
        |-- ClassesTypesUsedReverse.swift
        |-- ClassesTypesUsed.swift
        |-- ClassTypeUsed.ast
        |-- ClassTypeUsed.swift
        |-- EmptyClass.ast
        |-- EmptyClass.swift
        |-- EmptyEnum.ast
        |-- EmptyEnum.swift
        |-- EmptyMultiStruct.ast
        |-- EmptyMultiStruct.swift
        |-- EmptyNestedStruct.ast
        |-- EmptyNestedStruct.swift
        |-- EmptyStruct.ast
        |-- EmptyStruct.swift
        |-- generate_ast.sh
        |-- MethodUsed.ast
        |-- MethodUsed.swift
        |-- NestedPropertiesAndMethodsUsed.ast
        |-- NestedPropertiesAndMethodsUsed.swift
        |-- NestedStructWithProperties.ast
        |-- NestedStructWithPropertiesIn2ndType.ast
        |-- NestedStructWithPropertiesIn2ndType.swift
        |-- NestedStructWithPropertiesIn3Types.ast
        |-- NestedStructWithPropertiesIn3Types.swift
        |-- NestedStructWithProperties.swift
        |-- PropertiesAndMethodsUsed.ast
        |-- PropertiesAndMethodsUsed.swift
        |-- PropertyAndMethodUsed.ast
        |-- PropertyAndMethodUsed.swift
        |-- PropertyUsed.ast
        |-- PropertyUsed.swift
        |-- Protocol1.ast
        |-- Protocol1.swift
        |-- Return.ast
        |-- Return.swift
        |-- StructWithMethods.ast
        |-- StructWithMethods.swift
        |-- StructWithProperties.ast
        |-- StructWithProperties.swift
```

```
        `-- ViewController.ast
    |-- integration_test.rs
    `-- output
        |-- ClassesTypesUsed.json
        |-- ClassesTypesUsedReverse.json
        |-- ClassTypeUsed.json
        |-- EmptyClass.json
        |-- EmptyEnum.json
        |-- EmptyMultiStruct.json
        |-- EmptyNestedStruct.json
        |-- EmptyStruct.json
        |-- MethodUsed.json
        |-- NestedPropertiesAndMethodsUsed.json
        |-- NestedStructWithPropertiesIn2ndType.json
        |-- NestedStructWithPropertiesIn3Types.json
        |-- NestedStructWithProperties.json
        |-- PropertiesAndMethodsUsed.json
        |-- PropertyAndMethodUsed.json
        |-- PropertyUsed.json
        |-- Protocol1.json
        |-- Return.json
        |-- StructWithMethods.json
        |-- StructWithProperties.json
        `-- ViewController.json
```

En esta carpeta nos encontraremos con la implementación de la línea de comandos, el script para que el compilador genere los ast y el main que se empleará para ejecutar la herramienta. Además tendremos una carpeta llamada `tests` donde hemos incluido los archivos swift, los asts y las estructuras esperadas de la ejecución con los que hemos hecho los test de golden file descritos en la el capítulo 5. Destacar `ViewController.ast`, el cual es un ejemplo de código real encontrado en la página de desarrollo de Apple, y el cual usamos para determinar futuros pasos a seguir llegados a un punto durante el desarrollo. Por último, nos encontraremos el archivo `integration_test.rs`, el contiene las pruebas de integración realizadas entre esta parte y el procesamiento del ast.

```
ast2mdlib
|-- Cargo.toml
`-- src
    |-- lexer.rs
    |-- lib.rs
    |-- metrics.rs
    |-- node_types.rs
    |-- parser.rs
    `-- swift_dump_ast.rs
```

Aquí se encuentra la implementación de la generación de estructuras intermedias y la implementación del análisis del ast, que quedaría fuera del alcance de este TFG. Dentro del directorio nos encontraremos con varios archi-

vos, de los cuales nos interesarán: `metrics.rs`, `parser.rs` y `lib.rs`. `lib.rs` nos permitirá importar archivos, por ejemplo, importar `swift_dump_ast.rs` en `metrics.rs`. Por otro lado, `metrics.rs` y `node_types.rs` que son los que implementan la recepción de los *Nodos* y el procesamiento de estos *Nodos*, respectivamente, aunque se explicarán más en detalle en la siguiente sección.

```
type_data
|-- Cargo.toml
`-- src
    `-- lib.rs
```

En este directorio nos encontraremos con un único archivo que contendrá la definición y la implementación de las estructuras comunes para todas las partes del proyecto. Estas estructuras son *TypeData* y *Dependency*, las cuales se explicarán en la sección de la Metodología seguida.

4.2. Descripción

4.2.1. Entrada del proceso

El primer paso del desarrollo, el cual quedaría fuera del alcance de este TFG, es leer el árbol sintáctico y analizarlo, generando *Tokens* que contendrán la información traducida. Con la información de los *Tokens* podremos crear las estructuras *Nodos*, cuyo tipo dependerá de la línea de entrada que se haya analizado(clase, propiedad, función...). Cada tipo de *Nodo* será tratado de forma distinta, y se descartarán los tipos que no nos interesen para el análisis de las métricas.

4.2.2. Salidas del proceso

Por otro lado, recordar que el objetivo de este trabajo es generar una estructura intermedia en formato `json`. Para ello, se ha creado una estructura llamada *TypeData*, la cual contendrá información sobre un bloque de un tipo(*class*, **struct**, **enum**). La estructura resultante será un vector de *TypeData*. Esta estructura contiene 7 campos:

- *name*: contiene el nombre del tipo.
- *namespace*: contiene el contexto en el que se está analizando el código.
- *properties*: tendrá las propiedades de la declaración de datos.
- *methods*: tendrá las funciones de la declaración datos.
- *uses*: contendrá las referencias a otras declaraciones. Este campo será esencial para desarrollar las métricas de la última parte.
- *is_abstract*: indica si el tipo es un protocolo o interfaz
- *implements*: indica el protocolo o interfaz que implementa la clase. Este campo será un vector y podrá contener varios protocolos a la vez.

El campo uses será el que nos dará la mayor parte de información con la que sacaremos las métricas de acoplamiento. Este campo uses será un vector de objetos *Dependency*, los cuales contendrán los siguientes campos:

- *name*: contiene el nombre del tipo usado.
- *namespace*: Contiene el contexto en el que se encuentra el tipo concreto.
- *type_uses*: indica si se ha inyectado código
- *properties*: contiene las propiedades usadas.
- *methods*: contiene los métodos usados.
- *type_freq*: contiene la frecuencia con la que se declara un objeto.
- *property_freq*: contiene la frecuencia con la que se usa una propiedad de esta clase.
- *method_freq*: contiene la frecuencia con la que se usa un método de esta clase.

Con todo esto tendremos nuestra estructura intermedia creada, junto a toda la información que requerimos para realizar las métricas de acoplamiento, clases y sus nombres, métodos y propiedades, de una clase propia u otra, frecuencias de uso de estas, estructura del código inicial...

4.2.3. Clases

main.rs

En primer lugar tendremos la clase `main.rs` localizado en `ast2md-cli/src`, la cual se encargará de iniciar el proceso de generación de estructuras intermedias. Esto se hará de la siguiente manera:

```
fn main() {
    let cli = CliArgs::parse();

    env::set_var("RUST_LOG", cli.level.to_string());
    env_logger::init();

    match cli.command {
        Command::Manual { filename: filepath } => {
            run_manual(filepath);
        }
        Command::Generated { filename: filepath } => {
            run_generated(filepath);
        }
    }
}
```

Se deberá de llamar a la función con la opción *manual*, lo que ejecutará la siguiente función *run_manual*:

```
fn run_manual(filepath: Option<PathBuf>) {
    let reader: Box<dyn BufRead> = match filepath {
        None => Box::new(BufReader::new(io::stdin())),
        Some(filepath) => Box::new(
        BufReader::new(fs::File::open(filepath).unwrap())),
    };

    let mut lexer = Lexer::new(reader.bytes());
    let tokens_and_error = lexer.tokenize();
    info!("Tokens: {tokens_and_error:?}\n\n***\n");

    let mut parser =
    ast2mdlib::Parser::new(tokens_and_error.0.into_iter());
    let result_tree = parser.create_dump_ast();
    info!("\n\n***\n\nDump AST tree: {result_tree:?}");
    if let Ok(ast_tree) = result_tree {
        let mut data_extractor =
        DataExtractor::new(ast_tree.nodes);
        let type_data = data_extractor.extract_data();
        print!("{}", serde_json::to_string(&type_data)
        .unwrap());
    } else {
        eprintln!("Err: {:?}", result_tree.err());
    }
}
```

Esta función leerá el ast pasado como argumento y generará los *Tokens* y, posteriormente, los *Nodos* correspondientes para generar las estructuras. Este proceso se ha realizado el las 12 primeras líneas del código anterior, aunque esto estará explicado en detalle en el TFG de mi compañera. Finalmente, una vez generados los *Nodos* se llamará a la función *extract_data* de la estructura *DataExtractor* localizada en `metrics.rs`, que se encargará de gestionar los *Nodos* y de devolver la estructura intermedia. Tras la ejecución de la función *extract_data* se imprimirá la estructura intermedia por línea de comandos.

metrics.rs

Como se ha mencionado en la sección anterior, la clase `metrics.rs` se encargará de la lectura y gestión de los *Nodos*, llamando a la función *extract_data* de la estructura *DataExtractor*. La implementación de la función será de la siguiente manera:

```
pub fn extract_data(&mut self) -> Vec<TypeData> {
    let child_nodes = self.nodes.to_vec();
    let namespace = vec![];
    for node in child_nodes.iter() {
        info!("Processing node: {:?}", node.attributes);
        let reference = false;
```

```
        let reference_member = false;
        self.extract_node_data(
        node, 0, &namespace, reference, reference_member);
    }

    self.data.to_vec()
}
```

La función tratará de gestionar todos los nodos padres junto con sus hijos, llamando a la función `extract_node_data` por cada nodo padre que encuentre, pasando ese nodo padre como argumento de la función. Esta última función, `extract_node_data`, se encontrará en el mismo contexto que `extract_data`, es decir, en la implementación de la estructura `DataExtractor`. A continuación se muestra la implementación de dicha función:

```
    fn extract_node_data(
        &mut self,
        node: &Node,
        index: usize,
        namespace: &Vec<String>,
        reference: bool,
        reference_member: bool,
    ) {
        ...
        if !node.original {
            if let NodeTypes::DeclRefExpr
            | NodeTypes::MemberRefExpr
            | NodeTypes::LoadExpr
            | NodeTypes::DotSyntaxCallExpr = node.node_type
            {
            ...
            } else {
                node.node_type
                    .process(&mut new_domain,
                    &mut self.data, node.attributes.clone());
            }
        }
        for child in node.nodes.iter() {
            self.extract_node_data(child,
            new_index, &new_domain, refer, refer_member);
        }
    }
```

La función llamará a la función `process` de la clase `nodetypes.rs`, la cual se encargará de procesar cada nodo según su tipo, el cual se encuentra dentro de las propiedades de cada `Nodo`. Tras finalizar el procesamiento, se volverá a llamar a la función `extract_node_data` de forma recursiva, con el objetivo de procesar todos los nodos hijos pertenecientes al nodo padre inicial con el que

se llamó a la función por primera vez en *extract_data*. La función permitirá mantener la jerarquía entre nodos padres e hijos, debido a la recursividad que mantiene la variable *new_domain* actualizada (esta variable nos dirá en que contexto se encuentra cada nodo, es decir, quienes son sus nodos padre). Se han omitido ciertas líneas del código que no nos interesan en este momento.

```
self.data.to_vec()
```

Por último, cuando *extract_data* haya procesado todos los nodos, devolverá la estructura generada, que será del tipo **Vec**<*TypeData*>, es decir, un vector de estructuras *TypeData*, las cuales fueron descritas en una sección anterior.

nodetypes.rs

Esta clase se encargará de procesar cada *Nodo* de manera distinta según su tipo. En primer lugar, la clase contendrá un **enum** con todos los tipos de *Nodo* (Este **enum** ha sido usado también en la parte del procesamiento del ast):

```
pub enum NodeTypes {
    TypeDecl(String),
    FuncDecl,
    VarDecl,
    DeclRefExpr,
    MemberRefExpr,
    LoadExpr,
    DotSyntaxCallExpr,
    OtherType,
    ProtocolImpl,
}
```

Este **enum** será implementado en esta clase, encontrándose dentro de esta implementación la función *process* ya mencionada:

```
impl NodeTypes {
    pub fn process(
        &self,
        new_domain: &mut Vec<String>,
        data: &mut Vec<TypeData>,
        node_data: Vec<Attribute>,
    ) {
        match self {
            NodeTypes::TypeDecl(type_decl) => {
                ...
            }
            NodeTypes::FuncDecl => {
                ...
            }
            NodeTypes::VarDecl => {
                ...
```

```
                }
                NodeTypes::DeclRefExpr => {
                    ...
                }
                NodeTypes::MemberRefExpr => {
                    ...
                }
                NodeTypes::ProtocolImpl => {
                    ...
                }
                NodeTypes::OtherType => {
                    ...
                }
                _ => {
                    warn!("Not matching with any node type defined");
                }
            }
        }
}
```

Como se observa, según el tipo del nodo se ejecutará una parte del código u otra. Esto se ha realizado empleando un **match**. A continuación se describirá el tratamiento de cada uno de los tipos de *Nodo* en la función *process*:

- *TypeDecl*: Para los tipos de declaración tendremos un parámetro que nos dirá ante que tipo de declaración estamos. El procesamiento consistirá en crear una nueva estructura *TypeData* cada vez que llega una estructura de este tipo, con el nombre obtenido del parámetro, además, cambiaremos el dominio actual a ese nombre, lo que querrá decir que nos encontramos dentro de esa estructura y que lo que llegue posteriormente pertenecerá a ella (Nodos hijos). No obstante, hay un caso especial que requerirá de otro procesamiento, este es, el caso de que la estructura sea una implementación de un protocolo (interfaz en otros lenguajes). En ese caso, lo que haremos será buscar el protocolo que implementa para poder determinar directamente las características de este *TypeData*, ya que los métodos y propiedades deberían de ser las mismas. Finalmente, se añadirá al campo *uses* la información del protocolo implementado y se pondrá en el campo *implements* el nombre del protocolo que implementa.
- *FuncDecl*: Si nos llega un *Nodo* del tipo declaración de función meteremos en los métodos de *TypeData* el nombre del método. Este tipo de *Nodo* se refiere exclusivamente a los métodos propios de la clase.
- *VarDecl*: Este caso tratará las declaraciones de variables de una clase, colocando en el campo *properties* la variable declarada, al igual que el caso anterior. Además, en este tipo se ha añadido una funcionalidad para asegurar un funcionamiento correcto cuando hay una clase dentro de otra, ya que, de lo contrario, hay casos en los que las variables se colocan dentro de la clase ”interior” cuando en realidad están declaradas fuera de esa

clase, osea en el dominio de la clase ”exterior”.

- *DeclRefExpr*: Aquí trataremos los casos en los que una clase llama a una función de otra clase. Tras haber obtenido el nombre de la función y el nombre de la clase a la que pertenece dicha función, las cuales aparecerán en un formato incorrecto, deberemos de incluir el nombre de la función en el campo de uses que se corresponde a la clase de la función. En caso de que la clase no esté incluida aún se creará una instancia dentro del campo uses, con el nombre de la clase y la función incluidos.
- *MemberRefExpr*: En este caso realizaremos lo mismo que en el tipo de *Nodo* anterior, con la única diferencia de que trataremos con una propiedad en vez de una función.
- *LoadExpr y DotSyntaxCallExpr*: Este *Nodo* el siguiente ha sido incluido aquí, pero realmente no tienen ningún tratamiento en la función *process*. La función de estos dos *Nodos* es la de indicar si el siguiente *Nodo* será una referencia a una propiedad(*MemberRefExpr*) o un método(*DeclRefExpr*) de otra clase, respectivamente. Esto se debe a que los *Nodos MemberRefExpr* y *DeclRefExpr* se utilizan para otras funcionalidades, por lo que no todos se referirán a lo descrito anteriormente.
- *ProtocolImpl*: El procesamiento de este *Nodo* será igual que el del *Nodo TypeDecl* en el caso normal, con la única diferencia de que ahora tendremos que poner el campo *is_abstract* a true.
- *OtherType*: En este caso, estaremos ante un *Nodo* que no nos interesa y no lo procesaremos, únicamente devolveremos un warning.

4.3. Ejecución

El comando para ejecutar la herramienta completa será:

```
swiftc -dump-ast ast2md-cli/input/<Nombre Archivo>.swift|
target/debug/ast2md-cli manual |
target/debug/metcomp full-frequency
```

Esto se hará desde el directorio raíz. La ejecución consiste en un pipeline de comandos, los cuales se explican a continuación. Con el primer comando tratamos de generar el ast, compilando el archivo localizado en el directorio ast2md-cli/tests/input. Con el segundo, cogiendo el ast devuelto por el anterior comando como argumento, generaremos la estructura intermedia (Objetivo de este TFG) que le pasaremos al tercer comando, también como argumento. Por último, el tercer comando calculará las métricas leyendo la estructura intermedia generada, especificando la estrategia a seguir, en este caso se ha usado full-frequency, que consiste en contar todas las interacciones entre los componentes del código. Esto nos devolverá una estructura que contendrá el resultado de aplicar las métricas. Finalmente, el pipeline nos devolverá un informe sobre el grado de dependencia del código, haciendo uso del resultado del cálculo anterior, aunque

esta funcionalidad no está muy avanzada en el momento que se escribe esta memoria.

4.3.1. Ejemplos de uso

En esta sección se mostrarán ejemplos de uso de la herramienta completa. En primer lugar, se ejecutará la herramienta con el siguiente código, `Protocol1.swift`, en lenguaje Swift:

```
protocol Vehicle {
    var isOn: Bool { get set }
    func start()
    func stop()
}

class Car: Vehicle {
    var isOn: Bool

    init(isOn: Bool){
        self.isOn = isOn
    }

    func start() {
        isOn = true
    }

    func stop() {
        isOn = false
    }
}

class MyCar {
    let myCar: Vehicle = Car(isOn:false)

    func carOperations() {
        myCar.start()
        myCar.stop()
        print("My car")
    }
}
```

El comando a ejecutar será el descrito en la sección 4.3, que para este caso será el siguiente:

```
swiftc -dump-ast ast2md-cli/input/Protocol1.swift|
target/debug/ast2md-cli manual |
target/debug/metcomp full-frequency
```

`swiftc -dump-ast ast2md-cli/input/Protocol1.swift`, generará el ast, el cual se le pasará al segundo comando para la generación de estructuras intermedias.

ast generado:

```
(source_file "Protocol1.swift"
  (protocol range=[Protocol1.swift:1:1 - line:5:1]
  "Vehicle" interface
  type='Vehicle.Protocol' access=internal
  non-resilient requirement
  signature=<Self>
    (pattern_binding_decl range=
    [Protocol1.swift:2:5 - line:2:30]
      (pattern_typed type='Bool'
        (pattern_named type='Bool' 'isOn')
        (type_ident id='Bool' bind=Swift.
        (file).Bool)))
    (var_decl range=
    [Protocol1.swift:2:9 - line:2:9] "isOn"
    type='Bool' interface type='Bool'
    access=internal
    readImpl=getter writeImpl=setter
    readWriteImpl=modify_coroutine
 ...
              (argument label=terminator
               (default_argument_expr implicit
               type='String'
               location=Protocol1.swift:29:14 range=
               [Protocol1.swift:29:14 - line:29:14]
               default_args_owner=
               Swift.(file).print(_:separator:terminator:)
               param=2))))))
    (destructor_decl implicit range=
    [Protocol1.swift:23:7 - line:23:7]
    "deinit" interface type='(MyCar) -> () -> ()'
    access=internal
      (parameter "self")
      (parameter_list)
      (brace_stmt implicit range=
      [Protocol1.swift:23:7 - line:23:7]))
    (constructor_decl implicit range=
    [Protocol1.swift:23:7 - line:23:7]
    "init()" interface type=
    '(MyCar.Type) -> () -> MyCar'
    access=internal designated
      (parameter "self")
      (parameter_list)
      (brace_stmt implicit range=
      [Protocol1.swift:23:7 - line:23:7]
        (return_stmt range=
        [Protocol1.swift:23:7 - line:23:7])))))
```

La salida esperada tras ejecutar, `target/debug/ast2md-cli manual`, el segundo comando, será la siguiente:


```
[
    {
        "name": "Vehicle",
        "namespace": "",
        "properties": [
            "isOn"
        ],
        "methods": [
            "start()",
            "stop()"
        ],
        "uses": [],
        "is_abstract": true,
        "implements": []
    },
    {
        "name": "Car",
        "namespace": "",
        "properties": [
            "isOn"
        ],
        "methods": [
            "start()",
            "stop()"
        ],
        "uses": [
            {
                "name": "Vehicle",
                "namespace": "",
                "type": true,
                "properties": [
                    "isOn"
                ],
                "methods": [
                    "start()",
                    "stop()"
                ],
                "type_freq": 1,
                "property_freq": 1,
                "method_freq": 2
            }
        ],
        "is_abstract": false,
        "implements": ["Vehicle"]
    },
```

```
    {
        "name": "MyCar",
        "namespace": "",
        "properties": [
            "myCar"
        ],
        "methods": [
            "carOperations()"
        ],
        "uses": [
            {
                "name": "Vehicle",
                "namespace": "",
                "type": true,
                "properties": [],
                "methods": [
                    "start()",
                    "stop()"
                ],
                "type_freq": 1,
                "property_freq": 0,
                "method_freq": 2
            }
        ],
        "is_abstract": false,
        "implements": []
    }
]
```


Al tercer comando, `target/debug/metcomp full-frequency`, se le pasará esta estructura. El comando calculará las métricas de acoplamiento del código fuente inicial, empleando la información de la estructura intermedia siguiendo la estrategia **FullFreqMetric**, la cual calculará las métricas usando todas las frecuencias acumuladas. El resultado de ejecutar este tercer comando será el siguiente:


```
[
  {
    "type": "Vehicle",
    "namespace": "",
    "coupling": []
  },
  {
    "type": "Car",
    "namespace": "",
    "coupling": [
      {
        "type": "Vehicle",
```

```
        "namespace": "",
        "strategy": "full-frequency",
        "value": 4,
        "assessment": "high"
      }
    ]
  },
  {
    "type": "MyCar",
    "namespace": "",
    "coupling": [
      {
        "type": "Vehicle",
        "namespace": "",
        "strategy": "full-frequency",
        "value": 3,
        "assessment": "normal"
      }
    ]
  }
]
```

Como vemos en la estructura, la propiedad **assesment** será la que contenga el resultado del calculo de las métricas, que es la que medirá el grado de acoplamiento. Al terminar la ejecución se imprimirá la estructura por consola a modo de informe.

Otro ejemplo de uso podría ser el siguiente:

```
struct Type2 {
  var a: Int
  var b: Int
  func doSomething() {}
  func doSomethingToo() {}

}
struct Type1 {
  var c: Type2
  var d: Int
  mutating func doSomethingElse() {
      d = c.a
      d = c.b
      c.doSomething()
      c.doSomethingToo()
  }
}
```

Este código es `PropertiesAndMethodsUsed.swift` y el comando a ejecutar será:

```
swiftc -dump-ast
ast2md-cli/input/PropertiesAndMethodsUsed.swift|
target/debug/ast2md-cli manual |
target/debug/metcomp no-protocol-freq
```

En este caso, se volverá a generar el ast con el comando

```
swiftc -dump-ast ast2md-cli/input/PropertiesAndMethodsUsed.swift.
```

ast generado:

```
(source_file "PropertiesAndMethodsUsed.swift"
  (struct_decl range=
  [PropertiesAndMethodsUsed.swift:1:1 - line:7:1]
  "Type2" interface type=
  'Type2.Type' access=internal non-resilient
    (pattern_binding_decl
    range=[PropertiesAndMethodsUsed.swift:2:3
    - line:2:10]
      (pattern_typed type='Int'
        (pattern_named type='Int' 'a')
        (type_ident id='Int' bind=Swift.(file).Int)))
    (var_decl range=
    [PropertiesAndMethodsUsed.swift:2:7 - line:2:7]
    "a" type='Int' interface type='Int'
    access=internal readImpl=
    stored writeImpl=stored
    readWriteImpl=stored
      (accessor_decl implicit range=
      [PropertiesAndMethodsUsed.swift:2:7
      - line:2:7] 'anonname=0x55a95dec5700'
      interface type='(Type2)
      -> () -> Int' access=internal get_for=a
        (parameter "self" type='Type2'
        interface type='Type2')
        (parameter_list)
        (brace_stmt implicit range=
        [PropertiesAndMethodsUsed.swift:2:7
        - line:2:7]
          (return_stmt implicit
          ...
    (constructor_decl implicit range=
    [PropertiesAndMethodsUsed.swift:8:8 -
    line:8:8] "init(c:d:)"
    interface type='(Type1.Type)
    -> (Type2, Int) -> Type1'
    access=internal designated
      (parameter "self")
      (parameter_list range=
```

```
      [PropertiesAndMethodsUsed.swift:8:8
      - line:8:8]
        (parameter "c" apiName=c
        type='Type2' interface type='Type2')
        (parameter "d" apiName=d
        type='Int' interface type='Int')))))
```

Por otro lado, la salida esperada tras ejecutar el segundo comando `target/debug/ast2md-cli manual`será la siguiente:

```
[
 {
   "name": "Type2",
   "namespace": "",
   "properties": ["a","b"],
   "methods": ["doSomething()", "doSomethingToo()"],
   "uses": [],
   "is_abstract": false,
   "implements": []
},
{
   "name": "Type1",
   "namespace": "",
   "properties": ["c","d"],
   "methods": ["doSomethingElse()"],
   "uses": [{"name": "Type2","namespace": "","type": true,
   "properties": ["a","b"],"methods":
   ["doSomething()", "doSomethingToo()"],
   "type_freq": 1,"property_freq":2,"method_freq":2}],
   "is_abstract": false,
   "implements": []
}
]
```

Finalmente, en comparación con el ejemplo anterior, esta vez el tercer comando `target/debug/metcomp no-protocol-freq` usará la estrategia **NoProtocolFrec**, el cual usará las frecuencias que no procedan de protocolos o interfaces, dando el siguiente resultado:

```
[
   {
      "type": "Type2",
      "namespace": "",
      "coupling": []
   },
   {
      "type": "Type1",
      "namespace": "",
```

```
      "coupling": [
         {
            "type": "Type2",
            "namespace": "",
            "strategy": "no-protocol-freq",
            "value": 5,
            "assessment": "high"
         }
      ]
   }
]
```


Esta vez el análisis ha dado un valor alto de dependencia. Al finalizar se imprime el informe por consola.

Capítulo 5

Pruebas

Como se ha mencionado anteriormente en el capítulo 3, se ha empleado la metodología Test-Driven Development, lo que ha garantizado el funcionamiento de la herramienta cuando todas las funcionalidades pasan las pruebas. Además, nos ha permitido detectar errores de forma temprana, aumentando la productividad a la hora de depurar, refactorizar e implementar la herramienta. Por último, la realización de pruebas ha mejorado la mantenibilidad de nuestro código, modulando y haciendo más legible el código, lo que quiere decir que tras cada modificación del código se comprobará la funcionalidad modificada individualmente.

5.1. Tipos de pruebas

Durante el desarrollo de la herramienta se han usado diferentes tipos de pruebas, cada uno con un objetivo concreto. A continuación se muestran los tipos de pruebas empleadas y una breve descripción sobre cada una [12]:

- **Pruebas Unitarias**: Este tipo de pruebas de bajo nivel consisten en probar todos los métodos y funciones del software individualmente. Las pruebas unitarias nos permitirán detectar errores concretos de forma temprana y garantizar el funcionamiento de una funcionalidad concreta.

- **Pruebas de Integración**: Este tipo de pruebas tienen como objetivo comprobar que todos los componentes o módulos funcionan bien conjuntamente. Este tipo de pruebas nos permitirá paralelizar la implementación de varios módulos de la herramienta a la vez, reduciendo el tiempo de desarrollo y aumentando la productividad, con la certeza de que el comportamiento entre las diferentes partes se verificará posteriormente.

- **Pruebas de extremo a extremo**: Este tipo de prueba tratan de replicar la experiencia de un usuario desde que ejecuta la herramienta hasta que termina la ejecución. Esto nos servirá para verificar si el comportamiento del software es el que se espera.

5.2. Tests Golden File

Además de los tipos de pruebas mencionadas, hemos empleado los ”tests golden file” que consisten en comprobar que la salida de generada por el software no ha cambiado drásticamente. Este tipo de prueba no es diferente a una prueba común, pero tienen la peculiaridad de que la salida esperada se define en un fichero que es a lo que denominamos ”Golden File”. El uso de este tipo de ficheros en nuestro caso, se debe al gran tamaño que tiene cada estructura intermedia, lo que causaría que el código se extendiera significativamente. Esto se ha usado como prueba de integración entre la primera parte del proyecto, el procesamiento del ast, y la parte que documenta este TFG, y entre la parte del cálculo de métricas y la de este TFG. En la siguiente sección se mostrará un ejemplo.

5.3. Ejemplos de uso

5.3.1. Pruebas unitarias

En el proyecto se han implementado pruebas unitarios para garantizar el funcionamiento de la herramienta en los casos contemplados. Estos test unitarios nos ayudan a detectar errores mediante la comparación de los resultados esperados y los obtenidos. Se han empleado en `node_types.rs` para verificar el funcionamiento de la función *process*, por lo que se tuvieron que añadir *Nodos* manualmente para procesarlos y comparar la salida obtenida con la esperada.

A continuación se muestran algunos ejemplos de estos tests:

```
#[test]
fn process_func_decl() {
    let sut = Node::new(
        false,
        vec![
            Attribute::NonEmptyProperty(Property {
                key: "range".to_string(),
                value:
                    "Tests/ProQuaMeMeTests/testFiles/
                    StructWithProperties.swift:1:1
                    - line:6:1"
                        .to_string(),
            }),
            Attribute::NonEmptyProperty(Property {
                key: "range".to_string(),
                value:
                    "Tests/ProQuaMeMeTests/testFiles/
                    StructWithProperties.swift:1:1
                    - line:6:1"
                        .to_string(),
            }),
            Attribute::String("doSomething()"
```

```
                .to_string()),
            ],
            vec![],
            NodeTypes::FuncDecl,
        );

        let mut type_data_vector = vec![
        TypeData::new("TypeA".to_string(), "".to_string())];
        let mut new_domain = vec!["TypeA".to_string()];
        sut.node_type
            .process(&mut new_domain,
            &mut type_data_vector, sut.attributes);

        assert_eq!(
            type_data_vector.last().unwrap().methods[0],
            "doSomething()".to_string()
        )
    }
```

En este test se trata de comprobar el comportamiento de la función *process*, comprobando si se ha incluido correctamente la función *doSomething* en el vector de métodos. En este caso la variable *sut* contendrá el *Nodo* a procesar. Tras procesar el nodo se comprueba si el contenido de la lista de métodos es el correcto.

```
    #[test]
    fn process_var_decl() {
        let sut = Node::new(
            false,
            vec![
                Attribute::NonEmptyProperty(Property {
                    key: "range".to_string(),
                    value:
                        "Tests/ProQuaMeMeTests/testFiles/
                        StructWithProperties.swift:1:1
                        - line:6:1".to_string(),
                }),
                Attribute::NonEmptyProperty(Property {
                    key: "range".to_string(),
                    value:
                        "Tests/ProQuaMeMeTests/testFiles/
                        StructWithProperties.swift:1:1
                        - line:6:1".to_string(),
                }),
                Attribute::String("a".to_string()),
            ],
            vec![],
            NodeTypes::VarDecl,
```

```
        );

        let mut type_data_vector = vec!
        [TypeData::new
        ("TypeA".to_string(), "".to_string())];
        let mut new_domain =
        vec!["TypeA".to_string()];
        sut.node_type
            .process(&mut new_domain,
            &mut type_data_vector, sut.attributes);

        assert_eq!(
            type_data_vector.last().unwrap().properties[0],
            "a".to_string()
        )
    }
```

Este test es similar al anterior, con la diferencia de que ahora estaremos comprobando el comportamiento de la función *process* cuando recibe una variable. Se verificará, por tanto, si se ha añadido correctamente la propiedad a la lista de propiedades tras procesar el nodo.

5.3.2. Pruebas de integración

En el proyecto se han usado pruebas de integración para, por un lado, verificar el funcionamiento conjunto entre el módulo de procesamiento de nodos y el de generación de estructuras intermedias (tests golden file), y, por otro, para verificar el funcionamiento del módulo de generación de estructuras intermedias y el de cálculo de métricas (se hará referencia a este tipo de pruebas como pruebas B-C). Debido a que este TFG se encuentra entre las otras partes este tipo de pruebas son las más importantes, ya que son fundamentales para el correcto funcionamiento de la herramienta completa.

Tests Golden File

La ejecución de este tipo de pruebas se encuentra en el archivo `ast2md-cli/tests/integration_test.rs`, siendo de la siguiente manera:

```
#[rstest]
fn test_golden_file(
    #[files("tests/input/*.ast")] input: PathBuf,
) -> Result<(), Box<dyn std::error::Error>> {
    let mut cmd = Command::cargo_bin(env!("CARGO_PKG_NAME"))?;

    let dir = input.parent().expect("Invalid input directory");
    cmd.current_dir(dir);
    let file_name = input.file_name()
    .expect("Invalid input file name");
```

```
    cmd.arg("manual").arg(file_name);
    let output = cmd.output().
    expect("Failed to execute command");

    assert!(output.status.success(),
    "Unexpected execution status");

    let actual_string = String::from_utf8(output.stdout)
    .expect("Invalid output");
    let actual: Vec<TypeData> =
        serde_json::from_str(&actual_string)
        .expect("Invalid output JSON format");

    let mut output_path = input.clone();
    output_path.pop(); // filename
    output_path.pop(); // input
    output_path.push("output");
    output_path.push(Path::new(file_name)
    .with_extension("json"));
    let expected_string = fs::read_to_string(output_path)
    .expect("Invalid expected output");
    let expected: Vec<TypeData> =
        serde_json::from_str(&expected_string)
        .expect("Invalid expected JSON output");
    assert_eq!(actual, expected,
    "Actual and expected output don't match");
    println!("Actual: {actual:?}\nExpected: {expected:?}");

    Ok(())
}
```

Esto se ejecutará tantas veces como archivos ".ast" hayan en `ast2md-cli/tests/input`. Un ejemplo de esta ejecución puede ser el siguiente:

Código Swift:

```
    struct Type1 {
        struct Type2 {

            var prop1: Int
            let prop2: Float
            var prop3: String
            var prop4: [String]
        }
    }
```

ast:

```
(source_file "NestedStructWithPropertiesIn2ndType.swift"
```

```
(struct_decl range=
[NestedStructWithPropertiesIn2ndType.swift:1:1 - line:9:1]
"Type1" interface type='Type1.Type' access=internal
non-resilient
  (struct_decl range=
  [NestedStructWithPropertiesIn2ndType.swift:2:5
  - line:8:5] "Type2" interface type='Type1.Type2.Type'
  access=internal non-resilient
    (pattern_binding_decl range=
    [NestedStructWithPropertiesIn2ndType.swift:4:9
    - line:4:20]
      (pattern_typed type='Int'
        (pattern_named type='Int' 'prop1')
        (type_ident id='Int' bind=Swift.(file).Int)))
    (var_decl range=
    [NestedStructWithPropertiesIn2ndType.swift:4:13
    - line:4:13] "prop1" type='Int' interface type='Int'
    access=internal readImpl=stored writeImpl=stored
    readWriteImpl=stored
    ...
  (constructor_decl implicit range=
  [NestedStructWithPropertiesIn2ndType.swift:1:8 - line:1:8]
  "init()" interface type='(Type1.Type) -> () -> Type1'
  access=internal designated
    (parameter "self")
    (parameter_list)
    (brace_stmt implicit range=
    [NestedStructWithPropertiesIn2ndType.swift:1:8 - line:1:8]
      (return_stmt range=
      [NestedStructWithPropertiesIn2ndType.swift:1:8
      - line:1:8])))))
```

Estructura esperada:

```
[
  {
    "properties": [],
    "namespace": "",
    "methods": [],
    "uses": [],
    "name": "Type1",
    "is_abstract": false,
    "implements": []
  },
  {
    "properties": [
      "prop1",
      "prop2",
```

```
        "prop3",
        "prop4"
      ],
      "namespace": "Type1",
      "methods": [],
      "uses": [],
      "name": "Type2",
      "is_abstract": false,
      "implements": []
    }
]
```

En este ejemplo tratamos de comprobar si las propiedades se añaden correctamente. Para ello, compararemos el `json` con la salida devuelta por la herramienta al ejecutarla sobre el ast. El test nos proporcionará información sobre como ha ido la ejecución, en caso de fallo, mostrará el resultado obtenido y el esperado por la consola.

Pruebas B-C

Para simplificar la realización de este test, se ha dividido en dos funciones. La segunda función que se llame dependerá de la estrategia especificada. En este caso, se va a verificar el funcionamiento de la estrategia *full-frequency*:

```
#[rstest]
fn test_coupling_full_frec(
    #[files("tests/full_frequency/input/*.json")]
    input: PathBuf,)
    -> Result<(), Box<dyn std::error::Error>> {
        run_metrics_test(input, "full-frequency")
}
```

Esto facilitará la ejecución de pruebas cuando se implementen más estrategias en un futuro. Por otra parte, *run_metrics_test()* que es la primera función, la cual ejecuta la prueba, será así:

```
fn run_metrics_test(input: PathBuf, command_arg: &str)
-> Result<(), Box<dyn std::error::Error>> {
    let mut cmd = Command::cargo_bin(env!("CARGO_PKG_NAME"))?;

    let dir = input.parent().expect("Invalid input directory");
    cmd.current_dir(dir);
    let file_name = input.file_name().
    expect("Invalid input file name");
    cmd.arg(command_arg).arg(file_name);
    let output = cmd.output().
    expect("Failed to execute command");

    assert!(output.status.success(),
```

```
    "Unexpected execution status");

    let actual_string = String::from_utf8(output.stdout)
    .expect("Invalid output");

    let actual: Vec<MetricData> =
        serde_json::from_str(&actual_string)
        .expect("Invalid output JSON format");

    let mut output_path = input.clone();
    output_path.pop(); // filename
    output_path.pop(); // input
    output_path.push("output");
    output_path.push(Path::new(file_name)
    .with_extension("json"));
    let expected_string = fs::read_to_string(output_path)
    .expect("Invalid expected output");
    let expected: Vec<MetricData> =
        serde_json::from_str(&expected_string)
        .expect("Invalid expected JSON output");
    assert_eq!(actual, expected,
    "Actual and expected output don't match");
    println!("Actual: {actual:?}\nExpected: {expected:?}");

    Ok(())
}
```

A la función `run_metrics_test()` se le, a la cual se le pasan dos argumentos. El primer argumento es una ruta de entrada que contiene los archivos `.json` con las estructuras *TypeData* descritas anteriormente. El segundo argumento es una cadena de texto que especifica el comando de la estrategia a utilizar en la prueba, del cual depende la ejecución de la segunda función.

La función cogerá del directorio correspondiente la estructura intermedia de la cual se calcularán las métricas. Dependiendo de la estrategia seleccionada se ejecutará una función u otra. Tras el cálculo de las métricas, se preparará el resultado obtenido para ser comparado. Finalmente, se comparará el resultado esperado con el obtenido. Un ejemplo de ejecución puede ser el siguiente:

Estructura intermedia:

```
[
    {
        "name": "Type2",
        "namespace": "",
        "properties": [
            "prop1",
            "prop2"
        ],
```

```
        "methods": [],
        "uses": [
            {
                "name": "Type3",
                "namespace": "",
                "type": true,
                "properties": [
                    "a",
                    "b"
                ],
                "methods": [],
                "type_freq": 1,
                "property_freq": 0,
                "method_freq": 0
            },
            {
                "name": "Type1",
                "namespace": "",
                "type": true,
                "properties": [],
                "methods": [],
                "type_freq": 1,
                "property_freq": 0,
                "method_freq": 0
            }
        ],
        "is_abstract": false,
        "implements": []
    },
    {
        "name": "Type1",
        "namespace": "",
        "properties": [],
        "methods": [],
        "uses": [],
        "is_abstract": false,
        "implements": []
    },
    {
        "name": "Type3",
        "namespace": "",
        "properties": [],
        "methods": [],
        "uses": [],
        "is_abstract": false,
        "implements": []
    }
```

```
]
```

Resultado esperado del cálculo de métricas:

```
[
    {
        "type": "Type2",
        "namespace": "",
        "coupling": [
            {
                "type": "Type3",
                "namespace": "",
                "strategy": "full-frequency",
                "value": 1,
                "assessment": "low"
            },
            {
                "type": "Type1",
                "namespace": "",
                "strategy": "full-frequency",
                "value": 1,
                "assessment": "low"
            }
        ]
    },
    {
        "type": "Type1",
        "namespace": "",
        "coupling": []
    },
    {
        "type": "Type3",
        "namespace": "",
        "coupling": []
    }
]
```

En este ejemplo tratamos de comprobar si se realiza correctamente el cálculo de métricas para una estructura con una dependencia baja, aplicando la estrategia *`full frequency`*. Tras ejecutar el cálculo con la estructura, se comparará el resultado obtenido con el esperado.

5.3.3. Pruebas de extremo a extremo

En este TFG no se ha conseguido realizar este tipo de pruebas a tiempo, no obstante, se ha verificado el funcionamiento de la herramienta completa, como bien se ha mostrado en el capítulo 4. Por ello, aunque no se han definido pruebas de extremo a extremo, si que se han realizado este tipo de pruebas manualmente.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Conclusiones

Este TFG ha consistido en la realización de una primera versión de una herramienta de análisis de la calidad de un código Swift mediante el cálculo de métricas de acoplamiento. Con este proyecto esperamos ayudar en la implementación final de una herramienta capaz de medir la calidad de un diseño software aumentando la calidad de los códigos realizados en los procesos de desarrollo software.

La herramienta creada permitirá identificar aquellas áreas en las que se debería poner más atención, mejorando la legibilidad, mantenibilidad y el rendimiento del código. Asimismo, como las métricas de acoplamiento buscarán reducir la dependencia entre los elementos del código, reduciremos, por un lado, la complejidad del código, y por otro, la probabilidad y cantidad de errores. La herramienta proporcionará, también, una mayor eficiencia en el desarrollo de software, debido a la retroalimentación rápida y precisa que obtendrá el programador mientras trabaja. Esto permitirá a los desarrolladores reducir el tiempo dedicado a la depuración y refactorización del código.

Durante el periodo inicial de planificación, se determinó que el tamaño del proyecto era demasiado grande, por lo que se decidió implementar esta primera versión en un equipo de tres personas. Como ya se ha explicado, cada miembro ha documentado una de las 3 partes en la que se dividió el trabajo, el parser, la generación de estructuras intermedias y el cálculo de métricas.

Como ya se ha visto en la sección 4, donde se explica el desarrollo de la herramienta en detalle, la metodología para la implementación ha sido implementar pequeñas funcionalidades a la vez para posteriormente probarlas.

Entre los conocimientos adquiridos durante el desarrollo de este trabajo, podemos destacar:

- La formación en Rust, un lenguaje moderno muy demandado en el ámbito profesional.
- La coordinación y organización entre un equipo de desarrollo.
- La gestión eficiente de repositorios Git: ramas, **pull-requests** , **commits** organizados, resolución de
- conflictos en el repositorios...
- Conocimiento teórico sobre metodologías que fueron explicadas por los profesores en las reuniones semanales y que se desconocían, como los principios SOLID.
- Metodología eficiente para la realización de un proyecto: cuando subir un código, refactorización...
- Cómo realizar pruebas de un código de manera correcta y precisa.
- La depuración de un código Rust mediante el uso de pruebas unitarias.

Tras acabar este TFG, podemos concluir que se ha logrado implementar una primera versión de una herramienta muy potente capaz de mejorar la calidad del código Swift mediante el análisis de métricas de acoplamiento.

6.2. Dificultades afrontadas

Durante este TFG han surgido multitud de problemas que se han tratado de resolver entre todo el equipo.

En primer lugar, mencionar el desconocimiento del lenguaje Rust, lo que conllevó a un periodo de formación exhaustivo que trató de capacitar a los miembros del equipo para la implementación de la herramienta. Esto produjo numerosos errores de sintaxis en un inicio, afectando a la productividad del proyecto. Además, la incapacidad de entender ciertas partes del código en un inicio supuso una barrera al empezar con el proyecto.

Además, la gestión del repositorio Github también supuso un periodo de formación en el que se trató de implementar la metodología TDD, descrita en capítulos anteriores.

Asimismo, destacar los múltiples problemas que se tuvieron con la implementación de la herramienta. Problemas con el analizador léxico, la creación de nodos o el tratamiento de cada tipo de nodo. En concreto este último resultó ser que más tiempo conllevó, debido a la complejidad del tratamiento de los nodos y la información incluida en estos. Además, la implementación de nuevas funcionalidades daban lugar a refactorizaciones del código que, en ocasiones, daban lugar a errores inesperados.

La densidad del proyecto es otro de las dificultades que tuvimos que afrontar, ya que desde un primer momento se sabía que la herramienta era demasiado grande como para terminarla en el periodo que dura el TFG. Por ello, se tuvieron que priorizar unas funcionalidades frente a otras.

Finalmente, la realización de pruebas fue una barrera importante en un inicio, debido al desconocimiento sobre la implementación tanto de pruebas unitarias como de integración.

Ante todos estos problemas, la ayuda de nuestros tutores ha sido esencial, ya que trataban de resolver cualquier tipo de duda que surgiera de manera exhaustiva (explicación de metodologías, dudas del lenguaje, revisiones de código).

6.3. Impacto en ODS

Esta herramienta puede aportar significativamente en la consecución de varios Objetivos de Desarrollo Sostenible (ODS) de la ONU para 2030.

6.3.1. ODS 4: Educación de Calidad

La herramienta podrá ser utilizada en el ámbito de la educación para mostrar a los estudiantes, que estén adquiriendo formación en programación en Swift, la importancia de realizar un código de calidad, mediante la generación de informes detallados.

6.3.2. ODS 8: Trabajo Decente y Crecimiento Económico

La herramienta tratará de ayudar a los desarrolladores a mejorar la calidad del código, aumentando su productividad, la detección de errores en el código y la mantenibilidad de los programas.

6.3.3. ODS 9: Industria, Innovación e Infraestructura

La aplicación mejorará la calidad de los procesos de desarrollo software, mejorando la calidad de las aplicaciones y fomentando la innovación, al facilitar la creación de código de calidad.

6.3.4. ODS 10: Reducción de las Desigualdades

La herramienta puede reducir las desigualdades en las regiones menos favorecidas, proporcionando una herramienta que sirva para la formación de desarrolladores en este tipo de regiones.

6.3.5. ODS 16: Paz, Justicia e Instituciones Sólidas

La herramienta puede ayudar a aumentar la seguridad de las aplicaciones usadas por las instituciones públicas, al proporcionar un código de calidad que facilite la detección de anomalías.

Para finalizar esta sección, decir que creemos firmemente que la herramienta puede contribuir a los ODS de la ONU 2030. Entre las contribuciones se encuentran la educación y el aumento de productividad de programadores, la

mejora de los procesos de desarrollo software fomentando la innovación, la reducción de las desigualdades facilitando el acceso a una herramienta educativa y el aumento de seguridad en aplicaciones empleadas por organismos públicos.

6.4. Trabajo futuro

Una vez terminado este trabajo de fin de grado, se seguirá desarrollando la herramienta hasta terminarla, momento en el que se abrirá el repositorio de manera pública. Durante este desarrollo, se deberá ampliar la herramienta para que pueda leer e identificar cualquier funcionalidad de un código Swift, por ejemplo la herencia, algo que ahora no es capaz de leer. Al hacer lo anterior se deberán cambiar las estructuras intermedias para que recojan la nueva información que pueda ser importante para el cálculo de las métricas, y, realizar el cálculo de las métricas con la nueva información. Además, una vez terminado el desarrollo, se podría adaptar la herramienta para que sea capaz de analizar otros lenguajes, algo que se podría implementar con Rust. Por último, se deberá mantener la herramienta y resolver posibles errores que puedan tener los usuarios al usarla.

Bibliografía

[1] R. Nystrom, *Crafting Interpreters*. Genever Benning, 2021.

[2] C. N. Steve Klabnik, *The Rust Programming Language*. Rust community, 2022.

[3] J. R. Pascual. «Acoplamiento y Cohesión». (2019), dirección: https://www.disrupciontecnologica.com/acoplamiento-y-cohesion/.

[4] X. Mallón. «¿Qué es SonarQube?» (2022), dirección: https://keepcoding.io/blog/que-es-sonarqube/.

[5] T. Yin. «Lizard — a Cyclomatic Complexity Analyzer». (2024), dirección: https://github.com/terryyin/lizard.

[6] J. P. Simard. «SwiftLint: A tool to enforce Swift style and conventions». (2023), dirección: https://github.com/realm/SwiftLint (visitado 17-04-2024).

[7] Cubewise. «Arc for TM1 – Hands-on – Part 2». (2022), dirección: https://downloads.cubewise.com/Arc/Hands-on/HandsOn+-+Arc+for+TM1+-+Part+2.pdf.

[8] M. Campolese. «swift-code-metrics». (2023), dirección: https://github.com/matsoftware/swift-code-metrics?tab=readme-ov-file.

[9] A. D. Team. «swift-metrics». (2024), dirección: https://github.com/apple/swift-metrics.

[10] A. Team. «Flujo de trabajo de Gitflow». (2024), dirección: https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow.

[11] F. Fuentes. «¿Qué es Test-Driven Development (TDD)?» (2023), dirección: https://www.arsys.es/blog/test-driven-development.

[12] S. PITTET. «Los distintos tipos de pruebas de software». (), dirección: https://www.atlassian.com/es/continuous-delivery/software-testing/types-of-software-testing.

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Mon Jun 03 21:18:13 CEST 2024
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)