eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Ingeniería de Software

Herramienta para la Visualización de Arquitecturas de Líneas de Productos Software

Endika Varas

Dirección
Arantza Irastorza
Maider Azanza

2022

Agradecimientos

Muchas gracias a mi familia, en especial a mis padres, por apoyar todas y cada una de las decisiones que me han llevado a donde estoy.

Muchas gracias a mis amigos, que transforman cualquier momento difícil en fácil, que son mi segunda familia, y han estado ahí cada vez que lo he necesitado.

Muchas gracias a mis tutoras, que han tenido que aguantar mil errores y correos. Sin ellas este documento no pasaría del primer capítulo.

Finalmente, quiero darme las gracias a mi mismo, por creer en mí.

Resumen

Este trabajo tiene como objetivo principal facilitar la inserción en un equipo de trabajo a un desarrollador que va a trabajar con una línea de producto software. Para realizar esta tarea, se va a desarrollar una herramienta que, mediante ingeniería inversa, permita generar de manera automática unos diagramas que aporten información sobre la arquitectura.

En esta memoria se describe todo el proceso realizado para desarrollar esa herramienta que, posteriormente, ha sido integrada en *InsideSPL*, otra herramienta que aporta más información sobre una línea de producto software.

El objetivo de esa integración es crear una aplicación que, de manera conjunta, permita al desarrollador tener a su disposición información que le ayude a entender el proyecto con el que va a trabajar.

Índice de contenidos

Índice de figuras

Índice de tablas

CAPÍTULO 1

Introduccion

En el mundo del desarrollo del software, existen diferentes paradigmas para diseñar y generar productos. Uno de estos paradigmas es el de las líneas de producto software (SPL). Este paradigma plantea tener un código común y cierta parte variable para poder generar desde un mismo proyecto diferentes productos con similitudes y diferencias. Para ello, un proyecto puede tener diferentes características y la combinación de estas generará diferentes productos.

El hecho de que estas características afecten al código final añadiendo o quitando código hace muy difícil para un desarrollador recién incorporado a un equipo de desarrollo comprender un proyecto. Cada uno de estos productos cambiará la arquitectura del proyecto añadiendo o quitando componentes, relaciones o funciones dentro del código, así que puede ser un proceso muy largo tener que explorar todas estas opciones a mano.

El objetivo de este trabajo es desarrollar una herramienta que permita al desarrollador comprender la estructura del código y la manera en la que las diferentes características afectan a la arquitectura de cada producto.

Para ello, se seleccionará una herramienta que permita mostrar la arquitectura de un proyecto software y se adaptará para un proyecto *SPL*. Esta herramienta adaptada mostrará al desarrollador la arquitectura del proyecto automáticamente, con toda la información relativa a los productos o características que le interese conocer.

Después, se integrará en *InsideSPL* [1], la herramienta desarrollada por Iosu Salaberri en 2020 junto al grupo de investigación *Onekin*. El objetivo de esta integración será el de conseguir una herramienta que le dé al usuario información sobre la estructura y arquitectura de una línea de producto software.

CAPÍTULO 2

Planificación

Este capítulo está dedicado a la planificación del proyecto, definiendo así el alcance, el análisis de los riesgos, el control de la calidad, la gestión del tiempo, los sistemas de información, la comunicación y los interesados.
Esta planificación tiene como objetivo establecer el reparto del trabajo y generar una planificación que pueda sentar las bases para la correcta realización del proyecto.

2.1. Descripción

Las Líneas de Producto de Software o *SPL (Software Product Lines)* son un paradigma que permite la reutilización sistemática de diferentes partes de código para poder desarrollar un sistema que permita construir fácilmente software funcional, dependiendo de las necesidades de un cliente.

Este paradigma brinda al cliente una gran capacidad de personalización de un software deseado, pero tiene algunas complicaciones a la hora de introducir nuevos desarrolladores a un proyecto *SPL*. La falta de información en los proyectos es uno de estos problemas, ya que los desarrolladores recién llegados a un equipo de desarrollo tienen problemas para la comprensión del código, dada la complejidad derivada de este paradigma.

El proyecto se basa en *InsideSPL*, una herramienta diseñada por Iosu Salaberri, junto al grupo **Onekin**, para ofrecer la posibilidad de mejorar la comprensión de las *SPL*. Con una interfaz gráfica que ayuda al nuevo desarrollador que se incorpora al grupo de trabajo de desarrollo de una *SPL* a entender las características del proyecto. Este trabajo tiene como objetivo ampliar la herramienta agregando una nueva capa de información relacionada con la arquitectura de la *SPL*.

Esto facilitará los esfuerzos de un nuevo desarrollador para poder entender la estructura de una línea de producto software.

2.2. Gestión del Alcance

Para evitar en la medida de lo posible desviaciones en la gestión del alcance, se detallan los objetivos y requisitos del trabajo, así como los paquetes de desarrollo del proyecto.

2.2.1. Objetivos

El objetivo principal es el de ampliar la aplicación *InsideSPL*, añadiendo una nueva capa de información, que permita la visualización de la arquitectura de una *SPL*.

Basándonos en el objetivo principal, se han definido los siguientes objetivos:

- **Búsqueda de una herramienta de ingeniería inversa:** Buscar y seleccionar una herramienta que permita conseguir la arquitectura de los componentes de un proyecto a partir de su código.
- **Búsqueda de caso de prueba:** Buscar y seleccionar un proyecto *open source* con el que poder realizar las pruebas.
- **Adaptación de la herramienta:** Adaptar la herramienta seleccionada anteriormente para añadirle las funcionalidades necesarias para poder hacer ingeniería de una *SPL*.
- **Integración de la herramienta:** Integrar la herramienta adaptada en el proyecto *InsideSPL*, para que esta pueda mostrar la arquitectura de una *SPL*.

2.2.2. Requisitos

A continuación se detallan los requisitos de nuestro proyecto. Estos nos servirán como guía para saber si estamos cumpliendo correctamente con las necesidades del cliente.

1. La herramienta que se busque debe ser *open-source*, para poder realizarle las modificaciones necesarias para su funcionamiento con líneas de producto software.
2. La herramienta seleccionada debe tener soporte para JavaScript, ya que es uno de los lenguajes utilizados en los casos de prueba de los que dispone el grupo **Onekin**.
3. La herramienta deberá mostrar una descripción de los diferentes componentes que conforman un proyecto.
4. La herramienta final debe poder procesar código desarrollado en pure::variants.
5. La aplicación modificada final debe incluir aspectos de variabilidad en la arquitectura descrita.
6. Los casos de prueba deben estar implementados utilizando las pure::variants.

Además de estos requisitos, también se describen algunos requisitos extra deseables.

1. La herramienta soportará otros lenguajes de programación.
2. La herramienta tendrá una documentación que facilite la extensión/adaptación y la curva de aprendizaje.
3. La herramienta tendrá acceso vía Web, lo cual será útil para facilitar la integración con la aplicación *InsideSPL*

2.3. Gestión de las Tareas

A continuación se describen como se gestionaran las tareas, para ello, se muestra el EDT (diagrama de descomposición de tareas), los paquetes del proyecto junto a las tareas y las dependencias entre las mismas.

EDT

Se han descompuesto las tareas en el siguiente diagrama EDT. El objetivo de realizar este diagrama es el de dividir las tareas del proyecto en paquetes para poder facilitar la planificación.
Tras realizar la descomposición, se han podido identificar 7 paquetes que se muestran en la figura 5.7 y serán desglosados más adelante.

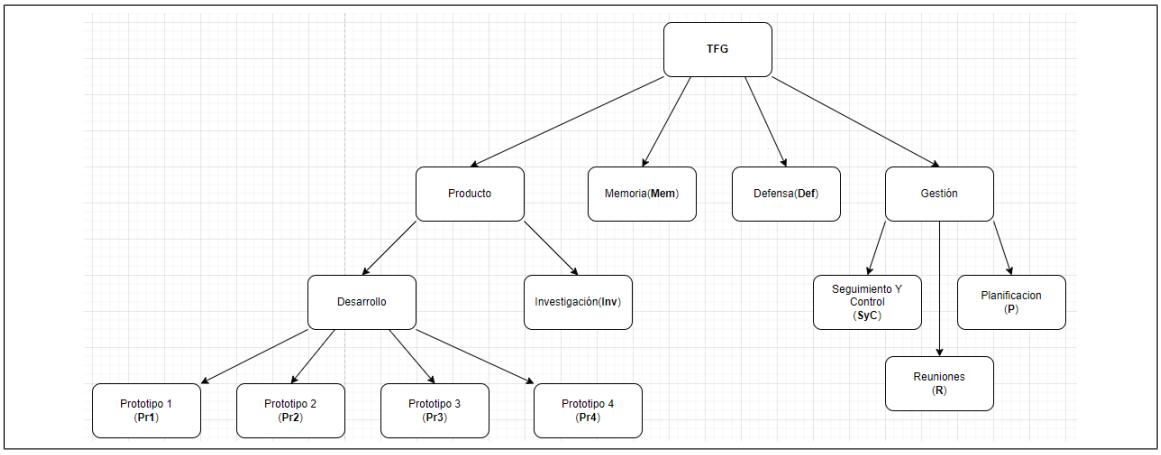

Figura 2.1: Diagrama EDT

Paquetes del Proyecto

A continuación se muestra el desglose de las tareas por paquetes.

- **Planificación (P)**
  - **P.1** Captura de requisitos y toma de decisiones iniciales.
  - **P.2** Planificación inicial orientada a preparación del entorno de desarrollo del proyecto.
  - **P.3** Actualización, si fuera necesaria, de la Planificación.
- **Seguimiento y control (SC)**
  - **SC.1** Recogida de Información relevante sobre el desarrollo del proyecto.
  - **SC.2** Contraste de la información de seguimiento con los planes realizados previamente, identificación de posibles desviaciones y tratamiento de riesgos emergentes.
  - **SC.3** Aseguramiento de las condiciones de calidad.
- **Reuniones (R)**
  - **R.1** Establecer las vías de comunicación.
  - **R.2** Realizar las reuniones periódicamente.
- **Investigación (Inv)**
  - **Inv.1** Revisión de las tecnologías *SPL* y conocimiento de las herramientas o plugins para su realización.

- **Inv.2** Investigación y análisis de la herramienta pure::variants.
- **Inv.3** Búsqueda y análisis de diversas herramientas *open source* que permitan la extracción de la arquitectura de un software a través de su código.
- **Inv.4** Búsqueda de un proyecto *open source* en *Github* con el que realizar las pruebas pertinentes.
- **Inv.5** Comprensión del proyecto *WacLine* de **Onekin** con el que haremos las pruebas finales
- **Inv.6** Comprensión de la aplicación *InsideSPL* para poder integrar la nueva herramienta.

- **Prototipo 1 (Pr1)**
  - **Pr1.1** Creación de una tabla de criterios de selección para la herramienta.
  - **Pr1.2** Selección de la herramienta a adaptar.
  - **Pr1.3** Comprension de la herramienta para su funcionamiento con un proyecto no implementado con pure::variants.
  - **Pr1.4** Pruebas con un proyecto implementado con un sistema tradicional (no *SPL*).
- **Prototipo 2 (Pr2)**
  - **Pr2.1** Adaptación de la herramienta para su uso con *SPL* implementado con pure::variants.
  - **Pr2.1** Pruebas con proyecto *SPL pure::variants* pequeño.
- **Prototipo 3 (Pr3)**
  - **Pr3.1** Mejora, en caso de ser necesaria, de la herramienta para utilizar el proyecto *WacLine* de **Onekin**.
  - **Pr3.2** Pruebas con el proyecto *WacLine* de **Onekin**.
- **Prototipo 4 (Pr4)**
  - **Pr4.1** Integración de la herramienta en *InsideSPL* para su uso con *WacLine*.
  - **Pr4.2** Pruebas con la herramienta en *WacLine*.
- **Memoria (Mem)**
  - **Mem.1** Redacción de la memoria.
  - **Mem.2** Revisión periódica de la memoria.
- **Defensa (Def)**
  - **Def.1** Preparación del póster
  - **Def.2** Presentación para la defensa.

Dependencias entre tareas

Las tareas listadas en el anterior apartado no siguen un desarrollo lineal, esto significa que algunos paquetes necesitan otros para poder ser completados. Estas relaciones son muy importantes y para especificarlas correctamente, se ha diseñado el diagrama que se ilustra en la figura 2.2.

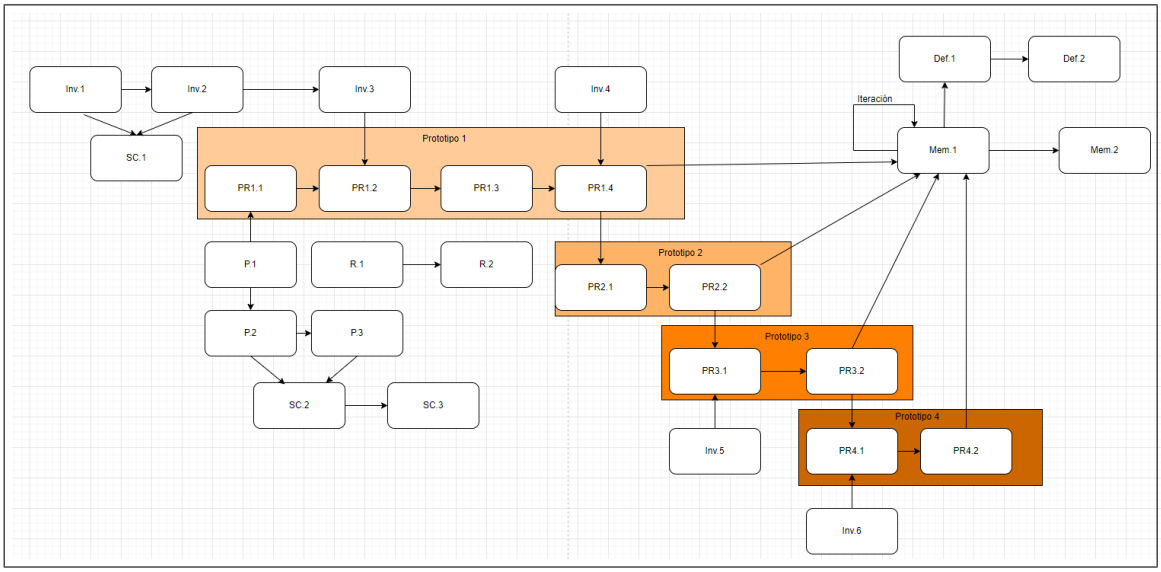

Figura 2.2: Dependencias entre tareas

Como se puede observar, la implementación depende en gran medida de la investigación, ya que es indispensable comprender la herramienta a adaptar antes de comenzar con el código.

Es por esto que es primordial realizar las tareas de investigación a tiempo, para poder evitar retrasos en el apartado de la implementación.

Además, se han resaltado los diferentes prototipos por paquetes, en los que se puede apreciar como cada una de las iteraciones se basa en el trabajo realizado en la implementación anterior.

2.4. Gestión del Tiempo

La gestión del tiempo es uno de los apartados con más peso a la hora de realizar una planificación, se debe generar un plan con tiempos asumibles que permitan gestionar el alcance correctamente, fragmentando el tiempo total disponible entre las tareas del proyecto.

Ciclo de Vida

En este proyecto existe un alto grado de incertidumbre a la hora de estimar la cantidad de trabajo que habrá que dedicar a la selección y adaptación de la herramienta, se trata de una herramienta desconocida y no podemos saber como de fácil será adaptarla al paradigma de *pure::variants*.

Como resultado de estas complicaciones, la investigación tiene un carácter muy importante, es por esto que resulta muy complicado estipular claramente el alcance y determinar de manera fiable la cantidad de trabajo que llevara cada uno de los apartados, por ello, se ha tomado la decisión de llevar a cabo el trabajo con un enfoque **Iterativo** e **Incremental**.

Este enfoque consiste en realizar procesos cíclicos, en los que podremos recibir información de nuestro cliente y presentar un producto de manera periódica, evitando así que el proyecto se exceda del alcance sin tener un producto finalizado.

Concretamente, serán cuatro ciclos, y en cada uno de estos se generara un prototipo que mejorara la implementación del anterior, tal y como se puede ver en la figura 2.2.

A la hora de realizar una estimación de los tiempos, hay un grado de incertidumbre alto, para poder atajar este problema, es necesario tomar algunos riesgos y diseñar un plan de acción para mitigarlos. Para ello se ha generado un plan de gestión de tiempo que ha dado como resultado un diagrama Gantt, tal y como se puede ver en la figura 2.3, que nos sirve como guía en la realización.

Tiempo estimado por tarea

Tras la descomposición de tareas, tal y como se puede ver en la tabla 2.1 se ha realizado un desglose de las horas estimadas por cada uno de los paquetes, y por cada tarea incluida en los mismos.

Paquete de Trabajo	Tarea	Descripción	Horas estimadas
Planificación	P.1	Captura de Requisitos	5
	P.2	Planificación inicial	12
	P.3	Actualización de la planificación	5
	Subtotal		22
Seguimiento y Control	SC.1	Recogida Información	3
	SC.2	Contraste información	2
	SC.3	Aseguramiento de condiciones	5
	Subtotal		10
Reuniones	R.1	Reunión Inicial	1
	R.2	Reuniones periodicas	15
	Subtotal		16
Investigacion	Inv.1	Revisión Tecnologias SPL	6
	Inv.2	Investigación sobre pure::variants	5
	Inv.3	Busqueda de herramientas	10
	Inv.4	Busqueda de Proyecto	5
	Inv.5	Comprensión de WacLine	8
	Inv.6	Comprensión de InsideSPL	8
	Subtotal		42
Prototipo 1	PR1.1	Creación Tabla de Criterios	3
	PR1.2	Seleccion Herramienta	5
	PR1.3	Adaptación Inicial	6
	PR1.4	Pruebas Iniciales	4
	Subtotal		18
Prototipo 2	PR2.1	Adaptación de la herramienta a pure::variants	30
	PR2.2	Pruebas con proyecto pure::variants pequeño	5
	Subtotal		35
Prototipo 3	PR3.1	Mejora de herramienta para SPL mas grande	40
	PR3.2	Pruebas con WacLine	5
	Subtotal		45
Prototipo 4	PR4.1	Integración en InsideSPL	35
	PR4.2	Pruebas finales	5
	Subtotal		40
Memoria	Mem.1	Redacción	70
	Mem.2	Revisión	10
	Subtotal		80
Defensa	Def.1	Póster	3
	Def.2	Presentación	4
	Subtotal		7
TOTAL			315

Tabla 2.1: Tabla de desglose de tareas

Como se puede ver en la tabla 2.1, existe una desviación de 15 horas en el cómputo final, se trata de una desviación normal con respecto a las 300 horas estipuladas para un TFG con carácter general.

Esta desviación se debe principalmente a una asignación de horas muy alta al tiempo de desarrollo, que supone un 55 % del tiempo final.

Diagrama de Gantt

Tal y como se ve en la figura 2.3, se ha realizado un diagrama Gantt para poder tener una idea general de la duración de cada tarea, con sus fechas de inicio y finalización. Además, también se identifican los hitos del proyecto, que son reconocibles por su forma de rombo.

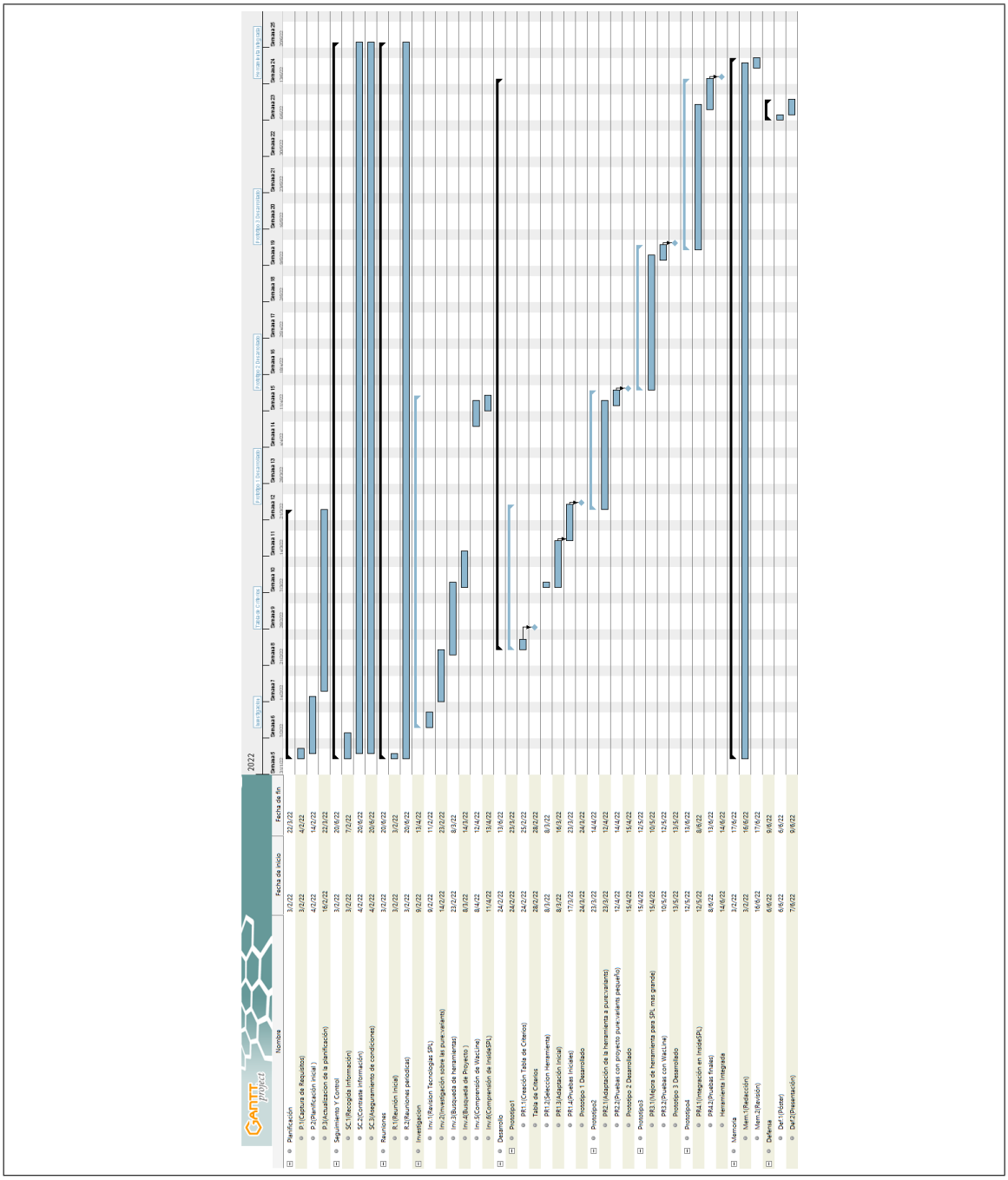

Figura 2.3: Diagrama de Gantt

2.4.1. Hitos

1. **Reunión inicial**. Esta reunión da inicio al desarrollo del trabajo, en esta reunión serán tratados los objetivos y requisitos del proyecto, además del alcance del mismo.

2. **Tabla de criterios**. Se llegará a este hito cuando estén definidos los requisitos y se haya creado una tabla para poder decidir cuál será la herramienta seleccionada.

3. **Herramienta seleccionada**. Este hito da comienzo a la parte de implementación del proyecto, teniendo ya seleccionada la herramienta a adaptar.

4. **Prototipo 1 desarrollado**. Este hito dará fin a la primera iteración del ciclo de vida del proyecto, teniendo ya el primero de los entregables preparado y listo para su demostración, este entregable consistirá en una herramienta capaz de extraer la arquitectura de un proyecto desarrollado de la forma tradicional (sin *SPL*).

5. **Prototipo 2 desarrollado**. Este hito dará fin a la segunda iteración del ciclo de vida del proyecto, con la adaptación de la herramienta para que pueda utilizar un proyecto implementado en pure::variants.

6. **Prototipo 3 desarrollado**. Con este hito finaliza la tercera iteración, teniendo ya creado un producto capaz de extraer la arquitectura del proyecto WacLine.

7. **Herramienta integrada**.Este hito dejará como resultado la herramienta integrada en *InsideSPL*, este es el objetivo final y, por lo tanto, uno de los hitos más importantes en el desarrollo.

8. **Pruebas realizadas**. Este hito se dará tras comprobar que todas las pruebas realizadas son satisfactorias, tanto con el proyecto seleccionado como con *WacLine*, el proyecto *SPL* ofrecido por **Onekin**.

9. **Reunión de cierre de proyecto**. Esta será la última de las reuniones que se realizaran durante el desarrollo. En esta reunión se asegurará que los requisitos, objetivos y estándares de calidad han sido cumplidos en el rango de tiempo establecido.

2.5. Análisis de Riesgos y Calidad

En este apartado se listarán los riesgos, ya que es primordial conocerlos para poder prevenir futuras derivaciones en los tiempos y evitar contratiempos que pongan en riesgo el futuro del proyecto.
También se hará una valoración de los estándares de Calidad que se consideraran necesarios para poder satisfacer las necesidades de los interesados.

2.5.1. Riesgos

- **Trabajo académico.** Existe la posibilidad de que el curso académico y las responsabilidades que conlleva supongan una carga de trabajo lo suficientemente importante como para afectar al desarrollo del proyecto. Para reducir este riesgo, se ha realizado un plan de estudios que permite compaginar ambas tareas con un margen de horas razonable para amortiguar las posibles desviaciones. Este plan deberá ser especialmente flexible en las semanas de horario agrupado, ya que serán esas semanas las que más carga de trabajo acumulen en el curso.

  Como se puede ver en la figura 2.3,las tareas desarrolladas durante las semanas de horario agrupado son las tareas **INV.3** y **PR2.1**. Estas tareas tienen una duración más prolongada que otras tareas con la misma cantidad de horas asignadas, ya que es posible que esas semanas supongan una carga de trabajo académico adicional.

- **Uso de tecnologías nuevas**. Algunas de las tecnologías que van a ser utilizadas en este proyecto son nuevas para el alumno y pueden suponer un problema para el correcto desarrollo del trabajo planificado, poniendo especial énfasis en los proyectos pure::variants.

Para evitar este riesgo, una de las tareas de investigación consistirá en estudiar previamente esta tecnología.

- **Captura de requisitos.** Es posible que la captura de requisitos no haya cubierto todas las necesidades del proyecto, cambiando el alcance del proyecto y generando la posibilidad de que existan futuras desviaciones para cumplir con nuevos requisitos en el futuro, para amortiguar este efecto, se realizara un estudio de los requisitos junto con el cliente antes de comenzar el paquete de desarrollo.

- **Carácter investigativo.** Este proyecto tiene un fuerte carácter de investigación y resulta muy complicado establecer un alcance que cumpla con las proporciones de un trabajo con este calibre. Para mitigar este riesgo, se toma un enfoque iterativo e incremental, tal y como se explica en el apartado 2.4

- **Pérdida de información.** Teniendo en cuenta que los activos de este proyecto tienen un carácter digital, hay que valorar la posibilidad de que exista una pérdida de información. Para minimizar el riesgo, se tendrán varias copias de seguridad en la nube, estas copias se generarán tras realizar cambios en el proyecto utilizando un gestor de repositorios.

- **Problemas con la compatibilidad.** Las diferentes herramientas que van a ser utilizadas en el desarrollo del trabajo pueden tener problemas de compatibilidad que lleven al proyecto a un estado de desfase respecto a la planificación. Para atajar este problema, se hará un estudio de estas compatibilidades antes de integrar nuevas herramientas al proceso.

- **Planificación incorrecta.** Puede que la planificación no abarque la totalidad del alcance en su primera versión, es por eso que se dedica una tarea a modificar la planificación tras una primera revisión.

2.5.2. Gestión de la Calidad

El objetivo final será el de mejorar la integración de nuevos desarrolladores en el paradigma de las líneas de producto software, por lo que se establecerán las siguientes acciones para asegurar la satisfacción de los interesados. Se establecen los siguientes criterios de calidad

- **Pruebas sobre el código implementado.** Se realizarán las pruebas sobre el código implementado de manera interna para poder confirmar el correcto desarrollo de la implementación.

- **Prototipos.** Todos los prototipos deberán ser mostrados a los principales interesados para que evalúen las funcionalidades, estas evaluaciones son esenciales para poder adaptar la herramienta a las necesidades del proyecto.

2.6. Gestión de Comunicaciones e Información

En este apartado hablaremos de los sistemas de información y de cómo estarán gestionados. Es importante tener unos sistemas fiables, que permitan mantener la seguridad y trazabilidad de los activos relacionados con el proyecto.

Además, se describirá cómo será el sistema de comunicación que mantendrá el alumno con los interesados del proyecto, ya que la comunicación será crucial para asegurar que el desarrollo se mantenga dentro del alcance definido.

2.6.1. Sistema de Información

El sistema de información está dividido de la siguiente forma:

- **Recursos:** En esta carpeta se almacenarán los recursos principales, tales como imágenes, diagramas o bibliografías.
- **Proyecto:** En esta carpeta se almacenarán los principales activos y la lógica de negocio del trabajo a realizar. Esta carpeta a su vez se fragmentará en las siguientes carpetas:
  - **Pruebas:** En esta carpeta se almacenarán las pruebas que se vayan realizando a lo largo del desarrollo.
  - **Código:** En esta carpeta se guardará todo el código que ha sido utilizado para adaptar e integrar la herramienta.
  - **Repositorios:** Aquí se almacenarán los proyectos usados para realizar las pruebas pertinentes. Además de esta carpeta, todo el código se guardará en un repositorio de *Github*.

El documento de la memoria se realizará a través de la plataforma *Overleaf*, que tiene un control de versiones integrado y permite compartir el documento a los interesados.
Todos los documentos se almacenarán tanto en los equipos físicos del alumno como en *Google Drive*, que nos brinda la posibilidad de acceder a ellos de manera muy sencilla desde cualquier dispositivo.

2.6.2. Sistema de Comunicación

Para poder comunicarse de manera correcta, se han establecido las siguientes vías de comunicación.

- **Correo electrónico:** Esta será la principal herramienta de comunicación para el día a día. Este medio permite compartir fácilmente archivos, llevar la comunicación con varias personas al mismo tiempo y tener un registro accesible de todos los mensajes.
- **Teléfono:** Este medio se usará en casos urgentes y para las comunicaciones a última hora. Permite avisar rápidamente en caso de contratiempos y adaptar fácilmente el horario de las reuniones.
- **Reuniones telemáticas:** Estas reuniones servirán de guía y control del proyecto, en ellas se recogerá en acta las decisiones tomadas y se podrán realizar demostraciones del proceso realizado.

2.7. Interesados

El interesado principal de que este proyecto concluya satisfactoriamente es **su autor**, ya que es quien defenderá el trabajo y recibirá la calificación. Además, será su responsabilidad terminar el trabajo en fecha y forma.

El grupo de investigación **Onekin**, ya que el objetivo final del trabajo, la herramienta *InsideSPL*, se alinea con la línea de investigación del grupo.

Las tutoras del trabajo, Arantza Irastorza y Maider Azanza, son también interesadas, ya que dirigirán y coordinarán la realización del mismo, y guiaran al alumno para garantizar el éxito del proyecto.

Finalmente, el último de los interesados será el tribunal, ya que este tendrá la responsabilidad de evaluar el trabajo realizado en función de los estándares de la facultad.

CAPÍTULO 3

Antecedentes

Para la correcta comprensión de este proyecto, es necesario explicar algunos conceptos que se nombran a lo largo del documento. Estos conceptos son las líneas de productos software, el proyecto *WacLine* [2] desarrollado por *Onekin*, e *InsideSPL* [1], la herramienta desarrollada por Iosu Salaberri como trabajo de fin de grado. Este capítulo está dedicado a presentar y explicar estos conceptos.

3.1. Líneas de Producto Software

A la hora de producir software, existe la costumbre de reutilizar codigo de otros proyectos para desarrollar un producto similar. Por ejemplo, una empresa que produce software para tiendas online, tendrán una parte común para mostrar y vender productos, pero cada tienda tendrá diferentes funciones. Por ejemplo, una tienda puede tener la opción de pagar con *PayPal* [3], mientras que otra tienda puede tener un sistema de usuarios. Todo esto dependiendo de las necesidades del cliente.

Ante este paradigma de copiar y reutilizar código, surgieron las *Software Product Line (SPL)* o líneas de producto software, una tecnología para producir código en función de las necesidades concretas de cada cliente [4].

Este paradigma de desarrollo de software se basa en productos y características, las líneas de producto software comparten un "nucleo" o base común. Este "nucleo" es de lo que se hablaba cuando se ejemplificaba la empresa que reutilizaba parte de su codigo para cada producto que implementaba.

Una vez se tiene ese "nucleo", entran en juego las características, que son las opciones que tiene el cliente para modificar el producto final a su gusto. Un ejemplo podría ser, en el caso de la empresa de software para tiendas, saber que tipo de sistema de pago se va a usar, tarjeta o *PayPal*. Otras características podrían ser la potencia, la cantidad de ruido máximo que puede generar, en que idioma estará la interfaz para regularlo y un largo etcétera.

Estas características pueden ser obligatorias, opcionales o alternativas, es decir, excluyentes entre ellas. Además, algunas características dependen de otras para poder ser seleccionables. Todo

este conjunto de características genera un diagrama de características como el que se puede ver en la figura 3.6.

Cuando un cliente selecciona las características necesarias para generar una implementación que cumpla con sus intereses, genera un producto. Estos productos son el objetivo final de las *SPL*, ya que se consigue de manera efectiva reutilizar la mayor parte del código cambiando únicamente las implementaciones necesarias para cada característica.

Pure::variants

Aunque existen varias herramientas para el desarrollo de las líneas de producto software, en este trabajo se utiliza *Pure::variants* [5]. *Pure::variants* es una herramienta de desarrollo de *SPL* basada en anotaciones de código. Estas anotaciones indican al proceso de generación del producto que partes del código deben añadir o retirar dependiendo de las características seleccionadas. Además, cuando se genera el producto final, todas estas anotaciones desaparecen automáticamente.

Las líneas de producto software implementadas en *Pure::variants* tienen los siguientes elementos principales:

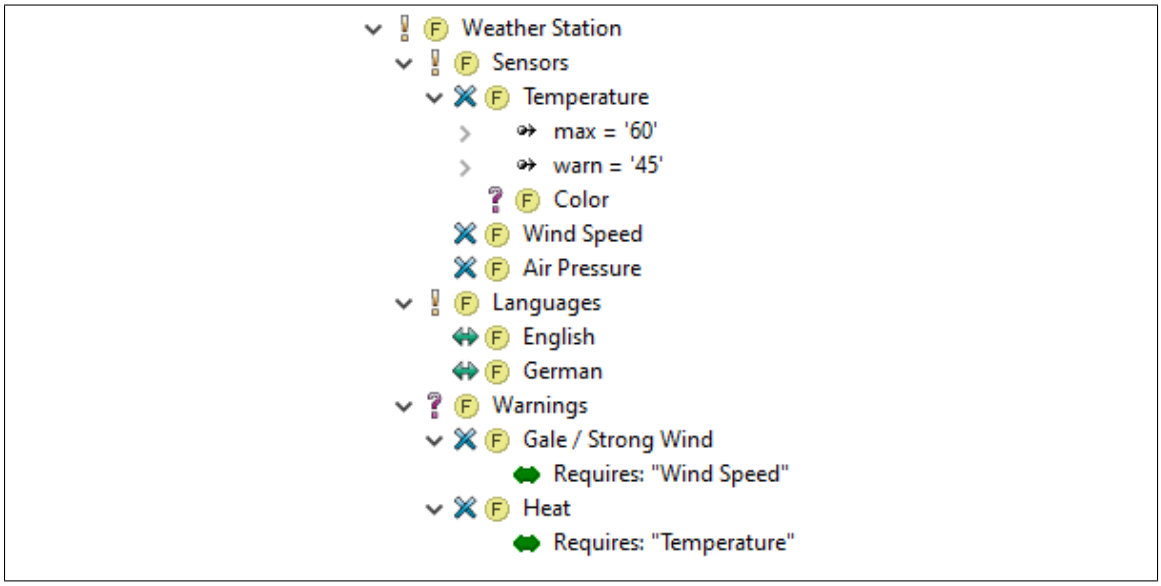

Figura 3.1: Ejemplo de un FeatureModel en *Eclipse*

- **Feature Model**: En estos ficheros se almacenan los modelos de características de un proyecto, con sus relaciones, dependencias y variabilidad.

  En la figura 3.1 se puede ver la representación de un *FeatureModel* en *Eclipse*. En este *FeatureModel* se puede ver como algunas características son dependientes de otras, por ejemplo, la característica "Heat" necesita que la característica "Temperature" esté previamente seleccionada. También hay algunas características alternativas, como las referentes a las lenguas. El símbolo de exclamación representa que es una característica obligatoria, por lo que en el caso de la lengua, una de las dos opciones tiene que estar seleccionada para que pueda generarse el producto final.

- **Family Model**: En este fichero se almacena la información de las distintas clases que se

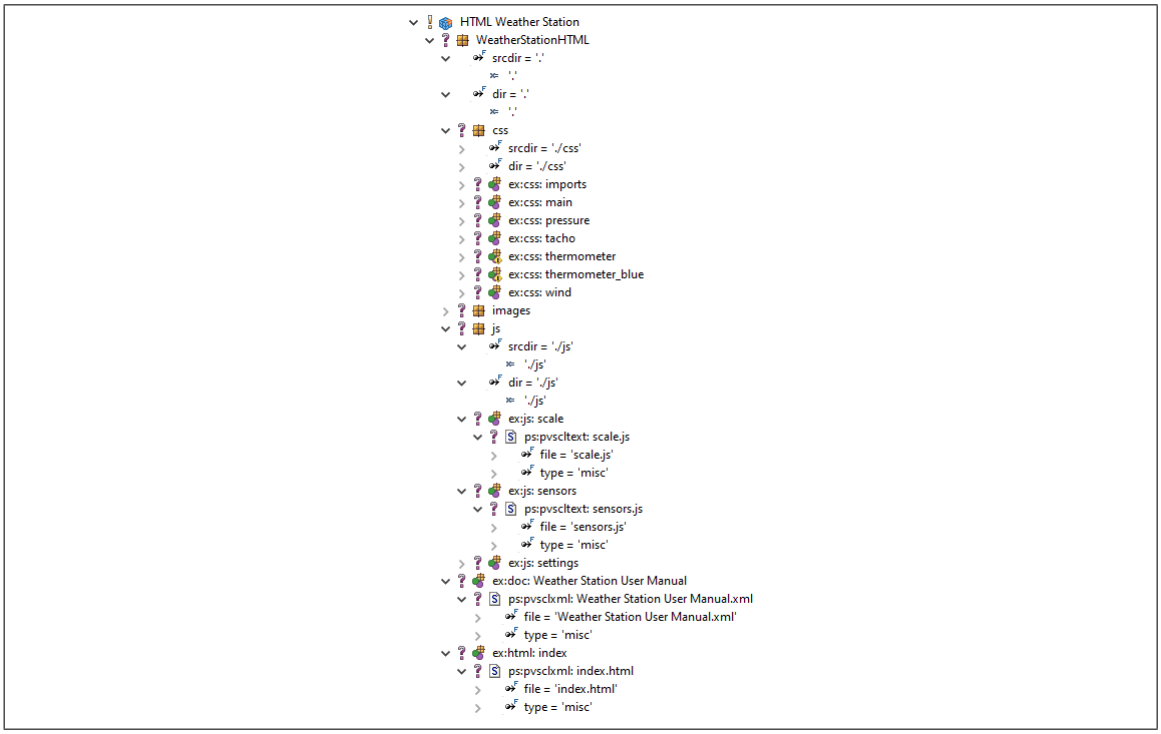

Figura 3.2: Ejemplo de un *FamilyModel* en *Eclipse*

ven afectadas por las características.
En la imagen 3.2 se pueden ver los ficheros que pueden ser modificados en función de las características seleccionadas. Los símbolos de interrogación representan que estos ficheros son opcionales, por lo que podría construirse un producto sin usarse algunos de estos ficheros.

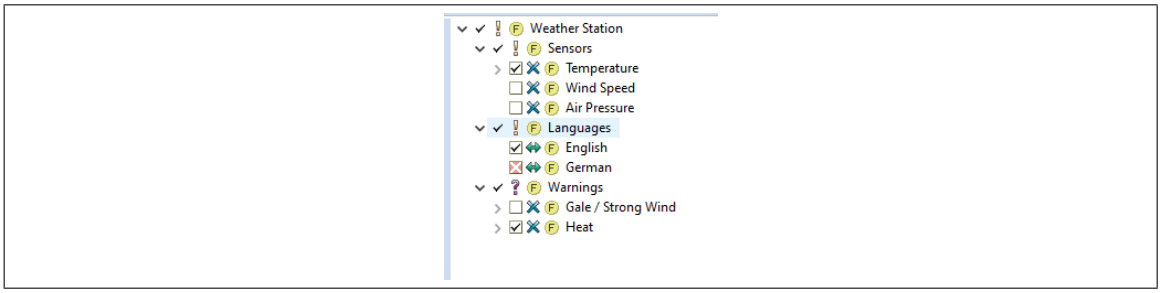

Figura 3.3: Ejemplo de las características seleccionadas en un *VariantModel*, visto en *Eclipse*

- **Variant Model**: En estos ficheros se almacena toda la información, tanto de características como de ficheros utilizados, de un producto concreto.
  En la figura 3.3 se pueden ver las diferentes características seleccionadas por un producto. En este ejemplo, la característica "German" pasa a estar bloqueada, ya que tiene carácter alternativo, por lo que no puede estar seleccionada si la característica "English" lo está. También se puede ver como la característica "Heat" está seleccionada, esta característica depende de la característica "Temperature" para poder ser seleccionada, tal y como se puede ver en la figura 3.1.

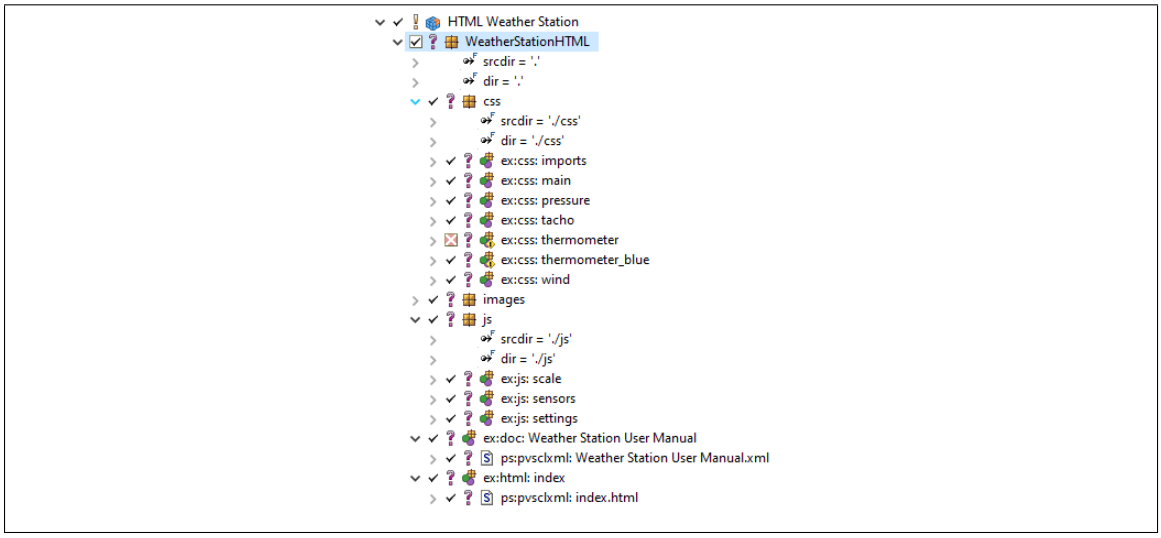

Figura 3.4: Ejemplo de los ficheros utilizados en un *VariantModel*, visto en *Eclipse*

En la figura 3.4 aparece el desglose de los ficheros que utiliza un producto concreto, se puede ver como hay ficheros que están seleccionados mientras otros aparecen con una cruz roja, simbolizando que estos ficheros no son utilizados por este producto.

Estos ficheros (*FeatureModel*, *FamilyModel* y *VariantModel*) son la principal diferencia entre una línea de productos software implementada en *Pure::variants* y una línea de productos software convencional. Estos proporcionan información sobre la organización de una línea de producto, software, sus características y sus productos. Gracias a estos ficheros, un desarrollador puede acceder mediante una interfaz gráfica a todas las opciones que proporciona un proyecto *SPL*.

3.2. WacLine

WacLine es un proyecto desarrollado por *Onekin*, un equipo de investigación de la facultad de ingeniería informática de Donosti. Este proyecto genera una extensión para navegadores *Chromium*(Google Chrome, Opera...) que trabaja con anotaciones web, permitiendo a usuarios dejar anotaciones en diferentes dominios. Estas anotaciones están en forma de texto y pueden ser accedidas por otros usuarios.

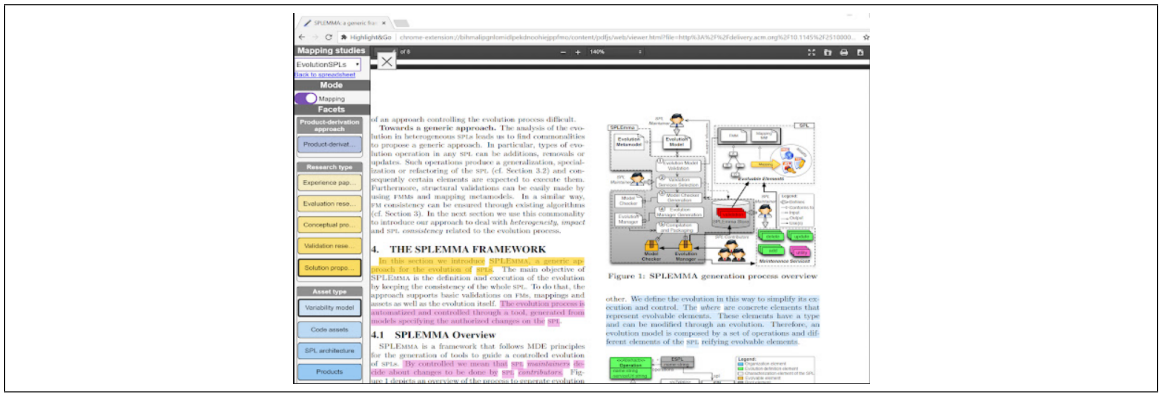

Figura 3.5: Ejemplo de ejecución de HighlightGo, producto de WacLine

En la figura 3.5 se puede ver un ejemplo de ejecución de *Highlight&Go* [6], un producto desarrollado

con WacLine. Los usuarios dejan anotaciones en el texto que aparece subrayado. De esta manera, un usuario con la extensión puede acceder a los comentarios que deje otro usuario para ver sus opiniones. Por ejemplo, esta extensión puede ser usada para hacer revisiones de documentos académicos por parte de profesores o tutores, de manera que el alumno pueda acceder a los comentarios desde su navegador.

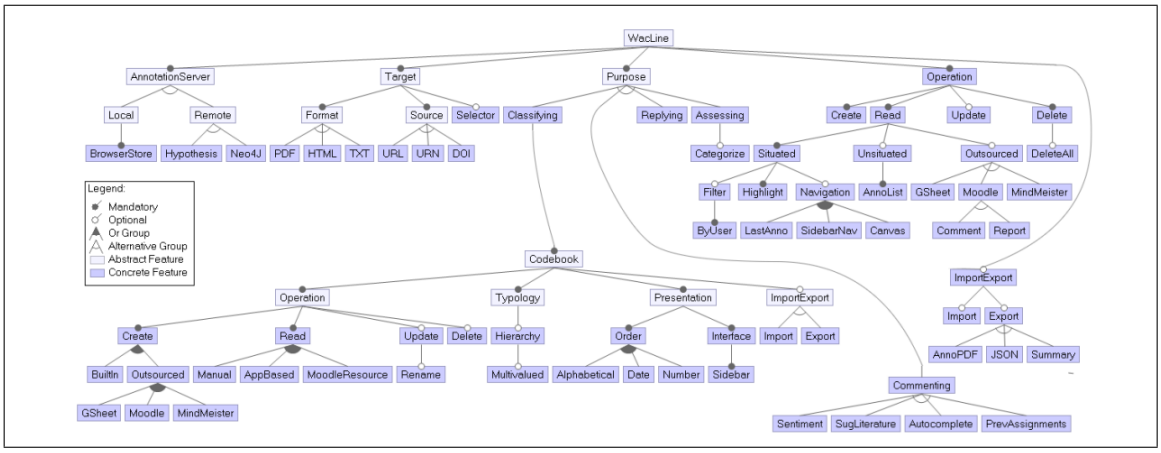

Figura 3.6: Diagrama de características de *WacLine*

Como se puede ver en la figura 3.6, WacLine cuenta con una gran variedad de características para personalizar el tipo de extensión que se generará como producto final. También se puede ver como algunas de estas características son obligatorias, opcionales o alternativas. Por ejemplo, la característica *Source* es obligatoria, pero abstracta, por lo que habrá que escoger por lo menos una de sus características-hijo. Estas, al ser alternativas, pueden ser escogidas a la vez.

3.3. InsideSPL

InsideSPL es una herramienta desarrollada por Iosu Salaberri como trabajo de fin de grado. Esta herramienta es capaz de mostrar a un usuario la información de una línea de producto desarrollada en *pure::variants*. Esta herramienta puede mostrarle al usuario las características, con sus relaciones y variabilidad. También puede mostrarle los diferentes productos, con la cantidad de características y código que utilizan. El objetivo de esta herramienta es facilitar el proceso de acogida a nuevos desarrolladores que se incorporan a un grupo que desarrolla una línea de producto software.

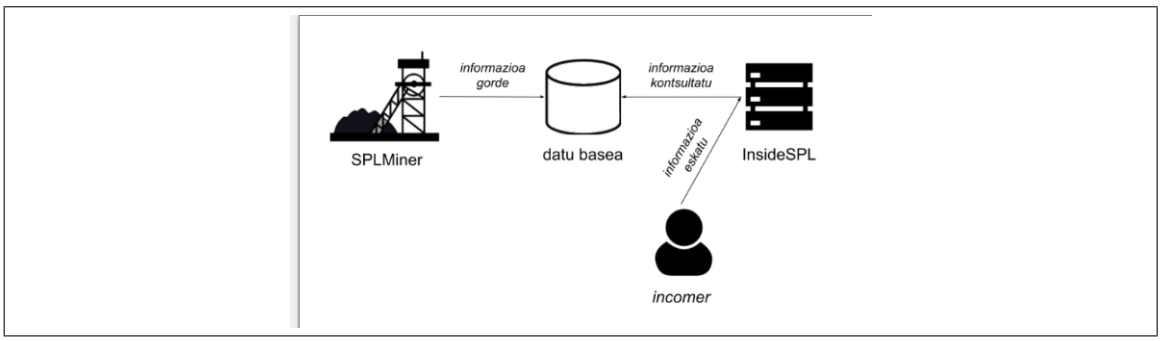

Figura 3.7: Diseño de la arquitectura realizado por Iosu Salaberri [1] (Pag 33)

Tal y como se ve en la figura 3.7, la herramienta se compone de dos grandes módulos:

- **SPLMiner**: Se encarga de procesar la *SPL* para obtener características y productos. Después almacena esos datos en un fichero *.SQL*.
- **InsideSPL**: Se encarga de consultar y mostrar al usuario los datos recopilados por el *SPLMiner*, filtrando y gestionando la información según las necesidades del usuario.

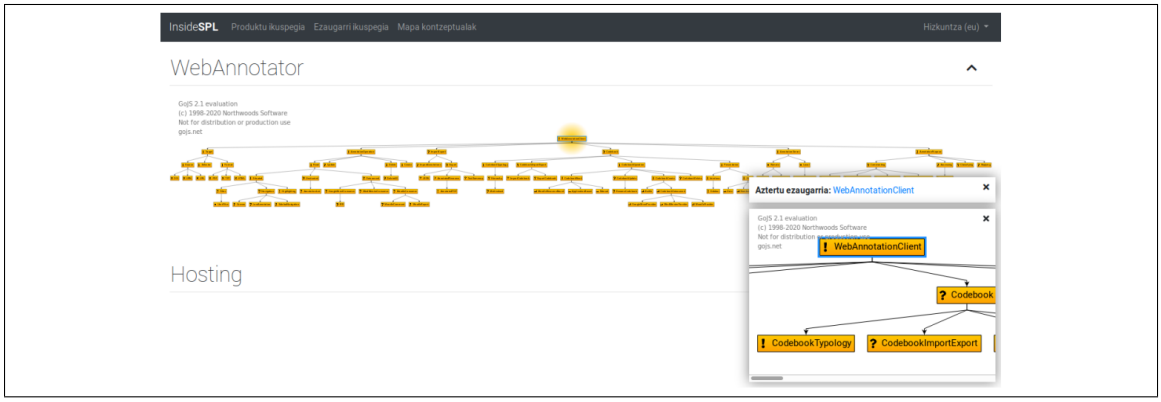

Figura 3.8: Ejemplo de visualización del diagrama de características de *WacLine* en *InsideSPL*

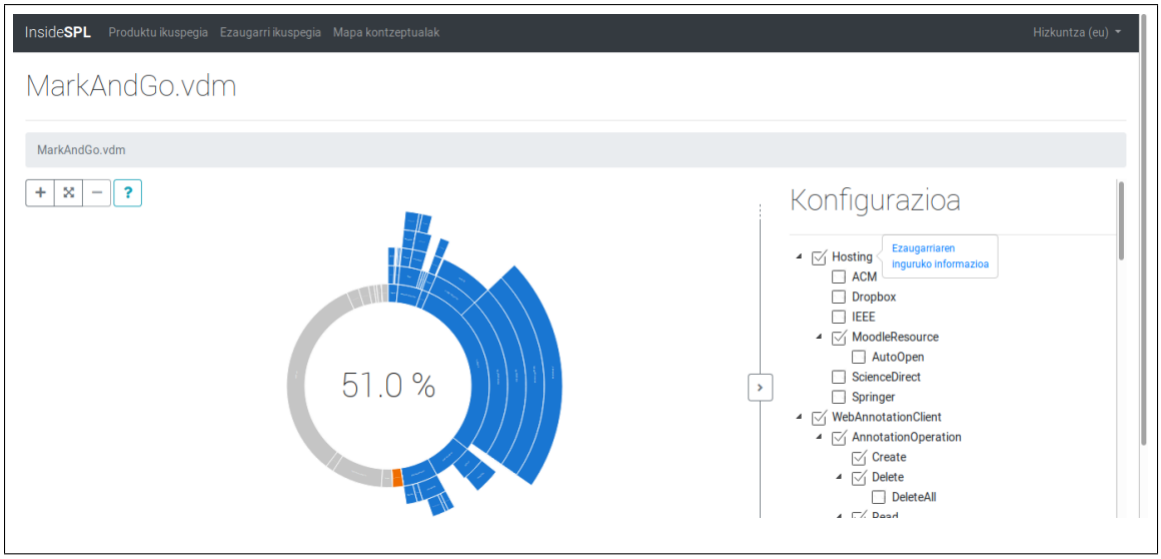

Figura 3.9: Ejemplo de visualización de un producto de *WacLine InsideSPL*

Como se puede ver en las figuras 3.9 y 3.10, *InsideSPL* permite obtener información sobre los productos y las características de una línea de productos software. También permite comparar productos para ver sus diferencias o su cantidad de características en común. Además, también ofrece información detallada sobre una característica en concreto, como sus relaciones o los ficheros que se ven afectados por la misma (ver figura 3.10).

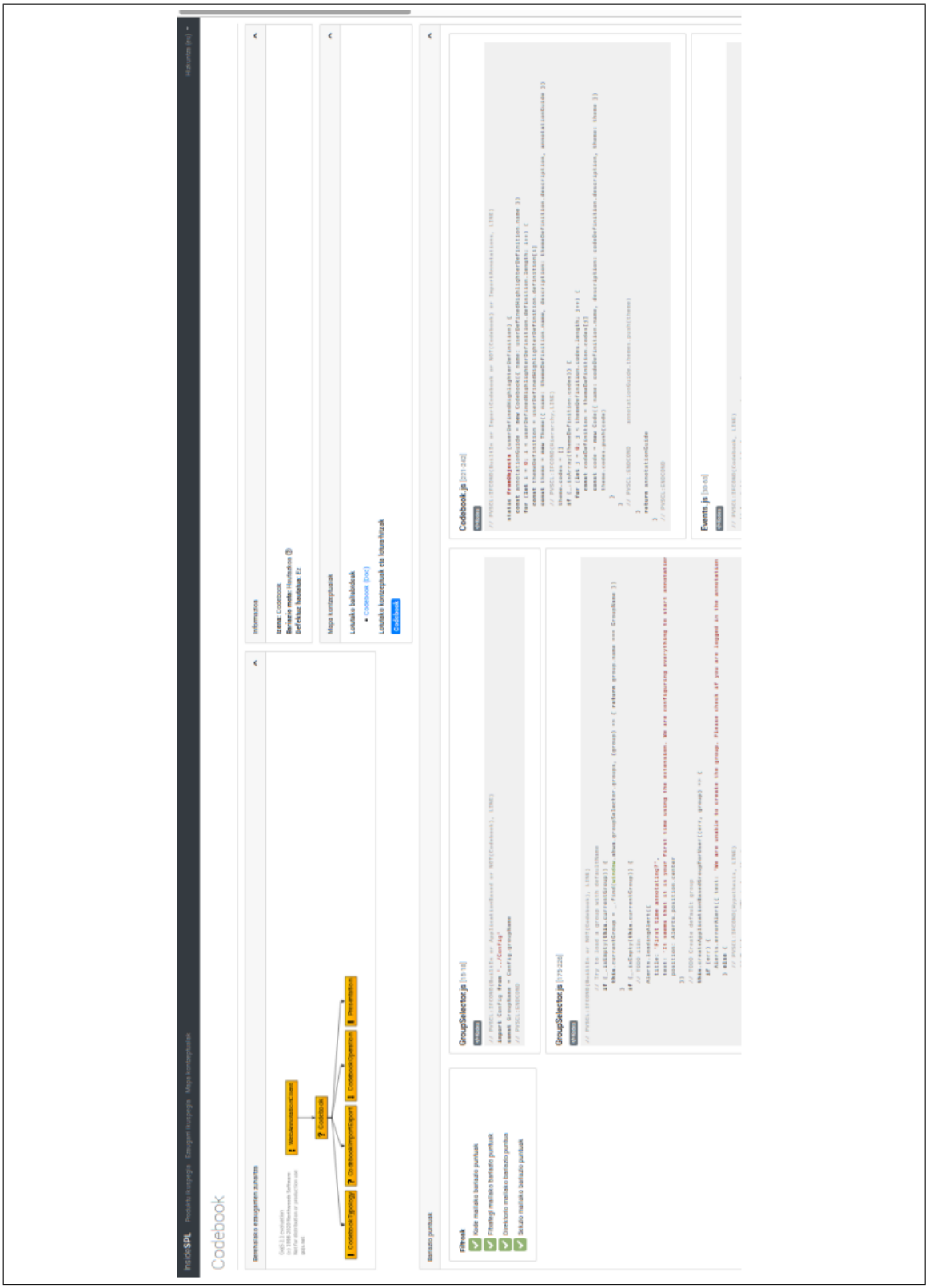

Figura 3.10: Ejemplo de visualización de una característica de *WacLine* en *InsideSPL*

CAPÍTULO 4

Estudio de herramientas para análisis de arquitectura

En este capítulo se documenta la metodología seguida para la selección de la herramienta, además de su posterior análisis y comprensión.

El objetivo de este apartado es encontrar y comprender una herramienta *OpenSource* que permita generar un diagrama de clases/dependencias para comprender la estructura del proyecto.

4.1. Parámetros del estudio

A la hora de realizar la búsqueda de la herramienta, se han tenido en cuenta los requisitos necesarios y deseables que han sido localizados en la planificación. Estos requisitos son los siguientes:

Requisitos necesarios

- La herramienta es capaz de procesar *JavaScript*.
- La herramienta es *OpenSource*.
- La herramienta deberá mostrar una descripción de los diferentes componentes que conforman un proyecto, bien de manera gráfica o vía *ADL(Architecture Description Language)* [7].

Requisitos deseables

- La herramienta soportará otros lenguajes de programación.
- La herramienta tendrá una documentación que facilite su extensión/adaptación y la curva de aprendizaje.
- La herramienta tiene guías de uso en vídeo para facilitar la comprensión de su ejecución.
- La herramienta tendrá acceso vía Web, lo cual será útil para facilitar la integración con la aplicación *InsideSPL* [1].

A la hora de escoger la manera de representar la arquitectura, se presentan dos opciones. La primera, la forma gráfica, consiste en generar información sobre la arquitectura mediante modelos, como podrían ser los diagramas de clase. Esto permite tener un grado de detalle más alto de cada componente específico del proyecto, dando, por ejemplo, información sobre las funciones, herencias o los patrones que sigue el proyecto. La segunda es mediante *ADL(Architecture Description Language)* [7], un sistema de documentación de la arquitectura. Se basa en diagramas que contemplen la cardinalidad y dependencia entre los componentes de un proyecto.

ADL describe la arquitectura a un nivel mayor que ningún otro sistema, pero también tiene algunas desventajas. Como el objetivo final de este trabajo es utilizar la herramienta que se seleccione para describir la arquitectura de una línea de productos, es importante saber qué clases o funciones son utilizadas por cada producto, o que características son usadas por estas clases y funciones. Como la cardinalidad entre componentes puede depender del producto o características que utilice la línea de producto software, *ADL* puede ser más difícil de implementar que una versión gráfica que muestre qué clases se utilizan en cada producto.

4.2. Herramientas estudiadas

Utilizando los requisitos descritos anteriormente, se han recopilado los criterios que servirán para crear una tabla de comparación de las herramientas. Estos criterios son los siguientes:

- **Herramienta:** Nombre de la herramienta.
- **Procesamiento de JavaScript:** Indica si es capaz de procesar *JavaScript*.
- **Descripción de clases vía ADL:** Indica si la descripción de las clases se hace vía *ADL*.
- **Descripción de clases gráfica:** Indica si la descripción de las clases se hace de manera gráfica.
- **Tipos de diagrama:** Indica de que tipos son los diagramas generados (dependencias, clases...).
- **Formato de la información:** Indica en que formato se genera el diagrama (UML, PlantUML...).
- **Posibilidad de cambiar el aspecto del diagrama:** Indica si es posible cambiar el aspecto del diagrama a través de los parámetros de ejecución.
- **Soporte para otros lenguajes:** Indica si puede procesar otros lenguajes además de *JavaScript*.
- **Acceso vía web:** Indica si la herramienta tiene soporte web.
- **Documentación:** Indica el grado de información que aporta la documentación.
- **Tutoriales en vídeo:** Indica si la herramienta cuenta con vídeos explicativos como parte adicional de la documentación.
- **Tiempo de adaptación a JavaScript:** En caso de no tener soporte para JavaScript, estimación del tiempo que supondría adaptarlo al mismo.

- **Razón descarte:** Esta columna se aplica solamente a las herramientas descartadas

A continuación se listan las herramientas consideradas para la adaptación. Algunas de estas herramientas no cumplen con los requisitos necesarios, esto se debe a que existe la posibilidad de adaptar una herramienta para que cumpla con los requisitos. Las capturas de ejecución están realizadas sobre diferentes proyectos, ya que no todas las herramientas podían procesar un proyecto en *JavaScript*. Tras la selección de estas cuatro herramientas potenciales, se procedió a rellenar la tabla de criterios (ver tabla 4.1).

Es importante tener en cuenta que muchas herramientas se han quedado fuera del proceso, bien porque no eran *openSource*, porque estaban programadas en lenguajes desconocidos para el autor u otros problemas listados en la tabla de criterios.

Estas últimas podrían haber entrado en el proceso de selección de no ser porque se encontraron alternativas mejores. La herramienta seleccionada ha sido la tercera, *Arkit* [8], de la que se habla en más profundidad en el capítulo 5.

4.2.1. Class Visualizer

Class Visualizer [9] es una herramienta que genera mediante ingeniería inversa un diagrama de clases UML con información adicional respecto a la arquitectura.

Al cargar un proyecto, genera también una lista de todos las clases, incluyendo las clases de *Java*, que están siendo usadas por el proyecto, haciendo que sea más fácil navegar por el proyecto.

Lo interesante de esta herramienta es que, como se puede ver en la figura 4.1, los diagramas que se crean son interactivos y permiten obtener mucha información sobre aspectos muy concretos de manera sencilla e intuitiva.

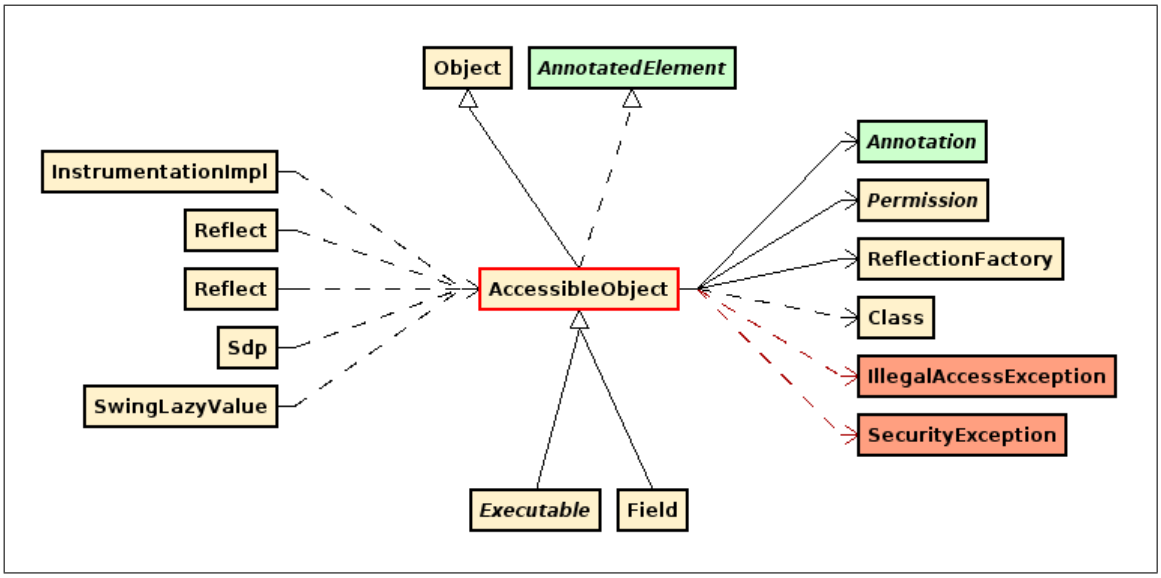

Figura 4.1: Diagrama de clases generado mediante la aplicación

Por ejemplo, al hacer clic sobre una clase, conseguimos diferente información. Como se puede observar en la imagen 4.2, esta información se divide en dos partes:

- **Contenido:** Todo lo que está propiamente incluido en la clase, como puede ser los importes, constructoras, métodos o la clase que implementan.

- **Relaciones:** En este apartado se puede ver con que otras clases interactúa la seleccionada.

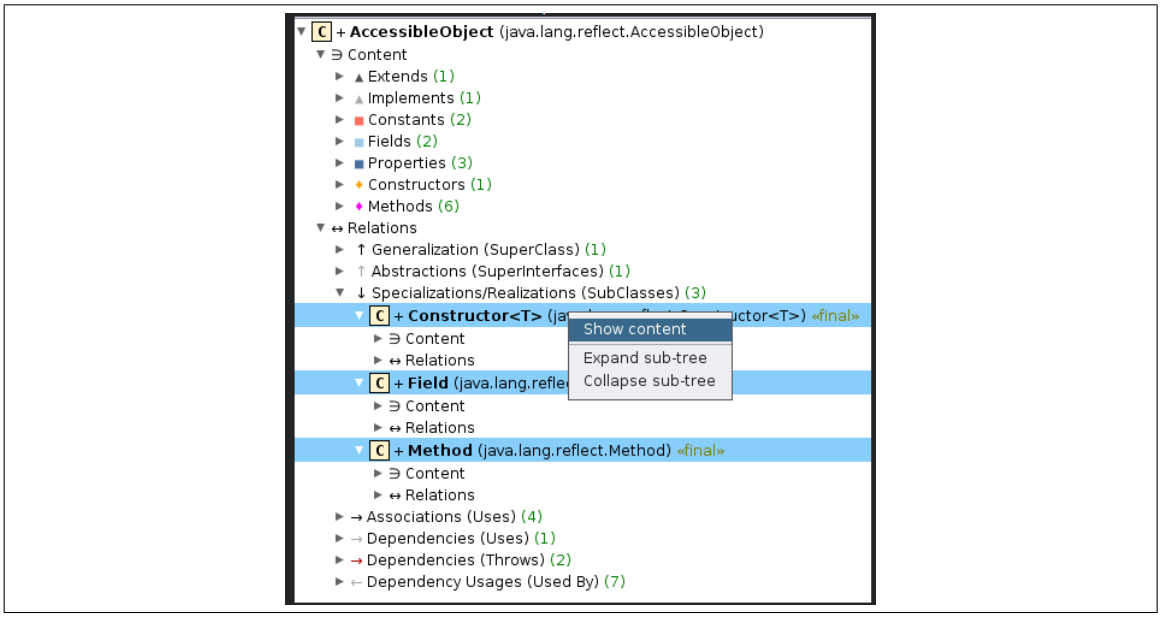

Figura 4.2: Información complementaria del diagrama generado por *Class Visualizer*

Además, la página web del proyecto cuenta con una documentación tanto escrita como en vídeo que ayudan a la comprensión de su funcionamiento.

El problema de la aplicación es que trabaja con clases en *Java* y no *JavaScript*, pero se mantuvo la opción pensando en la posibilidad de adaptarla para que funcionase con *JavaScript*. Esta adaptación se estimó en 12 horas.

4.2.2. Codeling

Codeling [10] es una aplicación que partiendo del código genera a diagramas de clase UML que pueden ser modificados. Además, se encarga de reprogramar el proyecto para adaptarlo a esas modificaciones.

Este proyecto también cuenta con una amplia documentación en su página web, aunque no cuenta con vídeos. La desventaja de usar esta herramienta es que tiene dos problemas:

- Da problemas con algunas implementaciones en *JavaScript*. Por ejemplo, si hay clases que se llaman igual, pero están en carpetas diferentes, solo aparece una de estas.
- Teniendo en cuenta cómo funcionan las líneas de producto software, es posible que al hacer cambios desde los diagramas generados, el programa borre las anotaciones generadas para el funcionamiento de la *SPL* y provoque errores en la ejecución.

```
// PVSCL:IFCOND(WindSpeed)
var windMeasure = 0;
function applyWindSpeed() {
    var measureText = document.getElementById("w_measure");
    windMeasure = measureText.value;
    var pointer = document.getElementById("w_point");

    applyTachoValue(minWind, maxWind, measureText, pointer);
    setWarnings();
    return false;
}
// PVSCL:ENDCOND
```

Figura 4.3: Anotaciones en una línea de producto software.

Por ejemplo, como se puede ver en la figura 4.3, el método *applyWindSpeed* se encuentra entre unas anotaciones que indican que este método debe ser usado solo si la característica *WindSpeed* está seleccionada. Al realizar un cambio en el diagrama, la clase se reescribe y aunque mantiene los comentarios, las funciones se reescriben, esto hace posible que, en caso de cambio, la función ya no esté entre las anotaciones.

Si se ejecuta *Codeling* en un proyecto de ejemplo se consigue un diagrama como el que se muestra en la figura 4.4. Como se ve en la figura, en el diagrama se recogen atributos y métodos de las clases, relaciones de asociación (con roles y cardinalidades) y herencia.

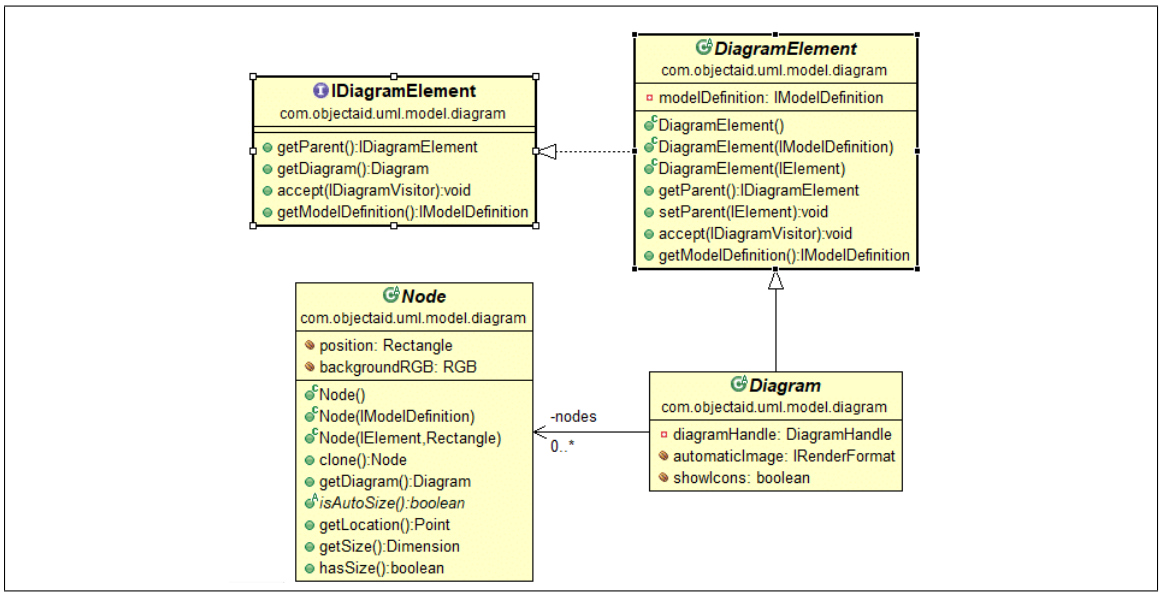

Figura 4.4: Diagrama de clases generado mediante *Codeling*

Además, tiene que ser adaptado para ser usado por *JavaScript*, ya que solo tiene soporte para Java. Se estima que se necesitaran unas 10 horas para adaptarlo.

Las pruebas realizadas en *ClassVisualizer* y *Codeling* están hechas sobre el proyecto de ejemplo que traen estas aplicaciones.

4.2.3. Arkit

Arkit [8] es una aplicación que genera diagramas de dependencias sobre proyectos *JavaScript*. Los diagramas muestran todos los componentes con sus relaciones, de manera que se pueda saber de que otros componentes dependen. Por ejemplo, en la figura 4.5 se pueden ver los diferentes componentes de la carpeta *Server* en el proyecto *ReactDOM*, un proyecto de ejemplo desarrollado por los creadores de *Arkit*. En la figura se ven sus dependencias y relaciones. En negrita aparecen los componentes de los que no dependen otros componentes, como por ejemplo el componente *Server*, este importa otros componentes pero no es importado por ninguno. Los componentes que son utilizados en otros componentes están en un color más claro, como pueden ser *header.vue* o *footer.vue*, ambos importados por *_slug.vue*.

Arkit genera estos diagramas en formato plantUML [11], un lenguaje que se utiliza para generar diagramas tal y como se explica en el apartado 5.4. Utilizar PlantUML permite modificar fácilmente el output, lo que es un punto a favor de la aplicación. Con el diagrama creado, *Arkit* genera un

PNG o SVG. Para poder generar este diagrama, *Arkit* genera un json de configuración en la carpeta del proyecto, que puede ser modificado para obtener diferentes resultados. Estas configuraciones pasan desde excluir carpetas, incluir ficheros en otros lenguajes como *TypeScript* [12] o incluso separar por partes las clases dependiendo de su localización en el proyecto. Las opciones de configuración se explican en el apartado 5.3, además, todas estas opciones están bien documentadas en su github.

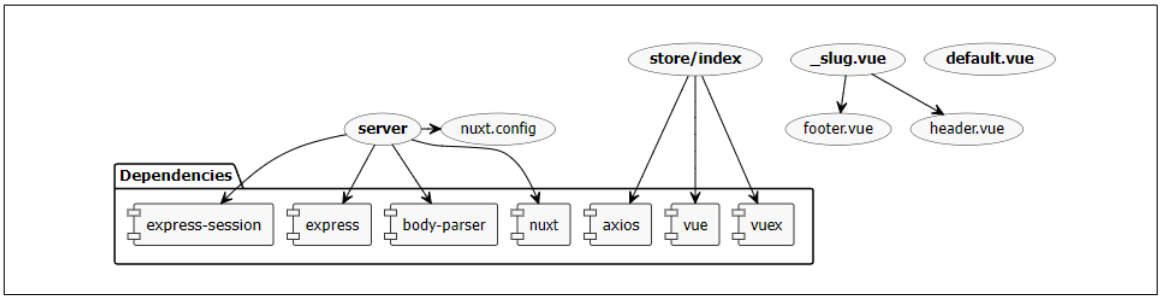

Figura 4.5: Diagrama de dependencias generado por Arkit

En la figura 4.6 se recoge el resultado de una ejecución que muestra el reparto de las clases por las carpetas *Server* y *Client*. También se puede ver como algunos componentes aparecen fuera de los cuadrados. Esto se debe a que la aplicación agrupa en estos cuadrados los componentes que se encuentran dentro de una carpeta, pero mantiene fuera aquellos que se encuentran en el directorio principal del proyecto.

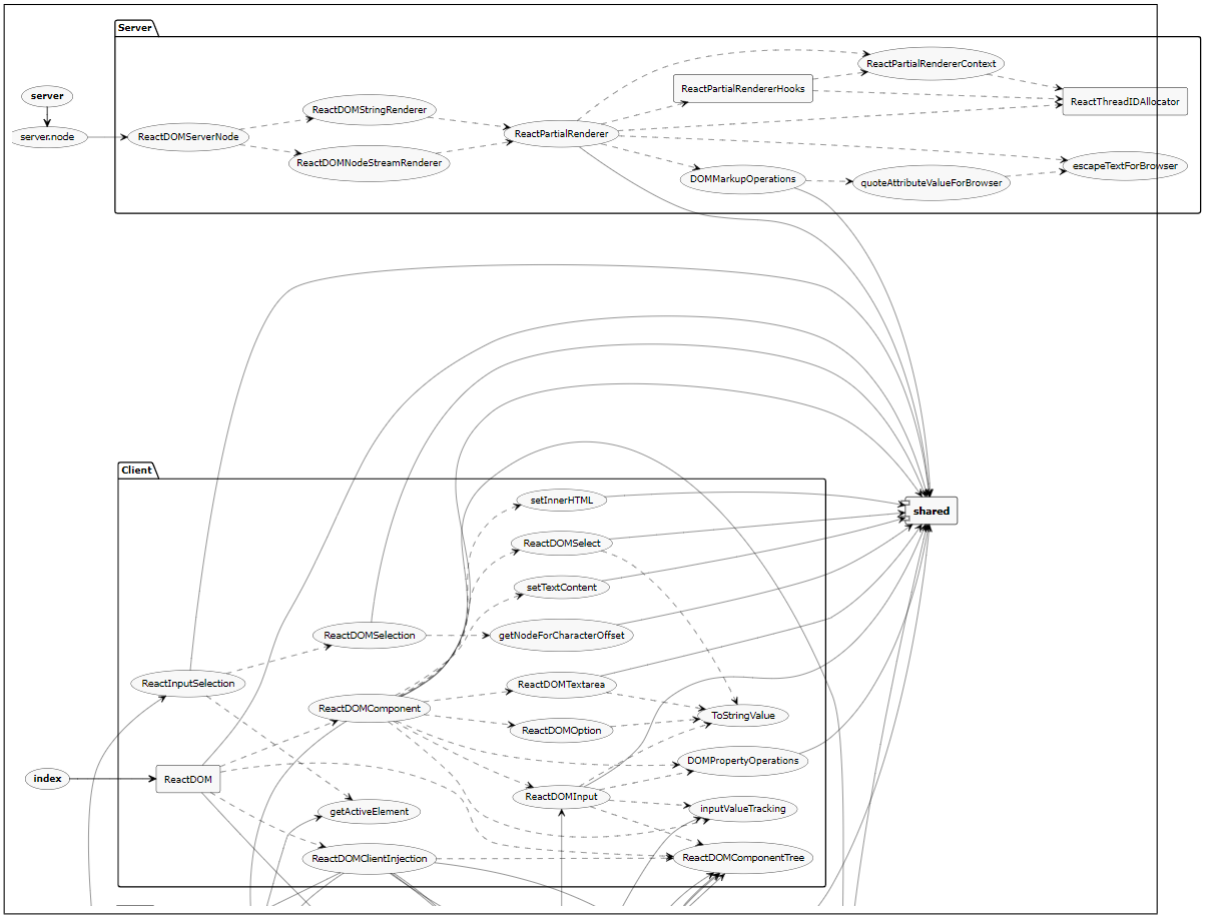

Figura 4.6: Diagrama de dependencias de un proyecto separado por carpetas

La aplicación permite ir profundizando en la visualización del diagrama. Por ejemplo, como se puede ver en la figura 4.6, se ha generado un diagrama sobre el proyecto entero, a diferencia de la figura 4.5, que contemplaba únicamente la parte del servidor. Para poder entender mejor el diagrama, se han separado los componentes por su origen, de esta manera, se puede, por ejemplo, ver que componentes de la parte del servidor interactúan con aquellos de la parte del cliente.

Para poder ver esta relación entre componentes de distintas carpetas, basta con fijarse en las relaciones que tengan una flecha continua, ya que las flechas discontinuas representan relaciones entre componentes de una misma carpeta.

Estas ejecuciones están realizadas sobre *Bootstrap* [13], una librería de CSS implementada en JavaScript.

4.2.4. js2UML

js2UML [14] es una herramienta que realiza diagramas UML sobre proyectos *JavaScript*, utilizando GraphViz, un software de visualización y generación de diagramas.

Esta herramienta es interesante porque a diferencia de *Arkit*, la herramienta descrita en el apartado 4.2.3, los diagramas que genera tienen también información sobre los métodos.

El problema de esta herramienta es que la gran mayoría de la documentación está en mandarín, complicando su comprensión.

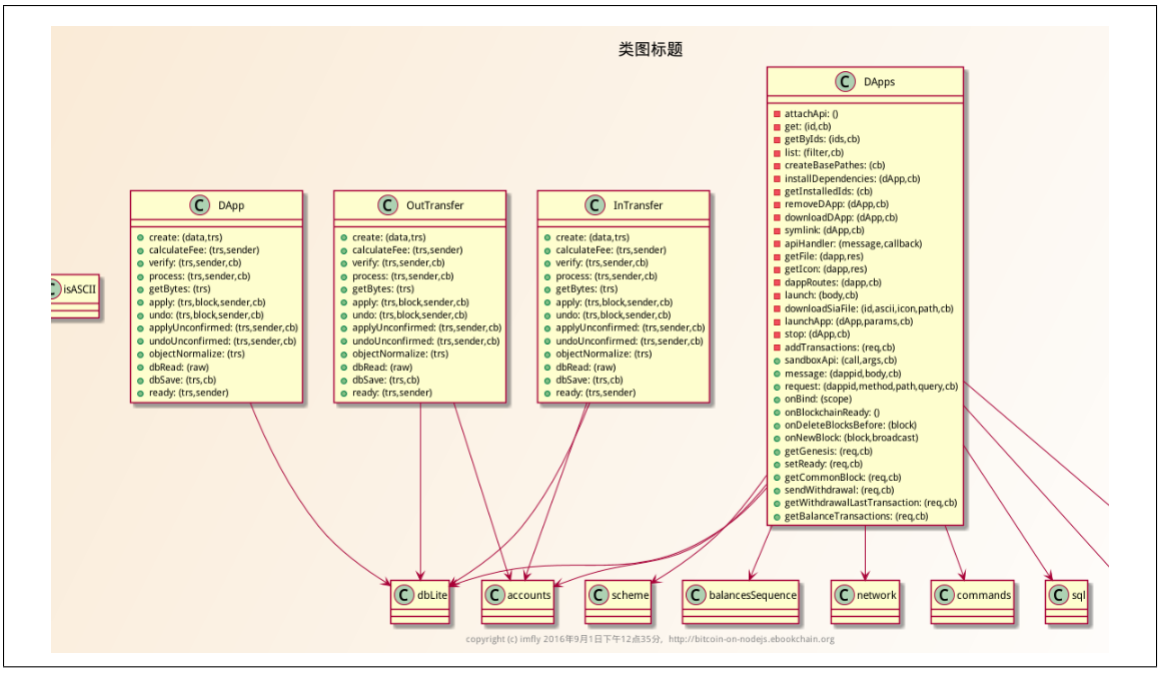

Figura 4.7: Ejemplo de ejecución de js2UML

Este ejemplo de ejecución está hecho con un proyecto diferente a *BootStrap*, herramienta con la que se han hecho las pruebas para *Arkit*, ya que da un código de error cada vez que se ejecuta con este proyecto, esto puede deberse al gran tamaño del proyecto.

Herramienta	Procesamiento de Javascript	Muestra descripcion de clases vía ADL	Muestra descripcion de clases Grafica	Tipos de diagrama	Formato de la información	Posibilidad de cambiar el aspecto del diagrama	Soporta otros lenguajes	Acceso via Web	Documentación	Tutoriales Video	Tiempo de adaptación a JavaScript	Razon descarte
Class Visualizer	FALSE	FALSE	Diagrama UML	Clase,Dependencias	UML	No	TRUE	FALSE	Extensa	TRUE	12	
Codeling	FALSE	FALSE	Diagrama UML	Clases,Dependencias	UML	No	TRUE	FALSE	Media	FALSE	10	
Arkit	TRUE	FALSE	Diagrama PlantUML	Componentes,Dependencias	PlantUML,PNG,SVG	Si	TRUE	TRUE	Media	FALSE		
Js2UML	TRUE	FALSE	Diagrama UML	Clase,Dependencias	UML	No	FALSE	TRUE	Incomprensible	FALSE		
Descartadas												
ArgoUML	FALSE	TRUE	Diagrama UML	Clase,Dependencias,Flujo	UML	Si	FALSE	FALSE	Ninguna	FALSE		Muy antigua, crasheo constante, no OpenSource
UModel	FALSE	FALSE	Diagrama UML	Clase,Dependencias	UML,PNG,PDF	No	FALSE	FALSE	Ninguna	FALSE		no OpenSource
eUML2	FALSE	FALSE	Diagrama UML	Clase,Dependencias	UML,PNG,PDF	Si	FALSE	FALSE	Ninguna	FALSE		no OpenSource
omondo	FALSE	TRUE	Diagrama UML	Clase,Dependencias	UML	No	FALSE	FALSE	Media	FALSE		Limite de procesamiento
MoDisco	FALSE	FALSE	Diagrama UML	Clase,Dependencias	UML,PNG	No	FALSE	FALSE	Extensa	FALSE		Solo trabja con .jar

Tabla 4.1: Tabla de criterios para selección de la herramienta

CAPÍTULO 5

Descripción de la herramienta Arkit

En este capítulo se describirá la arquitectura y funcionamiento de *Arkit* [8]. También se incluyen algunas capturas de ejecución y una sección dedicada a explicar *PlantUML* [11], el software utilizado para generar los diagramas.

Arkit es utilizado principalmente por desarrolladores de *JavaScript* para mantener, de manera actualizada, un diagrama con los componentes y dependencias de su proyecto. Esto permite ver, de una forma sencilla, cómo va evolucionando la estructura de un proyecto. La herramienta es capaz de procesar proyectos como *BootStrap* [13], que tienen más de 65000 líneas de código, en menos de cinco segundos. Por lo que es útil para cualquier tipo de proyecto en *JavaScript*.

La aplicación tiene más de 70000 líneas de código y cuenta con 50 clases diferentes. Como se puede ver en la figura 5.1, es perfectamente capaz de procesar su propio código y generar un diagrama de dependencias del mismo.

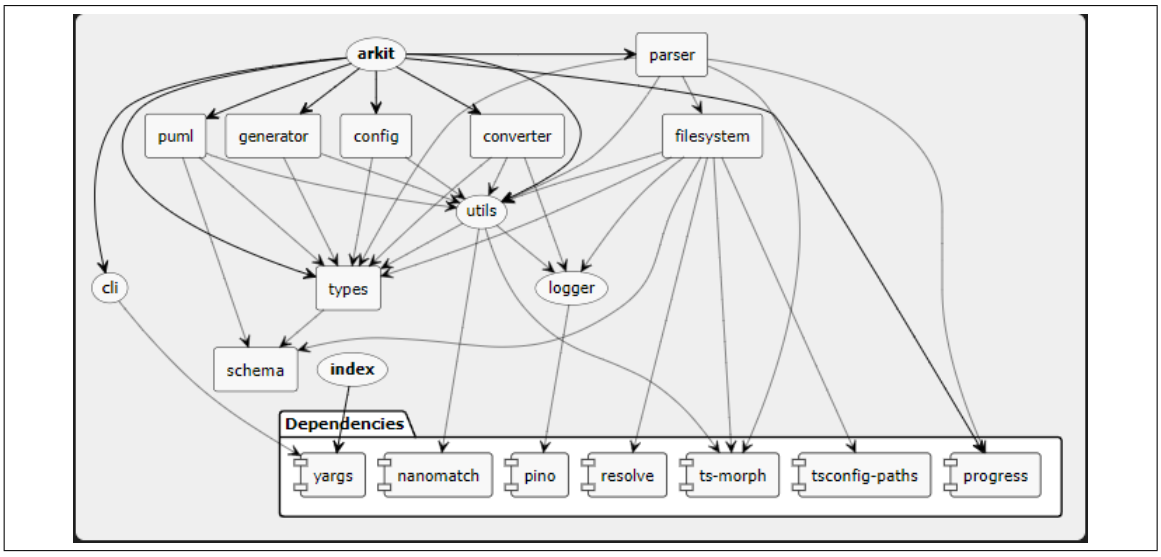

Figura 5.1: Diagrama de dependencias del proyecto *Arkit* generado con *Arkit*

En la figura 5.1 se pueden ver únicamente las clases implementadas en *JavaScript*, aunque existe la opción de incluirlas todas.

5.1. Arquitectura

Arkit está programado en *JavaScript* [15] y *TypeScript* [12], un lenguaje basado en *JavaScript* que añade clases y tipos estáticos.

Arkit se compone de dos carpetas principales, *Src* y *Dist*, estas carpetas engloban las clases necesarias para su funcionamiento. *Src* o *Source* contiene las clases en *TypeScript* que se forman importando funciones de los ficheros .js de la carpeta *Dist*. Por ejemplo, en la carpeta *Source* tenemos el componente *arkit.ts*, que importa todas las funciones definidas en el fichero *arkit.js* que se encuentra en la carpeta *Dist*.

En la figura 5.1 vemos las clases *JavaScript* que se encuentran en la carpeta *Dist*. En ella se encuentra *Arkit*, la clase principal que importa otros componentes. Vemos que algunos componentes tienen forma rectangular, representando que estos componentes son importados utilizando un "require", una librería de *JavaScript* para importar algunas funciones en forma de constantes.

Entre los diferentes componentes que conforman el proyecto, destacan los siguientes:

- **parser:** Procesa los archivos y los almacena en dos diccionarios, uno para las clases y otro para las carpetas, utilizando el componente *filesystem* para recorrer los archivos del proyecto.
- **config:** Procesa el archivo de configuración para poder establecer los diferentes parámetros de ejecución.
- **puml:** Escribe las clases con sus dependencias en un fichero .puml para que luego pueda ser procesado y transformado en un *PNG* o *SVG*.

La herramienta procesa todos los archivos que se encuentran en el proyecto y cumplen los requisitos para ello. Estos requisitos se definen en el archivo de configuración que se describirá en la sección 5.2.

Al procesar los archivos, *Arkit* genera unos diccionarios con cada archivo, carpeta a la que pertenece y los archivos que importa, y genera un fichero .puml que contiene el código para generar el diagrama. Este fichero .puml utiliza *PlantUml* [11], un software de generación de diagramas descrito en el apartado 5.4.

5.2. Configuración de la aplicación

Arkit cuenta con dos opciones para modificar el resultado de su ejecución. La primera opción es utilizar los argumentos básicos de la ejecución y la segunda es modificar el archivo de configuración. Los argumentos de ejecución son los siguientes:

- **-d**: Define el directorio sobre el que se ejecuta arkit.
- **-c**: Define el *path* del fichero de configuración.

- **-o**: Define el *path* del diagrama generado.
- **-f**: Define el patrón de los componentes que aparecerán primero en el diagrama.
- **-e**: Define qué clases no aparecerán en el diagrama.
- **-h**: Comando de ayuda. Muestra por pantalla las opciones de ejecución.
- **-v**: Muestra por pantalla la versión.

Estas opciones no permiten la suficiente personalización, así que se utilizó el fichero de configuración. En el archivo de configuración de *Arkit* se distinguen tres partes (como se ve en el ejemplo del código 5.1):

```
{
  "$schema": "https://arkit.pro/schema.json",
  "excludePatterns": ["test/**", "tests/**", "**/*.test.*", "**/*.spec.*"],
  "components": [
    {
      "type": "Dependency",
      "patterns": ["node_modules/*"]
    },
    {
      "type": "Component",
      "patterns": ["**/*.js", "**/*.jsx"]
    }
  ],
  "output": [
    {
      "path": "arkit.svg",
      "groups": [
        {
          "first": true,
          "components": ["Component"]
        },
        {
          "type": "Dependencies",
          "components": ["Dependency"]
        }
      ]
    }
  ]
}
```

Código JSON 5.1: Esquema de fichero de configuración

- **excludePatterns:** En este apartado se incluyen los componentes que no van a ser incluidos en la ejecución.
- **components:** En este apartado se incluyen los componentes que van a ser usados en la creación del diagrama, especificando cómo serán clasificados. Esta clasificación tiene dos parámetros:

1. ***type*** **:** Define el nombre del grupo de componentes que se procesara, este nombre puede ser cualquiera, ya que se procesara en el apartado "Output".

2. ***patterns*** **:** En este parámetro se define el patrón que cumplen las clases que han de incluirse.

En el ejemplo generado con el código 5.1, las clases que están en la carpeta "node_modules" serán las que tendrán como *type* "Dependency", mientras que el resto de las clases tendrán como *type* "Component".

- **output:** En este apartado se especifica cómo será el output, es decir, el diagrama, en qué formato se generará (PNG,SVG o .PUML), y cómo se fragmentarán los componentes descritos en el apartado *components.*

  Este apartado se divide en dos parámetros, *path*, que indica dónde y con qué extensión se creará el diagrama, y *groups*, que indica de que manera se organizaran los componentes que se han definido en *components*.

El parámetro *groups* se descompone de la siguiente manera:

- El parámetro *first* nos indica que componentes serán procesados íntegramente, ya que todos los componentes que no estén en *first* serán procesados únicamente si tienen relación con aquellos que se encuentren en *first*. Por ejemplo, si existiera algún componente en la carpeta "node_modules" que no tuviera relación con alguno de los componentes que aparecen en *first*, no aparecería en el diagrama final. Esto se debe a que los elementos de la carpeta "node modules" pertenecen al *type* "Dependency" y este tipo no esta incluido en *first*.

- El parámetro *type* sirve para definir una agrupación con el nombre asignado, añadiendo en *components* los componentes definidos anteriormente.

Para ilustrar mejor estas opciones, este fichero de configuración daría como resultado un *.SVG* con el diagrama mostrado en la figura 5.1. En ella podemos ver cómo se han generado primero los componentes que se han definido con *type* "components", es decir, todos los componentes .js que no se encontraban en el fichero "node_modules".

También se puede ver cómo se han agrupado los componentes que tenían como *type* "Dependency" en un rectángulo que se llama "Dependencies", tal y como se ha definido en el *type* del apartado "Output". Sin embargo, en la carpeta "node_modules" hay algunos componentes que no aparecen en el diagrama, esto se debe a que, al no estar en el grupo *first*, solo aparecen los que tienen relación con los que si están en el grupo *first*.

Este fichero es especialmente importante ya que habrá que modificar su funcionamiento para adaptarlo a las diferentes opciones de una línea de producto software. Por ejemplo, se podría añadir la opción de añadir al fichero de configuración un apartado llamado "Products", para que solo procesase las clases que tuvieran que ver con esos productos.

5.3. Ejemplos de uso de la aplicación

A continuación se muestran algunos ejemplos de uso de la herramienta *Arkit* sobre el proyecto *BootStrap* [13], una famosa librería de *CSS* [16] para desarrollo de *front-end*. El objetivo de estas pruebas es comprender el funcionamiento y output de la herramienta con un proyecto, además de demostrar que es capaz de procesar un proyecto real sin problemas.

Estas pruebas consisten en modificar el archivo de configuración de *Arkit* para poder obtener diferentes diagramas.

1. **Ejemplo de uso por carpetas:** A continuación se muestran diferentes ejecuciones por carpetas, tras modificar el archivo de configuración tal y como se ve en el código 5.2 para que procesase únicamente los componentes de una de las carpetas del proyecto.Esta opción es útil para poder ver las relaciones de los ficheros dentro de una carpeta.
   Para ello se modifica el archivo de configuración generando un nuevo tipo de componente llamado "Entry". El objetivo de este componente será el de indicar que componentes .js se procesaran, mientras que el componente "Component" contendrá los componentes que tienen relación con los que se encuentran en el grupo "Entry". Es decir, se procesarán todos los componentes .js que se encuentren en el patrón definido en el tipo "Entry", pero para aquellos que se encuentren en el patrón de "Component", solo se procesaran si tienen alguna relación con aquellos que se encuentren en "Entry".

   Como se puede ver en el código 5.2, el fichero de configuración para las pruebas sobre una carpeta en concreto no es muy diferente del fichero estándar de configuración que se puede ver en el código 5.1. La principal diferencia es que en esta nueva versión la entrada es únicamente los ficheros .js que están en la carpeta *Src*.

   Ejecutando *Arkit* con este fichero de configuración conseguimos el diagrama mostrado en la figura 5.3.

```
{
  "$schema": "https://arkit.js.org/schema.json",
  "excludePatterns": ["test/**", "tests/**", "**/*.test.*",
  ↪  "**/*.spec.*"],
  "components": [
   {
      "type": "Entry",
      "patterns": ["src/*.js"]
    },
    {
      "type": "Component",
      "patterns": ["**/*.ts", "**/*.tsx","**/*.js", "**/*.jsx"]
    }
  ],
  "output": [
    {
      "path": "arkit4.png",
      "groups": [
                {
```

```
        "first": "true",
        "components": ["Entry"]
                },
        {
          "components": ["Component"]
        }
      ]
    }
  ]
}
```

Código JSON 5.2: Fichero de configuración para diagrama de una única carpeta.

En la figura 5.3 se pueden ver las diferentes clases que conforman la carpeta *Source* del proyecto, en ella, están en negrita las clases que importan otras clases o no son importadas por ninguna otra.
También se puede ver como casi todas las clases utilizan el fichero *Util*, que importa todas las clases de la carpeta *Util*. Se puede ver el diagrama de esta carpeta en la figura 5.2.

Este componente no está en la carpeta *Src*, pero aparece porque es importado en alguno de los componentes definidos en el tipo "Entry" del fichero de configuración (ver código 5.2). Es decir, el componente *Util* pertenece al tipo "Component".

No se han incluido todos los diagramas porque el resto eran similares al de la carpeta *Util*.
Todas las ejecuciones se realizaron en menos de 5 segundos y no hubo ningún fallo, demostrando que la herramienta es capaz de procesar por carpetas.

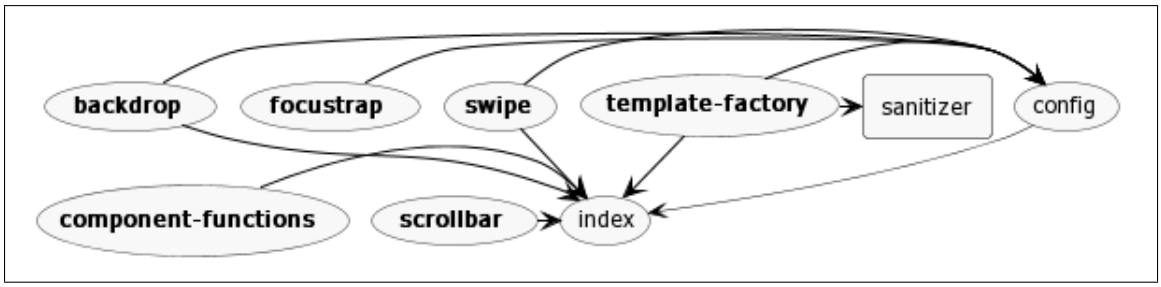

Figura 5.2: Diagrama generado para los componentes de la carpeta *Util*

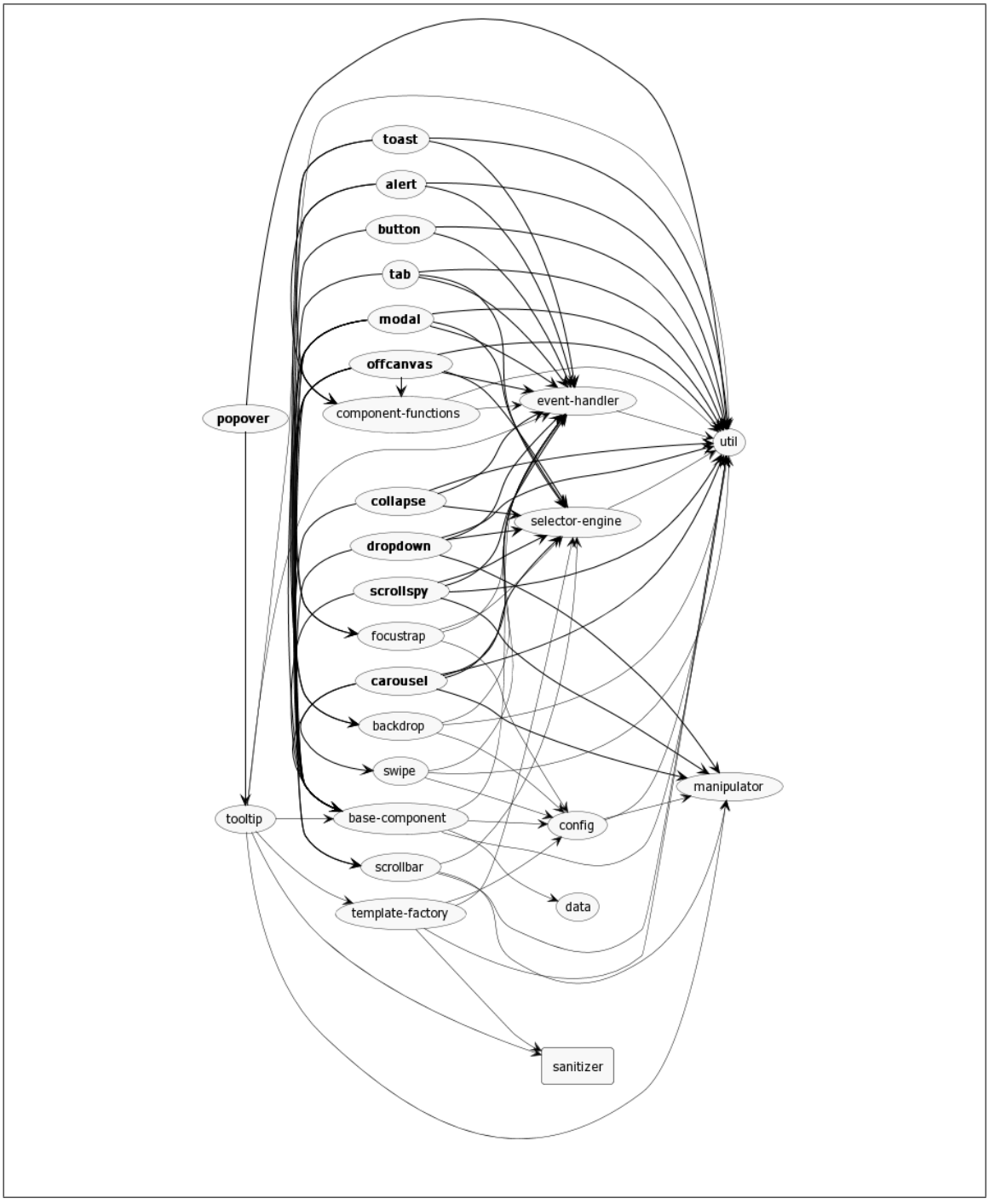

Figura 5.3: Diagrama generado para los componentes de la carpeta *SRC*

2. **Ejemplo de uso con el proyecto completo:** A continuación se muestra una ejecución con el proyecto completo, tras modificar el archivo de configuración mencionado en el apartado 5.2 para que generase rectángulos para representar cada carpeta del proyecto.

   Para poder entender mejor cómo se ha generado esta ejecución, se muestra a continuación el fichero de configuración 5.3.

```
{
  "$schema": "https://arkit.js.org/schema.json",
  "excludePatterns": ["test/**", "tests/**", "**/*.test.*",
  ↪  "**/*.spec.*"],
  "components": [
   {
      "type": "Entry",
      "patterns": ["**/*.js"]
    },
    {
      "type": "Component",
      "patterns": ["**/*.ts", "**/*.tsx","**/*.js", "**/*.jsx"]
    }
  ],
  "output": [
    {
      "path": "arkit4.png",
      "groups": [
        {
        "first": "true",
        "components": ["Entry"]
                },
                {
        "type": "JS Source",
        "patterns": ["js/src/*.js"]
                },
                {
        "type": "JS SourceDOM",
        "patterns": ["js/dist/dom/**"]
                },
                {
        "type": "JS SourceUTIL",
        "patterns": ["js/src/util/**"]
                },
                {
        "type": "Dist",
        "patterns": ["js/dist/**"]
                },
        {
          "components": ["Component"]
        }
      ]
    }
  ]
}
```

Código JSON 5.3: Fichero de configuración para diagrama por carpetas.

En el fichero de configuración se puede ver cómo se han ignorado las carpetas de *Test* y *Spec* en el apartado "excludePatterns". En el apartado components hay dos puntos:

- **Entry:** Da información de las clases que se van a procesar, en este caso, se tomaran como entrada todas las clases *JavaScript*.
- **Components:** En esta opción se añaden las clases que se quiere que aparezcan en las relaciones y no se encuentran en "Entry".

El apartado "Components" de este fichero de configuración es muy similar al fichero 5.2, con la única diferencia de que en este caso "Entry" contempla todos los ficheros .js. La principal diferencia se encuentra en el apartado "Output", en el que se generan 4 grupos, cada uno de ellos representando una de las carpetas del proyecto.

La diferencia entre este fichero de configuración y el fichero 5.1 a la hora de definir el grupo es que mientras que en fichero 5.1 se generaba un grupo introduciendo todos los componentes tipo "Dependency", en este fichero se añaden todos los componentes que cumplan un patrón, independientemente de si se encuentran en el tipo "Entry" o en el tipo "Component".

De este modo se genera un diagrama en el que aparecerán todos los componentes que tengan relaciones con los ficheros .js que cumplen el patrón definido en el tipo "Entry". Además, se separarán estos componentes por las carpetas en las que se encuentren. Como

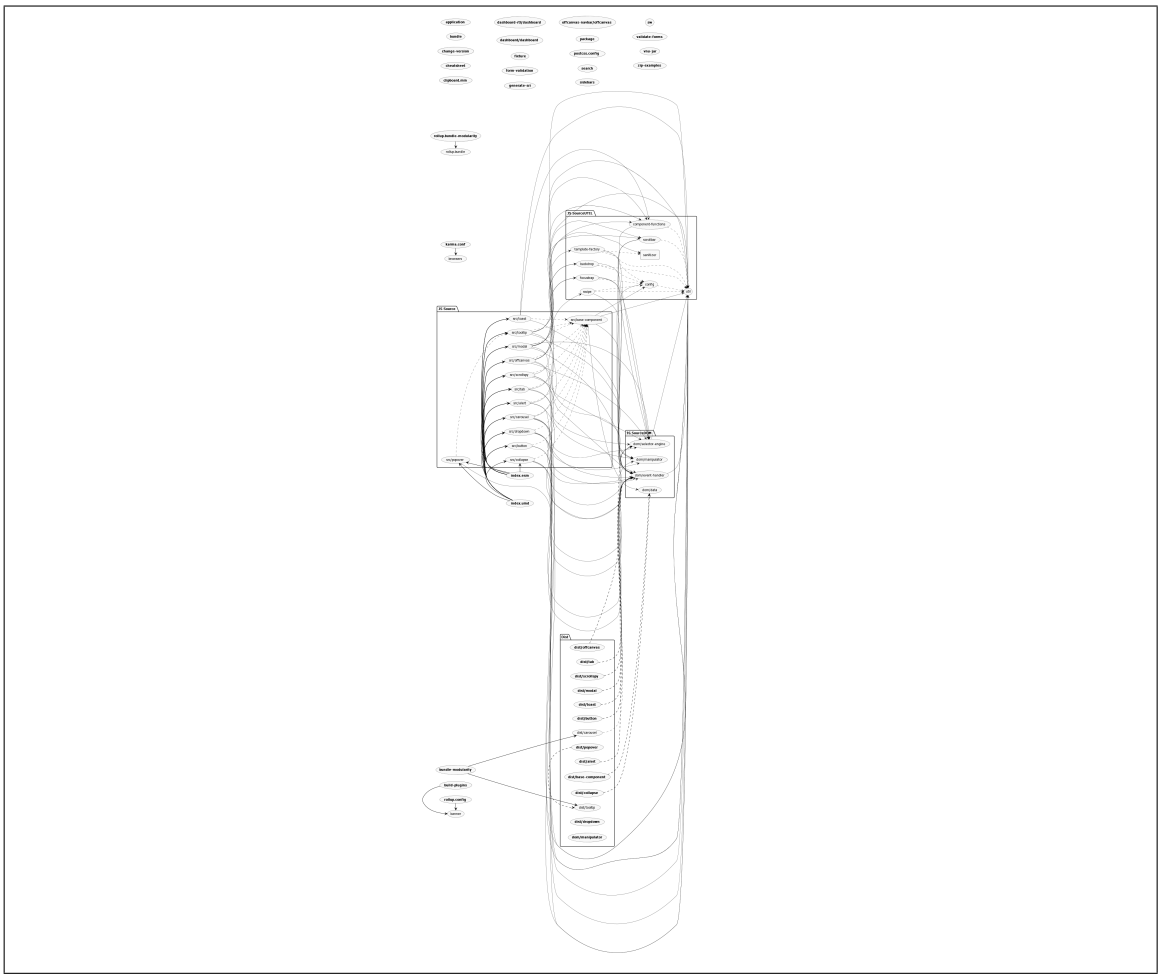

Figura 5.4: Diagrama generado para todos los componentes del proyecto

vemos en la figura 5.4, el diagrama contempla todos los ficheros del proyecto. Este diagrama es demasiado grande para analizarlo adecuadamente, así que basta con decir que se hizo una división de las clases por carpetas manteniendo todas las relaciones.

Además, los cuadrados que contienen las clases están renombrados, como se puede ver en el código 5.3, para poder clasificarlos mejor. Estas definiciones aparecen en el "output" tras cada "type".A continuación se pueden ver unos ejemplos de la solución anterior ampliada:

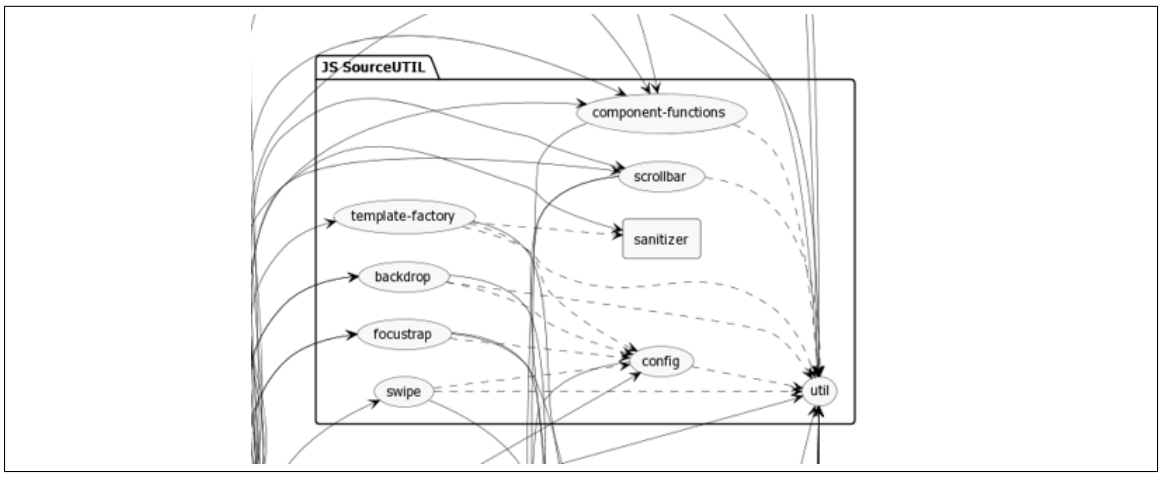

Figura 5.5: Parte del ejemplo que muestra el contenido de la carpeta *Util*

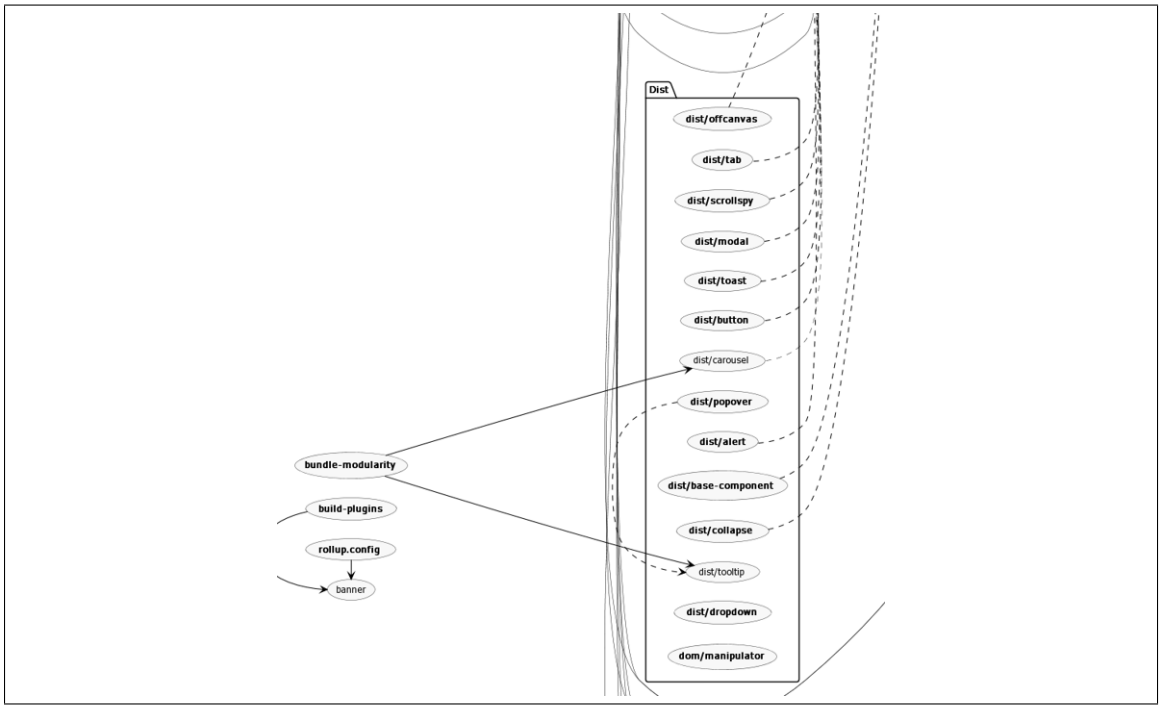

Figura 5.6: Parte del ejemplo que muestra el contenido de la carpeta *Dist*

Las clases que pertenecen a una carpeta aparecen su correspondiente cuadrado. En el ejemplo ejecución 5.6 hay algunas líneas discontinuas que apuntan a clases fuera del cuadrado. Esto es porque apuntan a clases que se encuentran en una sub carpeta de *Dist*.

5.4. PlantUml

Los ficheros generados por *Arkit* están escritos en PlantUML, el lenguaje que utiliza el software de mismo nombre, un software *OpenSource* de procesamiento de ficheros con extensión .puml.

Estos ficheros comienzan con la anotación @startuml y terminan con la anotación @enduml. Entre esas anotaciones están las clases, que se escriben entre paréntesis, y para generar las relaciones basta con escribir una flecha entre clase y clase, como se muestra en el código 5.1.

```
@startuml
(clase1)->(clase2)
(clase3)->(clase2)
(clase4)->(clase1)
@enduml
```

Código PlantUML 5.1: Descripción de un diagrama básico con tres clases.

Este código genera un diagrama como el siguiente:

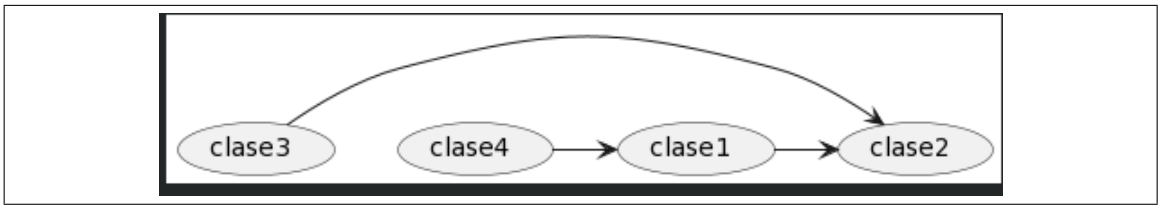

Figura 5.7: Diagrama generado con el código 5.1

Estos diagramas pueden extenderse añadiendo, por ejemplo, cardinalidad entre las relaciones, atributos a las clases o descripción de las funciones.

En el código 5.2 se muestra el fichero .puml que se utiliza para generar la figura 5.2.

Parámetros

Arkit no permite personalizar el estilo del diagrama generado, modificando, por ejemplo, el color de las clases, el grosor de las flechas o la forma de los componentes. Sin embargo, PlantUML si cuenta con algunos parámetros que pueden ser utilizados para personalizar el diagrama según las preferencias del usuario.
A continuación se listan algunos parámetros [17] que pueden ser modificados para cambiar el aspecto del diagrama.

- **scale**: Sirve para ajustar la proporción de la imagen generada dado un ancho o largo.
- **direction**: Marca la dirección del diagrama, tiene cuatro valores posibles, *left to right*, *right to left*, *top to bottom* y *bottom to top*.
- **nodesep**: Gestiona la distancia mínima a la que esta cada nodo.
- **defaultFontName y defaultFontSize**: Fuente y tamaño de la fuente del diagrama.
- **arrowColor y arrowThickness**: Tamaño y color de las flechas.

- **NodeBackgroundColor, NodeBorderColor y NodeFontColor**: Color del borde, interior y fuente de cada nodo.
- **dots per inch(dpi)**: Regula la cantidad de píxeles que utiliza el diagrama, es útil para ajustar el diagrama a la escala. Si el diagrama se sale por algún borde, conviene aumentarlo. Si no se define, su valor estándar es de 100.

```
@startuml
scale max 1920 width
top to bottom direction
skinparam nodesep 12
skinparam ranksep 25
skinparam monochrome true
skinparam shadowing false
skinparam defaultFontName Tahoma
skinparam defaultFontSize 12
skinparam roundCorner 6
skinparam dpi 150
skinparam arrowColor black
skinparam arrowThickness 0.5
skinparam packageTitleAlignment left
' oval
skinparam usecase {
  borderThickness 0.5
}
' rectangle
skinparam rectangle {
  borderThickness 0.5
}
rectangle "sanitizer" as _sanitizer
(<b>backdrop</b>) -[thickness=1]> (config)
(<b>backdrop</b>) -[thickness=1]> (index)
(<b>component-functions</b>) -[thickness=1]> (index)
(config) --> (index)
(<b>focustrap</b>) -[thickness=1]> (config)
(<b>scrollbar</b>) -[thickness=1]> (index)
(<b>swipe</b>) -[thickness=1]> (config)
(<b>swipe</b>) -[thickness=1]> (index)
(<b>template-factory</b>) -[thickness=1]> (config)
(<b>template-factory</b>) -[thickness=1]> (index)
(<b>template-factory</b>) -[thickness=1]> _sanitizer
@enduml
```

Código PlantUML 5.2: Descripción del diagrama mostrado en la figura 5.2

El código 5.2 muestra el fichero *plantuml* correspondiente a la figura 5.2. Dicho fichero comienza con una definición del tamaño del diagrama, seguido de un indicador de cómo está construido.

Primero, utilizando la palabra clave ”skinparam”, se definen los diferentes parámetros del diagrama,

tal y como se explica en el apartado 5.4. Después se definen los dos tipos de elemento que se pueden encontrar en el diagrama, el óvalo y el rectángulo, estableciendo el grosor del borde con el parámetro “borderThickness”.

Finalmente, se definen las clases, y después se escriben sus relaciones. En el caso de “sanitizer”, se define primero que es de tipo rectangle, ya que si se pone entre paréntesis aparecerá como óvalo.

Además, se define en las relaciones el grosor de cada flecha, como se puede ver en la figura 5.2, la flecha desde “config” a “index” es ligeramente más fina, ya que al definir el parámetro “arrowThickness” se le da el valor de 0.5, mientras que todas las demás relaciones modifican este parámetro en la propia flecha.

CAPÍTULO 6

Adaptación de Arkit para su funcionamiento con líneas de producto software

En este capítulo se explica el proceso de adaptación de la herramienta seleccionada para su uso con proyectos desarrollados en *pure::variants*. Primero se describe el proceso de adaptación a JavaScript, los problemas encontrados y la decisión de implementar la herramienta en Java. Finalmente, se describe la solución en Java y su proceso de implementación.

6.1. Estructura de un proyecto *SPL* implementado en pure::variants

A continuación se describe la estructura de un proyecto implementado en *pure::variants* para poder explicar cómo se debería procesar para obtener la información a nivel de arquitectura.

Weather Station

Para explicar la estructura de un proyecto *pure::variants*, se utiliza el proyecto WeatherStation [18]. Este proyecto implementa una aplicación de predicción del tiempo en *HTML* y *JavaScript*.

Model Elements	Level	Ankara	Athen	Berlin	Bern	Dubai	Hamburg	London	Madrid	Magde...	NewYork	Oslo	Paris	Prag	Rom	Stuttgart	Wien
WeatherStationFeatures																	
Weather Station		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Sensors	1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Temperature	1.1	☑	☑	☑	☑	☑	☐	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑
Wind Speed	1.2	☐	☑	☑	☑	☑	☑	☐	☐	☑	☑	☑	☐	☑	☐	☑	☐
Air Pressure	1.3	☐	☐	☐	☐	☐	☐	☑	☑	☑	☐	☐	☑	☐	☐	☑	☑
Languages	2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
English	2.1	☑	☑	☒	☒	☑	☒	☑	☑	☒	☑	☑	☑	☑	☑	☒	☒
German	2.2	☒	☒	☑	☑	☒	☑	☒	☒	☑	☒	☒	☒	☒	☒	☑	☑
Warnings	3	✓	✓	☐	☐	✓	✓	✓	☐	✓	✓	☐	☐	✓	☐	✓	✓
Gale / Strong Wind	3.1	☐	☑	☐	☐	☑	☑	☐	☐	☑	☑	☐	☐	☑	☐	☑	☐
Heat	3.2	☑	☑	☐	☐	☑	☐	☑	☐	☑	☑	☐	☐	☑	☐	☑	☑

Figura 6.1: Tabla de características de *WeatherStation* con sus selecciones por productos.

Para crear esta línea de productos se utilizan varias características como “German” o “English” para escoger el idioma en el que se implementan algunas alertas, o características como “AirPreassure” para saber si el producto contará con la función que da información sobre la presión atmosférica.

Los productos tienen nombres de ciudades, simulando que estas implementaciones se usarían para las estaciones meteorológicas de las diferentes ciudades. Por ejemplo, el producto Berlin utiliza la característica “German” pero no la característica “AirPreassure”(ver figura 6.1).

Features

Las *Features* son características de la implementación de una *SPL* que permiten añadir y quitar partes del código, tal y como se explica en el capítulo 3. Estas características se encuentran agrupadas en el fichero de configuración *.xfm* que genera el *FeatureModel*. Este fichero tiene la estructura de un *XML* y sus elementos conforman las diferentes características del proyecto tal y como se muestra en el codigo 6.1.

```
<cm:element cm:id="iypo4fjbnzCe3-d-j" cm:name="Gale" cm:type="ps:feature"
↪  cm:class="ps:feature" cm:default="off">
      <cm:relations cm:id="i1JzoTZqgvrdrtJvn" cm:class="ps:dependencies">
        <cm:relation cm:id="iWW37-gm6yp3Hfxio" cm:type="ps:requires">
          <cm:target cm:id="iAn20nsT3nGXNiop-">./iiB5wK-h-iipDy4tU</cm:target>
        </cm:relation>
      </cm:relations>
      <cm:relations cm:id="iTkL-LM1G66pw6xOz" cm:class="ps:parents">
        <cm:relation cm:id="iS9ZN98iUG8gRog36" cm:type="ps:parent">
          <cm:target cm:id="izXURZuUTIiuXOYWu">./iBFyOWLYHFpADbYjX</cm:target>
        </cm:relation>
      </cm:relations>
      <cm:properties>
      </cm:properties>
      <cm:vname>
        <cm:mimedesc cm:id="iEbFTNJNev65_lpoh" cm:mimetype="text/plain"
        ↪  cm:encoding="utf-8">Gale / Strong Wind</cm:mimedesc>
        <cm:mimedesc cm:id="iD7DyxnORH6QCYe8W" cm:mimetype="text/plain"
        ↪  cm:lang="de" cm:encoding="utf-8">Sturm / Starker
        ↪  Wind</cm:mimedesc>
        <cm:mimedesc cm:id="ieUNGcCl7dQVxoHtc" cm:mimetype="text/plain"
        ↪  cm:lang="zh" cm:encoding="utf-8">[U+70C8][U+98A8] /
        ↪  [U+5F37][U+98A8]</cm:mimedesc>
      </cm:vname>
    </cm:element>
```

Código XML 6.1: Declaración de una característica en el fichero de *FeatureModel*

Podemos ver que el atributo “cm:name” es “Gale”, que es el nombre de la característica. También nos da más información, como podrían ser las relaciones, es decir, qué características deben estar o no seleccionadas para que esta característica pueda ser utilizada y viceversa.

Por ejemplo, en este caso, en las etiquetas “cm:relations” aparecen dos elementos, el primero, con

tipo “ps:requires”, hace referencia a otra característica que deberá ser seleccionada para que esta esté disponible. A través del valor de la etiqueta “cm:target” se puede saber que la característica es “WindSpeed”.

La otra característica que aparece en sus relaciones es de tipo “ps:parent”, lo que nos indica la característica de la que deriva esta, que, como anteriormente, se puede conocer a través del valor de la etiqueta “cm:target”. Este valor nos da su *id*, en este caso, el *id* de la característica “Warnings”.

Productos

Los productos o variantes son las diferentes opciones de implementación que plantea una *SPL*. Estos productos forman un conjunto de características que han sido escogidas por un cliente.

Estas variantes están almacenadas en la carpeta “Variants” del proyecto. Las variantes están formadas por elementos, algunos de los cuales tienen el atributo “cm:type” con valor “selected”, esto significa que ese elemento, que referencia una característica, ha sido seleccionado y esa variante contendrá la característica en cuestión.

```
<cm:element cm:id="iqVOm1GCyO8DBaJ6k" cm:type="ps:selected"
↪  cm:class="ps:selection">
      <cm:relations cm:id="ipcCWflowwEIys8ZP" cm:class="ps:references">
        <cm:relation cm:id="ifjCx6E2bF8BKMhaD" cm:type="ps:references">
          <cm:target
          ↪  cm:id="i-uUbDCNGSzgfk3tq">iZDYEoOJvFKXojv_z/i3o2RSj4Fg7d7nNlj</cm:target>
        </cm:relation>
      </cm:relations>
      <cm:properties>
        <cm:property cm:id="i3L9ifSeF9lopc4ng" cm:type="ps:string"
        ↪  cm:invisible="true" cm:name="ps:selector">
          <cm:constant cm:id="id7C75guoEf6WDDxS"
          ↪  cm:type="ps:string">ps:auto</cm:constant>
        </cm:property>
      </cm:properties>
    </cm:element>
```

Código XML 6.2: Elemento de una variante

El elemento que se procesa en el codigo 6.2 tiene como tipo “selected”, eso significa que este producto contiene la característica en cuestión. Para saber qué característica es la utilizada, podemos ver que en el nodo “relation” existe un nodo hijo llamado “target”.

El valor de este nodo es el del *id* del nodo padre del documento de las características, seguido de un “/” y del *id* de la característica como tal. En este caso el *id* pertenece a la característica sensors.

Estos elementos también pueden darnos otro tipo de información como los ficheros que utilizan o las características que no pueden usar. Por ejemplo, en el código 6.3 se representa un elemento con tipo “ps:nonselectable”, lo que nos indica que este elemento es una característica

que no puede ser seleccionada en la variante. En este caso, el *id* pertenece a la característica "German", que, tal y como se ve en la figura 6.1, aparece con una cruz roja. Esto simboliza que esta característica no puede ser seleccionada, ya que es alternativa a la característica "English".

```
<cm:element cm:id="isXlwSMt7TIusN3P6" cm:type="ps:nonselectable"
↪  cm:class="ps:selection">
      <cm:relations cm:id="iLDCK7pgX55ji9OEa" cm:class="ps:references">
        <cm:relation cm:id="inUDEjusPNz8QjaUH" cm:type="ps:references">
          <cm:target
          ↪  cm:id="iC9AULjKLjEbnbdkS">iRZYU6x7KDUtPNAnI/iBMIyFNcg7KGy0qz6</cm:target>
        </cm:relation>
      </cm:relations>
      <cm:properties>
        <cm:property cm:id="itDSO3Ayyp0kRayn0" cm:type="ps:string"
        ↪  cm:invisible="true" cm:name="ps:selector">
          <cm:constant cm:id="iSSy8vX52yKAqdjWe"
          ↪  cm:type="ps:string">ps:auto</cm:constant>
        </cm:property>
      </cm:properties>
    </cm:element>
```

Código XML 6.3: Elemento no seleccionable de una variante

Esta categorización de los elementos implica tener que procesar todo el fichero comprobando el "cm:type" de cada elemento para saber si las características son no-seleccionables o seleccionadas, ya que nos interesan solo estas últimas.

6.2. Diseño de la solución en Arkit

A continuación se describe la solución diseñada para adaptar la herramienta al paradigma de las *SPL*.

La idea para la primera adaptación es que los componentes muestren información sobre qué variante o producto forma parte de las mismas, para ello, habrá que parsear los documentos FeatureModel, FamilyModel y VariantModel explicados en el capítulo 3. Estos contienen la información sobre estos productos y sus relaciones con los componentes.

Una vez obtenida la relación entre las variantes y los componentes, hay que modificar el fichero plantUML.js de *Arkit*, que se encarga, dados los diccionarios sobre los componentes, de generar el fichero plantUML que permite generar el diagrama.

Esta modificación pasa por añadir en la creación del fichero *.puml* la funcionalidad de escribir tras cada componente los productos en los que aparece.

Para ejemplificar este caso, se utiliza el diagrama de ejemplo mencionado en el apartado 5.4. Se modifica pensando que pertenece a un *SPL* con dos productos, Variante1 y Variante2.
Si se simula que Variante1 utiliza solo los componentes 1 3 y 2, mientras Variante2 utiliza los componentes pares, el resultado de la ejecución sería el siguiente:

```
@startuml
(<b>clase1</b> \n Variant1)->(<b>clase2</b>\n Variant2 \n Variant1)
(<b>clase3</b> \n Variant1)->(<b>clase2</b>\n Variant2 \n Variant1)
(<b>clase4</b> \n Variant2)->(<b>clase2</b>\n Variant2 \n Variant1)
@enduml
```

Código PlantUML 6.1: Generación de diagrama con variantes

De esta manera generaríamos el siguiente diagrama:

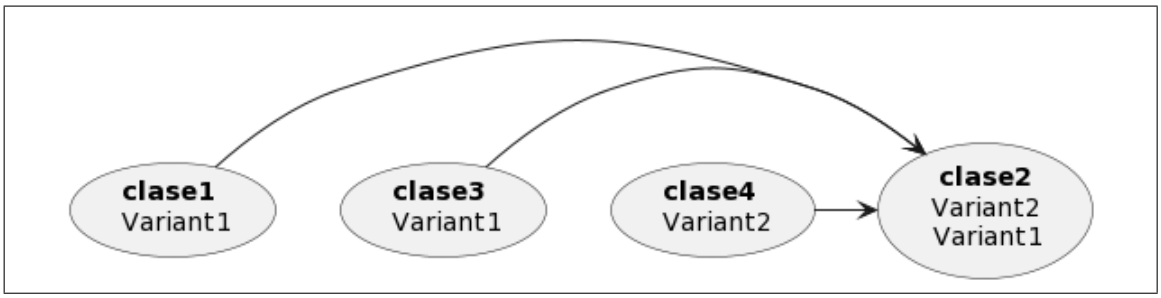

Figura 6.2: Diagrama con variantes generado

Así podríamos entender fácilmente a qué producto pertenece cada componente, o incluso mostrar solo los componentes que pertenecieran a uno o varios productos, manteniendo la información sobre las relaciones entre componentes.

También se tienen en cuenta las características, por lo que otra opción es mostrar en cada componente que característica es parte del componente, por ejemplo, en nuestro ejemplo de la figura 6.2, se podrían añadir tres características de manera que el diagrama quedase como en la figura 6.3

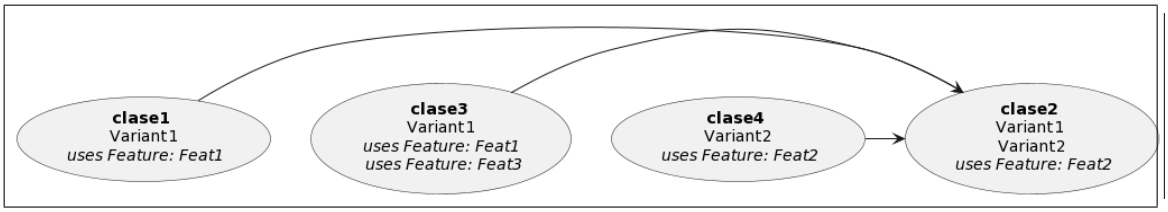

Figura 6.3: Diagrama con variantes generado

De este modo, la figura 6.3 da información sobre que característica es utilizada por cada componente, ofreciendo una capa de información extra sobre la que aporta *Arkit*.

Otra opción es mostrar solo los componentes que cumplen unos requisitos, por ejemplo, que pertenezcan a un producto concreto o que los componentes usen alguna de las características de una lista seleccionada por el usuario.

Solución en Arkit

Como se puede ver en la figura 5.1, el componente "arkit" utiliza diferentes componentes para realizar la solución, para poder implementar la adaptación a las *SPL*, se realizaran los siguientes cambios en la arquitectura.

- **Componente variantProcessor**: Este nuevo componente procesará los ficheros de *FamilyModel*, *FeatureModel* y *VariantModel*, generando diccionarios para las relaciones entre los componentes, las características y los productos. Con estos diccionarios se puede saber, por ejemplo, a qué productos pertenece un componente o a que características pertenece un componente en concreto.
- **Componente config**: Este componente se encarga de procesar el fichero de configuración explicado en la sección 5.2, hay que modificarlo para que contemple las opciones relativas a las líneas de producto software, tales como qué características contemplar o qué productos procesar.
- **Componente puml**: Este componente procesa la información generada por el componente parser y genera el fichero *.puml* que posteriormente se transforma en archivos *SVG* o *PNG*. Hay que modificar este componente para que contemple los productos y las características a la hora de generar el diagrama.

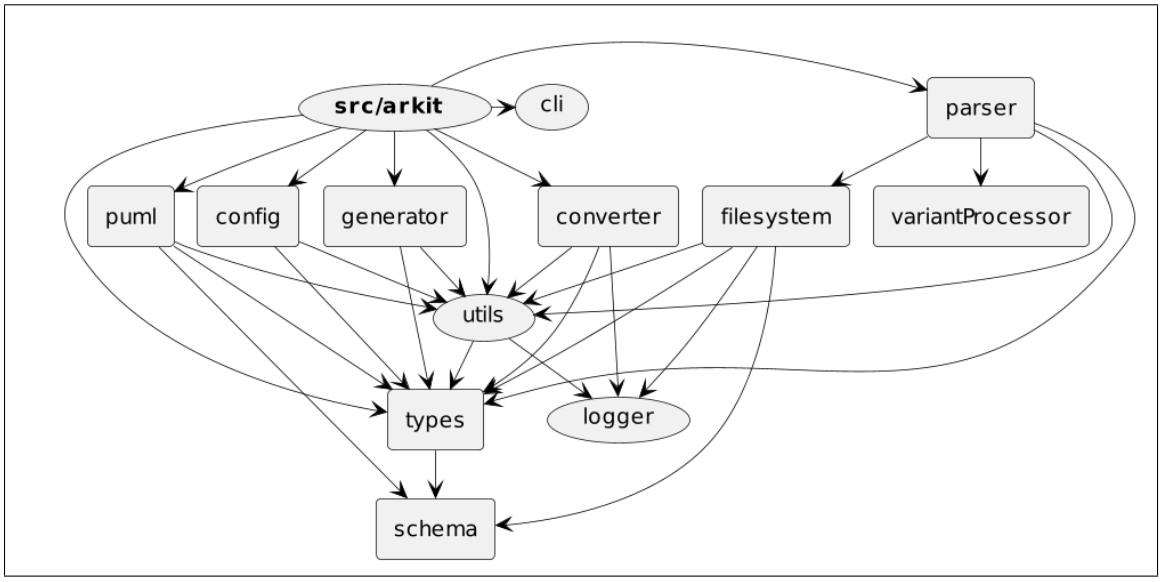

Figura 6.4: Arquitectura de Arkit tras la adaptación

Como se puede ver en la figura 6.4, el cambio principal está en el componente *parser*, que ahora implementa el nuevo componente *variantProcessor*. El resto de la arquitectura no se ve modificada, ya que los cambios estarán en el codigo de los componentes.

6.3. Implementación en JavaScript

En esta sección se describe la implementación realizada para llevar a cabo la solución diseñada, así como los problemas para la adaptación y la justificación para una "reimplementación" de la herramienta.

6.3.1. variantProcessor

Este componente procesa los ficheros *FamilyModel*, *FeatureModel* y *VariantModel* para generar diferentes diccionarios desde los que acceder a la información de la línea de producto software.

Para ello, utiliza los siguientes métodos:

- **GetFeatures**: Esta función devuelve las diferentes características del proyecto. Para conseguir la lista de características, basta con procesar el *FeatureModel*, devolviendo el valor del atributo "cm:name" de cada elemento. En la figura 6.5 se ve un ejemplo de ejecución de este método con el proyecto *WeatherStation*.

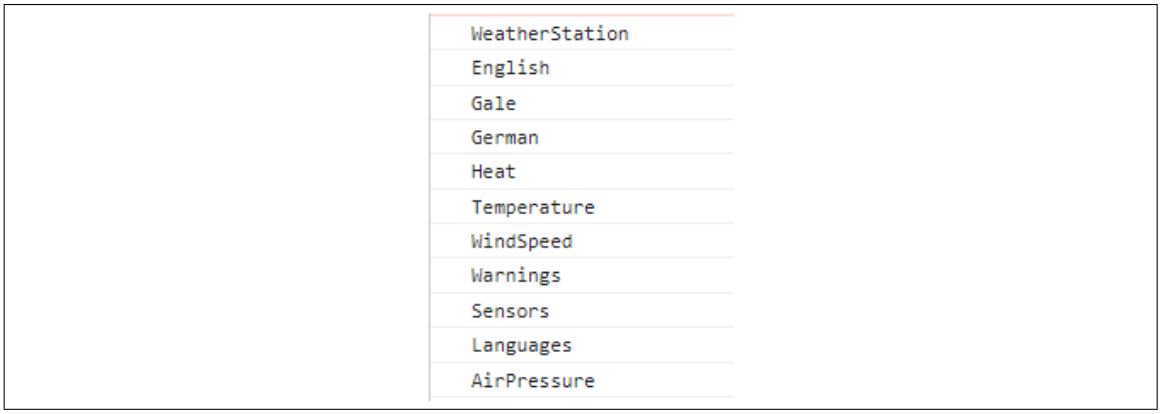

Figura 6.5: Ejemplo de ejecución de getFeatures con el proyecto *WeatherStation*

- **GetClassByID(idgiven)**: Este método devuelve el nombre de un componente dado su *id*. El método se encarga de recorrer el *FamilyModel* hasta encontrar el *id* proporcionando, después busca en las relaciones los diferentes nombres de los componentes que tienen relación con el *id* proporcionado. De esta manera, se genera un array de los componentes que utilizan esta característica.

- **GetClasses(feature)**: Esta función devuelve los componentes afectadas por una característica. Por ejemplo, que una característica esté seleccionada puede hacer que dentro de un componente aparezcan o no algunos métodos.

  En este método se va almacenando el *id* de cada elemento hasta encontrar el elemento con el nombre de la característica, para después llamar al método *getClassById*, que recorre el *FamilyModel* para conseguir los componentes asociadas al *id* de la característica.

  En el ejemplo de la figura 6.6 vemos como para la característica "Sensors" este método nos devuelve el componente "sensors.js", que es el único componente que es implementado por la característica. En el ejemplo de *WeatherStation*, cada característica se implementa únicamente en un componente, pero en una línea de producto software como *WacLine*, es normal que una característica se implemente en varios componentes.

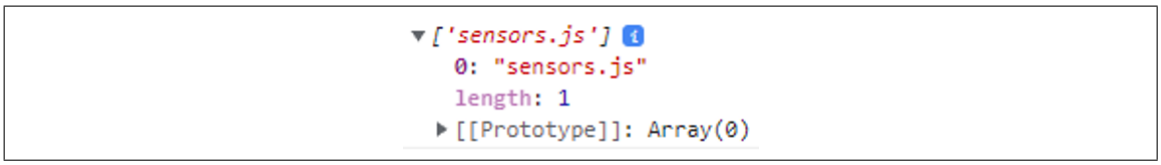

Figura 6.6: Ejemplo de ejecución de getClasses con el proyecto *WeatherStation*

- **GetVariantClass(variant)**: Este método devuelve todos los componentes de una variante. Funciona de manera similar al método *getClasses*. La principal diferencia es que para conseguir los componentes, este método recorre el *VariantModel* de la variante introducida como parametro.

  En este método se recorre el *VariantModel* de un producto, buscando todos los elementos que tengan como tipo "ps:selected", después se llama al método getVname con el *id* de ese elemento para obtener el nombre de ese componente.

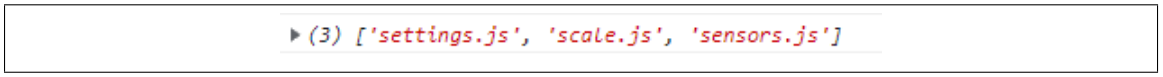

Figura 6.7: Ejemplo de ejecución de *GetVariantClass* con el proyecto *WeatherStation*

  En la figura 6.7 se puede ver el resultado de ejecutar este método, pasando como parámetro de entrada el producto "Ankara.vdm", del cual se puede ver su información en la figura 6.1. En el caso de *WeatherStation*, solo hay tres componentes, por lo que todos los productos utilizan esos tres componentes.

- **getVname(id)**: Es similar a *getClassById*, solo que en este caso el *id* es la del propio elemento y no la de sus relaciones, por lo que en *getClassById* se obtenía un array de los componentes relacionadas con ese *id*, mientras que en este método se obtiene directamente el componente. En este método se recorre el *FamilyModel* hasta encontrar el elemento con el *id* introducida, después se obtiene su nombre y se devuelve.

- **getVariantsOfClass(component)**: Este método devuelve los productos que aparecen en un componente. Para ello, recorre las variantes generando una lista de componentes utilizadas por cada una de estas variantes. Después, comprueba si el componente está en alguna de estas listas para saber si la clase aparece en alguna de las variantes.

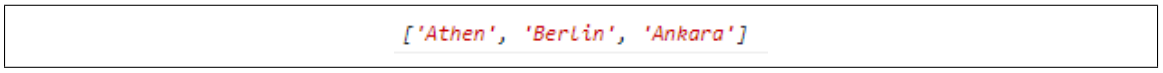

Figura 6.8: Ejemplo de ejecución de *getVariantsOfClass* con el proyecto *WeatherStation*

  En la figura 6.8 aparecen las variantes en las que se utiliza el componente "sensors.js". Solo aparecen tres productos, ya que fueron los que se utilizaron para las pruebas. En este método se llama a la función *getVariants()*, esta solo devuelve los tres productos que se ven en la figura para hacer más simples las pruebas.

Problemas de Integración

Tras desarrollar y probar el nuevo componente, hubo algunos problemas para integrarlo en la herramienta. Para entender el origen de estos errores, hay que explicar que *Arkit* cuenta con un bot llamado "renovate-bot" [19] que se encarga de actualizar las dependencias para que la herramienta esté actualizada.

Arkit utiliza *Pino* [20], una herramienta para crear logs de la ejecución. Con esta herramienta, *Arkit* genera las diferentes trazas de ejecución que utiliza. *Arkit* utiliza estos logs para almacenar un registro de la ejecución, pero también las utiliza para mostrar por pantalla el progreso de la ejecución. Para ello, utiliza la función *PrettyPint*, que permite escribir por consola información de los logs.

En marzo de 2022, los creadores de *Pino* crearon *pino-pretty*, un sistema para mejorar las funcionalidades de *PrettyPint*, pero en abril, *PrettyPint* dejo de ser funcional y las herramientas con *Pino* no funcionaban si se usaba sin la herramienta *pino-pretty* implementada.

El bot que se encargaba de renovar las dependencias actualizó *Pino*, pero no integró *pino-pretty* en el codigo de *Arkit*, por lo que cada vez que se intentaba ejecutar *Arkit* daba el error que podemos ver en la figura 6.9.

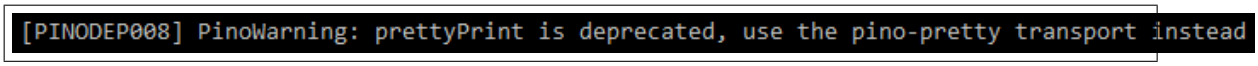

Figura 6.9: Error de Arkit en la ejecución

Después de recibir estos errores, se intentó descargar una versión anterior de *Arkit* para su uso hasta que se arreglara el problema, pero resultó que *Pino* realiza llamadas a un servidor interno, por lo que las versiones atrasadas no funcionaban. Tras esperar 3 días sin que actualizaran *Arkit*, se tomaron en consideración las siguientes opciones:

- **Implementar pino-pretty manualmente**: Leer la documentación de *pino-pretty* para saber que métodos habían sido modificados en *Pino*, para después implementar estos métodos en *Arkit*. El problema era el total desconocimiento sobre el funcionamiento de *Pino*, por lo que la estimación de tiempo para esta opción era poco precisa.
- **Reimplementar *Arkit* para no depender de Pino**: Esta opción pasaba por comprender en que componentes se utilizaba *Pino* para eliminarlo de la implementación. De esta manera se evitaría el error de las dependencias. El problema de esta opción es que es muy difícil determinar la estimación de tiempo total por el desconocimiento de la importancia de *Pino* para la ejecución.
- **Reimplementar la herramienta en Java**: Se estimó esta opción como la más viable, ya que tras comprender como funcionaba *Arkit*, la estimación de tiempo para realizar la reimplementación era la más precisa. Además, si la herramienta estaba en *Java*, sería mucho más fácil de integrar en *InsideSPL*, por lo que, aunque se utilizara más tiempo del estimado para la tarea de adaptación, se estaría recortando en tiempo de la integración, reduciendo el impacto de tener que implementar la herramienta en *Java*.

Además, escoger una opción que pasara por modificar *Arkit* suponía, para la integración en *InsideSPL*, tener que implementar un módulo en *Java* que hiciera llamadas a *Arkit* modificando el archivo *JSON* de configuración. De esta manera, la propia herramienta queda implementada en Java y se evita tener que implementar este módulo.

6.4. Diseño de la solución en Java

Solución en Java

Para la solución en Java, se contemplaron 3 módulos diferentes:

- **Generator**: Este módulo se encarga de replicar el trabajo realizado por el proyecto de *Arkit* original, genera los diccionarios para los componentes, sus relaciones y dependencias. También genera el fichero *.puml* y realiza la transformación *.puml* a *.PNG*.

- **Graphics**: Este módulo se encarga de gestionar la interfaz de la aplicación, esta interfaz sirve de sustituto del fichero de configuración de *Arkit*, ya que se pueden seleccionar las opciones de manera visual.

- **VariantMiner**: Este módulo se encarga de parsear los ficheros del *FeatureModel*, *FamilyModel* y *VariantModel* para obtener la información sobre las características y productos de un proyecto, de manera similar al componente *variantProcessor* diseñado para la solución en *JavaScript*.

Como se puede ver en la figura 6.10, los componentes del módulo *Graphics* generan los menús para que el usuario pueda escoger el proyecto sobre el que trabajar, que se pasa como parametro al módulo *VariantMiner*.

VariantMiner obtiene los productos y las características de este proyecto, y el módulo *Graphics* se encarga de preguntar al usuario mediante una interfaz qué productos y características quiere procesar.

Con esta información, el componente *VariantMiner* generará todos los diccionarios que tengan que ver con los productos y características seleccionadas. Por ejemplo, en el proyecto *WeatherStation*, tal y como se explica en la sección 6.1, se podría procesar uno de los productos (o varios) para obtener los componentes que son utilizados por este producto. Después, con las características seleccionadas, se generan diccionarios sobre los componentes que las utilizan.

En la última fase de la ejecución, se generan los diccionarios de los componentes con sus dependencias y toda la información recabada se utiliza para generar el diagrama *.puml*. Después se utiliza el diagrama para generar una imagen *.PNG* del diagrama.

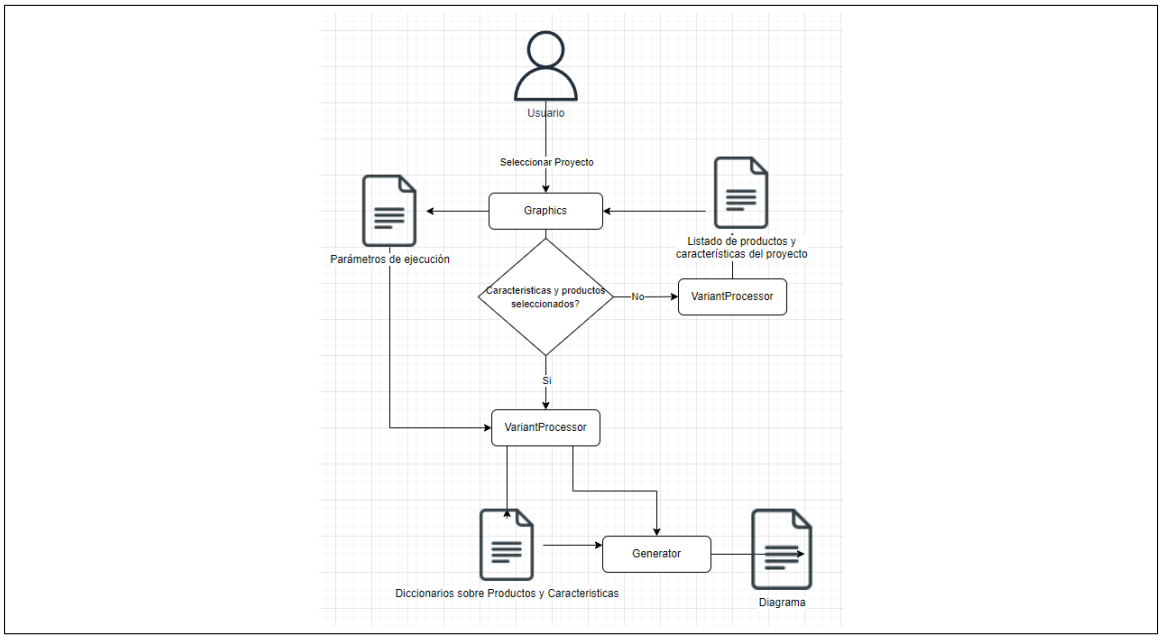

Figura 6.10: Diagrama de flujo de la ejecución para la solución diseñada

6.5. Implementación en Java

En esta sección se describe el proceso de implementación de *Arkit* en *Java* y el producto obtenido.

6.5.1. Módulo variantMiner

En el módulo *variantMiner* hay un solo componente, *VariantProcessor*, que implementa las mismas funciones que en la implementación de JavaScript descrita en el apartado 6.3. En la versión de Java, se añade un método nuevo, *getFeaturesOfVariant*, que se utiliza para generar las características que utiliza un producto en concreto.

Este método es importante para ofrecer al usuario la posibilidad de procesar las características que utilice el producto o los productos seleccionados. Por ejemplo, para el proyecto *WeatherStation*, si el usuario escogiese el producto "Ankara", no tiene sentido ofrecer la posibilidad de procesar la característica "German", ya que este producto no la utiliza.

6.5.2. Módulo Generator

El módulo *Generator* se encarga de replicar el proceso que realizaba *Arkit*, es decir, procesar los componentes y sus relaciones y generar el diagrama. Para realizar esta tarea, cuenta con dos componentes, *ClassList* y *PumlGenerator*.

ClassList contiene los métodos para generar los diccionarios que almacenan información sobre los componentes. Esta información puede ser las funciones de un componente, sus relaciones o la manera en la que las características afectan a las funciones. Por ejemplo, como se puede ver en la figura 6.11, en el componente *scale* del proyecto *WeatherStation* la característica *AirPreassure* afecta a la función *initiateScales*.

```
function initiateScales() {
    // PVSCL:IFCOND(AirPressure)
    var parent = document.getElementById('p_main');
    setScale(parent, minPres, maxPres, presScale, 14);
    // PVSCL:ENDCOND
```

Figura 6.11: Ejemplo de anotación de una característica en una función.

PumlGenerator se encarga de generar el diagrama *.puml* y transformarlo a *PNG*. Para ello recibe los diccionarios generados por *VariantProcessor* y *ClassList* y genera los diagramas teniendo en cuenta los parámetros de ejecución descritos en el módulo *Graphics*.

6.5.3. Módulo Graphics

El módulo *Graphics* se encarga de generar las interfaces para poder pedir al usuario los parámetros de ejecución. Estos parámetros son los siguientes.

- **Path del proyecto**: El primer parámetro es la dirección del proyecto que se quiere procesar.

- **Productos a procesar**: Este parámetro almacena los productos que se van a tener en cuenta para realizar el diagrama. Es importante aclarar que se procesaran todos los componentes que pertenezcan por lo menos a uno de los productos.
- **Características a procesar**: Este parámetro almacena las características que aparecerán en el diagrama, si algún componente utiliza una característica almacenada en este parámetro, esta se verá reflejada en el diagrama.
- **Carpetas seleccionadas**: Las carpetas que se utilizan para realizar el diagrama. Si el proyecto es demasiado grande, es recomendable separar el diagrama por carpetas.
- **Información sobre las funciones**: Este es un parámetro booleano que indica si se quiere o no mostrar información sobre las funciones. Esto se debe a que hay algunos proyectos que cuentan con componentes con demasiadas funciones como para que el diagrama sea legible. De esta manera, el usuario puede escoger si desea o no recibir información sobre estas funciones. Si este parámetro se selecciona, el diagrama muestra las funciones de cada componente con las características que le afectan.

Para realizar estas funciones, este módulo cuenta con cuatro clases:

1. **Menu**: Esta clase genera la interfaz para conseguir el path del proyecto.
2. **FolderSelector**: Esta clase se encarga de preguntar al usuario qué carpetas del proyecto quiere procesar.
3. **VariantSelector**: Esta clase genera la interfaz para preguntar al usuario que productos quiere procesar.
4. **FeatureSelector** Esta clase genera la interfaz para preguntar al usuario que caracteristicas quiere procesar. También pregunta al usuario si desea mostrar información sobre las funciones.

6.6. Ejemplos de ejecución

A continuación se muestran diferentes ejemplos de ejecución de la aplicación modificando los parámetros para generar diferentes diagramas.

6.6.1. Interfaz

Lo primero que hace el programa es preguntar por el path del proyecto, tal y como se ve en las figuras 6.12 y 6.13.

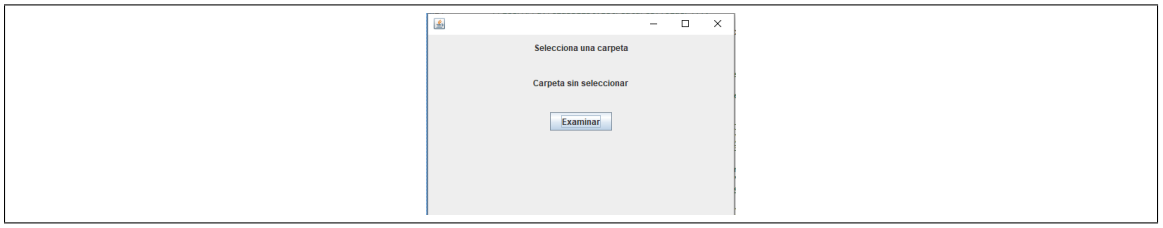

Figura 6.12: Menú de selección de la carpeta.

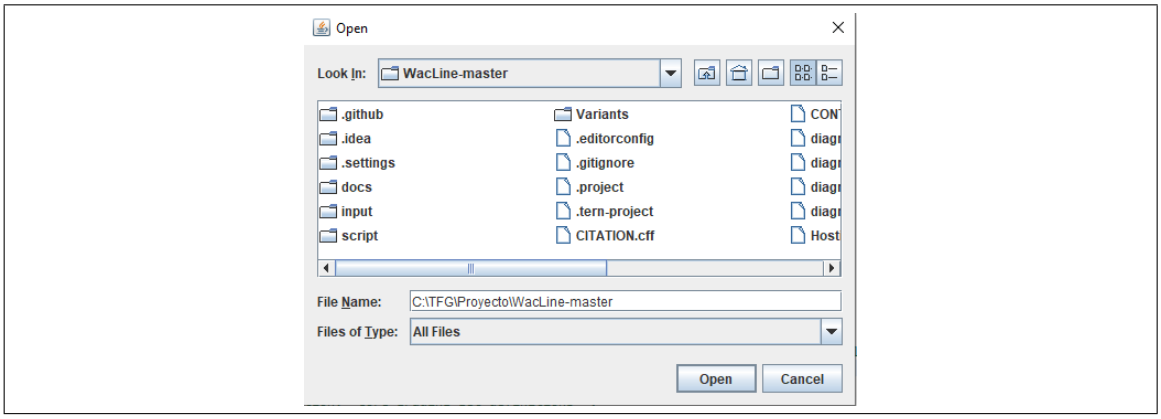

Figura 6.13: Menú de selección de la carpeta.

Una vez obtenido el path, se pregunta al usuario qué carpetas quiere procesar mediante la interfaz tal y como se ve en la figura 6.14

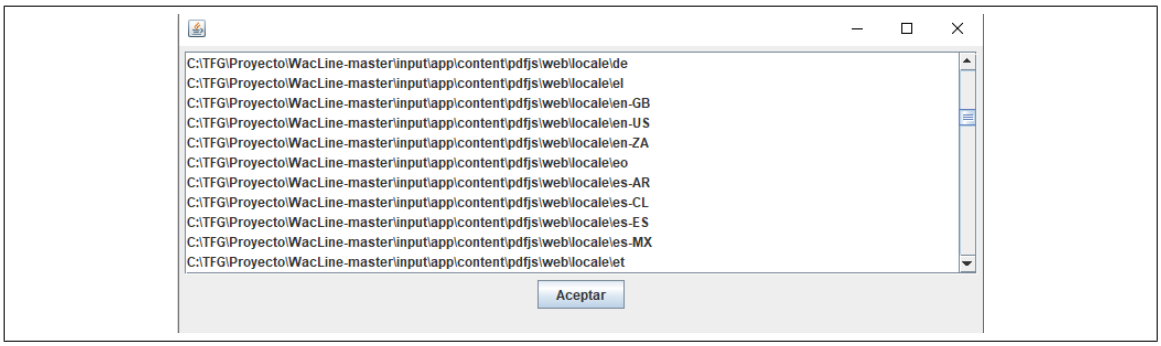

Figura 6.14: Menú de selección carpetas para procesar.

Una vez realizada esta selección, el módulo *VariantMiner* procesa el proyecto para conseguir los productos. El siguiente paso es preguntar al usuario qué productos quiere procesar, tal y como se ve en la figura 6.15 Con los productos conseguidos, el módulo *VariantMiner* utiliza el método *getFeaturesOfVariant* descrito en la sección 6.5.1 para obtener las características de los productos seleccionados. Con estas características se genera una última interfaz que pregunta al usuario por las características que desea seleccionar, además de incluir la opción de procesar las funciones (ver figura 6.16). Una vez que se han seleccionado estos parámetros se obtiene el diagrama.

Figura 6.15: Menú de selección de productos a procesar.

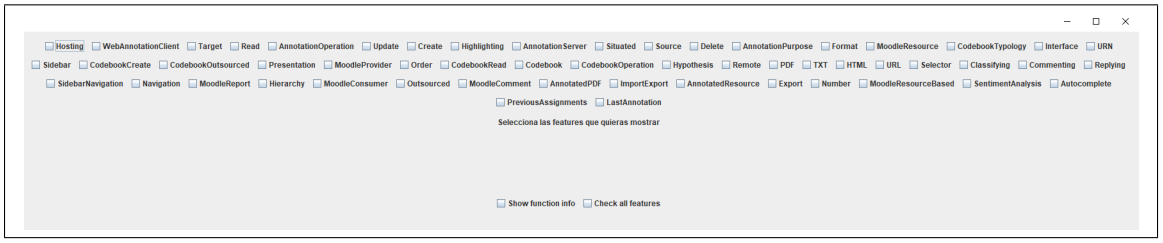

Figura 6.16: Menú de selección de características a procesar.

6.6.2. Ejemplos de diagramas

La primera ejecución (ver figura 6.17) muestra los componentes de *WeatherStation* que aparecen en los productos "Berlin","Bern","Oslo" y "Paris". Se puede ver como se generan los componentes con sus relaciones de manera correcta, demostrando que la aplicación es capaz de procesar un proyecto pequeño. En la segunda ejecución (ver figura 6.18) se puede ver como además del pro-

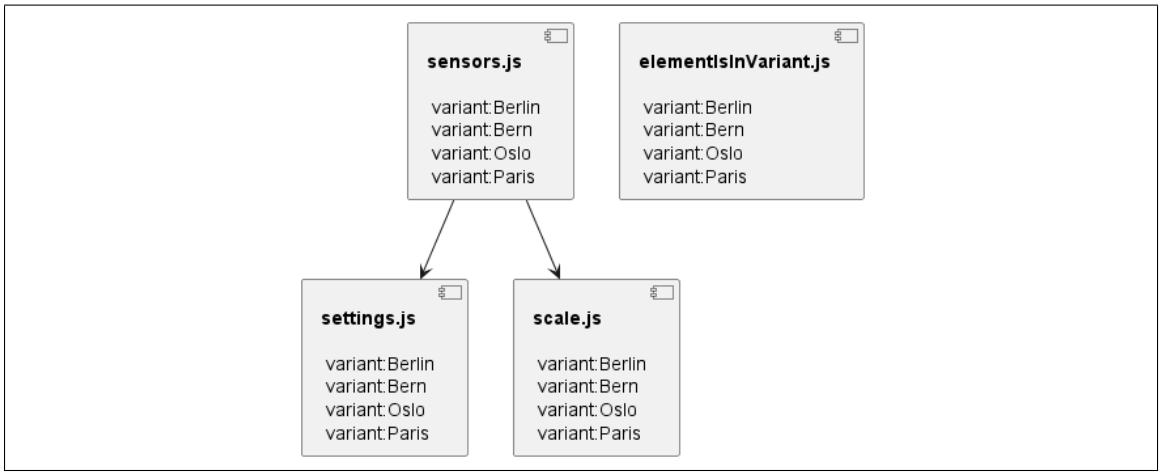

Figura 6.17: Diagrama de componentes de *WeatherStation* sin características.

ducto "Bern" se muestra información de algunas características que son usadas por los distintos componentes del proyecto *WeatherStation*.

En la tercera ejecución (ver figura 6.19) se puede ver como, además de la información de los productos y las características, se muestran las funciones. Debajo de cada función se muestra qué características modifican su comportamiento.

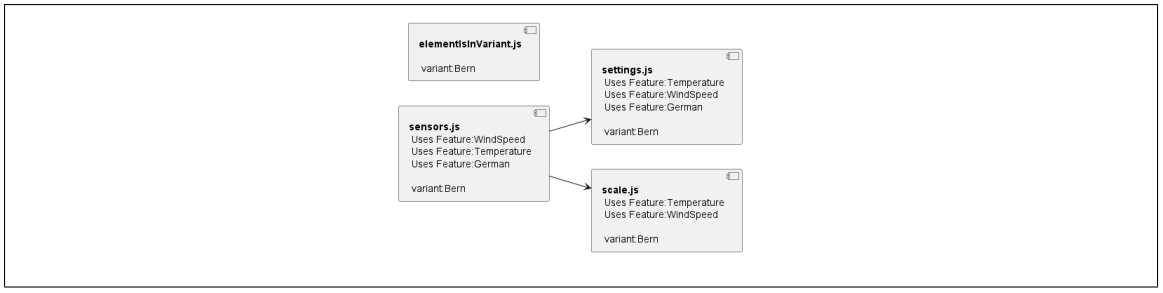

Figura 6.18: Diagrama de componentes de *WeatherStation* con características.

Por ejemplo, en la figura 6.20 se puede ver como la característica *WindSpeed* afecta a la función *applyWindSpeed*, por lo que en el diagrama aparece debajo de esta función.

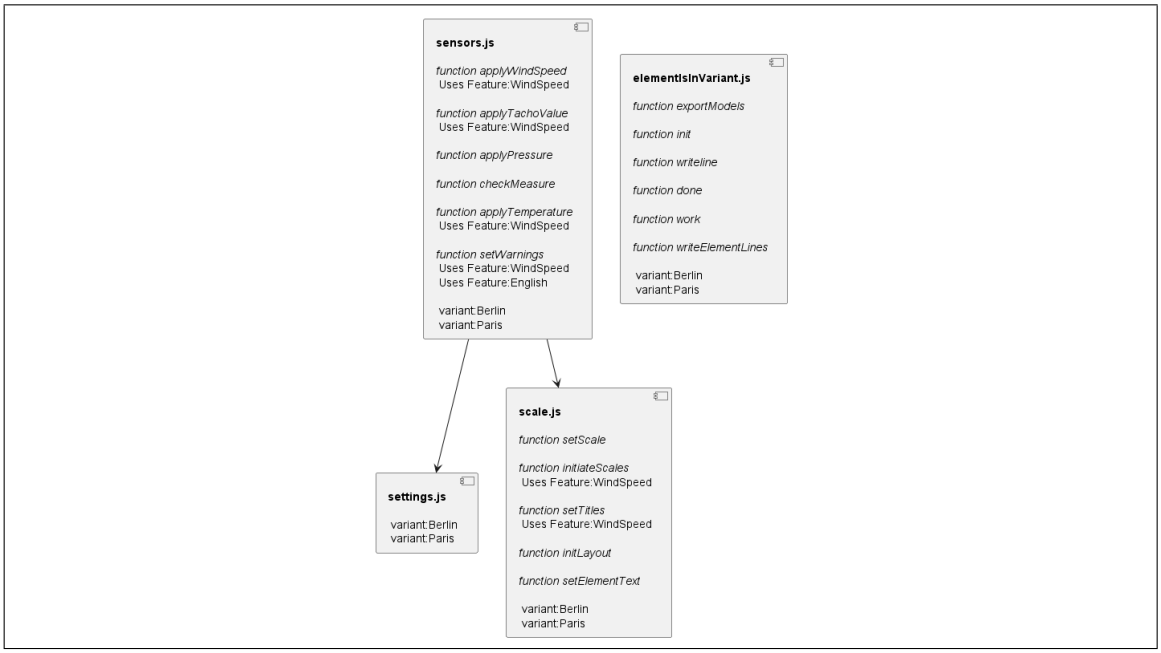

Figura 6.19: Diagrama de componentes de *WeatherStation* con características e información sobre las funciones.

```
// PVSCL:IFCOND(WindSpeed)
var windMeasure = 0;
function applyWindSpeed() {
    var measureText = document.getElementById("w_measure");
    windMeasure = measureText.value;
    var pointer = document.getElementById("w_point");

    applyTachoValue(minWind, maxWind, measureText, pointer);
    setWarnings();
    return false;
}
// PVSCL:ENDCOND
```

Figura 6.20: Ejemplo de anotación en *sensors.js*

Trás estos ejemplos, se puede ver como la aplicación funciona correctamente con un proyecto pequeño, ya que *WeatherStation* solo cuenta con cuatro componentes. Además, estos cuatro componentes están incluidos en todos los productos.

Para demostrar que también funciona con proyectos más grandes, a continuación se muestran diferentes ejecuciones en *WacLine*, el proyecto explicado en la sección 3.2. Este proyecto cuenta con más de 100 componentes y el uso de estos sí varía dependiendo de los productos utilizados.

Ejecución en WacLine

En la primera ejecución (ver figura 6.21) se puede ver como se genera el diagrama de todo el proyecto para el producto *OATS*. Este diagrama es demasiado grande como para ser comprendido correctamente. Es por este tipo de proyectos por lo que se desarrolló la opción de generar los diagramas por carpeta.

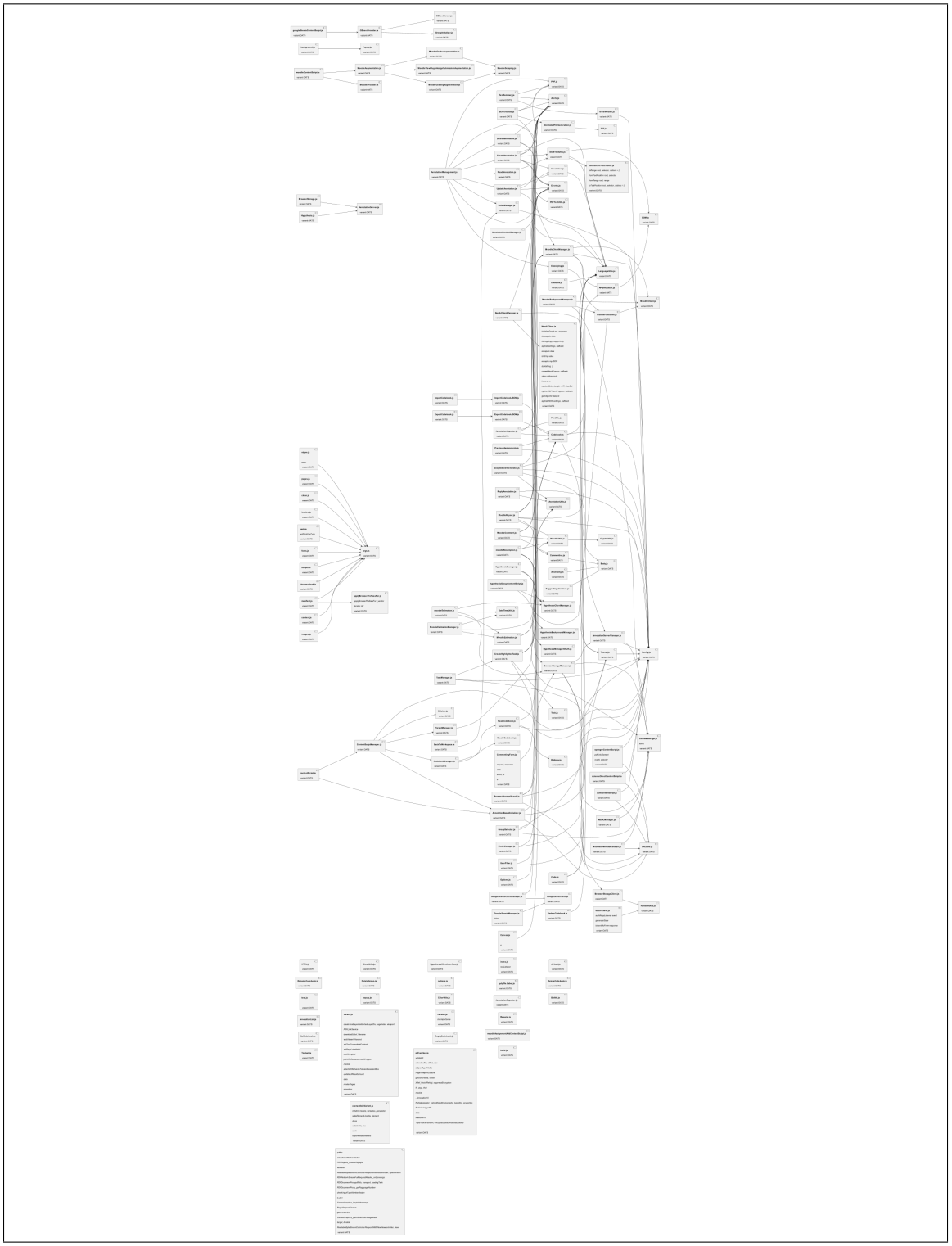

Figura 6.21: Ejemplo de diagrama de proyecto completo

En la segunda ejecución (ver figura 6.22) se puede ver como se generan las relaciones correctamente, mostrando el producto al que pertenecen y sus características. En este caso, todos los componentes están en todos los productos.

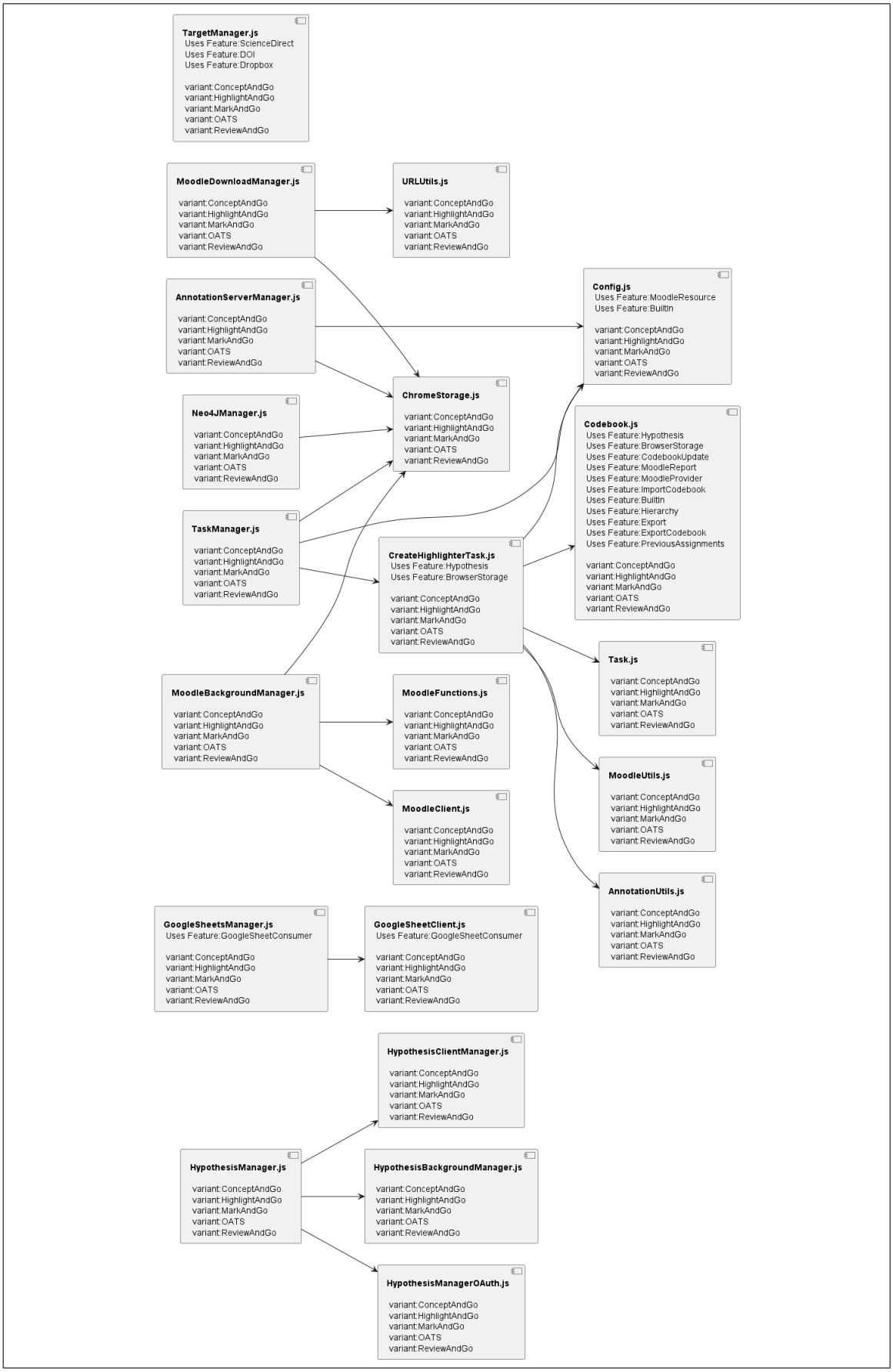

Figura 6.22: Ejemplo de diagrama de proyecto utilizando las carpetas “Moodle”, “Codebook” y “GoogleSheet”

En algunos casos, no todos los componentes pertenecen a los mismos productos. En la figura 6.23 se puede ver como algunos componentes no pertenecen a todas las variantes.

Figura 6.23: Ejemplo de diagrama de proyecto de la carpeta *Operations*

En la tercera ejecución (ver figura 6.24) se puede ver otro ejemplo que utiliza características. En este caso se ejecuta sobre los componentes encargados de trabajar con Moodle.

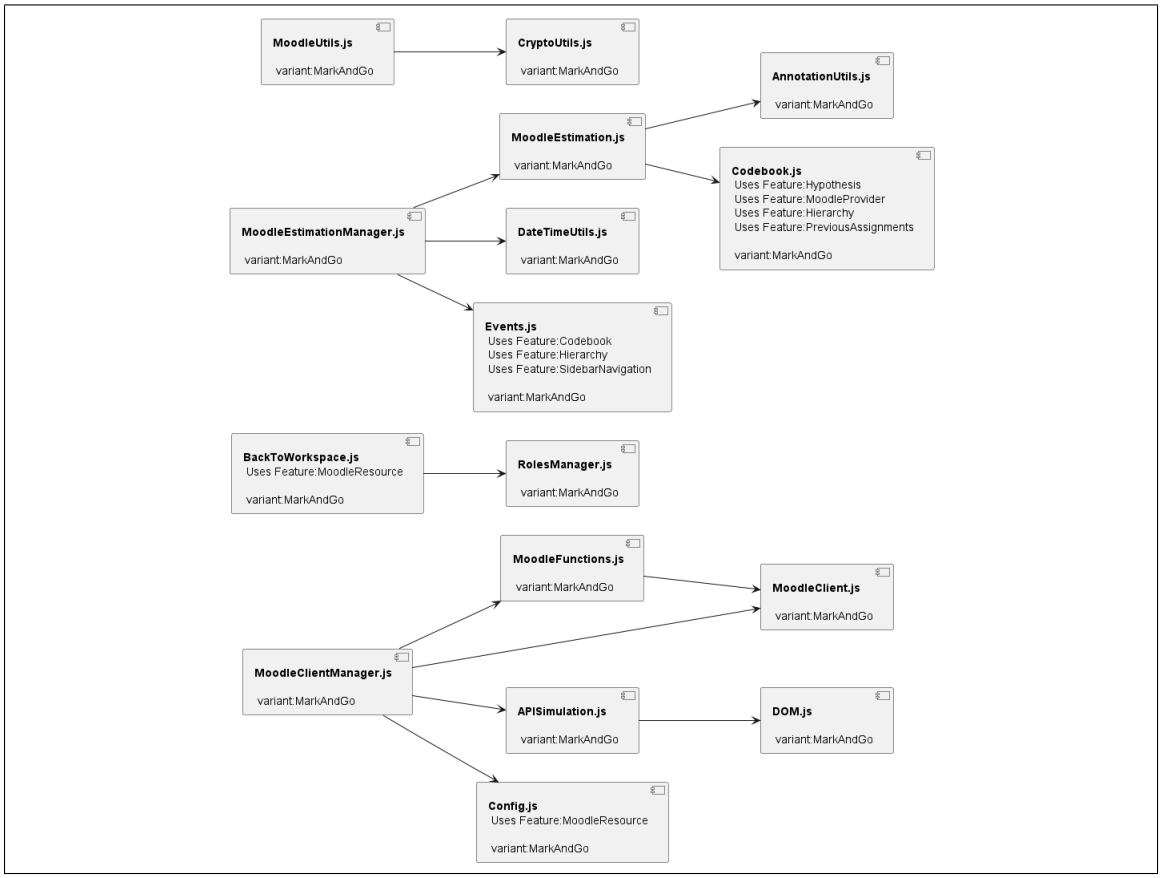

Figura 6.24: Ejemplo de diagrama de proyecto de la carpeta *Moodle*

Tras estas ejecuciones se demuestra que la aplicación es capaz de procesar proyectos más grandes. En el caso de *WacLine*, seleccionando todas las carpetas, productos y características, se procesa sin problemas, dando como resultado la figura 6.21. Esta ejecución tarda algo más de un minuto, tal y como se ve en la figura 6.25.

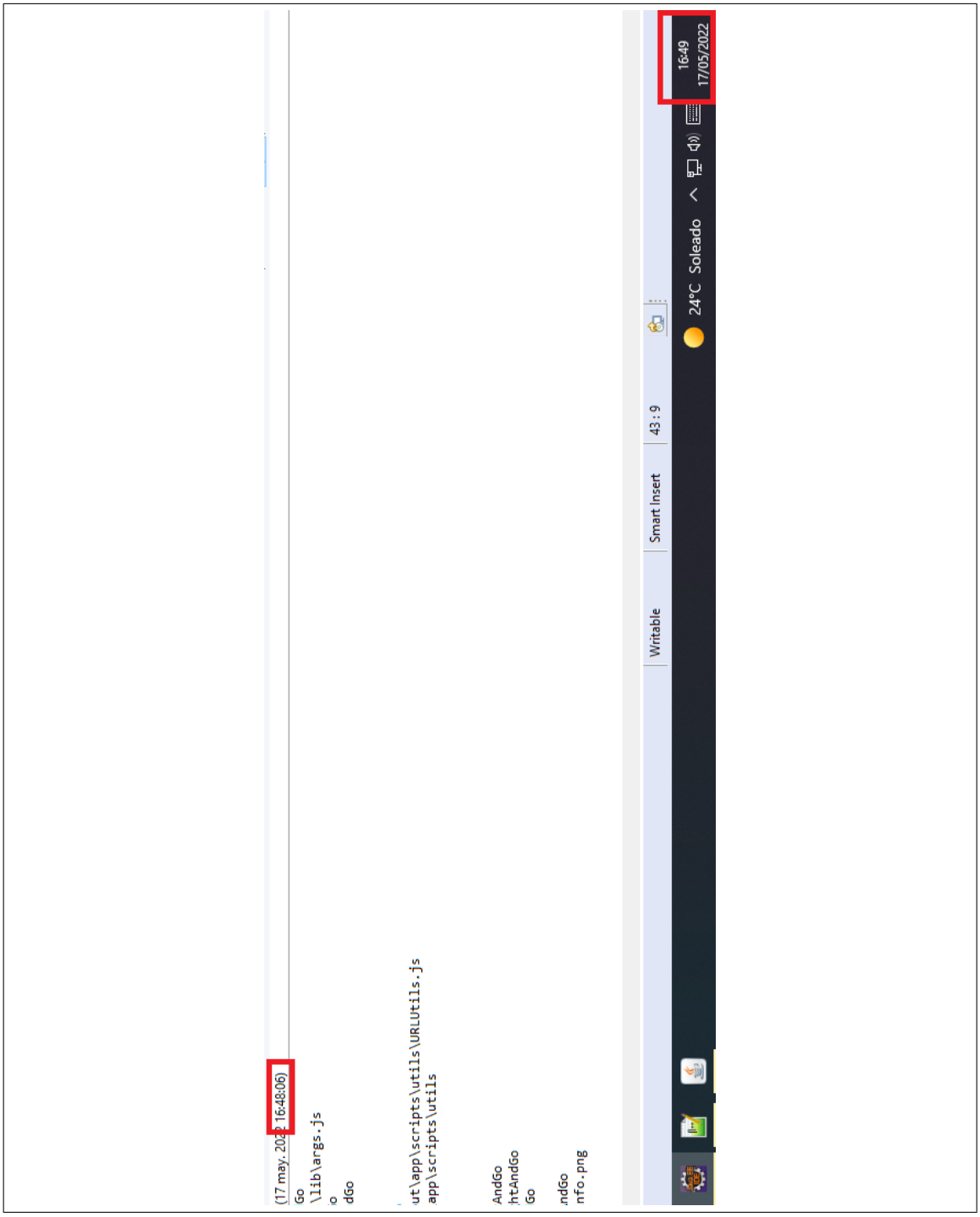

Figura 6.25: Tiempo de ejecución para generar la figura 6.21

CAPÍTULO 7

Evolución de la aplicación InsideSPL

En este capítulo se explica el proceso de evolución de la herramienta *InsideSPL* desarrollada en este trabajo de fin de grado. Para ello, se explicará la estructura a modificar, los nuevos requisitos de la aplicación integrada, el diseño de la solución y el resultado final.

7.1. Estructura del proyecto

InsideSPL [1] es un módulo de visualización de datos relacionados con la implementación de líneas de producto software (SPL). Estos datos son obtenidos tras un proceso de minado realizado por el módulo *SPLMiner*, tal y como se explica en la sección 3.3. *SPLMiner* se encarga de generar un archivo SQL tras procesar el código de una SPL. Este archivo SQL es procesado por *InsideSPL* para visualizar los diferentes datos que se recogen en el archivo. *InsideSPL* está implementado en *Spring* [21], un *framework* de *Java* para desarrollar, entre otras, aplicaciones web. Para ello, *InsideSPL* cuenta con diferentes módulos, a continuación se listan los importantes para la integración.

- **insideSPL**: Este módulo es el encargado de inicializar la aplicación.
- **BussinessLogic**: En este módulo se encuentran las clases que se encargan de realizar las diferentes acciones que tengan que ver con la lógica de negocio.
- **controller**: En este módulo se encuentran todos los controladores que se encargan de la parte del servidor de la aplicación, hacen llamadas a las clases que se encuentran en los módulos de lógica de negocio y cargan las diferentes vistas *HTML*.
- **domain**: En este módulo se encuentran todas las clases que serán utilizadas como objetos en las vistas, por ejemplo, la clase *Feature*. Esta clase cuenta con diferentes atributos, como el nombre, la característica de la que dependen (si existe) o el *FeatureModel* al que pertenecen.
- **resources**: En este módulo se encuentran diferentes ficheros de configuración en los que se definen los mensajes predefinidos de la aplicación. Es necesario definir estos mensajes predefinidos para que la aplicación cambie los valores de los textos en función del idioma seleccionado. Actualmente, *InsideSPL* está implementado para euskera, castellano e inglés.
- **views**: En este módulo se almacenan las diferentes vistas que son cargadas por los controladores.

Actualmente, *InsideSPL* muestra las características definidas en la *SPL*, los productos que se pueden derivar, que características participan en la implementación de dichos productos y la posibilidad de comparar los productos a nivel de sus características.

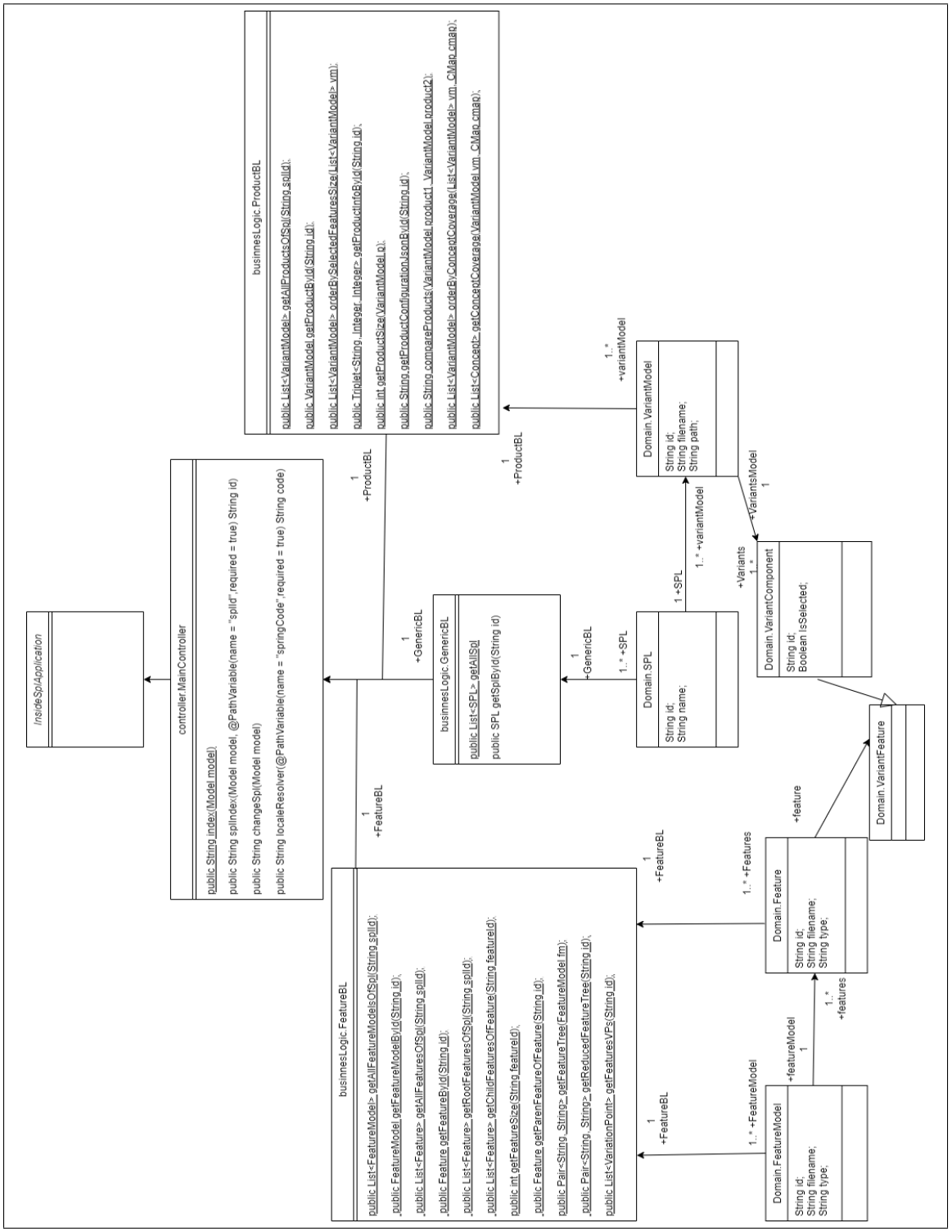

Figura 7.1: Diagrama de clases del *InsideSPL* sobre las clases importantes para la integración

En la figura 7.1 se pueden ver las diferentes clases que son interesantes para la integración, *InsideSPLApplication* es la encargada de inicializar la aplicación y hacer la llamada al controlador *MainController*, que carga la página principal, para esto utiliza las diferentes clases de la lógica de negocio. Estas, a su vez, utilizan las clases de *Domain* para cargar los datos. Para entender mejor la importancia y funcionamiento de estos módulos, a continuación se explica un ejemplo que muestra datos sobre el producto *MarkGo* derivado de la *SPL WacLine*. Uno de los objetivos del trabajo de fin de grado que se presenta en esta memoria es añadir más funcionalidades a *InsideSPL*. La funcionalidad añadida es la visualización de datos relacionados con la arquitectura de la SPL. Esta funcionalidad es la que se ha descrito en el capítulo 6.

Para inicializar la aplicación, se ejecuta la clase que se encuentra localizada en el módulo *InsideSPL*, ésta carga la página inicial (ver figura 7.2), que es una vista almacenada en el módulo *views*.

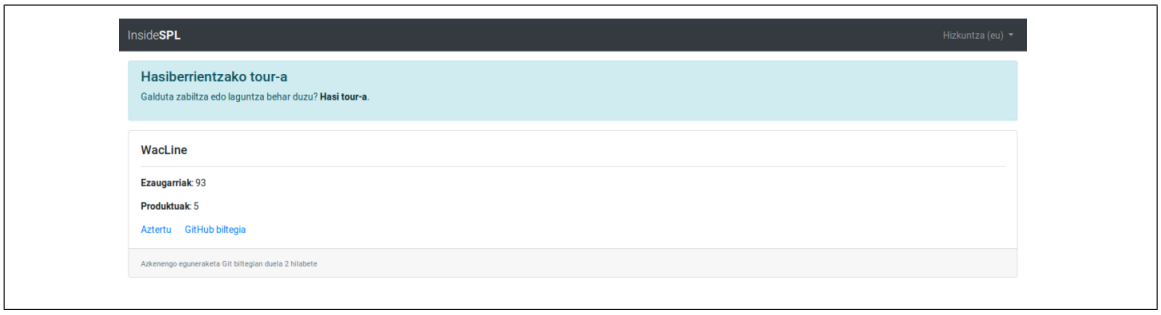

Figura 7.2: Menu principal de *InsideSPL*

A continuación, el usuario decide ver la información sobre un producto, hace clic en *aztertu* y selecciona el producto *MarkAndGo.vdm*. Esta petición pasa por el controlador de los productos, pasándole el *id* del producto a una clase del módulo *BusinessLogic* que se encarga de devolverle el producto en función de ese *id*. Ese producto es un objeto de la clase homónima definida en el módulo *domain*. Con ese producto ya cargado, el controlador carga la vista pasándole el producto como parámetro mediante un método *POST*.

Como se puede ver en la figura 7.3, la vista carga y muestra la información del producto, junto a sus características. También se puede ver algunos mensajes como *Konfigurazioa* o *Produktu ikuspegia* que están en euskera porque es el lenguaje seleccionado. Si se cambia el lenguaje, estos mensajes se actualizan, ya que han sido definidos en los archivos de configuración del módulo *resources*.

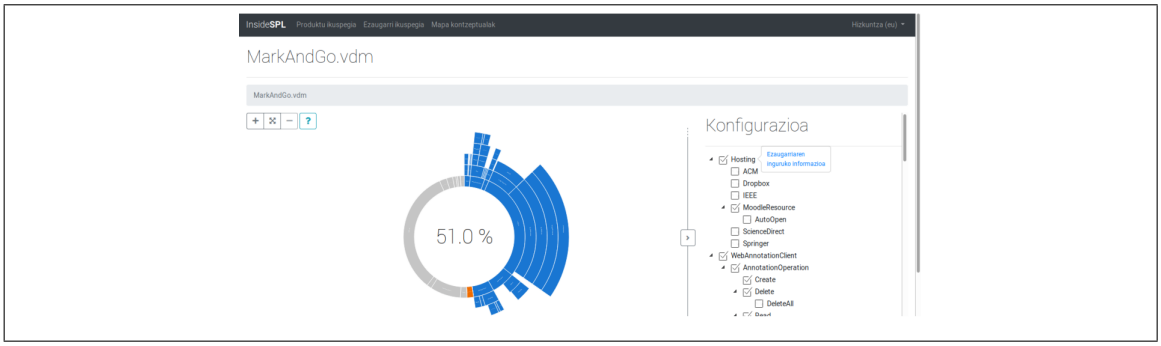

Figura 7.3: Menú del producto *MarkAndGo*

7.2. Requisitos para ampliar InsideSPL

A continuación se muestran los requisitos sobre la aplicación que se han definido para la ampliación de *InsideSPL*.

- **Mostrar un listado de todos los módulos que participan en el diagrama de la arquitectura implementada en la SPL**: Para ello, el usuario escogerá un nivel (desde el 1 hasta el máximo) de profundidad de módulos que quiere escoger. Si este usuario escoge el nivel 1, solo podrá escoger entre los módulos que cuelgan directamente desde la raíz del proyecto. Si escoge el nivel 2, podrá escoger entre los módulos que cuelgan directamente de la raíz y también entre los que están directamente dentro de estos (no aparecerán las carpetas que están incluidas dentro de las del segundo nivel).

- **Una vez seleccionado un módulo, mostrar un listado de variantes o productos implementadas con dicho módulo**: Tras seleccionar los módulos que se quieren procesar, dar opción de escoger únicamente los productos que sean implementados con dichos módulos. Por ejemplo, si hay una carpeta que implementa exclusivamente las funciones de una característica, tras seleccionar únicamente esa carpeta solo serán seleccionables aquellos productos que tengan dicha característica.

- **Mostrar el diagrama de componentes de varios productos o variantes**: Aparecerán en el diagrama todas las clases que sean utilizadas por al menos un producto. Aquellas clases que sean utilizadas por todos los productos seleccionados aparecerán en gris y aquellas que no aparecerán en blanco.

Tal y como se explica en la sección 7.6, estos no son todos los requisitos propuestos por el cliente, pero tras una valoración de viabilidad, se decidió implementar estos requisitos, ya que se encontraban dentro del alcance.

El análisis de los requisitos ha dado lugar a los diferentes diagramas de secuencia que se muestran la sección 7.3

7.3. Diseño de la integración con InsideSPL

A continuación se detalla cómo se ha diseñado la solución a implementar. Para ello, se muestran el diagrama de clases del nuevo diseño, junto a un listado de funcionalidades. Estas funcionalidades vienen acompañadas de diferentes diagramas de secuencia para explicar su caso de uso. De esta manera, será más fácil comprender las diferentes funcionalidades que se han añadido con la expansión de la herramienta.

Como se puede ver en la figura 7.4, se han añadido algunas clases al diagrama original. Respecto al *Domain*, donde se encuentran las clases que son utilizadas como objetos en las vistas, se ha añadido la clase *Folder*, en esta clase se almacenan la información sobre los módulos que se usarán en el diagrama, cada uno con una lista de los ficheros que se encuentran dentro. Se han

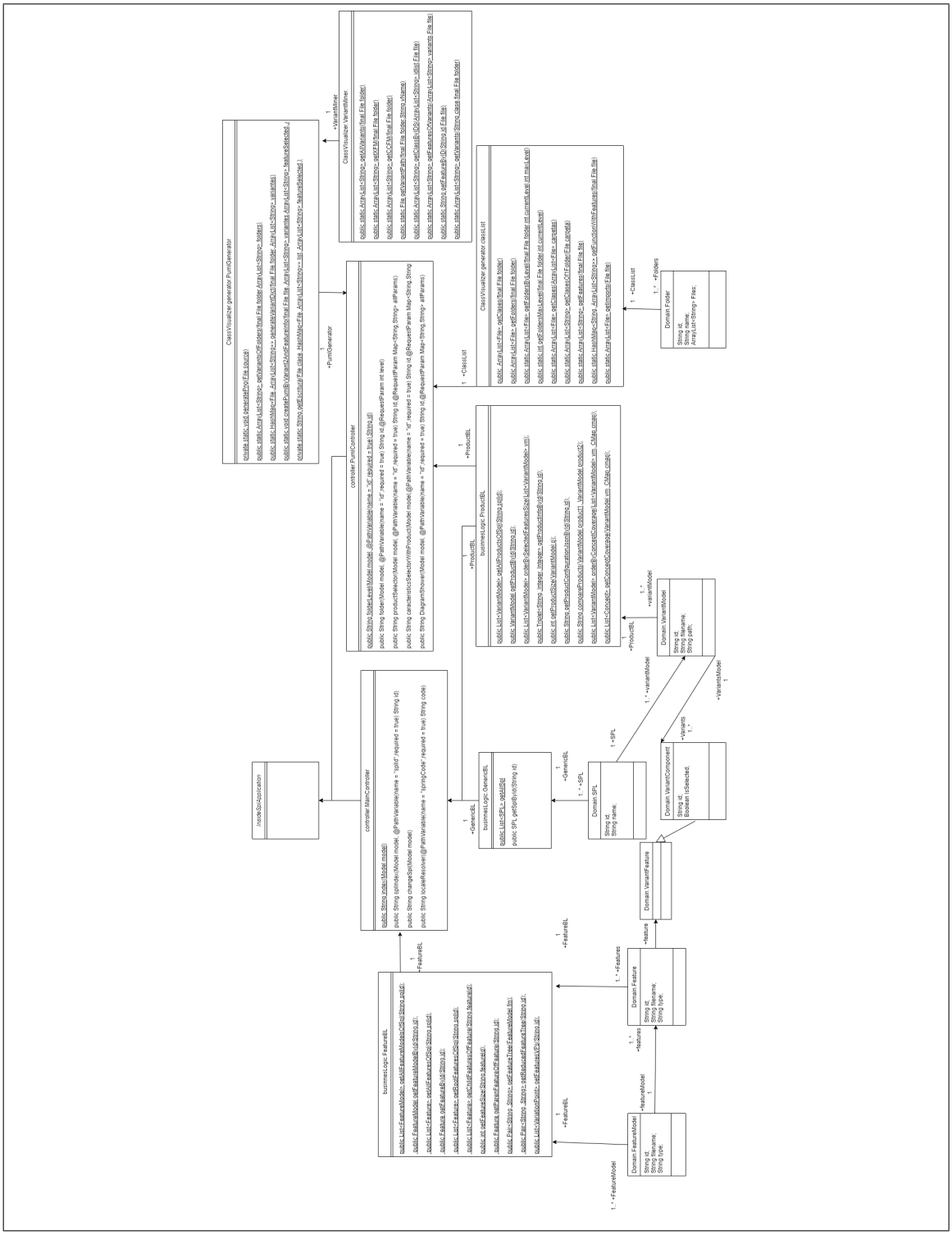

Figura 7.4: Actualización del diagrama de clases de la figura 7.1 con las nuevas clases implementadas

modificado los cuatro módulos de *InsideSPL* que han sido descritos en la sección 7.1, a continuación se describen esos cambios

A la lógica de negocio se han añadido tres clases, *ClassList*, *VariantMiner* y *PumlGenerator*. *ClassList* se encarga del procesamiento de los módulos, esta clase cuenta con los métodos necesarios para obtener toda la información sobre los ficheros, como las características o los productos o

las relaciones con otros ficheros. *VariantMiner* se encarga de implementar algunas funciones que no estaban implementadas en la aplicación original, por ejemplo, una función para conseguir los productos que utilizan un grupo específico de módulos. *PumlGenerator* implementa los métodos necesarios para generar el diagrama que se mostrara al final de la ejecución.

Respecto al módulo de los controladores, se añade un nuevo controlador, *PumlController*. Este controlador cuenta con los métodos necesarios para procesar las peticiones de cada una de las vistas. Estos métodos reciben una serie de parámetros, bien por la *URL* parametrizada o por métodos *POST*. Con esos parámetros utilizan los diferentes métodos de la lógica de negocio y generan un modelo. Este modelo cuenta con diferentes objetos del módulo *Domain*. Una vez tiene el modelo generado, carga la vista correspondiente, pasándole el modelo. Estas vistas utilizan el modelo para mostrar por pantalla diferente información. Por ejemplo, en la vista de selección de productos, tal y como se explica en la sección 7.3.1.2, el controlador genera un modelo que contiene los productos, los ficheros seleccionados y sus componentes. Esta vista utiliza ese modelo para mostrar toda la información.

En el módulo resources se modifican los ficheros de configuración para añadir los mensajes necesarios para la nueva implementación. Este módulo tiene tres ficheros, *messages_en*, *messages_es* y *messages_eu*. Cada uno de estos ficheros corresponde con un idioma, inglés, español y euskera respectivamente. Para mantener la posibilidad de cambiar el idioma, hay que añadir en estos ficheros los diferentes textos que aparecen en las vistas y utilizar la etiqueta *<spring:message>* en los ficheros *HTML*, que carga estos textos.

Además, se añaden al módulo "views" las nuevas vistas, tal y como se explica en la sección 7.3.1.5. Estas vistas son las encargadas de mostrarle al usuario las diferentes opciones para poder generar el diagrama. Para ello, cargan el modelo que reciben del controlador con la información necesaria y le pasan al controlador las diferentes opciones seleccionadas por el usuario, mediante un método *POST*.

7.3.1. Funcionalidades del nuevo diseño

A continuación se describen las nuevas funcionalidades que se han diseñado para la aplicación, siguiendo el hilo de los nuevos requisitos descritos en la sección 7.2. Estas funcionalidades se encuentran ordenadas por orden de uso, es decir, cada una de ellas es dependiente de la anterior. Al estar ordenadas, es más fácil entender su funcionamiento.

7.3.1.1. Selección de los módulos que participan en el diagrama

Para poder ilustrarse mejor este proceso, se ha generado el diagrama de secuencia del proceso de selección de carpetas (ver figura 7.5). En este diagrama se puede ver cómo el usuario hace la petición al servidor de cargar la vista de selección de nivel. El servidor recibe la URL y manda la petición al controlador. Este hace una llamada a la lógica de negocio para cargar el nivel máximo de carpetas disponibles y genera un modelo, este modelo contiene el nivel máximo que podrá seleccionar el usuario. Con ese modelo creado, hace una llamada al servidor para que este cargue la vista de selección de nivel. Esta vista cuenta con un desplegable de los niveles disponibles y utiliza la información del modelo para generar el desplegable.

Después, el usuario selecciona el nivel que desea y le da a un botón, este botón forma parte

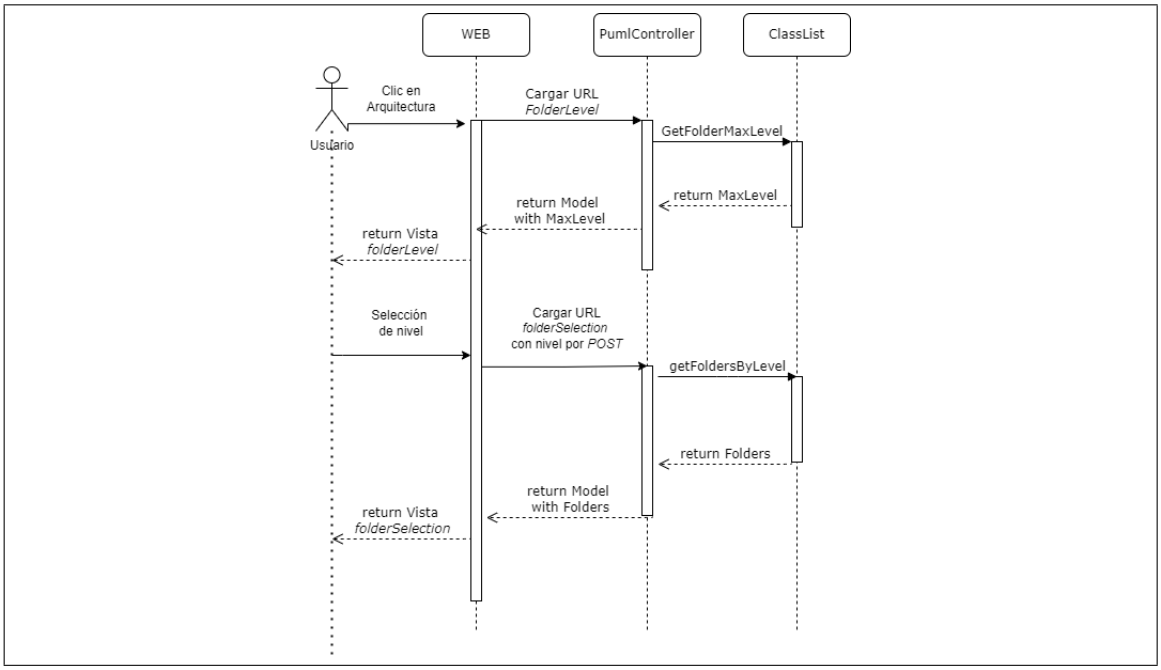

Figura 7.5: Diagrama de secuencia del proceso de selección de carpetas

de un formulario *HTML* que hace una llamada a cargar la URL de selección de carpetas, pasando por un método *POST* el nivel seleccionado. El controlador recibe el parámetro de nivel y llama a la lógica de negocio para conseguir las carpetas que están dentro de este nivel. Una vez más genera un modelo con la lista de carpetas y se lo devuelve a la vista, que le da al usuario la lista de carpetas para que seleccione cuáles quiere procesar.

7.3.1.2. Selección de productos basándose en los módulos seleccionados

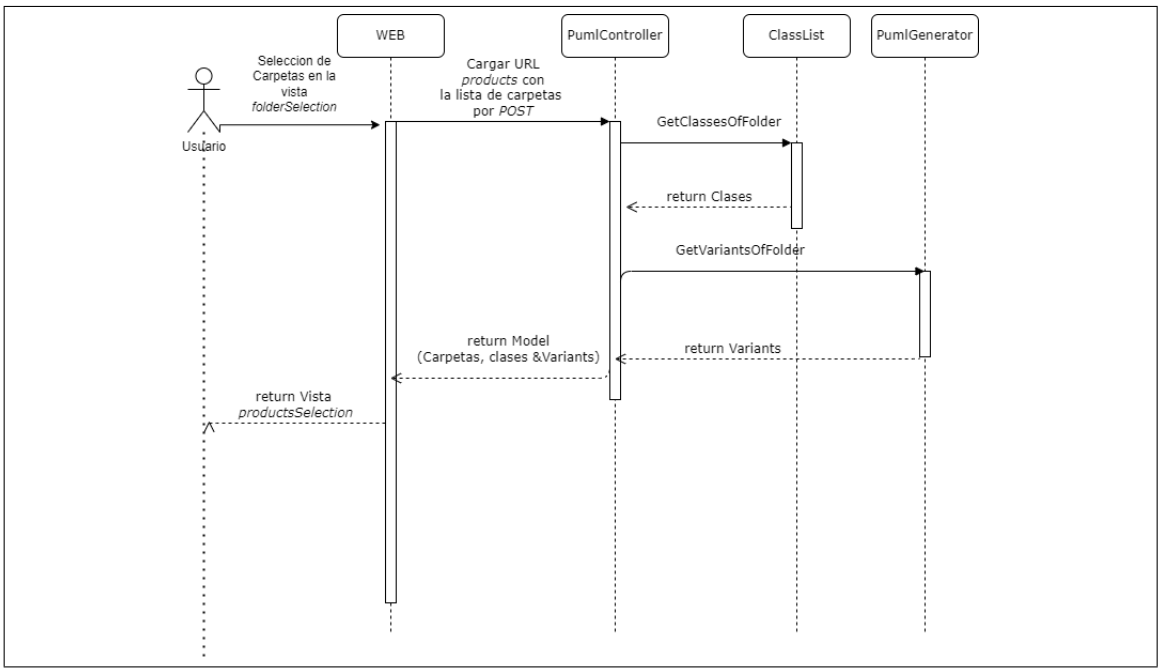

Figura 7.6: Diagrama de secuencia del proceso de selección de productos

Para ilustrar este proceso, se ha generado el diagrama de secuencia del proceso de selección de productos (ver figura 7.5). En este diagrama se puede ver como el usuario, tras seleccionar las carpetas, pasa a cargar la vista de los productos. Antes de que esta vista cargue, el servidor manda la petición al controlador pasándole la lista de carpetas seleccionada por el usuario. Con esta lista de carpetas, el controlador utiliza la lógica de negocio para generar la lista de productos que después le pasa a través de un modelo a la vista.

Con este modelo la vista genera un desplegable de productos para que el usuario seleccione los que desee procesar, además, en el modelo también van incluidas las carpetas seleccionadas y sus ficheros. De esta manera, en la vista también aparecen, a modo informativo, las carpetas seleccionadas anteriormente y los ficheros que las componen.

7.3.1.3. Selección de características basadas en los productos seleccionados

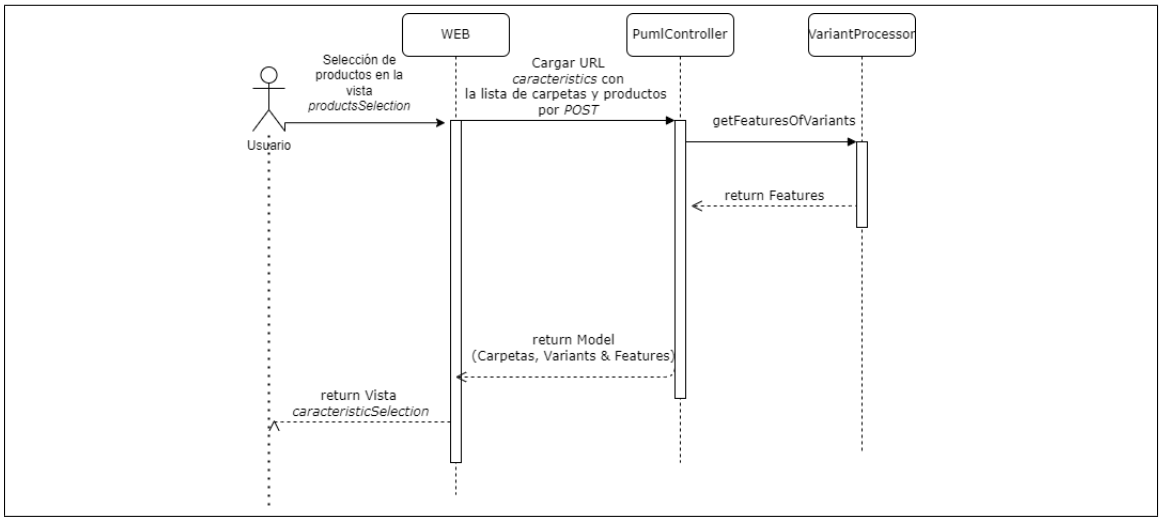

Figura 7.7: Diagrama de secuencia del proceso de selección de características

Para ilustrar este proceso, se ha generado el diagrama de secuencia del proceso de selección de características (ver figura 7.7). Como se puede ver en el diagrama, esta funcionalidad está disponible para el usuario tras seleccionar los productos que desea procesar. Una vez ha hecha esa selección, el servidor hace una petición al controlador para cargar la vista de selección de productos. En ese momento, el controlador hace una llamada a la lógica de negocio para recibir la lista de características que son utilizadas en los productos seleccionados. Con esta lista de características, carga la vista de selección pasándole, a través de un modelo, la lista de carpetas seleccionadas, productos seleccionados y características disponibles para seleccionar.

Con este modelo, la vista genera un desplegable de características seleccionables, que el usuario escogerá para poder generar el diagrama según las necesidades del cliente.

7.3.1.4. Generación y muestra del diagrama

Para ilustrar este proceso, se ha generado el diagrama de secuencia del proceso de selección de productos (ver figura 7.8). Como se ve en la figura, el usuario selecciona las características que desea procesar y el servidor hace una petición al controlador para cargar la vista que muestra el diagrama.

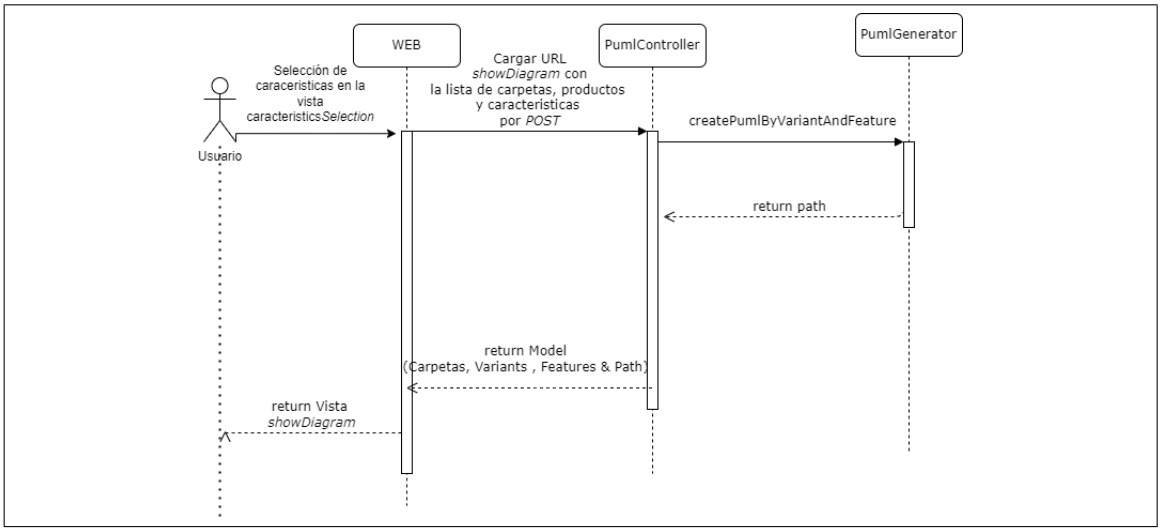

Figura 7.8: Diagrama de secuencia del proceso de generación del diagrama

Cuando eso sucede, el controlador, que ha recibido mediante un método *POST*, hace una llamada a la lógica de negocio para generar el diagrama. En esta llamada pasa como parámetros las carpetas, los productos y las características seleccionadas por el usuario. Esta llamada genera una imagen y devuelve al controlador el *path* de la imagen. Con este *path*, el controlador genera un modelo que contiene todas las selecciones del usuario y el *path* de la imagen. Así, cuando carga la vista, esta utiliza el modelo para mostrar la información del diagrama junto al mismo.

7.3.1.5. Diagrama de vistas

A continuación (ver figura 7.9) se muestra un diagrama de las diferentes vistas que se han añadido al módulo *views* de *InsideSPL*. En este diagrama, se explica el proceso que siguen las vistas para generar el diagrama final.

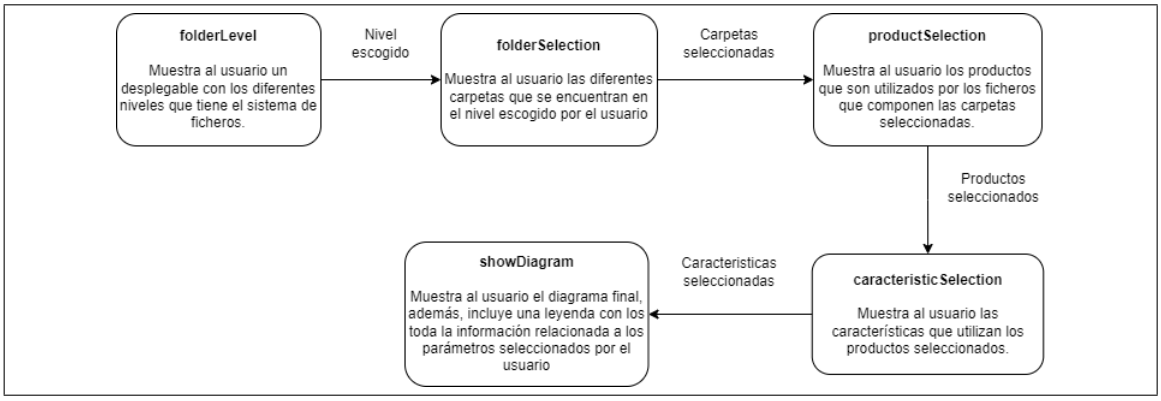

Figura 7.9: Diagrama de secuencia del uso de las vistas

En el diagrama se puede ver cómo cada vista es directamente dependiente de la anterior. Las dos primeras vistas, *folderLevel* y *folderSelection* responden a la funcionalidad de selección de los módulos que participan en el diagrama descrita en la sección 7.3.1.1. La vista *productSelection* responde a la funcionalidad de selección de productos según los módulos escogidos descrita en la sección 7.3.1, ya que en esta vista se seleccionan los productos. La vista *caracteristicSelection* responde a la funcionalidad de selección de características descrita en la sección 7.3.1.3, en esta

vista, se pueden seleccionar las diferentes características que formaran parte del diagrama final. Finalmente, la vista *showDiagram* responde a la funcionalidad de muestra del diagrama descrita en la sección 7.3.1.4, ya que muestra el diagrama final, añadiendo una leyenda con todos los datos de las selecciones del usuario.

7.3.1.6. Funciones implementadas

Para poder adaptarse a las funcionalidades de la aplicación, se implementan las siguientes funciones:

- **getFoldersByLevel (final File folder,int currentLevel,int maxLevel)**: Este método recibe como parámetros la carpeta raíz del proyecto, el nivel donde se quiere empezar y el nivel que ha seleccionado el usuario. En todas las llamadas, el parámetro *currentLevel* es 1, pero puede usarse para conseguir las carpetas de un solo nivel o a partir de un nivel. *currentLevel* debe ser menor que *maxLevel* o devolverá una lista vacía.
- **getFoldersMaxLevel (final File folder,int currentLevel)**: Este método recibe como parámetro la carpeta raíz del proyecto y el nivel por el que se quiere comenzar. Como en el método anterior, *currentLevel* tiene en todas las llamadas valor 0. En este caso este valor se usa como parámetro de entrada porque el método tiene un carácter recursivo. Llamar a este método con un valor diferente a 0 devolverá como tamaño máximo un nivel que no es el correcto.
- **getVariantsOfFolders (final File folder,ArrayList<String>folders)**: Este método recibe como parámetros la carpeta raíz y la lista de las carpetas seleccionadas. El método recorre la lista añadiendo a una lista interna los diferentes productos en los que participan las clases de estas carpetas. Cuando se acaba la lista de carpetas o la lista interna es del mismo tamaño que la lista de productos completa, para. Después, devuelve la lista interna. Si la lista de carpetas seleccionadas está vacía o es *null*, devuelve una lista vacía de productos.

7.4. Aspectos de la implementación

Módulo	Clase	Lineas de Código
Domain	FeatureModel	66
	Feature	123
	VariantFeature	22
	VariantComponent	43
	SPL	91
	VariantModel	69
	Folder	**45**
BusinessLogic	FeatureBL	234
	GenericBL	70
	ProductBL	144
	ClassList	**254**
	PumlGenerator	**250**
	VariantMiner	**415**
Controllers	MainController	129
	PumlController	**210**
Views	**folderLevel**	**16**
	folderSelection	**13**
	productSelection	**64**
	caracteristicSelection	**20**
	showDiagram	**35**

Tabla 7.1: Tabla de líneas de código por clase utilizada en la integración

A continuación (ver tabla 7.1) se muestra el tamaño de los diferentes módulos de *InsideSPL*, mostrando las líneas de código que tiene cada clase, y el tamaño de la parte implementada para la integración. Para facilitar la comprensión, solo se muestran las clases que aparecen en los diagramas de clase 7.1 y 7.4, además de las nuevas vistas. Este desglose se hace por módulos.

Con este desglose se quiere mostrar la cantidad de código que se ha tenido que estudiar, además del trabajo realizado para la integración. Las clases nuevas añadidas aparecen en negrita, las demás son clases que ya estaban implementadas. Como se puede ver, se han tenido que estudiar 991 líneas de código y se han implementado 1322 líneas. Además, la parte de estudio sólo corresponde al estudio relacionado con diseño de la integración, ya que realmente se han tenido que estudiar el proyecto entero, que cuenta con 54 clases.

7.5. Interfaz de la nueva aplicación

A continuación se muestran las capturas de la interfaz del proyecto, haciendo referencia a las diferentes funcionalidades y vistas descritas en la sección 7.3.1.

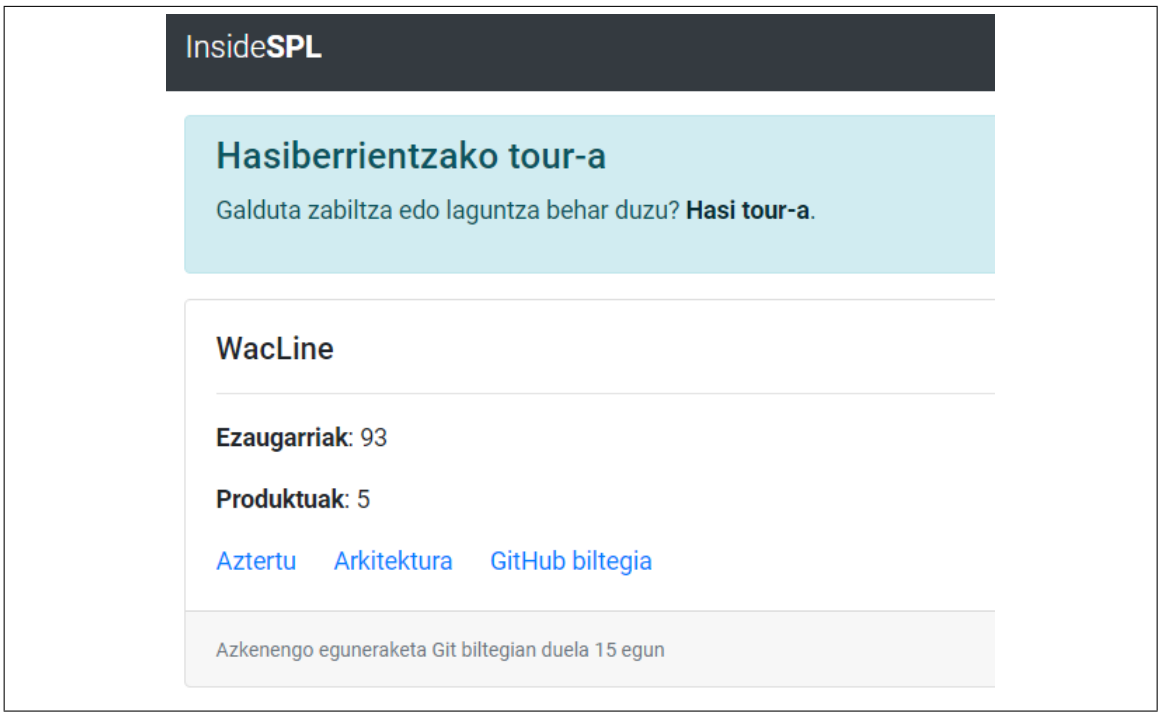

Figura 7.10: Menú inicial modificado

En la figura 7.10 se puede ver cómo tras inicializar la aplicación se muestra el menú inicial, en este menú existe ahora una nueva opción para generar el diagrama. Esta vista no corresponde a ninguna de las mencionadas en la sección 7.3.1.5, ya que es original de *InsideSPL*. Unicamente se ha modificado para añadir un link nuevo que se llame *Arquitectura*.

7.5.1. Selección de los módulos

Tal y como se explica en la sección 7.3.1, la primera funcionalidad que se le ofrece al usuario tras hacer clic en la opción de *Arquitectura* es la de la selección de nivel. La figura 7.11 se corres-

Figura 7.11: Menú de selección de nivel

ponde con la vista *folderLevel*. En ella, aparece un desplegable con los diferentes niveles que el usuario puede seleccionar. Estos niveles van desde el 1 (carpetas que están en la raíz) hasta el valor que devuelva la función *getFoldersMaxLevel*.

Tras seleccionar el nivel, se pasa a la vista *folderSelection* (ver sección 7.3.1). Antes de cargar esta vista, el controlador ha generado una lista de carpetas (ver figura 7.12). En este caso, el nivel seleccionado es el 3. Como se puede ver, no hay ninguna carpeta con más de 3 niveles de profundidad desde la raíz.

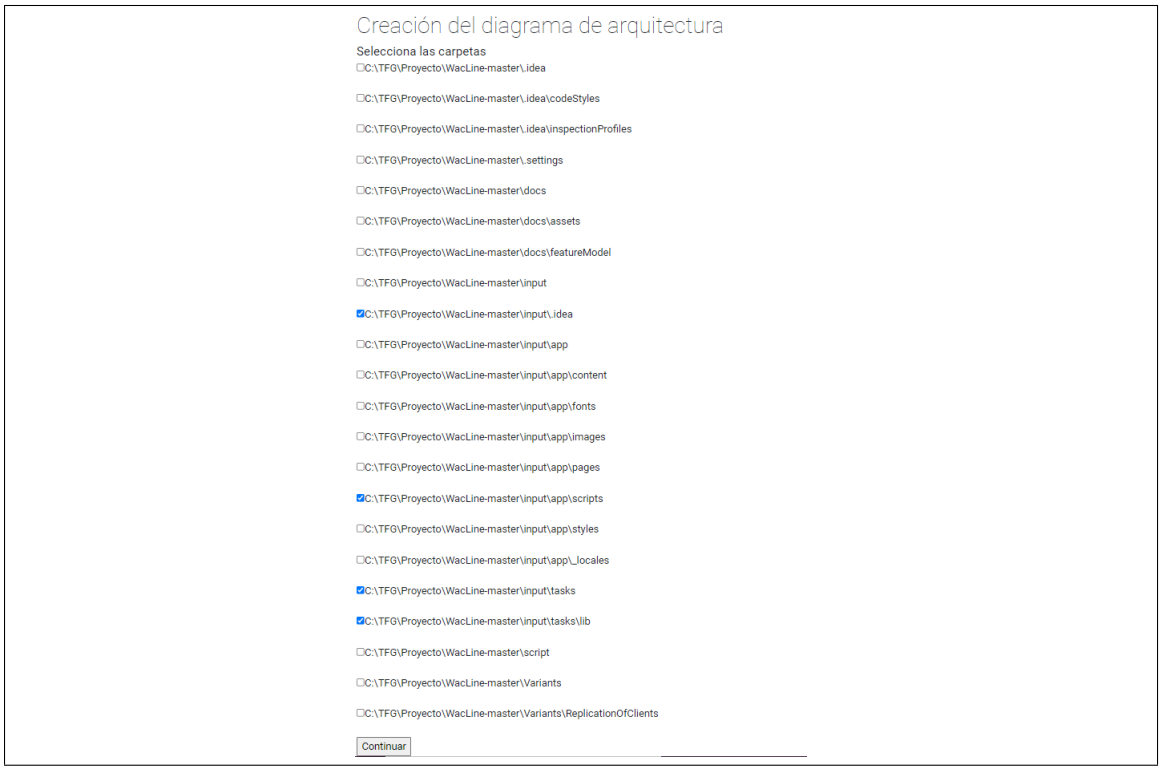

Figura 7.12: Menú de selección de carpetas con nivel 2

7.5.2. Selección de los productos en función de los módulos escogidos

Tras seleccionar los módulos deseados, como se explica en la sección 7.3.1.2, el controlador se encarga de llamar a la función *getVariantsOfFolders* para devolver los productos que utilizan estos módulos. Tras generar la lista de los productos, el controlador carga la vista *productSelection* (ver sección 7.3.1.5), pasándole a esta vista un modelo que contiene las carpetas seleccionadas, los ficheros que compone cada carpeta y los productos disponibles.

Figura 7.13: Menú de selección de productos

En la figura 7.13 se pueden ver los diferentes productos disponibles tras escoger los módulos que aparecen seleccionados en la figura 7.12. Se puede ver cómo también aparece una lista informativa de las carpetas que forman la selección. Además, debajo de cada carpeta hay un botón que permite mostrar u ocultar los ficheros que contiene cada carpeta (ver figura **??**.

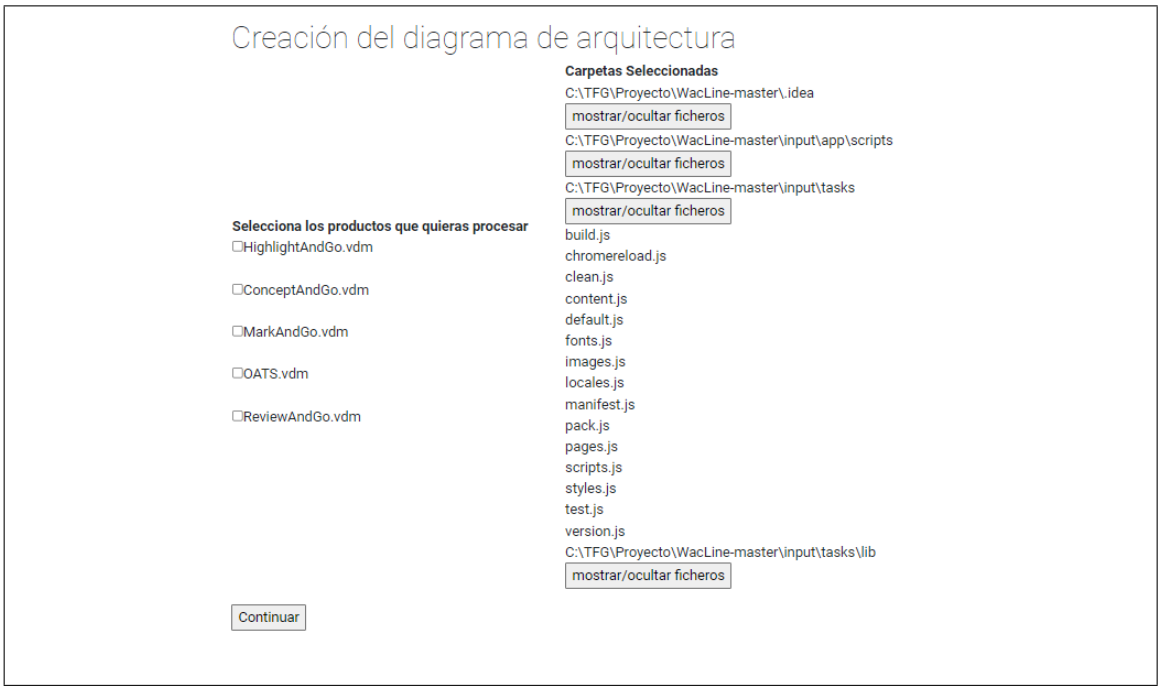

Figura 7.14: Menú de selección de productos con lista de ficheros expandida

7.5.3. Selección de las características basadas en los productos seleccionados

Una vez el usuario selecciona los productos que quiera usar, el servidor pasa mediante un *POST* los parámetros al controlador y este se encarga de generar el modelo con los datos necesarios para cargar la vista de selección de características. Esta vista se corresponde con la vista *caracteristicSelection* descrito en la sección 7.3.1.5. Como se puede ver en la figura 7.15 en este menú se pueden escoger las diferentes características que utilizan los productos seleccionados.

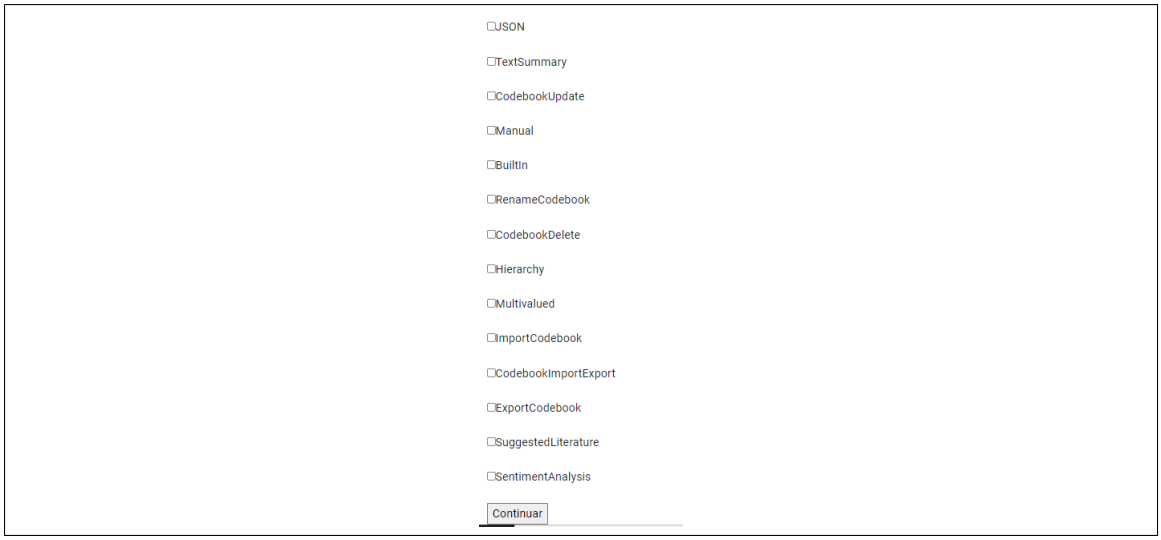

Figura 7.15: Menú de selección de características

7.5.4. Generación y muestra del diagrama

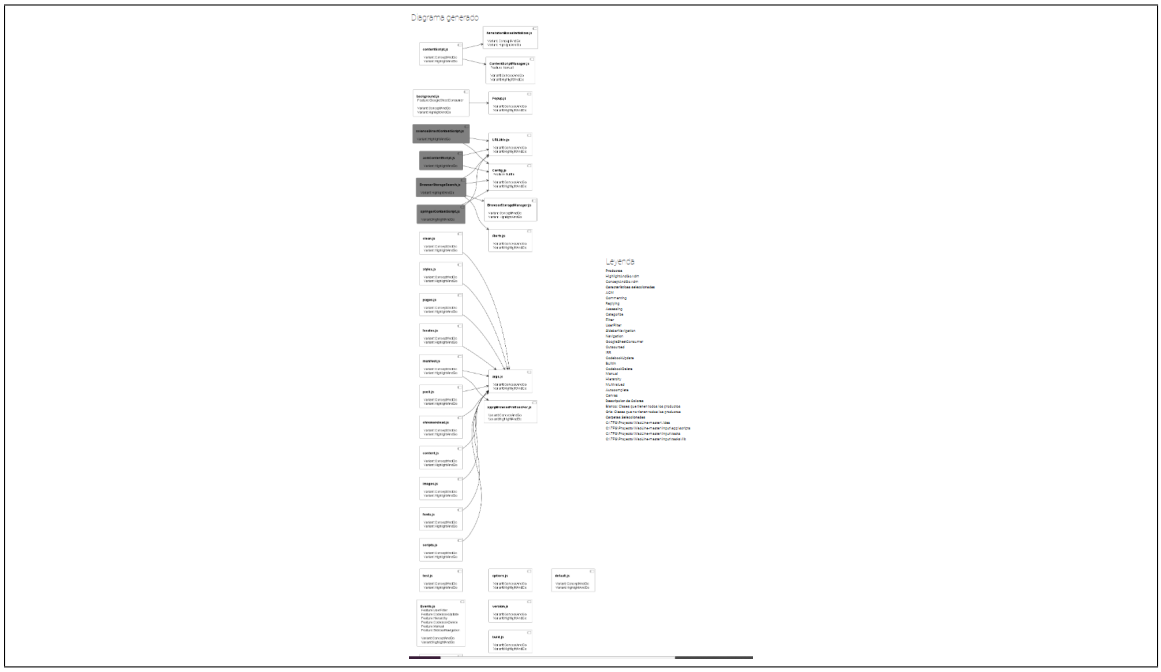

Figura 7.16: Vista con el diagrama final generado

Después de seleccionar los módulos, los productos y las características, el servidor hace una petición para cargar la vista en la que se muestra el diagrama final. Entonces, el controlador, que ha recibido todos los parámetros necesarios mediante un método *POST*, se encarga de hacer una llamada a la lógica de negocio para generar el diagrama. Una vez generado, obtiene el *path* de la imagen y genera un modelo con todos los datos relativos al diagrama, incluyendo el *path*. Con este modelo carga la vista *showDiagram* descrita en la sección 7.3.1.5.

Como esta figura es demasiado grande, en la figura 7.17 se muestra un fragmento del diagrama. En este fragmento se ve cómo las clases aparecen en la implementación de todos los productos seleccionados están en blanco y aquellas que solo son parte de la implementación de algunos productos están en gris.

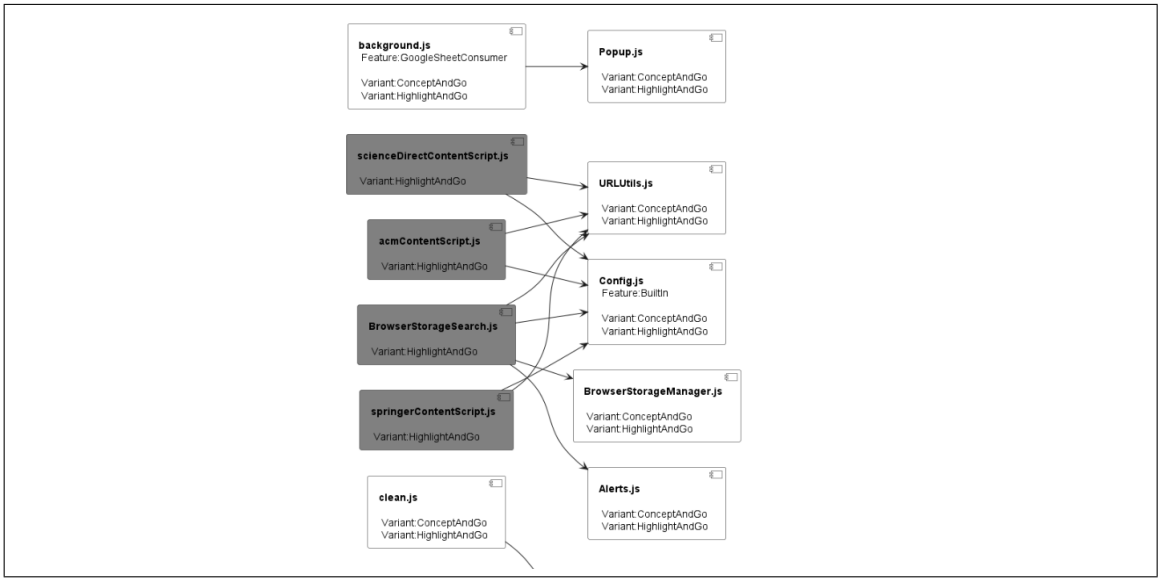

Figura 7.17: Parte del diagrama final

Como se puede ver en la figura 7.18, también se ha generado una leyenda que permita al usuario que vea este diagrama tener todos los datos de los parámetros seleccionados.

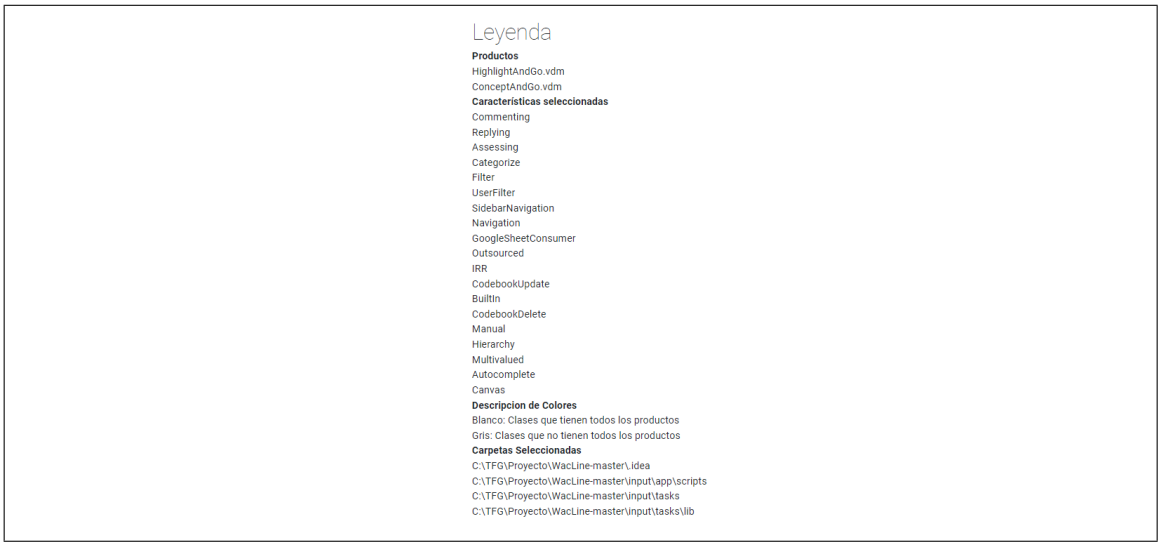

Figura 7.18: Leyenda del diagrama

7.6. Requisitos para la visualización de la arquitectura de una SPL

A continuación se muestran los requisitos sobre la aplicación que se han definido a lo largo del desarrollo del proyecto y que no estaban contemplados en la planificación. Algunos de estos requisitos no han sido finalmente implementados, por falta de viabilidad o tiempo. Para decidir qué requisitos se implementarían, se hizo una estimación de tiempo (en caso de ser viable) para cada requisito y se decidió junto al cliente qué requisitos podían implementarse sin modificar el alcance.

- **Mostrar un listado de todos los módulos que participan en el diagrama**: El primer requisito planteado fue el de generar una lista de todos los módulos que participaban en el diagrama. Como esta lista era demasiado larga en algunos casos, se decidió diseñar el sistema de niveles. Este sistema consiste en dejar al usuario escoger un nivel (desde el 1 hasta el máximo) de profundidad de módulos que quiere escoger. Si este usuario escoge el nivel 1, solo podrá escoger entre los módulos que cuelgan directamente desde la raíz del proyecto. Si escoge el nivel 2, podrá escoger entre los módulos que cuelgan directamente de la raíz y también entre los que están directamente dentro de estos (no aparecerán las carpetas que están incluidas dentro de las del segundo nivel).

  Este requisito fue finalmente implementado porque su coste en horas de trabajo se estimó en unas 3 horas de trabajo. Esta estimación es tan baja porque ya estaba implementado un sistema de selección de ficheros. Esta implementación está descrita en la sección 6.6.1.

- **Una vez seleccionado un módulo, mostrar un listado de variantes implementadas en dicho módulo**: Otro de los requisitos planteados y seleccionados para su implementación fue el de mostrar el listado de los productos disponibles para el diagrama en función de los módulos seleccionados.

  Este requisito se implementó porque no tenía sentido ofrecer una lista de productos sin tener en cuenta los módulos seleccionados. En un sistema real, el desarrollador puede no conocer todos los ficheros y su relación con los productos. Esto puede desembocar en una selección de productos que no son utilizados por los módulos seleccionados y, por tanto, un diagrama que no aporta ninguna información relativa a la arquitectura.

- **Mostrar el diagrama de componentes de varios productos**: El último requisito que se implementó fue el de mostrar el diagrama de varios productos. En las versiones descritas en los capítulos 6 y 5, el diagrama ya mostraba información sobre varios productos, pero estas versiones generaban el diagrama únicamente de la intersección de estos productos. Por ejemplo, si se escogían dos productos, solo mostraba el diagrama de los componentes que participaban en ambos productos, nunca si solo participaban en uno de ellos.

  Por este motivo, se planteó la idea de modificar el diagrama para que mostrara los componentes que pertenecían a todos los productos de un color, mientras se mostraba de otro color aquellos que solo pertenecían a algunos. Tras hacer un estudio de la viabilidad, se estimó que la implementación de este requisito no afectaría al alcance, por lo que se implementó.

- **Interactividad del diagrama:** Uno de los requisitos planteados por el cliente era el de la interactividad del diagrama. Esta interactividad consistía en darle al usuario la posibilidad de hacer clic en los diferentes componentes del diagrama, recibiendo información sobre las características, los productos y los otros componentes con los que interactúa. De esta

manera se facilita la tarea de comprender la arquitectura.

El problema con este requisito es que, actualmente, el diagrama generado es una imagen, y habría que rediseñar el sistema de generación del diagrama. Este rediseño pasaría por cambiar la tecnología que se utiliza actualmente para generar el diagrama, que es *PlantUML*, ya que este solo genera imágenes. Otra solución posible sería programar un script de JS que hiciese interactivo el diagrama. Ambas soluciones se salían del alcance, por lo que se descartó el requisito.

- **Tabla Component x Variant:** Otro de los requisitos planteados era el de obtener una tabla que muestre en las filas los componentes y en las columnas los productos. En las casillas (i,j) se indicaría si el componente C_i aparece en la variante V_j. Este requisito se descartó por falta de tiempo.
- **Diagrama de módulos:** Otro de los requisitos planteados era el de generar el diagrama, mostrando los módulos en vez de los ficheros. Este requisito iba fuertemente ligado al requisito de implementar la parte interactiva, ya que el usuario seleccionaba los módulos y recibía información sobre los ficheros que contenía dicho módulo, además de que productos y características utilizaba. El problema de este requisito es el lenguaje sobre el que se genera la arquitectura, *JavaScript*.

  Esto se debe a que en *JavaScript*, muchas veces los ficheros dentro de un módulo solo interactúan con otros ficheros del mismo módulo. ya que son scripts que se llaman desde ficheros HTML. Por ejemplo, en *WacLine* existe el módulo *codebook*, que tiene 18 ficheros, y solamente uno interactúa con un fichero externo al módulo, mientras que todos los demás ficheros interactúan entre ellos, importando funciones. Si se implementase este sistema, se perdería mucha información de las relaciones que tienen los ficheros de un mismo módulo.

CAPÍTULO 8

Seguimiento y Control

En este capítulo se detalla el seguimiento y control realizado a lo largo del proyecto. Para ello se explican las incidencias, la gestión del alcance, las desviaciones de tiempo surgidas y el control de los riesgos y la calidad.

8.1. Incidencias

En esta sección se detallan las incidencias que han surgido a lo largo del proyecto.

Errores de compatibilidad de Arkit

Esta incidencia se corresponde con uno de los riesgos planteados en la planificación. Durante la adaptación de la herramienta *Arkit*, surgieron algunos problemas de compatibilidad entre las dependencias del proyecto.

Para atajar este problema, se decidió migrar el trabajo realizado, implementar la aplicación en *Java*. Esto supuso un retraso en los plazos establecidos para cada tarea, además de una desviación del tiempo estimado. Esta desviación se explica mejor en el apartado 8.3.

8.2. Gestión del alcance

A continuación se detalla cómo han afectado las incidencias al alcance del proyecto. Explicando que impacto han tenido sobre las tareas y como se han modificado las mismas.

Al tomar la decisión de migrar la herramienta a *Java*, se generó una nueva tarea en el paquete de trabajo *PR.2 (Prototipo 2)*, esta nueva tarea supuso 30h extras en el paquete de trabajo (ver tabla 8.1). Como se explica en la sección 8.4, esta inversión de horas extra en el trabajo de migrado permitió reducir las horas necesarias para realizar el resto de tareas de implementación. Esta reducción de horas consiguió que la nueva tarea no tuviera impacto sobre la planificación total del proyecto.

8.3. Gestión del tiempo

A continuación se muestran las desviaciones de tiempo frente a las planificadas al comienzo del desarrollo del proyecto. (ver tabla 8.1)

En la tabla 8.1 se puede observar que existe una desviación de 9 horas respecto a las horas planificadas. Esta dedicación final supone una desviación del 3 % para un trabajo de 315 horas.

La mayor parte de la desviación ha sucedido en las tareas de redacción y en el paquete del prototipo 2. La desviación en el paquete del prototipo 2 ha sido producto del impacto de un riesgo, tal y como se explica en la sección 8.4. Este impacto ha generado una nueva tarea, la tarea *PR2.2*. La tarea no tiene un tiempo estimado, ya que no se contempló en la planificación original. Sin embargo, el impacto de introducir una nueva tarea ha sido amortiguado por el resto de tareas de implementación. En cambio, se estimaron de manera incorrecta las horas que finalmente se dedicaron a la memoria.

Paquete de Trabajo	Tarea	Descripción	Horas estimadas	Horas actuales	Diferencia
Planificación	P.1	Captura de Requisitos	5	5	0
	P.2	Planificación inicial	12	10	-2
	P.3	Actualización de la planificación	5	6	+1
	Subtotal		22	21	-1
Seguimiento y Control	SC.1	Recogida Información	3	3	0
	SC.2	Contraste información	2	3	+1
	SC.3	Aseguramiento de condiciones	5	3	-2
	Subtotal		10	9	-1
Reuniones	R.1	Reunión Inicial	1	1	0
	R.2	Reuniones periodicas	15	9	-6
	Subtotal		16	10	-6
Investigacion	Inv.1	Revisión Tecnologias SPL	6	5	-1
	Inv.2	Investigación sobre pure::variants	5	5	0
	Inv.3	Busqueda de herramientas	10	12	+2
	Inv.4	Busqueda de Proyecto	5	5	0
	Inv.5	Comprensión de WacLine	8	5	-3
	Inv.6	Comprensión de InsideSPL	8	8	0
	Subtotal		42	40	-2
Prototipo 1	PR1.1	Creación Tabla de Criterios	3	5	+2
	PR1.2	Seleccion Herramienta	5	5	0
	PR1.3	Adaptación Inicial	6	6	0
	PR1.4	Pruebas Iniciales	4	5	+1
	Subtotal		18	21	+3
Prototipo 2	PR2.1	Adaptación de la herramienta a pure::variants	30	30	0
	PR2.2	Migración de la herramienta a Java	—	30	+30
	PR2.3	Pruebas con proyecto pure::variants pequeño	5	10	+5
	Subtotal		35	70	+35
Prototipo 3	PR3.1	Mejora de herramienta para SPL mas grande	40	15	-25
	PR3.2	Pruebas con WacLine	5	5	0
	Subtotal		45	20	-25
Prototipo 4	PR4.1	Integración en InsideSPL	35	25	-10
	PR4.2	Pruebas finales	5	5	0
	Subtotal		40	30	-10
Memoria	Mem.1	Redacción	70	85	+15
	Mem.2	Revisión	10	10	0
	Subtotal		80	95	+15
Defensa	Def.1	Póster	3	3	0
	Def.2	Presentación	4	4	0
	Subtotal		7	7	0
TOTAL			315	323	+9

Tabla 8.1: Tabla de desviaciones del tiempo asignado para cada tarea

8.4. Gestión de los riesgos

A continuación se listan los riesgos y se explica el impacto que han tenido en el desarrollo del proyecto.

- **Trabajo académico:** Existía el riesgo de que el curso académico afectase al desarrollo de las diferentes tareas del proyecto. Para amortiguar el impacto de este riesgo, se aumentó la duración de las tareas que estaban previstas para ser realizadas en las semanas de horario agrupado, de forma que la dedicación durante esas semanas fuera menor. Gracias a esta gestión, las horas de estudio dedicadas en estas semanas no afectaron al desarrollo del proyecto.
- **Uso de tecnologías nuevas:** A la hora de trabajar con *pure::variants*, resultaba complicado estimar algunas tareas, como el tiempo de adaptación de la herramienta. Esto se debe a que esta tecnología era nueva para el alumno. Para evitar este riesgo, se añadió una tarea de estudio y análisis de esta tecnología. Esta tarea permitió establecer unos tiempos realistas para los trabajos de adaptación.
- **Captura de requisitos:** En el comienzo del proyecto, se definieron algunos requisitos con el cliente, tal y como se describe en el apartado 2.2.2, esto se hizo para evitar desviaciones en caso de que el alumno hubiera definido incorrectamente los mismos. Aun así, no se tuvo en cuenta la posibilidad de que el cliente añadiera nuevos requisitos a lo largo del proyecto.

  Además, los requisitos añadidos en la última fase de implementación y que se describen en el capítulo 7.6, no supusieron una desviación en el tiempo, porque fueron definidos por el cliente tras conocer la dedicación disponible en ese momento y tras realizar una correcta estimación de su coste en horas de trabajo. En este caso, el hecho de añadir nuevos requisitos no supuso un impacto para la realización del proyecto, pero habría que haberlo tenido en cuenta a la hora de definir los riesgos.
- **Carácter investigativo:** A la hora de definir el alcance, se decidió tomar un enfoque iterativo e incremental para poder decidir tras cada iteración si tenía sentido seguir con el proyecto, teniendo en cuenta el cómputo de horas realizadas en el momento de terminar dicha iteración. Tras tomar este enfoque, se han realizado todas las iteraciones definidas, por lo que se asume que el riesgo se ha gestionado correctamente.
- **Pérdida de información:** Gracias al control de versiones y las copias de seguridad en la nube, se han podido recuperar los datos que se han perdido en hasta dos ocasiones por problemas de sobrescritura.
- **Problemas con la compatibilidad:** El riesgo de los problemas de compatibilidad generó una incidencia que se describe en la sección 8.1, tal y como se puede ver en la tabla 8.1, esta incidencia hizo que se tuviera que crear una nueva tarea en el paquete del prototipo 2, causando una desviación de 30h respecto a la dedicación planificada. Una correcta gestión del impacto permitió invertir más horas en esta tarea para optimizar el resto de la implementación. Esta gestión consistió en migrar la herramienta a *Java* para ahorrar tiempo en la integración y compensar de esta manera la desviación de tiempo.
- **Planificación incorrecta:** Tras una primera revisión, se actualizó la planificación para intentar abarcar de la mejor manera el alcance del proyecto. Aun así, ha habido algunas desviaciones, pero, tal y como se explica en la sección 8.3, esta desviación ha sido de un 2 % del tiempo estimado, por lo que se entiende que el proyecto se planificó correctamente.

8.5. Control de la calidad

Para asegurar la calidad del proyecto, se han llevado las acciones descritas en el apartado 2.5.2 de la planificación.

- **Pruebas internas de código:** Se realizaron pruebas en cada una de las etapas de desarrollo del producto. Estas pruebas se realizaron manualmente cambiando los parámetros de ejecución. Las pruebas no fueron realizadas mediantes una manera estandarizada como podrían ser los *JUnits* por dos razones. La primera es el tiempo, ya que suponía demasiado tiempo para el estimado en las pruebas de cada prototipo. La segunda razón es la complicación de estas pruebas, ya que los parámetros de ejecución cambian por cada opción escogida. Por ejemplo, tras seleccionar un producto, cambian las características disponibles para procesarse, esto produce un gran número de posibilidades, que consumirían demasiado tiempo de pruebas. Además, siendo un proyecto que pretende analizar la viabilidad de obtener la arquitectura de una SPL, el objetivo era tener un prototipo de herramienta, que es lo que se ha conseguido. Por tanto, se considera que las pruebas son suficientes. Si la aplicación fuera a ponerse en producción, necesitaría una fase de pruebas más exhaustiva y sistemática, pero que quedaba fuera del alcance de este proyecto.
- **Muestra de prototipos al cliente:** Durante el desarrollo del proyecto, se han hecho reuniones periódicas con el cliente. En estas reuniones, se han presentado los diferentes prototipos al cliente, para asegurar la calidad del producto desarrollado. Además, estas reuniones han servido para añadir nuevos requisitos a lo largo del proceso de desarrollo.

8.6. Gestión de las comunicaciones

Respecto a las comunicaciones, tal y como se explicaba en la sección 2.6, se ha utilizado el correo electrónico para las comunicaciones del día a día. El uso del correo electrónico ha permitido una comunicación rápida y cómoda, además de la posibilidad de compartir fácilmente ficheros.

También se han realizado reuniones telemáticas mediante la plataforma *Webex*, estas reuniones han permitido mostrar a los interesados las diferentes versiones del producto. Gracias a estas demostraciones, se ha recibido un *feedback* que ha resultado muy útil para el desarrollo de la aplicación.

8.7. Gestión de los interesados

En esta sección se describen cómo se ha gestionado la interacción con los interesados de este proyecto y el impacto que han tenido en el proyecto.

La interacción principal ha sido con el cliente, ya que las reuniones telemáticas se han utilizado principalmente para mostrar las diferentes versiones del trabajo realizado. En estas reuniones, se mostraba al cliente el prototipo implementado para recibir su *feedback*. Durante algunas de estas reuniones, el cliente pidió añadir requisitos para la aplicación. Para gestionar esta petición, se hizo un desglose de los nuevos requisitos, explicándole al cliente la viabilidad de los mismos y el tiempo estimado para implementarlos. Este desglose está descrito en la sección 7.2.

Una vez hecho el desglose, se decidió junto al cliente que requisitos se implementarían. Una vez implementados estos requisitos, se le mostró al cliente el prototipo final.

CAPÍTULO 9

Conclusiones y líneas futuras

En este capítulo se enumeran las conclusiones que se han obtenido tras finalizar el TFG. Estas conclusiones se dividen en conclusiones técnicas y personales. También se incluyen las posibles mejoras que podría tener la herramienta desarrollada y que no han podido ser desarrolladas por falta de tiempo.

9.1. Conclusiones

Al ser tan complicado estimar el alcance de los objetivos del proyecto, este TFG fue planteado con un carácter iterativo e incremental. Cada una de estas iteraciones daba como resultado un producto que podía considerarse como el producto final de este proyecto. A pesar de las incidencias del proyecto, se han cumplido todos los objetivos del proyecto.

Conclusiones a nivel técnico

Las conclusiones a nivel técnico son positivas. En la tabla 9.1, se listan los diferentes requisitos del proyecto, indicando su grado de prioridad y su grado de cumplimiento, añadiendo, en caso de no haberse cumplido, la razón para ello. El grado de cumplimiento tiene tres valores posibles:

- **Satisfactorio:** Este valor indica que se ha cumplido correctamente el requisito.
- **Suficiente:** Este valor indica que el requisito se ha cumplido con posibilidad de mejora.
- **Incompleto:** Este valor indica que el requisito no se ha cumplido.

Requisito	Prioridad	Grado de cumplimiento	Razón
Herramienta open-source	Principal	Satisfactorio	
Soporte para JavaScript	Principal	Satisfactorio	
Descripción de los componentes	Principal	Suficiente	Falta información sobre las constantes y variables..
Procesamiento de código pure::variants	Principal	Satisfactorio	
Aspectos de variabilidad en el diagrama	Principal	Satisfactorio	
Casos de prueba implementados usando las pure::variants	Principal	Suficiente	Faltan casos de prueba unitarios.
Listado de módulos en la herramienta integrada	Principal	Satisfactorio	
Listado de productos implementados en los módulos escogidos	Principal	Satisfactorio	
Diagrama de componentes generado para varios productos	Principal	Satisfactorio	
Soporte para otros lenguajes de programación	Secundario	Incompleto	La herramienta solo soporta JavaScript
Documentación de la herramienta	Secundario	Suficiente	Se ha generado una guía de uso pero falta la documentación sobre el codigo y sus metodos
Herramienta con acceso web	Secundario	Incompleto	Al migrar la herramienta no hacía falta que tuviera acceso web

Tabla 9.1: Tabla de requisitos con su grado de cumplimiento

Estos son los objetivos técnicos implementados en el proyecto.

- **Herramienta de ingeniería inversa adaptada para un proyecto *SPL*:** Se ha adaptado una herramienta para obtener, mediante ingeniería inversa, la arquitectura de un proyecto *SPL*.
- **Migración de la herramienta a Java:** Se ha migrado la herramienta a *Java*, obteniendo una herramienta más fácil de integrar en *InsideSPL*.
- **Integración en InsideSPL:** Se ha integrado la herramienta en *InsideSPL*, cumpliendo los requisitos establecidos por el cliente.

Conclusiones a nivel personal

En lo que se refiere a las conclusiones a nivel personal, estas son también positivas. Se ha obtenido conocimiento en diferentes áreas relacionadas con el trabajo realizado. Se muestran a continuación las diferentes conclusiones.

- **Aprendizaje sobre *pure::variants*:** Los conocimientos que se tenían sobre el funcionamiento de una línea de producto software son los que se adquieren en el tercer curso del grado, en la asignatura de "Diseño Industrial de Software". Además, solo se ven SPLs de tamaño pequeño e implementadas con un enfoque compositivo, a diferencia de las que se usan normalmente en las empresas, que están anotadas. Para poder completar correctamente este proyecto, no solo se han adquirido conocimientos sobre el paradigma de las *SPL*, sino que también se han adquirido conocimientos sobre el sistema de anotaciones que utiliza *pure::variants*, conocimientos que han sido indispensables para la realización del trabajo.

- **Aprendizaje sobre *PlantUML*:** A la hora de generar diferentes diagramas, en el grado se muestran diferentes tecnologías al alumno. Con la realización de este proyecto, se ha podido conocer *PlantUML*, gracias a este software, se ha facilitado la tarea de generar el diagrama.
- **Aprendizaje sobre *Spring*:** *InsideSPL* está programado en *Spring*, un *framework* de *Java*, que permite trabajar con aplicaciones web. Para el desarrollo de este proyecto, ha sido absolutamente necesario adquirir conocimientos sobre esta tecnología.
- **Gestión de los riesgos del proyecto:** Uno de los riesgos definidos en la sección 2.5.1 era el de los problemas con la compatibilidad. Este problema causó una desviación del 100 % del tiempo estimado para la tarea. Tras una correcta gestión del impacto que supuso este riesgo, se decidió migrar la herramienta a *Java*, recuperando el tiempo perdido en las tareas de integración y mejora de la herramienta.

9.2. Líneas Futuras

Aunque se hayan cumplido los diferentes objetivos y requisitos definidos a lo largo del proyecto, hay algunas mejoras que podrían implementarse en el futuro.

- **Optimización del código:** Algunas de las funciones implementadas son ineficientes. Una optimización de estas funciones mejoraría los tiempos de ejecución.
- **Mejora del *frontend*:** En la implementación, se generaron diferentes vistas. Una mejora a considerar sería la de mejorar el aspecto de estas vistas.
- **Diagrama interactivo:** Sería interesante hacer el diagrama final interactivo, de manera que el usuario pudiera hacer clic en un componente en concreto para recibir información sobre el mismo.
- **Soporte en otros lenguajes:** Actualmente, la aplicación funciona con proyectos *JavaScript*. Una mejora a tener en cuenta sería hacer que trabajase con proyectos implementados en otros lenguajes.

Anexos

A. Guía de Uso

A continuación se muestra una guía de uso de la aplicación. En esta guía se describen los pasos necesarios para poder ejecutar la aplicación, explicando que tecnologías instalar y como configurar la aplicación para su correcto funcionamiento.

UPV - EHU

Facultad de Informática

Guia de uso para la aplicación desarrollada

Autor:
Endika Varas

2022

Índice general

Capítulo 1

Instalación de la aplicación

En este capítulo se describen los pasos a seguir para instalar la aplicación.

1.1. Tecnologías necesarias

A continuación se muestran las tecnologías que han de instalarse para poder utilizar la aplicación.

1.1.1. XAMPP

El primer paso es instalar XAMPP, para ello, accedemos a la página web de apache y descargamos la versión que más nos convenga.

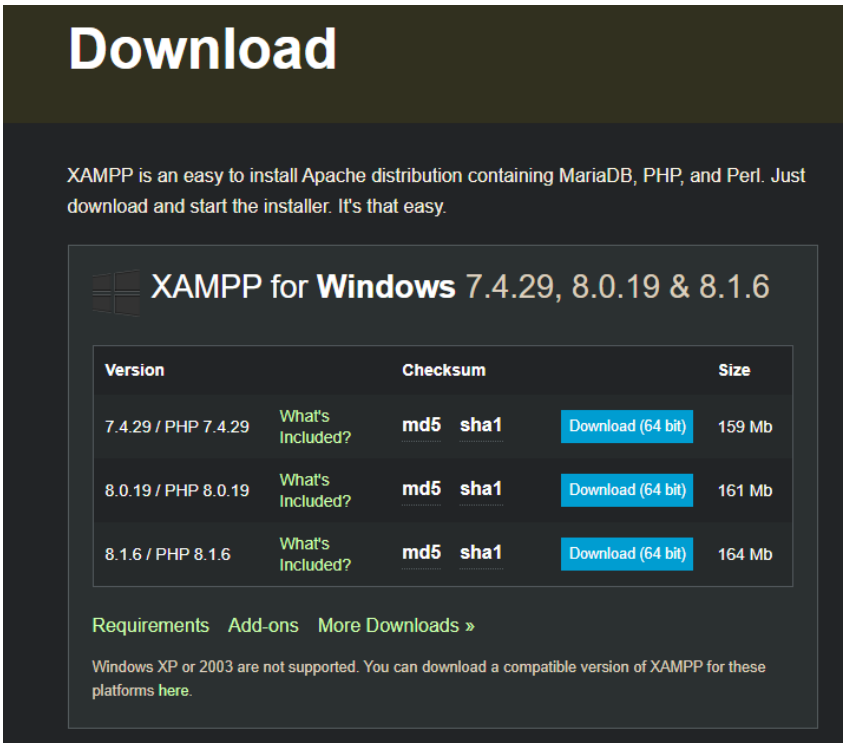

Figura 1.1: Página web de Apache

En la figura 1.1 vemos como se nos muestran varias versiones para descargar. Se recomienda descargar la última versión. Una vez descargado el instalador, lo ejecutamos e instalamos la aplicación.

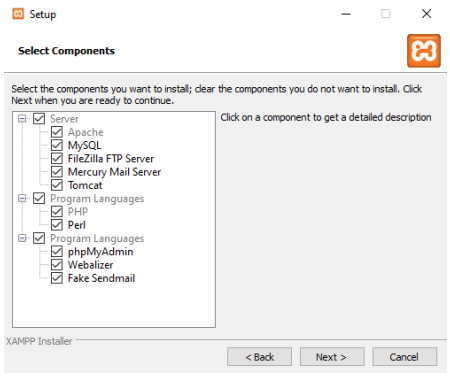

Figura 1.2: Menú de selección de tecnologías

En la imagen 1.2 podemos ver que al instalarlo, conviene marcar todas las tecnologías disponibles.

Tal y como se ve en la figura 1.3 es importante instalar la aplicación en el disco C, ya que normalmente da problemas instalarlo en otros discos.

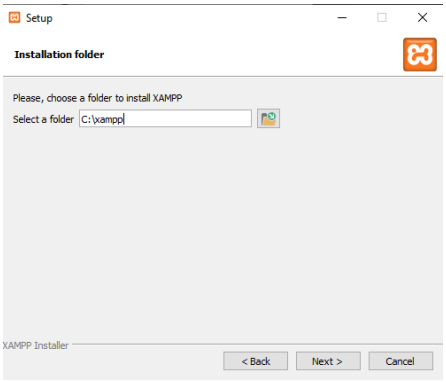

Figura 1.3: Menú de selección de *path* de instalación

1.1.2. Eclipse

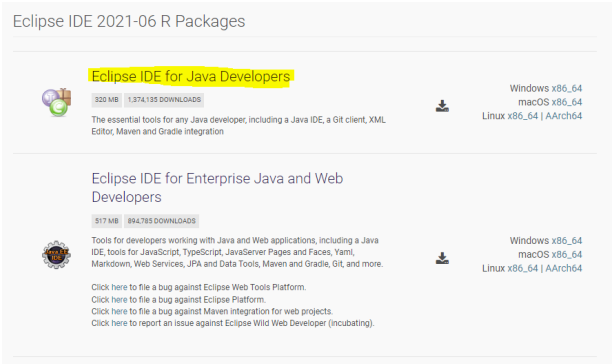

Figura 1.4: Página de descargas de *Eclipse*

Para poder ejecutar las aplicaciones, es necesario tener instalado *Eclipse*, para ello, nos dirigimos a la página web de Eclipse y seleccionamos la opción de *Eclipse IDE for developers*, tal y como se ve en la imagen 1.4. Una vez hayamos descargado el fichero, lo descomprimimos y ejecutamos el instalador. En este caso, no es importante el lugar de instalación.

1.1.3. Git Bash

Para algunas tareas del módulo *SPLMiner*, es importante generar ficheros .git en las carpetas de los proyectos a minar. Por eso, tras acceder a la página web de GIT, descargamos el instalador para la máquina que tengamos.

Como anteriormente, la carpeta de instalación no es importante a la hora de instalar la aplicación

1.2. Proyectos necesarios

A continuación se describen los proyectos necesarios para poder utilizar la aplicación.

1.2.1. SPLMiner

Este proyecto se encarga de generar el fichero SQL que será procesado por *InsideSPL* para obtener los diferentes datos necesarios para la visualización.

Para instalar este proyecto, el primer paso es descargarlo del repositorio donde se encuentra. Una vez descargado, abrimos eclipse y seguimos los siguientes pasos.

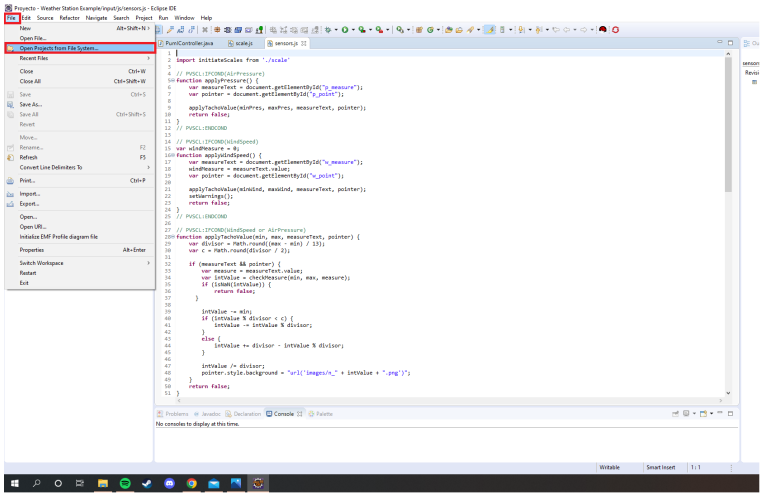

Figura 1.5: Interfaz de eclipse

Tal y como se ve en la figura 1.5, hacemos clic en *File* y después en *Open projects from file system*. Esta opción abrirá un navegador de archivos y tendremos que seleccionar la dirección donde se haya guardado el proyecto.

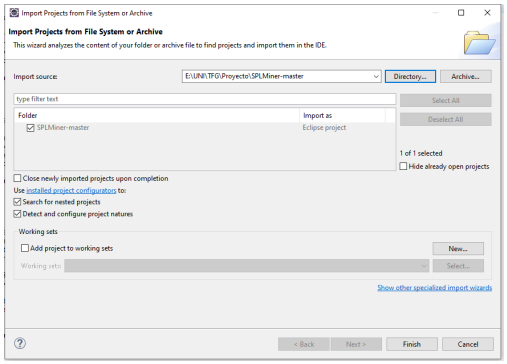

Figura 1.6: Interfaz de selección de carpeta

Una vez que el menú este como en la figura 1.6, hacemos clic en *finish* y el proyecto estará importado.

1.2.2. InsideSPL

En el caso de *InsideSPL* tendremos que descargar el repositorio desde el siguiente link. Una vez este descargado tendremos que realizar el mismo proceso de importado que en *SPLMiner*.

Capítulo 2

Minado de la SPL

2.1. Creación del fichero .git

Para que el minado funcione correctamente, se necesita un fichero .git en la carpeta del proyecto a minar. Antes de generar el fichero .git, creamos un repositorio en *Github*. Para generar el fichero .git, necesitamos abrir *git bash*.

Con *git bash* abierto escribimos los siguientes comandos:

- ***cd *path del proyecto****: Este comando abrirá la carpeta como dirección objetiva para el resto de comandos.
- ***git init***: Este comando inicializa él .git en la carpeta.
- ***git add****: Este comando añade los ficheros del proyecto a la cola para ser *commiteados*.
- ***git commit***: Este comando deja el *commit* preparado para poder hacer un *push*.
- ***git commit***: Este comando deja el *commit* preparado para poder hacer un *push*.
- ***git remote add origin https://github.com/username/new_repo***: Este comando establece como destino del *push* el repositorio creado.
- ***git push -u origin master***: Este comando sube todos los archivos al repositorio.

2.2. Creación de usuario MySQL

Para esta parte, necesitamos abrir *XAMPP*. Una vez abierto, hacemos clic en *Shell*, tal y como se ve en la figura 2.1.

Figura 2.1: Interfaz de XAMPP

Una vez en *shell*, ejecutamos los siguientes comandos:

- **mysql**: Este comando inicializa la aplicación mysql.
- **CREATE USER 'spl'@'localhost' IDENTIFIED BY 'spl';**: Este comando creará al usuario que utiliza *SPLMiner* para generar el fichero .SQL.
- **GRANT ALL PRIVILEGES ON * . * TO 'spl'@'localhost';**: Este comando dará al usuario todos los permisos necesarios para poder generar el fichero.

Una vez hecho esto ya tenemos el usuario preparado para generar el fichero.

2.3. Creación de la base de datos

Para crear la base de datos, necesitaremos usar *XAMPP* de nuevo. En la interfaz hacemos clic en el boton *Admin* que se encuentra en la línea de *MySQL*, tal y como se ve en la figura 2.2.

Figura 2.2: Interfaz de XAMPP

Una vez hecho clic, se abrirá un menú online para la gestión de la base de datos. En este menú hacemos clic en la opción *nueva* que se encuentra en la esquina superior izquierda (ver figura 2.3).

Figura 2.3: Interfaz de MySQL

Una vez seleccionada esta opción creamos una nueva base de datos vacía y la llamamos *spl*.

Figura 2.4: Interfaz de MySQL

Después de crear la base de datos, hacemos clic en *SQL* (ver figura 2.4) y ejecutamos el siguiente script (puede ser ejecutado por partes).

```
SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
START TRANSACTION;
SET time_zone = "+00:00";

CREATE TABLE `attribute` (
```

```
  `ID` varchar(255) NOT NULL,
  `NAME` varchar(255) NOT NULL,
  `TYPE` varchar(255) NOT NULL,
  `VALUE` varchar(255) NOT NULL,
  `TARGET_FEATURE` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `codefile` (
  `ID` varchar(255) NOT NULL,
  `FILENAME` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `code_element` (
  `ID` varchar(255) NOT NULL,
  `PATH` varchar(255) NOT NULL,
  `TYPE` varchar(255) NOT NULL,
  `PARENT` varchar(255) DEFAULT NULL,
  `SPL_ID` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `code_variation_point` (
  `VP_ID` varchar(255) NOT NULL,
  `START_LINE` int(11) NOT NULL,
  `END_LINE` int(11) NOT NULL,
  `CONTENT` varchar(255) NOT NULL,
  `NESTING_LEVEL` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `dependency` (
  `ID` varchar(255) NOT NULL,
  `TYPE` varchar(255) NOT NULL,
  `SOURCE_FEATURE` varchar(255) NOT NULL,
  `TARGET_FEATURE` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `directory` (
  `ID` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `feature` (
  `ID` varchar(255) NOT NULL,
  `NAME` varchar(255) NOT NULL,
  `TYPE` varchar(255) NOT NULL,
  `PARENT` varchar(255) DEFAULT NULL,
  `FEATURE_MODEL` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `feature_model` (
  `ID` varchar(255) NOT NULL,
  `FILENAME` varchar(255) NOT NULL,
  `PATH` varchar(255) NOT NULL,
  `SPL` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `feature_size` (
```

```
  `FEATURE_ID` varchar(255) NOT NULL,
  `SIZE` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `git_spl` (
  `ID` varchar(255) NOT NULL,
  `URL` varchar(255) NOT NULL,
  `LAST_CHANGED` date NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `part` (
  `ID` varchar(255) NOT NULL,
  `PART_TYPE` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `spl` (
  `ID` varchar(255) NOT NULL,
  `NAME` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `variant_code` (
  `VC_ID` varchar(255) NOT NULL,
  `CODE_ELEMENT_ID` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `variant_component` (
  `ID` varchar(255) NOT NULL,
  `VARIANT_MODEL` varchar(255) NOT NULL,
  `IS_SELECTED` tinyint(1) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `variant_feature` (
  `VC_ID` varchar(255) NOT NULL,
  `FEATURE_ID` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `variant_model` (
  `ID` varchar(255) NOT NULL,
  `FILENAME` varchar(255) NOT NULL,
  `PATH` varchar(255) NOT NULL,
  `SPL_ID` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `variation_point` (
  `ID` varchar(255) NOT NULL,
  `CODE_ELEMENT_ID` varchar(255) NOT NULL,
  `EXPRESION` varchar(255) NOT NULL,
  `VP_SIZE` int(11) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE `variation_point_feature` (
  `VP_ID` varchar(255) NOT NULL,
  `FEATURE_ID` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

```
ALTER TABLE `feature`
  ADD PRIMARY KEY (`ID`);
ALTER TABLE `spl`
  ADD PRIMARY KEY (`ID`);
COMMIT;
```

Código SQL 2.1: Script de creación de la base de datos

Una vez hecho esto ya tenemos la base de datos configurada.

2.4. Obtención del fichero .SQL

Para obtener el fichero .sql que necesitamos, abrimos *Eclipse* y seleccionamos el proyecto *SPLMiner* importado. En él abrimos la clase *MainClass* del módulo *Main*.

```
private static final String SPL_LOCAL_GIT_REPO = "C:/TFG/Proyecto/WacLine-master";

// Folder where are all the code files, images...
private static final String SPL_CODE_FOLDER = SPL_LOCAL_GIT_REPO + "/input";

// SPL info
private static final String SPL_NAME = "WacLine";

// Mining type
private static final int MINING_TYPE = 3;

// CMap configuration (for MINING_TYPE == 2 OR 3)
private static final String CMAPS_FOLDER = "/cmaps";
private static final String[] CMAPS = {
            SPL_LOCAL_GIT_REPO + CMAPS_FOLDER + "/WebAnnotation.cxl"
        };
```

Figura 2.5: Clase *Main* de *SPLMiner*

Tal y como se ve en la figura 2.5, al comienzo de la clase hay varios parámetros que hay que modificar.

- **SPL_LOCAL_GIT_REPO:** En este parametro debe ir la direccion del proyecto.
- **SPL_NAME:** El nombre con el que queremos guardar la información.
- **CMAPS:** en caso de haber un fichero *.cxl*, aquí se escribe su dirección, si no existe, en el parámetro **MINNING_TYPE** debemos poner 1.

Una vez hecho esto ejecutamos el *main* y se generará un fichero .sql en la carpeta del proyecto.

2.5. Importación del fichero a la base de datos

Para importar el fichero .SQL, entramos en la interfaz de *MySQL* y hacemos clic en *Importar*, que está en la barra superior (ver figura 2.6)

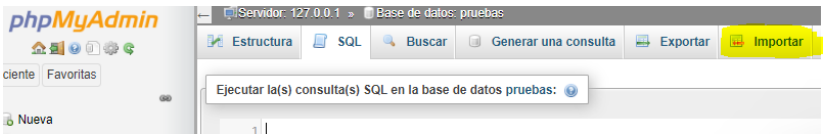

Figura 2.6: Interfaz de MySQL

Después hacemos clic en examinar y cargamos el fichero. Hacemos clic en continuar y se importará la base de datos.

Capítulo 3

InsideSPL

En este capítulo se explica como inicializar la aplicación para acceder a las diferentes funcionalidades.

3.1. Uso de la aplicación

Para inicializar la aplicación, lo primero es abrir *XAMPP* y darle a *start* en las aplicaciones *Apache* y *MySQL*. Después, abrimos eclipse y abrimos el proyecto *InsideSPL*. Una vez abierto, buscamos la clase *InsideSPLApplication* del módulo *com.onekin.insideSPL*. Ejecutamos esa clase, debería mostrarse por pantalla un mensaje como el de la figura 3.1.

```
2022-06-23 17:44:29.697  INFO 4036 --- [           main] c.onekin.insideSpl.InsideSplApplication  : Started InsideSplApplication in 3.73 seconds (JVM running for 10.255)
```

Figura 3.1: Mensaje de inicio de *InsideSPL*

Cuando aparezca este mensaje, bastará con entrar al navegador e introducir la dirección *http://localhost:8080/*. Cuando introduzcamos esta dirección, cargara la interfaz de la aplicación. (ver figura 3.2).

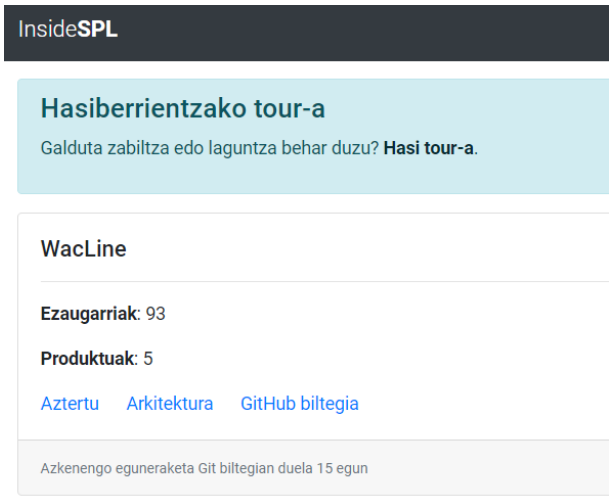

Figura 3.2: Menú de inicio de *InsideSPL*

Este menú tiene tres opciones:

- **Examinar:** Este menú redirige al usuario a la interfaz de visualización de productos. Este menú permite al usuario ver que características implementa cada producto, comparar los productos o ver el árbol de características del proyecto.
- **Arquitectura:** Este menú lleva a la interfaz de generación del diagrama. Esta interfaz permite al usuario seleccionar diferentes parámetros como los módulos, productos y características que se tienen en

cuenta para generar el diagrama. Una vez seleccionado, se genera el diagrama para los componentes que están en los módulos seleccionados y que además implementan alguno de los productos seleccionados, mostrando información sobre las características.

- **Github:** En esta opción se muestra el repositorio de GitHub donde se encuentra el proyecto.

Además, la aplicación está implementada en Euskera, Castellano e Inglés. Para cambiar el idioma, hay un desplegable en la esquina superior derecha que permite modificar el idioma (ver figura 3.3).

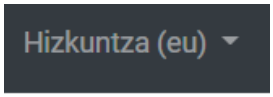

Figura 3.3: Menú de selección de idioma

Bibliografía

[1] Software Produktu-Lerroen (SPL) arkitektura bistaratzeko aplikazioaren garapena, iosu salaberri intxaurrondo, 2020. https://addi.ehu.es/handle/10810/48777.

[2] GitHub - onekin/WacLine: WacLine is a software product line that allows configuration and derivation of web annotation browser extensions for specific domains using pure::variants. https://github.com/onekin/WacLine.

[3] Comprar, vender y enviar dinero por Internet | PayPal España. https://www.paypal.com/es/home.

[4] Software Product Lines. https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=513819.

[5] pure-systems: Home. https://www.pure-systems.com/.

[6] Highlight&Go. https://highlightandgo.haritzmedina.com/.

[7] Architecture Description Language (ADL) - CIO Wiki. https://cio-wiki.org/wiki/Architecture_Description_Language_(ADL).

[8] GitHub - dyatko/arkit: JavaScript architecture diagrams and dependency graphs. https://github.com/dyatko/arkit.

[9] Class Visualizer. http://class-visualizer.net/features.html.

[10] Codeling. https://codeling.de/sourcecode.html.

[11] herramienta de código abierto que utiliza descripciones textuales simples para dibujar diagramas UML. https://plantuml.com/es/.

[12] TypeScript: JavaScript With Syntax For Types. https://www.typescriptlang.org/.

[13] GitHub - twbs/bootstrap: The most popular HTML, CSS, and JavaScript framework for developing responsive, mobile first projects on the web. https://github.com/twbs/bootstrap.

[14] GitHub - imfly/js2uml: Convert JS to UML class diagrams. https://github.com/imfly/js2uml.

[15] JavaScript | MDN. https://developer.mozilla.org/es/docs/Web/JavaScript.

[16] CSS | MDN. https://developer.mozilla.org/es/docs/Web/CSS.

[17] All Skin Parameters — Ashley's PlantUML Doc 0.2.01 documentation. https://plantuml-documentation.readthedocs.io/en/latest/formatting/all-skin-params.html.

[18] GitHub - onekin/WeatherStation: Pure::variants example of a Weather Station system - HTML. https://github.com/onekin/WeatherStation.

[19] Renovate · GitHub Marketplace · GitHub. https://github.com/marketplace/renovate.

[20] pino - npm. https://www.npmjs.com/package/pino.

[21] Spring | Home. https://spring.io/.