Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA
Mención en Ingeniería del Software

Reglas y control de calidad automatizado para el lenguaje Vensim

---

Alumno: Juan Herruzo Herrero

Tutora: Yania Crespo González-Carvajal

A mi padre, por haber sido el mejor modelo a seguir.

Agradecimientos

A mis padres y familia, por haberme apoyado y cuidado durante tantos años, sin ellos no habría llegado a donde estoy ahora.

A mi novia, por haberme ayudado en cualquier momento aunque estuviese diluviando o granizando y haberme empujado a seguir día a día. Gracias por toda esa confianza que me has dado.

A mis amigos, por estar siempre para quedar y hablar.

A mi tutora, Yania Crespo, por haberme ayudado tanto y guiarme en este proyecto.

A mi compañero veterano Daniel Bazaco, por el gran trabajo que hizo en el plugin inicial y por ofrecerme su ayuda durante el transcurso del proyecto. Y al resto de la universidad por haberme dado una gran calidad de enseñanza la cual he podido aprovechar para realizar este proyecto.

Resumen

Este Trabajo Fin de Grado se ha realizado en el marco del proyecto europeo H2020 LOCOMOTION, en este proyecto europeo, trece instituciones europeas trabajan codo con codo para desarrollar un modelo que permita estudiar cómo poder reducir la huella de carbono generada por la sociedad para conseguir un mundo más sostenible.

El proyecto que se desarrolla en este TFG, tiene como objetivo principal el desarrollo de en plugin para Sonarqube dedicado a analizar la calidad de Integrated Assessment Models (IAMs) desarrollados con el software de simulación Vensim. Se parte del TFG de Daniel Bazaco Velasco llamado "Definición y comprobación de estándares de calidad en la programación de IAMs en Vensim" y se continúa con el desarrollo implementado durante el mismo.

Mediante el plugin desarrollado en este trabajo, se puede comprobar estándares de calidad, convenciones de nombres y reglas de programación en los modelos de Vensim. Además, el plugin permite la comunicación con un diccionario de datos externo que sirve para mantener un registro de todos los elementos existentes en los modelos. Dicho registro favorece la coordinación en el desarrollo del modelo, así como la comunicación con otros stakeholders del proyecto Locomotion.

Para la dirección del proyecto se ha utilizado el marco de trabajo Scrum, y se ha desarrollado utilizando Java con las librerías de Sonarqube y ANTLR4. El software obtenido como resultado final se ha publicado en GitHub, con la finalidad de permitir su uso públicamente.

Abstract

This capstone project has been carried out within the framework of the European project H2020 LOCOMOTION.In this European project, thirteen European institutions work hand in hand to develop a model that allows studying how to reduce the carbon footprint generated by society to achieve a more sustainable world.

The project that is developed in this capstone project, has as its main objective the development of a plugin for Sonarqube dedicated to analyzing the quality of Integrated Assessment Models (IAMs) developed with the Vensim simulation software. It starts from Daniel Bazaco Velasco's capstone project called " Definition and verification of quality standards in the programming of IAMs in Vensim " and the development implemented during it continues.

Using the plugin developed in this work, you can check quality standards, naming conventions and programming rules in Vensim models. In addition, the plugin allows communication with an external data dictionary that serves to keep a record of all the existing elements in the models. This record favors coordination in the development of the model, as well as communication with other stakeholders of the Locomotion project.

The Scrum framework has been used to manage the project, and it has been developed using Java with the Sonarqube and ANTLR4 libraries. The software obtained as a final result has been published on GitHub, in order to allow its use publicly.

Índice general

Lista de Figuras

Lista de fragmentos de código

Lista de Tablas

Capítulo 1

Introducción

1.1. Contexto

Este Trabajo Fin de Grado está realizado en convenio con GEEDS (GIR Grupo e Investigación en Energía, Economía y Dinámica de Sistemas) [13], grupo coordinador del proyecto europeo LOCOMOTION (Low-carbon society: an enhanced modelling tool for the transition to sustainability). [27]. El proyecto LOCOMOTION es el sucesor del proyecto MEDEAS [31], ambos pertenecientes al programa Horizon 2020 [12].

El alumno Daniel Bazaco Velasco publicó el curso pasado su TFG, llamado "Definición y comprobación de estándares de calidad en la programación de IAMs en Vensim" [4]. El cual, tiene como objetivo la creación de un *plugin* para la plataforma SonarQube [45] junto a la definición y desarrollo de un conjunto de reglas de calidad necesarias para la comprobación de convención de nombres y otras reglas para el lenguaje de programación Vensim[55] en el que está escrito el proyecto LOCOMOTION.

Tomando el TFG de Daniel Bazaco como predecesor, se tratará de ampliar el trabajo ya realizado por este alumno. Se seguirá desarrollando el *plugin* para SonarQube, teniendo en cuenta la sección "Líneas de trabajo futuras"descrita en el TFG de Daniel Bazaco.

1.1.1. Introducción a LOCOMOTION

LOCOMOTION es un IAM (Integrated Assessment Models) [5], es decir, un modelo que permite hacer simulaciones a nivel global para poder comprobar el efecto en diferentes aspectos de distintas políticas. Esto permite al interesado poder tomar decisiones teniendo de base un modelo que estima la factibilidad, eficiencia y coste de las medidas consideradas. Las estimaciones generadas pertenecerán a distintos ámbitos como el económicos, energético y tecnológico.

Este modelo es desarrollado apoyándose en otros modelos como pueden ser World6 [44], TIMES [18], LEAP [48], GCAM [40], C.Roads [9] y MEDEAS [31], su predecesor. El objetivo es realizar un modelo más robusto, útil y transparente.

El lenguaje principal en el que está desarrollado LOCOMOTION es Vensim, un lenguaje de modelado de dinámica de sistemas. El problema de este lenguaje es que es privativo y niega el acceso libre al uso del modelo, por eso, existe un proyecto interno en LOCOMOTION con el objetivo de transpilar [23], tanto el modelo, como el motor de simulación a python. Esta traspilación permitiría el acceso libre al código por cualquier usuario.

1.1.2. Introducción a Vensim

Vensim es un programa utilizado para la generación de modelos de dinámicas de sistemas. Destaca por su eficiencia, funcionalidad y flexibilidad, además cuenta con una interfaz gráfica.

Cada modelo generado está compuesto por símbolos, vistas y grupos. Un símbolo define un aspecto del modelo, puede tener una unidad, un comentario y una o varias ecuaciones que lo definen. Estas ecuaciones pueden a su vez contener más símbolos y de este modo generar una red de interconexiones.

En el momento de hacer la simulación del modelos, Vensim va realizando saltos de tiempo fijos en los cuales recalcula el valor de los símbolos y así ver la evolución del sistema.

Existen varios tipos de símbolos en Vensim, a continuación se encuentra una lista con los tipos de símbolos que son relevantes para LOCOMTION. [61]:

- **Variables auxiliares:**
  Cualquier símbolo que dependa del tiempo o de otros símbolos que sean variables.
- **Niveles:**

  Son variables dinámicas, que se calculan a partir de su valor en la iteración anterior de la simulación.
- **Datos:**
  Estas son variables independientes de las demás variables del modelo, sin embargo si que dependen de información externa al modelo. Se suelen utilizar para comparar el modelo respecto a la realidad.
- **Constantes:**

  Un símbolo cuyo valor no cambia respecto al tiempo. Aun así, estas variables pueden ser modificadas por el usuario durante la simulación.

  También existen un tipo especial de constantes llamadas Contantes inmutables, las cuales no pueden ser modificadas durante la simulación.

Por último, también podemos utilizar las constantes iniciales, las cuales tienen al inicio de la simulación su valor generado utilizando diversos símbolos del modelo. Una vez generadas se mantienen constantes en el tiempo.

- **Subscripts:**

  Son símbolos que funcionan como un diccionario, es decir, pueden almacenar pares de información clave-valor, que podrá ser recuperada más adelante mediante una clave. Pueden ser subconjuntos de otros *subscripts* o copias de los mismos, en este caso no pueden tener más valores propios o de un tercer *subscripts*.

- **Lockups:**

  Son funciones numéricas no lineales, en las cuales se definen los valores de x e y. Mediante estas funciones se puede obtener el valor asignado a y, si damos el valor de x.

- **Reality checks:**

  Estas ecuaciones se utilizan para aportar información adicional sobre el modelo y poder verificar la consistencia de unidades. Son añadidas a las ecuaciones del resto de tipos de símbolos.

Una vista es la representación gráfica de un subconjunto de símbolos del sistema. En cada vista, se pueden añadir los símbolos que el usuario desee y observar las conexiones entre ellos mediante flechas, los símbolos añadidos en una vista pueden ser de tipo `defined` o `primary` si en la vista se está definiendo como se genera el valor del símbolo, es decir, se muestra su ecuación mostrando todos los símbolos que aparecen en dicha ecuación, o de tipo `shadow` o `secondary` si solo se lee el valor del símbolo [56]. También se pueden añadir gráficas que muestran como se ha ido modificando el estado de uno a varios símbolos seleccionados.

Un grupo es una forma de agrupar a los símbolos en conjuntos que pueden tener relación semántica, su principal objetivo es mantener el código fuente organizado y poder realizar inspecciones de los resultados más rápidas, actualmente, solo existe un grupo en el código fuente de LOCOMOTION, el cual contiene las variables internas de Vensim [57].

1.1.3. Introducción a SonarQube

La plataforma web de código abierto llamada SonarQube [45], permite analizar de forma estática códigos escritos en diversos lenguajes de programación con el fin de evitar malas prácticas y *bugs* no detectados.

SonarQube utiliza una serie de reglas creadas por el usuario u obtenidas de un paquete por defecto. Cuando se activa una de estas reglas para el análisis, en el momento que falle generará un *issue*. Un *issue* puede indicar un problema en el código, su gravedad, la ubicación, el tipo y una posible forma de resolverlo.

Las reglas pueden ser agrupadas en *Quality Profiles*, los cuales se pueden automatizar para que se activen o desactiven dependiendo el tipo de archivo o manualmente. Esto se hace para desactivar masivamente reglas en masa sin necesidad de ir de una en una.

SonarQube es únicamente un servidor que gestiona y almacena los proyectos analizados. El propio análisis se hace por separado por un escáner, por ejemplo el oficial de Sonarqube llamado SonarScanner [46]. El escáner cuando es ejecutado se comunica con el servidor para cargar las reglas requeridas.

El escáner tiene una serie de parámetros que pueden ser modificados al gusto del usuario a la hora de realizar la ejecución, en la sección Manual de usuario del apéndice A se encuentra un ejemplo de esta parametrización. Una vez acaba, el escáner envía el informe a SonarQube que, además tiene un *dashboard* para donde se puede encontrar la colección de *issues* detectada y su evolución.

1.1.4. Introducción al diccionario de símbolos

A la vez que el *plugin* existe otro proyecto, el diccionario de símbolos [26], el cual tiene como objetivo servir de base de conocimiento de todos los elementos que existen el LOCOMOTION, estos pueden ser símbolos, pero también otro tipo de elementos como pueden ser módulos o categorías las cuales se explicarán a lo largo de la memoria. Este diccionario ha sido implementado por el grupo `CRES` con Panos Stravis como desarrollador principal, su estructura está basada en el TFG de David Gómez Pedriza llamado "Una arquitectura de microservicios Python como soporte a un diccionario de datos en un sistema de aseguramiento de la calidad y asistencia al desarrollo de IAMs por grandes equipos globalmente distribuidos"[17].

Este diccionario almacena toda la información relevante a cada elemento como pueden ser sus dependencias con otro elementos y sus características como por ejemplo un comentario descriptivo.

Con toda esta información se puede conseguir mantener consistencia entre los símbolos del modelo y así poder reutilizarlos manteniendo todas las características que originalmente tenía.

Cuenta con una API para poder hacer inyecciones y consultas [49] de forma automatizada la cual será y es utilizada por el *plugin*.

1.1.5. Introducción a las reglas de control del plugin

El *plugin* a desarrollar tendrá como una de sus principales funciones el comprobar la calidad de los modelos mediante el uso de un conjunto de reglas de control de calidad.

Estas reglas se utilizarán para comprobar el uso del convenio de nombrado en los distintos tipos de símbolos que existen, asegurar la consistencia de los elementos del modelo con los

almacenados en el diccionario de datos o por lo general, prevenir malas prácticas como el uso de números mágicos. Existe un documento de LOCOMOTION en el cual se establece el marco de trabajo del proyecto donde se puede encontrar el convenio de nombres entre otras prácticas a seguir [28].

En el *plugin* de partida existe una serie de reglas ya definidas e implementadas en función del marco de trabajo de LOCOMOTION. A continuación se expondrá una lista de estas reglas ya definidas, más adelante se expondrán también las nuevas a implementar.

Reglas de control de calidad ya implementadas en el plugin

Control de nombrado de las variables El nombre de las variables declaradas puede contener caracteres en minúscula, números y el caracter '_'. Pueden contener caracteres en mayúscula si y solo si pertenecen a un acrónimo aceptado por LOCOMOTION los cuales se encuentran almacenado en el diccionario de símbolos.

Control de nombrado de las constantes El nombre de las variables declaradas puede contener caracteres en mayúscula, números y el caracter '_'.

Control de nombrado de los Lookups Siguen las mismas reglas que las variables pero deben tener el sufijo '_lt'.

Control de nombrado de los Reality check Siguen las mismas reglas que las variables pero deben tener el sufijo '_check'.

Control de nombrado de los Subscript Siguen las mismas reglas que las constantes pero deben tener el sufijo '_I'.

Control de nombrado de los Subsccript value Siguen las mismas reglas que las constantes.

Control de uso de números mágicos No se deben utilizar números arbitrarios en las definiciones de los símbolos, es necesario crear una constante que exprese el valor de dicho símbolo. Existe una serie de números excluidos de esta regla por su utilidad inherente estos son: 0;1;2;3;4;5;6;7;8;9;-1;100. Esto se debe a su uso habitual para hacer inversiones de valores, exponentes o porcentajes.

Control de símbolos definidos en el diccionario Todos los símbolos utilizados en el modelo deben de estar almacenados en el diccionario de símbolos.

Control de existencia de comentario en los símbolos Todos los símbolos del modelos deben incluir un comentario descriptivo para ayudar con la compresión del modelo.

Control de existencia de unidad en los símbolos Todos los símbolos deben tener una unidad o expresar explícitamente que no tienen unidad, esto se realidad con la unidad especial 'Dmnl'.

Control de consistencia con el diccionario de datos Todos los símbolos utilizados en el modelos deben mantener la consistencia de su definición guardada en el diccionario de datos. La consistencia se debe mantener en su unidad, comentario y tipo. En caso de tratarse de un *subscript* debe mantener la consistencia de sus valores.

Reglas de control de calidad a implementar en este proyecto

Control de nombrado de las variables delayed Las variables de tipo *delayed*, siguen la misma regla que las variables normales pero deben tener el prefijo 'delayed_', si además la variable *delayed* depende del tiempo transcurrido en cada paso de la simulación deben tener el prefijo 'delayed_TS_'. La explicación más detallada de este tipo de variables puede ser encontrada en la sección implemetación de la historia de usuario 4.9.

Control de nombrado de las vistas Todas las vistas deben incluir un módulo, una categoría y opcionalmente una subcategoría. El separador por defecto entre módulo y categoría es '-' y entre categoría y subcategoría es '.'. En la sección de análisis 'Estructura, datos y control de flujo de la versión de partida' puede encontrarse una definición de los módulos y categorías.

Control de nombrado de las copias de los Subscripts Siguen las mismas reglas que las constantes pero deben tener el sufijo 'MAP_I'.

En la sección implementación de la historia de usuario 4.6 se puede encontrar una descripción de que son las copias de los *subscripts*.

Control de duplicación de las categorías No pueden existir dos categorías ni subcategorías que compartan el mismo nombre, es decir, no puede haber ni dos categorías con el mismo nombre, ni dos subcategorías ni una categoría y una subcategoría.

La explicación de esta regla puede ser encontrada en la sección implementación de la historia de usuario 4.8.

Control de consistencia de las referencias Excel de los símbolos con el diccionario de datos Se debe mantener la consistencia entre el diccionario de símbolos y el modelo de todas las referencias a fichero de excel que contengan los símbolos.

La descripción de esta regla puede ser encontrada en la sección de implementación de la historia de usuario 6.5

Control de uso de unidades válidas Todos los símbolos deben utilizar una de las unidades válidas definidas en el diccionario de símbolos, con la excepción de no tener unidad que en ese caso deben de tener como unidad 'Dmnl'.

Control de definición de los símbolos en el grupo apropiado Todos los símbolos deben pertenecer a su grupo apropiado, en este caso solo los símbolos especiales para la configuración de la simulación de Vensim deben pertenecer al grupo `Control`, el resto no deben tener grupo.

Control de prevención de declaración de Lookups incrustados No se debe incrustar la información de un *lookup* en el propio modelo ya que esta puede contener cientos de relaciones. Es necesario realizar una referencia a una tabla externa almacenada en un fichero excel.

1.2. Justificación del proyecto

El motivo del siguiente TFG reside en la necesidad que le surge al proyecto europeo LOCOMOTION de controlar la calidad del código que se desarrolla en el proyecto. Esto se debe a que el equipo que trabaja en este proyecto es multidisciplinar y puede tener diferentes perspectivas sobre la forma de programar. Para prevenirlo se suelen utilizar herramientas de evaluación automática de posibles malas prácticas en el código.

Actualmente, existe la herramienta desarrollada en el proyecto de partida de este TFG, esta cuenta con una serie de comprobaciones del las posibles malas prácticas. En este TFG se ampliará esta lista de comprobaciones haciendo asi una herramienta más completa que permita detectar problemas en más zonas del modelo.

No existe una herramienta pública capaz de hacer esta evaluación automática para Vensim, por lo que también podría servir para otros proyecto. Para permitir su uso público, se priorizará la utilización de parámetros para poder adecuar el funcionamiento del analizador al gusto del usuario final. Esto permitiría adaptarse a las convenciones establecidas en otros proyectos.

A parte, también existe la necesidad de generar un registro automático con todos los elementos existentes en el proyecto LOCOMOTION. En la actualidad el modelo cuenta con varios cientos de símbolos, módulos, categorías, índices, etc. Cada uno de ellos con sus características internas. Toda esta información sería imposible de ser manejada sin llevar un registro de que elementos existen y de como han ido evolucionando en el tiempo.

Existe actualmente el diccionario de símbolos [26] en el cual se inyectan de forma automática lo símbolos detectados gracias al *plugin* que existe actualmente. Esta información debe se extendida de forma manual añadiendo todas las categorías, índices, y módulos que existen en el modelo además de características extra de cada símbolo. Este TFG ampliará esta automatización para evitar tener que añadir todos estos elementos de forma manual y además ampliará la información inyectada para poder mantener un registro lo más amplio posible de todas las características de los elementos del modelo.

1.3. Objetivos

El objetivo principal de este Trabajo Fin de Grado es continuar con el desarrollo de un *plugin* para SonarQube con el que podamos analizar la calidad de los modelos (archivos terminados en .mdl) de la aplicación Vensim. Algunos ejemplos de comprobaciones de la calidad pueden ser: nombrado correcto de símbolos, coherencia con el diccionario de símbolos, prevenir de “números mágicos”, etc.

Para conseguir este objetivo he definido varios objetivos:

1. Realizar el análisis y estudio del TFG predecesor realizado por Daniel Bazaco Velasco [4] para conocer los flujos y la estructura del código. Además en el Capítulo 4 se realizará

un análisis de la situación de partida, exponiendo algunos conceptos que hayan podido quedar en dudas del TFG predecesor.

2. Ampliación de los estándares de calidad para los modelos Vensim en colaboración con el grupo GEEDS. Al tratarse de un hito que al inicio del TFG es desconocido, la lista de todas las ampliaciones hechas durante todo el proyecto se pueden encontrar en la tabla 2.6

3. Implementación en el *plugin* de los estándares no implementados en el TFG anterior y de los estándares de nueva creación.

4. Mantenimiento del código base llevando a cabo refactorizaciones cuando fuera necesario. Esto podría conllevar a la reestructuración de ciertas partes del código para facilitar la comprensión.

   Un objetivo secundario de este trabajo es crear un marco de trabajo que involucre el funcionamiento tradicional de realización de un TFG con el marco de trabajo ágil *Scrum*. Esto se explicará en la Sección 2.1 Este marco de trabajo es necesario por el motivo de que los objetivos 2 y 3 son difusos y no se sabe su alcance, por lo que es necesaria una gestión y monitorización contante de los estándares que se desean crear e implementar.

1.4. Estructura de la memoria

Este documento se estructura de la siguiente forma:

Capítulo 2 Requisitos y planificación: Describe el marco de desarrollo ágil utilizado y su adaptación al proyecto. También la lista de requisitos e historias de usuario del proyecto. Adicionalmente se documenta el análisis de riesgos detectados con su explicación. Por último, se describen la planificación y el presupuesto del proyecto.

Capítulo 3 Tecnologías utilizadas: Describe las tecnologías utilizadas para la gestión del TFG y para el desarrollo del proyecto.

Capítulo 4 Análisis: Describe el análisis realizado de las diversas secciones del TFG de partida además de los principales cambios detectados que se deberán realizar.

Capítulo 5 Diseño: Describe las estructuras que tendrán los diferentes elementos del proyecto como pueden ser archivos que se generan, comunicaciones con APIs externas o la jerarquía de clases y paquetes del proyecto.

Capítulo 6 Implementación y pruebas: Describe detalladamente todas las implementaciones realizadas fragmentadas en la historia de usuario a la que pertenece. Además se explican las prácticas realizadas para mantener la calidad del código como pueden ser refactorizaciones, eliminación de malas prácticas o mantener una buena cobertura de los tests.

Capítulo 7 Seguimiento del proyecto: Describe *sprint* a *sprint* la tareas que había que hacer y un resumen de lo que se ha hecho en cada sprint. También tiene un resumen del proyecto en sí teniendo en cuenta el total de horas y costes finales.

Capítulo 8 Conclusiones: Describe un resumen de todo el proyecto, analiza qué cosas se han hecho bien y cuales mal y una reflexión en general del TFG. Además incluye líneas de trabajo futuras y los usos reales que ya se ha dado a este proyecto.

Anexo A Manuales: Incluye manuales de mantenimiento, de instalación, despliegue, y de uso.

Anexo B Resumen de enlaces adicionales: Incluye enlaces de interés sobre el proyecto, como el repositorio de código, el diccionario de símbolos o la instancia de SonarQube utilizada en LOCOMOTION.

Anexo C Estructura de los endpoints del diccionario de símbolos: Incluye las estructuras utilizadas para realizar las comunicaciones con el diccionario de símbolos.

Capítulo 2

Requisitos y Planificación

2.1. Marco de desarrollo ágil Scrum

Para el desarrollo de este proyecto se ha seguido el marco de desarrollo ágil *Scrum*. Podemos definir *Scrum* como un *framework* (marco de trabajo) para la gestión de productos, proyectos y servicios complejos que facilita un desarrollo mantenido e incremental [24].

Se decidió usar *Scrum* porque al ser un marco de desarrollo ágil nos resultaría más sencillo implementar cambios, debido a unos requisitos iniciales poco específicos y que pueden ir cambiando a lo largo del proyecto según varíen las necesidades del cliente. Otro de los motivos para utilizar este marco de trabajo son los resultados obtenidos cada poco tiempo que pueden ser entregados al cliente para su aceptación o propuesta de cambios.

A continuación se definirán los artefactos, roles y eventos de*Scrum*.

2.1.1. Artefactos

Durante el proyecto se crean tres artefactos [24], que son:

- ***Product Backlog*:**
  Lista de funcionalidades, productos o acciones que conforman el producto a crear. Se compone de historias de usuario que pueden irse completando, detallando o añadiendo más historias de usuario a medida que se va desarrollando el proyecto y priorizando unas sobres otras.
- ***Sprint Backlog*:**
  Lista de funcionalidades o tareas seleccionadas del *Product Backlog* que se incorporan al Sprint actual para llevarse a cabo durante el mismo. Una vez a comenzado el *Sprint* (Sprint se define en el apartado 2.1.3), el *Sprint Backlog* no puede ser modificado.

- ***Incremento*:**
  Resultado del Sprint que incrementa el valor del producto. Se conforma por todas las tareas, escenarios, historias de usuario y cualquier elemento que se haya desarrollado durante el Sprint.

2.1.2. Roles

Scrum tiene tres roles [37], que son:

- ***Product Owner*:**
  El *Product Owner* es el responsable del producto final, se ocupa de decidir cuáles serán las tareas a realizar para maximizar el valor del producto. Esto lo lleva a cabo generando el *Product Backlog*, es el único que puede editar dicho artefacto.

  El valor del producto es subjetivo y dependerá de qué producto se desarrolla. Por ejemplo en el caso de un producto comercial, el valor se podría ver como el ROI (*Return of Investment*) del producto. En cualquier caso, será decisión del *Product Owner* cuál es el valor a maximizar y cómo lograrlo.

- ***Scrum Master*:**
  El *Scrum Master* vela por el buen funcionamiento del proyecto, ayudando y enseñando las prácticas de *Scrum* y protegiendo al equipo de interferencias externas. Se asegura de que todo el mundo entiende y sigue las guías de *Scrum*. Hay que entender que no es el gestor del proyecto, el *Scrum Master* no manda qué hacer a nadie, ya que es el propio equipo el que decide en qué se trabaja.

- **Equipo de desarrollo:**
  El Equipo de desarrollo (*development team*) es el responsable de construir el producto que especifica el *Product Owner*. Es un equipo multidisciplinar formado habitualmente de 5 a 9 personas, con la capacidad de poder realizar todas las tareas necesarias para el desarrollo, desde hacer análisis, hasta las implementaciones y despliegues. Son un equipo auto gestionado y pueden funcionar de forma independiente.

  Para llevar el desarrollo a cabo, el equipo decide la cantidad de tareas del *Product Backlog* (respetando el orden dictado por el *Product Owner* en la medida que se pueda) que se van a hacer en el *Sprint*. Estas tareas son añadidas al *Sprint Backlog*.

2.1.3. Eventos

Scrum tiene cinco eventos [43] que ocurren durante el ciclo de vida del proyecto. Estos eventos tienen una duración previamente fijada y se realizan periódicamente en unas fechas marcadas para facilitar la comunicación eficaz entre los miembros del equipo. Estos eventos son:

- ***Sprint*:**
  Es un ciclo de trabajo de menos de 4 semanas, normalmente de 2 semanas. Se hacen

de forma continuada durante todo el proyecto, es decir nada más acaba uno empieza el siguiente. En cada *Sprint* se realiza un *Sprint Backlog* y una vez ha llegado la fecha fin de dicho Sprint se cierra, se hayan acabado todas las tareas o no. Si han quedado tareas pendientes, será decisión del equipo qué hacer. Algunas opciones son: Pasar la tarea automáticamente al siguiente *Sprint* o devolver la tareas al *Product Backlog* para más adelante.

- ***Sprint Planning*:**
  Cada Sprint necesita su *Sprint Backlog*, este artefacto es generado en el *Sprint Planning*, en estas reuniones de aproximadamente dos horas se reúne todo el equipo (el *Product Owner* es opcional pero recomendable) y se decide qué llevar a cabo en el Sprint. Una vez decidido, la reunión se centra en cómo realizar dichas tareas, realizar estimaciones de tiempo y si fuese necesario añadir o eliminar alguna de las tareas previamente aceptadas.

- ***Daily Scrum*:**
  Son reuniones típicamente diarias de máximo 15 minutos, suelen hacerse de pie para fomentar que se realice de forma rápida y efectiva. Su objetivo principal es la coordinación entre los miembros del equipo de desarrollo. No debería de haber decisiones extendidas sino solo sincronizar trabajos y prever futuros obstáculos que puedan ocurrir.

- ***Sprint Review*:**
  Una vez ha acabado el *Sprint* se realiza esta reunión de aproximadamente 1-2 horas. En ella de analiza el incremento de funcionalidad del producto que se ha llevado a cabo en el Sprint. En dicha reunión puede asistir gente externa del equipo de desarrollo y preguntar dudas sobre el incremento. También esta reunión sirve para que el *Product Owner* aprenda de la situación actual y como deberá enfocar o adaptar el *Product Backlog* para el futuro. Si hubiese que resumir esta reunión en dos palabras serían: Inspección y adaptación.

- ***Sprint Retrospective*:**
  Reunión que se hace acto seguido del *Sprint Review*, con una duración de alrededor de hora y media. Es muy similar a su reunión predecesora, pero con una diferencia fundamental, si el *Sprint Review* se centraba en el producto, el *Sprint Retrospective* se centra en el proceso de desarrollo. Es un momento para que el equipo discuta qué problemas existen y qué cosas no se están haciendo de forma eficiente y qué soluciones se proponen. Pero no solo hay que centrarse en los problemas, también es conveniente comunicar qué medidas sí funcionan y hacen que el trabajo sea más eficiente. Esta reunión es una buena oportunidad para que el *Scrum Master* tome note de posibles problemas y actúe ante ellos para actuar como facilitador para el resto del equipo.

2.2. Adaptación de Scrum al proyecto

Scrum es un marco de trabajo abierto que puede ser adaptado para cada proyecto, se puede concretar qué individuos toman el papel de cada rol, la duración de los *Sprints*, las características de las reuniones, etc.

Este proyecto involucra únicamente a dos personas: la tutora (Yania Crespo) y el alumno (Juan Herruzo). Estas condiciones no son las ideales para poder aplicar *Scrum* y por ello se realizará una adaptación de este marco de trabajo.

La tutora (Yania Crespo) conoce más en profundidad el proyecto LOCOMOTION, al formar parte del equipo del mismo. Ella hará de intermediaria entre las directrices y requerimientos pedidos por GEEDS y el equipo de este proyecto. Por lo tanto, tendrá el rol de *Product Owner*.

Además, la tutora cuenta con una gran experiencia en el ámbito de las metodologías ágiles y en *Scrum* al haber dirigido varios proyectos utilizando este marco de trabajo. Por lo tanto, también realizará el rol de *Scrum Master* a pesar de no considerarse una buena práctica según *Scrum*.

Por último, el equipo de desarrollo estará compuesto únicamente por el estudiante Juan Herruzo, el autor de este TFG.

Cada Sprint tendrá una duración de dos semanas y, se ha considerado que no es necesario hacer una *daily* (reunión diaria). Por ello se realizará una *weekly*, es decir una reunión a la semana y se utilizará la plataforma de mensajería instantánea *rocket* para poder comunicarse en cualquier otro momento.

En cada Sprint se realizará un *Sprint Planning* al inicio del mismo, una *weekly* a la mitad y un *Sprint Review* y *Sprint Retrospective* al finalizar el *Sprint*.

2.3. Requisitos

Los requisitos mostrados en la Tabla 2.1 son los requisitos que han sido recolectados a lo largo del proyecto.

Tabla 2.1: Tabla de requisitos.

Código	**Nombre**	**Descripción**
R1	**Generación de una nueva gramática que permita el análisis de las *views* de Vensim.**	Ampliación de la gramática con el objetivo de poder analizar léxica y sintácticamente la sección del modelo donde se definen las vistas. Este análisis debe contener al menos la información relativa al nombre (módulo, categoría y subcategoría) e información sobre qué símbolos forman parte de la vista. Se diferenciará entre cuáles son símbolos principales y cuáles son símbolos secundarios o *shadows*.

Continúa en la página siguiente

Tabla 2.1 – *Viene de la página anterior*

Código	Nombre	Descripción
R2	**Filtrado de los símbolos del modelo mediante el módulo al que pertenecen.**	Creación de la funcionalidad para poder seleccionar qué símbolos del modelo pueden tener *issues* en la ejecución del control de calidad. Los símbolos seleccionados son aquellos que tienen como módulo principal aquel por el que se desea filtrar.
R3	**Actualización de la regla sobre el nombrado de variables para tener en cuenta los acrónimos.**	Actualización de la regla de nombrado de las variables para que el uso de los acrónimos guardados en el diccionario de datos no genere *issues*.
R4	**Parametrización de las características mutables de las reglas.**	Actualización de las reglas que contengan una característica modificable, concretamente, expresiones regulares y números, generar propiedades de estas características que puedan ser modificadas con *Quality profiles* de SonarQube.
R5	**Inyección en el diccionario de datos de los módulos detectados como nuevos .**	Inyección en el diccionario de datos de los módulos que han sido detectados como nuevos, lo módulos que ya estén almacenados en el diccionario de símbolos no deben ser reinyectados.
R6	**Inyección en el diccionario de símbolos de categorías y subcategorías detectadas como nuevas.**	Inyección en el diccionario de símbolos de categorías y subcategorías que han sido detectados como nuevas, las categorías y subcategorías que ya estén almacenados en el diccionario de símbolos no deben ser reinyectados. Si se detecta una subcategoría nueva de una categoría ya almacenada en el diccionario, solo se deberá inyectar la subcategoría.
R7	**Inyección de las referencias a tablas Excel externas detectadas nuevas al diccionario de datos.**	Inyección en el diccionario de datos de las referencias a tablas Excel externas que han sido detectados como nuevas tiendo la referencia al símbolo al que pertenecen, las referencias a tablas Excel externas que ya estén almacenados en el diccionario de símbolos no deben ser reinyectados.

Continúa en la página siguiente

Tabla 2.1 – *Viene de la página anterior*

Código	Nombre	Descripción
R8	**Validación de las inyecciones.**	Limitar la inyección al diccionario de símbolos a elementos considerados como válidos por el análisis, es decir, que no hayan generado ninguna *issue* y que además no existan ya en el diccionario de símbolos. Este requisito se aplica a cualquier elemento que haya que subir al diccioaniro de símbolos.
R9	**Actualización de la inyección de símbolos.**	Ampliación de la inyección de símbolos para que contengan la nueva información sobre sus módulos y categorías.
R10	**Creación de una nueva regla que detecte *lookups* incrustados.**	Creación de una regla que genere una *issue* cuando se detecten *lookups* que estén incrustados en el modelo. Un lookup incrustado hace referencia a que los puntos que definen el lookup se encuentran declarados en el propio código. La *issue* generada debe tener una gravedad de `INFO` si la cantidad de datos incrustados es menor o igual a 4 o `MAJOR` si es mayor a este número. Este número debe de ser una propiedad modificable en SonarQube.
R11	**Creación de una nueva regla que detecte la declaración de símbolos en el grupo de control.**	Creación de una nueva regla que genere una *issue* cuando exista un símbolo que no sea de control declarado en la zona de control generada por defecto por Vensim, también se debe generar una *issue* en el caso en el que un símbolo de control no esté declarado en esta zona. Los símbolos de control por defecto son: TIME, TIME STEP, INITIAL TIME, FINAL TIME, SAVEPER. Esta lista de símbolos deben de ser una propiedad modificable en SonarQube.
R12	**Creación de una nueva regla que compruebe el nombre de las vistas.**	Creación de una nueva regla que genere una *issue* cada vez que el nombre de una vista no siga las directrices de la convención de nombres.

Continúa en la página siguiente

Tabla 2.1 – *Viene de la página anterior*

Código	Nombre	Descripción
R13	**Creación de una nueva regla que detecte fallos en las unidades de cada símbolo.**	Creación de una nueva regla que genere una *issue* cada vez que las unidades de un símbolo no estén contenidas en la lista de unidades almacenada en el diccionario de datos del proyecto. Existe una excepción, si un símbolo no tiene unidades se declarará con la unidad `Dmnl` (dimensionless).
R14	**Creación de una nueva regla que revise el nombre de las copias de los *subscripts*.**	Creación de una nueva regla que genere una *issue* cada vez que el nombre de un *subscript copy* no siga las directrices de la convención de nombres.
R15	**Creación de una nueva regla que revise la información sobre las referencias a tablas Excel externas de cada símbolo.**	Creación de una nueva regla que genere una *issue* cada vez que la información sobre las referencias a tablas Excel externas detectada en local y la almacenada en el diccionario de datos no sean idénticas. Esta *issue* se generará sobre el símbolo al que pertenece la referencia externa.
R16	**Creación de una nueva regla que compruebe que no existen categorías o subcategorías con el mismo nombre**	Creación de una nueva regla que genere una *issue* en las subcategorías que tengan el mismo nombre que otras subcategorías o que otras categorías, el nombre de cada categoría y subcategoría tiene que ser único.
R17	**Creación de una nueva regla que compruebe que se cumple la convención de nombres en las variables delayed**	Creación de una nueva regla que genere una *issue* en las variables de tipo *delayed* (las cuales son las que utilizan las funciones `DELAYED` o `SMOOTH` y sus derivadas) cuando no cumplan con la convención de nombres. Por convenio estas variables deberán tener como prefijo —delayed_— a no ser que estén vinculadas con la contante `TIME STEP` en cuyo caso deberán tener el prefijo —delayed_TS_—
R18	**Generación de documentación que contenga las diferencias encontradas entre los símbolos de modelo programado en el .mdl y los símbolos del diccionario de datos.**	Generación de un documento *json* que contenga todas las diferencias que se han detectado entre los elementos locales y los almacenados en el diccionario de datos del proyecto. El nombre de este documento será *dictionaryDiff.json*. Este documento se generará si y solo si, ha sido especificado en las propiedades del sonar scanner.

Continúa en la página siguiente

Tabla 2.1 – *Viene de la página anterior*

Código	Nombre	Descripción
R19	**Actualización de la documentación generada al ejecutar sonar scanner.**	Actualización del documento json generado con nombre `symbolTable.json`, se debe añadir como mínimo, información relativa al módulo, categoría y subcategoría de cada símbolo y las referencias a tablas excel externas de los símbolos que lo requieran.
R20	**Creación automática de una carpeta que contenga los archivos generados por el *plugin* y la parametrización de su nombre.**	Creación de una carpeta en la cual se guardarán todos los documentos generados en local al realizar el escáner. Esta carpeta debe estar ubicada en la ruta donde se ejecuta dicho escáner y debe tener por defecto el nombre *auxiliary_files*. Este nombre debe de ser modificable utilizando una propiedad de sonar scanner.
R21	**Ampliación del conjunto de propiedades del archivo de configuración de sonar scanner.**	Ampliación del conjunto de propiedades existentes en el archivo de configuración de sonar scanner, como mínimo se debe añadir la opción de inyectar o no los elementos nuevos detectados.
R22	**Mantenimiento del alcance actual del plugin**	Mantenimiento de todas las funcionalidades y reglas actuales del *plugin*, se deben modificar las funcionalidades del *plugin* inicial para que sigan funcionando si cambian aspectos críticos del *plugin*.

2.4. Product Backlog

Utilizando los requisitos de la Tabla 2.1, se genera un *Product Backlog* inicial para poder ser utilizado en el marco de trabajo *Scrum* adaptado al proyecto.

El Product Backlog contiene historias de usuario, la cuales explican de forma general una funcionalidad del software escrita desde la perspectiva del usuario final o cliente. La estructura general de una historia de usuario es: “Como ⟨rol del cliente⟩, quiero ⟨intención u objetivo del cliente⟩ para ⟨beneficio que le aportará al cliente⟩.”.

Si una historia de usuario tiene una complejidad demasiado grande como para ser afrontada en un *Sprint* esta pasará a ser denominada una épica o *epic* en inglés y es recomendable dividirla en historias de usuario más pequeñas.

En el *plugin* solo existe un rol, el de un programador Vensim en el proyecto LOCO-

MOTION o en cualquier otro proyecto que adopte una serie de convenciones y reglas de programación, este rol será referenciado con la palabra "usuario".

En la Tabla 2.2 se puede encontrar la lista de historias de usuario ordenadas de mayor a menor prioridad, esta ordenación se realizó en conjunto con la tutora del TFG, Yania Crespo, al inicio del proyecto, actuando como Product Owner. Este *Product Backlog* inicial intenta estructurar las ideas generales de los requisitos en grupos para empezar a comprender los flujos de desarrollo que existirán en el futuro, es necesario hacer una revisión y subdividir las historias en otras más pequeñas para así representar con una mayor granularidad la lista de requisitos existente.

Código	**Descripción**
US1	Como usuario quiero poder seguir realizando todas las acciones que se pueden realizar con el *plugin* en su estado inicial.
US2	Como usuario quiero poder analizar la representación textual de la parte de las vistas de los modelos de Vensim almacenada en archivos `.mdl` para verificar que la sintaxis es correcta.
US3	Como usuario quiero poder filtrar que símbolos generan *issues* mediante el módulo al que pertenecen.
US4	Como usuario quiero recibir más avisos sobre errores en el modelo de los que existen actualmente para poder mejorar la calidad del modelo de Vensim.
US5	Como usuario quiero poder configurar partes del funcionamiento del escáner.
US6	Como usuario quiero que todos los elementos nuevos sean inyectados al diccionario de datos del proyecto para poder llevar un registro de ellos.

Tabla 2.2: Product Backlog inicial.

La historia de usuario US1 no es una historia realizable en un Sprint en sí, sino que deberá ser hecha durante todo el desarrollo. Se define para prevenir que se puedan realizar modificaciones el en código original sin una justificación. Viene siendo similar a una restricción en la descripción de requisitos funcionales, de información, no funcionales y restricciones.

Las historias US4, US5 y US6 son consideradas épicas y es necesario subdividirlas en historias de usuario menos complejas, pero que sigan cumpliendo los requisitos establecidos. Estas subdivisiones se fueron realizando a lo largo del proyecto.

Las subdivisiones que se consiguieron al final pueden ser encontradas en la Tabla 2.3 para la historia de usuario US4, en la Tabla 2.4 para la historia de usuario US5 y en la Tabla 2.5 para la historia de usuario US6.

Código	Descripción
US4.1	Como usuario quiero que no se generen *issues* si una variable sigue el convenio de nombres, pero contiene en su nombre un acrónimo definido en el diccionario de datos del proyecto.
US4.2	Como usuario quiero ser informado si existe algún símbolo de tipo *lookup* el cual tenga incrustada su declaración en el modelo.
US4.3	Como usuario quiero ser informado si existe algún símbolo dentro del grupo de control que no sea de control y viceversa.
US4.4	Como usuario quiero ser informado si existe alguna vista que no siga las reglas de nombrado.
US4.5	Como usuario quiero ser informado si existe algún símbolo cuya unidad no se encuentre en el sistema de unidades definido en el diccionario de datos del proyecto.
US4.6	Como usuario quiero ser informado si existe alguna copia de un símbolo de tipo *subscript* y no siga las reglas de nombrado.
US4.7	Como usuario quiero ser informado si existe alguna discrepancia entre las referencias a tablas Excel encontradas en local y las referencias guardadas en el diccionario de datos del proyecto.
US4.8	Como usuario quiero ser informado si existe alguna subcategoría cuyo nombre no sea único en el conjunto de categorías y subcategorías.
US4.9	Como usuario quiero ser informado si existe alguna variable de tipo *delayed* que no cumpla la convención de nombres.

Tabla 2.3: División de la historia de usuario 4.

Código	Descripción
US5.1	Como usuario quiero poder configurar la regla de nombrado utilizada en cada regla que contenga una.
US5.2	Como usuario quiero poder configurar lo límites numéricos en cada regla que contenga uno.
US5.3	Como usuario quiero poder decidir cuáles serán los separadores utilizados en el nombre de una vista para diferenciar módulo, categoría y subcategoría.
US5.4	Como usuario quiero poder decidir si se inyectan nuevos elementos al diccionario de símbolos del proyecto o no.

Tabla 2.4: División de la historia de usuario 5.

Código	Descripción
US6.1	Como usuario quiero que los módulos nuevos detectados sean inyectados al diccionario de símbolos del proyecto.
US6.2	Como usuario quiero que los símbolos inyectados contengan el módulo al que pertenecen (módulo principal).
US6.3	Como usuario quiero que las categorías y subcategorías nuevas detectadas sean inyectadas al diccionario de símbolos del proyecto.
US6.4	Como usuario quiero que los símbolos inyectados contengan la categoría y subcategoría a la que pertenecen.
US6.5	Como usuario quiero que los símbolos inyectados contengan las referencias a tablas Excel externas cuando existan en dicho símbolo.
US6.6	Como usuario quiero que los índices se inyecten y se recuperen de forma independiente al resto de símbolos.

Tabla 2.5: División de la historia de usuario 6.

Durante el desarrollo se añadió una nueva historia de usuario US7, por ello también se añadieron algunas historias de usuario extra como la historia US5.5 y US5.6

OJO En la tabla 2.6 se muestra como quedó el *Product Backlog* después de subdividir las historias de usuario.

Código	Descripción
US1	Como usuario quiero poder seguir realizando todas las acciones que se pueden realizar con el *plugin* en su estado inicial.
US2	Como usuario quiero poder analizar la representación textual de la parte de las vistas de los modelos de Vensim almacenada en archivos `.mdl` para verificar que la sintaxis es correcta.
US3	Como usuario quiero poder filtrar qué símbolos generan *issues* mediante el módulo al que pertenecen.
US4.1	Como usuario quiero que no se generen *issues* si una variable sigue el convenio de nombres, pero contiene en su nombre un acrónimo definido en el diccionario de datos del proyecto.
US4.2	Como usuario quiero ser informado si existe algún símbolo de tipo *lookup* el cual tenga incrustada su declaración en el modelo.
US4.3	Como usuario quiero ser informado si existe algún símbolo dentro del grupo de control que no sea de control y viceversa.

US4.4	Como usuario quiero ser informado si existe alguna vista que no siga las reglas de nombrado.
US4.5	Como usuario quiero ser informado si existe algún símbolo cuya unidad no se encuentre en el sistema de unidades definido en el diccionario de datos del proyecto.
US4.6	Como usuario quiero ser informado si existe alguna copia de un símbolo de tipo *subscript* y no siga las reglas de nombrado.
US4.7	Como usuario quiero ser informado si existe alguna discrepancia entre las referencias a tablas Excel encontradas en local y las referencias guardadas en el diccionario de datos del proyecto.
US4.8	Como usuario quiero ser informado si existe alguna subcategoría cuyo nombre no sea único en el conjunto de categorías y subcategorías.
US4.9	Como usuario quiero ser informado si existe alguna variable de tipo *delayed* que no cumpla la convención de nombres.
US5.1	Como usuario quiero poder configurar la regla de nombrado utilizada en cada regla que contenga una.
US5.2	Como usuario quiero poder configurar lo límites numéricos en cada regla que contenga uno.
US5.3	Como usuario quiero poder decidir cuáles serán los separadores utilizados en el nombre de una vista para diferenciar módulo, categoría y subcategoría.
US5.4	Como usuario quiero poder decidir si se inyectan nuevos elementos al diccionario de símbolos del proyecto o no.
US5.5	Como usuario quiero poder decidir el nombre de la carpeta donde se generarán los documentos auxiliares.
US5.6	Como usuario quiero poder decidir si se genera el archivo auxiliar con las diferencias encontradas entre local y el diccionario de símbolos.
US6.1	Como usuario quiero que los módulos nuevos detectados sean inyectados al diccionario de símbolos del proyecto.
US6.2	Como usuario quiero que los símbolos inyectados contengan el módulo al que pertenecen.
US6.3	Como usuario quiero que las categorías y subcategorías nuevas detectadas sean inyectadas al diccionario de símbolos.
US6.4	Como usuario quiero que los símbolos inyectados contengan la categoría y subcategoría a la que pertenecen.
US6.5	Como usuario quiero que los símbolos inyectados contengan las referencias a tablas excel externas cuando existan en dicho símbolo.

US6.6	Como usuario quiero que los índices se inyecten y se recuperen de forma independiente al resto de símbolos.
US7	Como usuario quiero que al ejecutar un escáner se generen documentos auxiliares que contengan información relativa a los elementos que contiene el modelo.

Tabla 2.6: Product Backlog final.

2.5. Riesgos

Tanto al inicio del proyecto, como también durante su ejecución, se ha ido realizando un análisis de los posibles riesgos que incumben a este proyecto. Cada riesgo ha sido caracterizado en función de su probabilidad y de su impacto en una escala del 1 al 10, además de la importancia como es el producto de la probabilidad y el impacto. [38]. Dependiendo de la importancia, probabilidad e impacto se categorizarán en prioritarios, principales y secundarios. Cada riesgo además tiene asignada una estrategia a seguir elegida teniendo en cuenta la importancia de cada uno de ellos.

Riesgo 1	**Comprensión del proyecto de partida**
Descripción	Debido a que este TFG parte de otro Trabajo, existe el riesgo de que no se comprenda correctamente su desarrollo y conlleve una mala comprensión de la funcionalidad real de plugin.
Amenaza u Oportunidad	Amenaza
Probabilidad	3
Impacto	8
Importancia (P*I)	24
Categoría	Secundario
Estrategia	Aceptado. Se reservará un Sprint en exclusiva a la lectura y comprensión del TFG de partida.

Tabla 2.7: Riesgo “Comprensión del proyecto de partida”.

Riesgo 2	**Flexibilidad horaria del equipo de trabajo**
Descripción	Durante del desarrollo del TFG no existirá una carga extra de asignaturas de otros años, por lo que se podrá dedicar más tiempo a su desarrollo.
Amenaza u Oportunidad	Oportunidad
Probabilidad	9
Impacto	3
Importancia (P*I)	27
Categoría	Secundario
Estrategia	Aprovechado. Se solicitará el reconocimiento de 6 créditos por otras actividades que permitirá no realizar la única asignatura impartida en el segundo cuatrimestre para aumentar la flexibilidad del horario.

Tabla 2.8: Riesgo “Flexibilidad horaria del equipo de trabajo”.

Riesgo 3	**Dificultad de comprensión del código de partida**
Descripción	Se parte de una gran base de código, de la cual hay que entender tanto la estructura como el flujo de datos que existe. Durante el estudio del código puede descubrirse que sea más complejo o extenso de lo esperado.
Amenaza u Oportunidad	Amenaza
Probabilidad	8
Impacto	8
Importancia (P*I)	64
Categoría	Prioritario
Estrategia	Mitigado. Aparte de dedicar un Sprint inicial al estudio del TFG de partida, se dispondrá de contacto con el desarrollador del Trabajo anterior y se generarán los diagramas que sean necesarios para facilitar la comprensión.
Plan de contingencia	En caso de que no se comprenda la estructura o el flujo de datos del código de partida, se deberá: ▪ Consultar nuevamente los diagramas, tanto del TFG de partida, como de los nuevamente generados. ▪ Consultar a la tutora, Yania Crespo. ▪ Consultar al desarrollador del TFG de partida, Daniel Bazaco.

Tabla 2.9: Riesgo “Dificultad de comprensión del código de partida”.

Riesgo 4	**Cambio en la estructura del archivo Vensim**
Descripción	Los desarrolladores del software de simulación, Vensim podrían modificar la estructura del fichero del modelo, haciendo que la gramática desarrollada quede obsoleta.
Amenaza u Oportunidad	Amenaza
Probabilidad	1
Impacto	10
Importancia (P*I)	10
Categoría	Secundario
Estrategia	Aceptado. Se fijará una versión oficial en el proyecto LOCOMOTION para evitar el riesgo durante el TFG. Si se modificase la estructura y se quisiese utilizar el *plugin* nuevamente, habría que rehacer la gramática para que pudiese volver a ser utilizable.

Tabla 2.10: Riesgo "Cambio en la estructura del archivo Vensim".

Riesgo 5	**Gramática no completa**
Descripción	Vensim tiene una gran cantidad de funcionalidades, puede que alguna de ellas no estén previstas en la gramática desarrollada.
Amenaza u Oportunidad	Amenaza
Probabilidad	9
Impacto	7
Importancia (P*I)	63
Categoría	Prioritario
Estrategia	Mitigado. Se utilizará la documentación de Vensim para cubrir la mayor parte de las funcionalidades que ofrece.
Plan de contingencia	En caso de que se descubra una funcionalidad sin cubrir, se modificará la gramática para que se tenga en cuenta.

Tabla 2.11: Riesgo "Gramática no completa".

Riesgo 6	**Historia de usuario mal definida**
Descripción	Al analizar una nueva funcionalidad a implementar, puede comprenderse de manera errónea y llevar a cabo un desarrollo no útil.
Amenaza u Oportunidad	Amenaza
Probabilidad	5
Impacto	10
Importancia (P*I)	50
Categoría	Principal
Estrategia	Mitigado. Se cerciorará durante los *Sprint planing* cuales son las historias de usuario a desarrollar, confirmando por parte del equipo que se ha entendido correctamente y siendo verificado por el *Scrum Master*. Adicionalmente, en la weekly se revisará y analizará lo realizado a mitad de Sprint para cerciorarse de ir por el camino adecuado.

Tabla 2.12: Riesgo "Historia de usuario mal definida".

Riesgo 7	**Mutabilidad de las historias de usuario**
Descripción	Al proyecto LOCOMOTION involucra a una gran cantidad de organizaciones de todo Europa. Esto implica que pueda haber más variabilidad en los requisitos por culpa de retrasos en la coordinación entre organizaciones.
Amenaza u Oportunidad	Amenaza
Probabilidad	8
Impacto	9
Importancia (P*I)	72
Categoría	Prioritario
Estrategia	Mitigado. Se utilizará un marco de trabajo ágil para poder asimilar la variabilidad del cambio.
Plan de contingencia	Si se produce un cambio en una o varias historias de usuario, se deberán de volver a analizar y asegurar de que el cambio requerido ha sido validado por todas las partes involucradas, una vez asegurado, se realizarán las modificaciones oportunas.

Tabla 2.13: Riesgo "Mutabilidad de las historias de usuario".

Riesgo 8	**Falta de organización en el equipo de trabajo**
Descripción	Este TFG es el proyecto más grande en el que he participado hasta el momento, pueden darse casos en los que la complejidad de organización sea demasiado grande y haya problemas organizativos.
Amenaza u Oportunidad	Amenaza
Probabilidad	6
Impacto	7
Importancia (P*I)	42
Categoría	Principal
Estrategia	Mitigado. Se realizarán reuniones semanalmente con la tutora para ir informando de la situación actual del proyecto y de posibles inconvenientes que se hayan descubierto.

Tabla 2.14: Riesgo "Falta de organización en el equipo de trabajo".

Riesgo 9	**Comunicación entre las partes poco fluida**
Descripción	Durante el desarrollo del proyecto puede que sea necesario ponerse en contacto con el tutor. Podría darse la posibilidad de que no se pueda hallar ninguna vía de comunicación.
Amenaza u Oportunidad	Amenaza
Probabilidad	6
Impacto	10
Importancia (P*I)	60
Categoría	Prioritario
Estrategia	Evitado. Se utilizará la plataforma de mensajería instantánea de la escuela *rocket* [42] para mantener una vía de comunicación siempre abierta.

Tabla 2.15: Riesgo "Comunicación entre las partes poco fluida".

Riesgo 10	**Incompatibilidad horaria en el equipo de trabajo**
Descripción	El equipo de trabajo tendrá otras tareas externas al proyecto, las cuales pueden dificultar los horarios para las reuniones.
Amenaza u Oportunidad	Amenaza
Probabilidad	6
Impacto	3
Importancia (P*I)	18
Categoría	Secundario
Estrategia	Mitigado. Antes de comenzar el proyecto, cada miembro del equipo de trabajo expondrá sus incompatibilidades horarias, Teniendo en cuenta estas incompatibilidades, se puede buscar el mejor momento para realizar las reuniones. Si un día en concreto uno de los miembro no pudiese asistir a la reunión, lo avisará para así poder cambiarla de hora.

Tabla 2.16: Riesgo "Incompatibilidad horaria en el equipo de trabajo".

Riesgo 11	**Barrera lingüística**
Descripción	El proyecto LOCOMOTION involucra a gente con lenguas maternas distintas, gran parte de esas lenguas no son comprendidas por el equipo de desarrollo y esto provocaría problemas a la hora de comunicarse con otros miembro del proyecto.
Amenaza u Oportunidad	Amenaza
Probabilidad	10
Impacto	7
Importancia (P*I)	70
Categoría	Prioritario
Estrategia	Evitado. Desde el principio el proyecto impuso como lengua oficial el inglés. Este idioma no presenta problemas al equipo de desarrollo.

Tabla 2.17: Riesgo "Barrera lingüística".

Riesgo 12	**Pérdida del código fuente**
Descripción	El desarrollo del *plugin* es realizado solo por una persona en un único dispositivo, cabe el riesgo de que ese dispositivo deje de funcionar y se pueda perder parte o la totalidad del código desarrollado.
Amenaza u Oportunidad	Amenaza
Probabilidad	2
Impacto	10
Importancia (P*I)	20
Categoría	Prioriatio
Estrategia	Mitigado. Se usará el controlador de versiones, *git* [14] con un repositorio en remoto, este será el repositorio oficial de la escuela, Gitlab.
Plan de contingencia	En caso de fallo del dispositivo, se reemplazará dicho dispositivo y se recuperará al última versión resguardada en el repositorio remoto.

Tabla 2.18: Riesgo "Pérdida del código fuente".

Riesgo 13	**Falta de conocimientos del *stack* de desarrollo**
Descripción	El proyecto utiliza ANTLR y java con librerías de SonarQube. El equipo de desarrollo nunca ha trabajado con ANTLR ni con las librerías de SonarQube. El tiempo requerido para comprender su funcionamiento puede ser mayor al esperado.
Amenaza u Oportunidad	Amenaza
Probabilidad	4
Impacto	6
Importancia (P*I)	24
Categoría	Secundario
Estrategia	Mitigado. Se estudiará ANTLR y el funcionamiento de las librerías de SonarQube a priori de empezar el proyecto

Tabla 2.19: Riesgo "Falta de conocimientos del stack de desarrollo ".

Riesgo 14	**Limitaciones del *stack* actual de desarrollo**
Descripción	Al depender de librerías y software de terceros, puede darse el caso de que estos no permitan ciertas funcionalidades requeridas.
Amenaza u Oportunidad	Amenaza
Probabilidad	3
Impacto	9
Importancia (P*I)	27
Categoría	Secundario
Estrategia	Aceptado. En caso de que se encuentre una limitación, se buscarán alternativas a la implementación. No se contempla un cambio en el *stack* de desarrollo debido a la gran cantidad de tiempo que llevaría traspilar el código actual.
Plan de contingencia	En caso de no encontrar alternativas a la implementación, se pedirá una revisión de la historia de usuario pertinente para que sea complaciente con las limitaciones.

Tabla 2.20: Riesgo "Limitaciones del stack actual de desarrollo ".

Riesgo 15	**Problemas de sincronización con el diccionario de datos**
Descripción	El *plugin* tiene conexión con un diccionario de datos del proyecto, el cual está también en desarrollo. Su conexión se hace mediante una API, la cual podría cambiar en el futuro o carecer de *end points* necesarios para el correcto funcionamiento del *plugin*.
Amenaza u Oportunidad	Amenaza
Probabilidad	8
Impacto	7
Importancia (P*I)	56
Categoría	Principal
Estrategia	Mitigado. Se establecerá una línea de comunicación con el equipo de desarrollo del diccionario de datos del proyecto. Esta comunicación se realizará mediante un issue tracker definido en el repositorio gitlab oficial de LOCOMOTION.

Tabla 2.21: Riesgo "Problemas de sincronización con el diccionario de datos del proyecto ".

Riesgo 16	**Cancelación del proyecto LOCOMOTION**
Descripción	El TFG forma parte del proyecto LOCOMOTION, puede darse el caso de que dicho proyecto fuese cancelado durante el desarrollo del TFG.
Amenaza u Oportunidad	Amenaza
Probabilidad	1
Impacto	4
Importancia (P*I)	4
Categoría	Secundario
Estrategia	Mitigado. El *plugin* no solo se utilizará en LOCOMOTION, al finalizar el desarrollo se hará publico en un repositorio en *github* [15]. Para que así pueda ser utilizado por otras personas. Si el proyecto LOCOMOTION fuese cancelado, se seguiría teniendo un objetivo real para llevar a cabo el desarrollo del *plugin*.

Tabla 2.22: Riesgo "Cancelación del proyecto LOCOMOTION".

Riesgo 17	**Restricciones debidas a fuerza mayor**
Descripción	En la actualidad pueden ocurrir eventos que impongan ciertas restricciones que interfieran con el desarrollo del *plugin*. Por ejemplo, este TFG se empieza durante la pandemia causada por el virus SARS-CoV-2, existen restricciones de movilidad y en los entornos de trabajo.
Amenaza u Oportunidad	Amenaza
Probabilidad	10
Impacto	10
Importancia (P*I)	100
Categoría	Prioritario
Estrategia	Mitigado. Como se prevé que sigan activas las restricciones, las reuniones se harán de manera *online* usando las plataformas *jitsi* [1] y *WebEx* [7]
Plan de contingencia	Si se impusieran nuevas restricciones que afectasen al entorno de desarrollo, se haría, si fuese posible, una reunión de emergencia en la cual se analizaría en qué afectan las nuevas restricciones y qué pasos realizar para poder continuar con el desarrollo. En caso de no ser posible siquiera una reunión *online*, se utilizará el sistema de mensajería instantánea *rocket* o el servicio de email de la universidad. Mientras no se pudiese analizar la situación, el equipo de desarrollo continuara implementando el *Sprint Backlog* del *Sprint* actual en el que se llegue.

Tabla 2.23: Riesgo “Restricciones debidas a fuerza mayor ”.

2.6. Planificación

Este TFG se realiza asociado a una beca en el proyecto LOCOMOTION. El periodo de beca del TFG es de 6 mesas ampliables hasta 12. Para realizar la planificación inicial se asumirá que la duración será de 6 meses. Si fuese necesario se ampliará y reestructurará en el futuro, al saber que el inicio de la beca es el 14 de Septiembre de 2020, tenemos como fecha de fin de beca el día 14 de Marzo de 2021. Durante este periodo de beca se priorizará el desarrollo del *plugin* frente a la escritura de la memoria, esto se debe a que en la beca solo se incluye la realización del *plugin*. Pese a ello, se irán tomando notas de los avances que se vayan haciendo por Sprint para poder ser luego utilizados a la hora de escribir la memoria.

El TFG está definido en el Grado de Ingeniería Informática de la UVa con 12 créditos ECTS [54]. Cada uno de estos créditos equivale a 25 horas de trabajo, [53] por lo que el el proyecto supondrá 300 horas de trabajo aproximadamente.

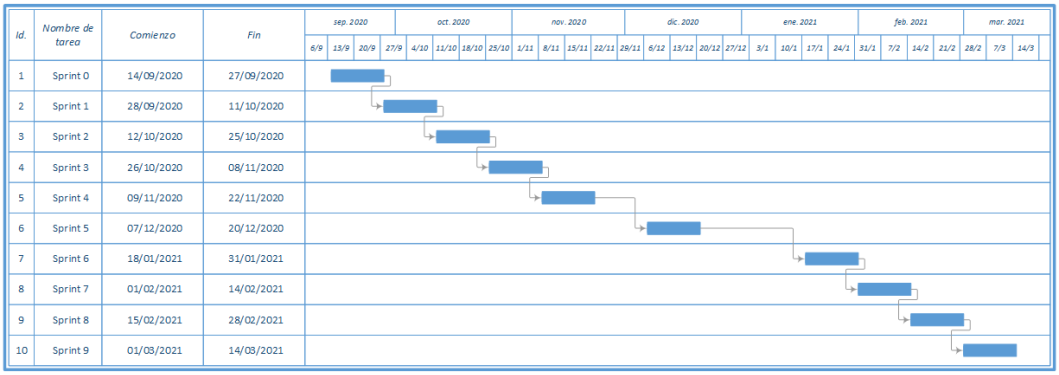

Figura 2.1: Planificación inicial.

Desde el principio se decidió tener *Sprints* de 2 semanas, con un total de 10 *Sprints* de desarrollo, 9 de ellos para implementación de nuevas funcionalidades y 1 extra como contingencia. Estos *Sprints* se iniciarían en Lunes.

Por otra parte, los primeros 3 meses del TFG se realizarán en paralelo con otras 5 asignaturas del grado. Por ello se decidió incluir un *Sprint* de descanso en el momento que se estimó que habría más carga por parte de las asignaturas, esta semana se implantó entre el día 23 de Noviembre de 2020 hasta el 6 de Diciembre de 2020.

Además, durante el periodo de vacaciones de Navidad no se realizó ningún Sprint en activo. Pese a ello se pensó utilizar este tiempo para otro tipo de tareas como el *rafactoring* y limpieza del código. El periodo de vacaciones de Navidad se estipuló entre los días 20 de Diciembre de 2020 y 18 de Enero de 2021. En este período se incluyó también la realización de exámenes del primer cuatrimestre en convocatoria ordinaria.

En la Figura 2.1 puede encontrarse la planificación inicial en forma de diagrama de Gantt.

2.7. Presupuesto

2.7.1. Presupuesto simulado

El equipo de desarrollo está constituido por un programador junior, teniendo en cuenta el "XVII Convenio colectivo estatal de empresas de consultoría y estudios de mercado y de la opinión pública."[32], se considera que el salario de un programador junior a partir del 31 de Diciembre de 2019 es de 15.860,56 € al año, realizando 1800 horas anuales.

A parte de este salario, la organización debe pagar alrededor de un 30,9 % extra del salario base a la Seguridad Social [19]. Por lo tanto el precio total anual es 20.761,47 € que implica un precio de 11,53 € por hora. Como la duración del TFG será de 300 horas, conlleva un precio total de 3.459 € respecto a los recursos humanos.

En cuanto a hardware, el desarrollo del TFG se realizará en un portátil Lenovo Y520-15IKBN valorado en 850 €. Según la agencia tributaria, los equipos para procesos informáticos tienen un coeficiente de amortización máximo del 25 % [2]. Teniendo en cuenta que la duración estimado del TFG es de seis meses, resulta en una amortización de 106,25 €.

Respecto al software, el desarrollo se ha desarrollado utilizando el entorno de desarrollo integrado de JetBrains, llamado Intellij IDEA cuyo precio es de 14,90 € por mes [21], por lo tanto tenemos un total de 89,40 €. El resto de software utilizado para el desarrollo del proyecto y la comunicación entre los miembros del equipo de proyecto son gratuitos.

No se tendrán en cuenta costes de las necesidades básicas como luz y agua, al haberse realizado el desarrollo desde las instalaciones universitarias y desde el alojamiento del alumno.

Se tendrá un margen de contingencia del 12 % para poder afrontar posibles futuros imprevistos. En la Tabla 2.24 se puede ver el presupuesto calculado.

Sueldo	3.459 €
Hardware	106,25 €
Software	89,40 €
Sub-total	3.654,65 €
Margen de contingencias	438,56 €
Total	**4.093,21 €**

Tabla 2.24: Presupuesto simulado.

2.7.2. Presupuesto real

El presupuesto real incluye la remuneración por la beca y la amortización del portátil utilizado para el desarrollo. El entorno de desarrollo no ha sido necesario adquirirlo, esto se debe a que el entorno de desarrollo es obtenido de forma gratuita gracias a ser estudiante universitario. [20]

El sueldo de la beca es de 300 € brutos mensuales, a estos hay que añadir 48,41 € extras de seguridad social que dan un total de 348,41 € mensuales. La beca tiene una duración de seis meses prorrogables hasta doce meses. Por ello la base prevista para el sueldo es de $348,41x6 = 2090,46$ €

La amortización del portátil es de 106,25 €.

Por lo tanto el presupuesto total final es de 2.196,71 €

Capítulo 3

Tecnologías utilizadas

En este Capítulo se presenta un resumen de las tecnologías utilizadas tanto para la gestión del proyecto, como para el desarrollo.

3.1. Tecnologías para la gestión del TFG

Estas tecnologías han servido para poder mantener organizado el desarrollo del proyecto, esto ha sido posible gracias a poder haber mantenido una línea de comunicación con la tutora y al poder gestionar qué tareas y/o incidencias quedan por realizar y cuánto tiempo se ha invertido en cada una.

3.1.1. Rocket.chat

Es una plataforma de mensajería instantánea en la cual se pueden tener multitud de conversaciones privadas así como grupos o hilos de comunicación. Este software es parecido a Slack, aunque en el caso de Rocket es *open-source* [42].

La Escuela de Ingeniería Informática cuenta con un servicio privado de Rocket.chat para todos los alumnos y profesores (https://rocket.inf.uva.es/). Se ha utilizado este medio para poder mantener una línea de comunicación con la tutora para cualquier transferencia de información pertinente sin tener que esperar a la próxima *weekly*, además para poder modificar los horarios de las *weeklys* en caso extraordinario de que uno de los integrantes no pudiese en la hora concretada.

3.1.2. Jitsi

Jitsi [1] es un software de vídeo conferencias de código abierto. Tiene una buena integración con Rocket y es por ese motivo que se utiliza esta plataforma para realizar las *weeklys*.

La Escuela de Ingeniería Informática cuenta con un servidor con este software en ejecución.

3.1.3. WebEx

WebEx [7] es otra plataforma de vídeo conferencias, se utilizará como medio de respaldo en caso de que el servicio de Jitsi esté caído.

Para poder utilizarlo se necesita licencia, la Universidad de Valladolid cuenta con dicha licencia.

3.1.4. GitLab Issue Tracker

GitLab Issue Tracker es el gestor de incidencias de la plataforma GitLab [16].

Este gestor de incidencias se utiliza para anotar qué tareas quedan por hacer, cuáles se están haciendo y cuáles han sido hechas, además de poder añadir comentarios, estimaciones de tiempos, etiquetas y *milestones* a cada una de estas incidencias.

Para este proyecto, se utiliza una tabla para representar los tres estados en los que puede estar una incidencia: `Open`, `Doing` y `Closed`. Adicionalmente se han utilizado las siguientes etiqueta para categorizar cada incidencia:

Data dictionary para incidencias relativas al diccionario de símbolos.

Documentation para incidendias de documentación, tanto del código como de la memoria.

File generation para incidencias de la generación de archivos como la tabla de símbolos o las diferencias con el diccionario de símbolos.

Grammar para incidencias relativas a la gramática y ANTLR.

Refactor para incidencias relativas a las refactorizaciones hechas en el código.

Release para incidencias de los despliegues de nuevas versiones.

Rule added para incidencias de desarrollo de nuevas reglas.

Rule modification para incidencias de modificación de reglas ya desarrolladas.

Por otra parte se han utilizado también los *milestones*, se ha generado un *milestone* para cada *Sprint* que existe para así poder agregar todas las indecencias del *Sprint*.

En la figura 3.1 se puede ver un ejemplo del uso de esta plataforma.

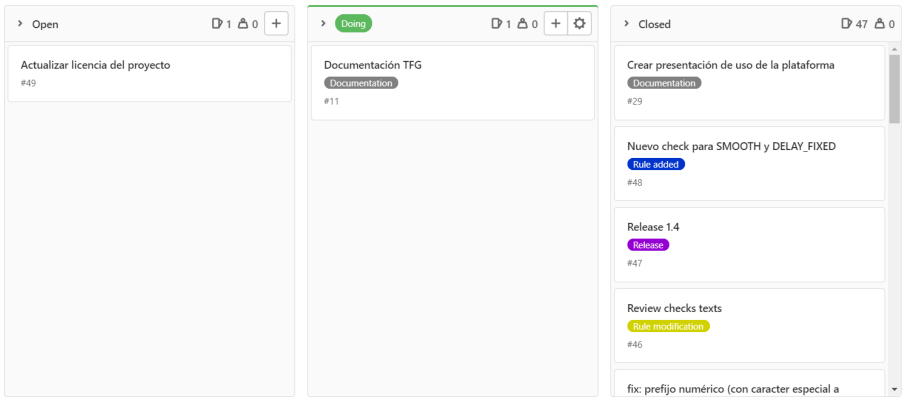

Figura 3.1: Tabla de GitLab Issue Tracker.

3.1.5. Toggl

Toggl [52] es un software para el registro de tiempo, utilizado para poder anotar la cantidad de tiempo empleada por cada tarea a realizar. Cuenta con una aplicación para Android la cual ha sido la forma principal de medir la cantidad de tiempo empleada.

3.1.6. Overleaf

Overleaf [36] es un editor de LaTeXonline que permite escribir de forma cooperativa con otros usuario, anotar comentario e incluso guardar un histórico de todos los cambios realizados en el documento a lo largo del tiempo.

Este servicio web es el que se utilizado para escribir esta memoria.

3.1.7. Visual Paradigm y Astah

Visual Paradigm [62] y Astah [6] son dos herramientas de modelado de diagramas, sirven para poder plasmar las ideas del equipo y diseños mentales en un lenguaje gráfico formal estándar. Gracias a esto, se puede generar una documentación de los proyectos que pueda ser entendible por cualquier otra persona que conozca el estándar.

Para este proyecto se ha utilizado astah para generar los diagramas de clases y paquetes escritos en UML y Visual Paradigm para generar el diagrama de la API del diccionario de símbolos.

3.2. Tecnologías para el desarrollo del plugin

Para realizar un correcto desarrollo es necesario tener un conjunto de herramientas apropiadas con las que el desarrollador esté a gusto. En esta sección encontraremos qué aplicaciones han sido utilizadas para llevar a cabo este desarrollo.

3.2.1. IntelliJ IDEA

IntelliJ IDEA [22]es un entorno de desarrollo integrado especializado en la creación de sistemas basados en Java desarrollado por JetBrains. Cuenta con funcionalidades que facilitan el desarrollo como refactorización semi-automatizada, indexación de proyectos y autocompletado entre otros.

Además, tiene una gran cantidad de *plugins* que permiten incrementar la capacidad de la tecnología.

Se ha utilizado este IDE (Integrated Development Environment) para realizar el desarrollo del *plugin*.

3.2.2. Git

Git [14] es un sistema de control de versiones con el cual se puede realizar un historias de todos los cambios realizados en un proyecto.

En git existe el concepto de "repositorio", un repositorio es el lugar donde se almacena el proyecto con todas las versiones del código incluidas. Un repositorio puede existir en cualquier máquina, por ejemplo se puede tener un repositorio en la misma máquina donde se está desarrollando y aparte otro repositorio en remoto en otra máquina, con la ventaja de que estos dos repositorios pueden ser sincronizados para que almacenen la misma información.

Por otra parte, para poder llevar a cabo múltiples desarrollos a la vez en el mismo proyecto existe el concepto de ramas, cada rama es una línea de desarrollo. En este proyecto se seguirá el modelo de ramas presentado por Gitflow [8], en el cual se tienen constantemente dos ramas: *master* y *develop*. En *master* se encuentra una versión del código estable y funcional y en *develop* una versión la cual no debe contener errores críticos y seguir siendo funcional, cuando se tiene un incremento funcional se agrega a *master*. Cada nueva funcionalidad que se quiera agregar se hará mediante una nueva rama que nazca de *develop*.

3.2.3. GitLab

GitLab [16] es una tecnología que alberga una gran cantidad de funcionalidades, una de ellas la gestión de repositorios Git.

La Escuela de Ingeniería Informática cuenta con un servicio activo de GitLab, se utilizará este servicio para mantener un repositorio en remoto del proyecto.

Es en este mismo repositorio donde se llevarán a cabo la gestión de incidencias comentadas en la subsección 3.1.4.

A parte, GitLab cuenta con un sistema de CI/CD [41], este se configuró para ejecutar los tests generados y además desplegar el *plugin* en un servidor de pruebas de la escuela cada vez que se realizara un *merge* contra la rama `develop` o contra `master`.

3.2.4. SonarQube

SonarQube [45] es la plataforma para la que se está desarrollando el *plugin*, se trata de un servicio de control de calidad de código.

Existen *plugins* para diversos lenguajes de programación como Java, Python, C, etc. Cada uno de ellos con sus propias reglas de control y asegurar que el código que analizan sea lo más seguro y siga las mejores prácticas posibles.

Para hacer pruebas con el *plugin*, se despliega una instancia de SonarQube en la máquina usada para el desarrollo. Aparte también se tiene otra instancia desplegada en una máquina virtual de la escuela para poder mostrar versiones en desarrollo a posibles personas interesadas.

Finalmente el plugin se despliega para ser utilizando en producción en una máquina virtual que realiza funciones de servidor de desarrollo para el proyecto LOCOMOTION.

3.2.5. Maven

Maven [50] es una herramienta utilizada para la gestión de dependencias y compilación de proyectos escritos en Java de una forma semi automatizada gracias al uso de comandos.

La forma de configurar Maven en un proyecto es utilizando un archivo llamado `pom.xml` el cual se encuentra en la raíz del proyecto.

En este proyecto se utiliza Maven en su versión 3.6.3.

3.2.6. ANTLR4

ANTLR4 [3] es una tecnología que permite crear gramáticas y generar analizadores léxicos y sintácticos a partir de estas definiciones. En el caso de este proyecto, estos analizadores serán generados en el lenguaje Java.

Al tratarse de una dependencia del proyecto, esta puede ser gestionada con Maven, además gracias es esto, se puede incrustar la generación de la gramática a partir de su definición en el proceso de compilación y empaquetado el *plugin*.

3.2.7. Dyson

Dyson [25] es una tecnología que permite generar *mocks* de APIs RESTful en node de forma rápida y sencilla, tan solo se necesita un archivo con la configuración escrita en JavaScript.

Esta API mockeada se utiliza para poder realizar pruebas de integración en aislamiento respecto a la comunicación entre el *plugin* y el diccionario de datos del proyecto sin necesidad de comunicarse con el servidor real y tener el riego de hacer inyecciones de elementos no válidos.

3.2.8. PostMan

PostMan [39] es una tecnología que permite realizar peticiones y consultas a una API y poder ver la respuesta que devuelve.

Su uso puede ser similar a curl. Se decide utilizar PostMan sobre curl por su forma de poder guardar peticiones para poder ser repetidas rápidamente aparte de tener interfaz gráfica que en este caso se prefiere.

Se utiliza para poder verificar un correcto funcionamiento del diccionario de datos del proyecto ya que, al tratarse este de otro proyecto en desarrollo, pueden darse situaciones en las cuales alguno de los *endpoints* deje de funcionar correctamente. En estos casos se avisa a los responsables del desarrollo del diccionario de datos del proyecto creando una incidencia en el servidor GitLab dedicado de LOCOMOTION.

3.2.9. Junit y Mockito

JUnit [51] es un *framework* utilizado para crear test unitarios para el lenguaje de programación Java.

Este *framework* es utilizado junto a Mockito [33], otro *framework*. Mockito es utilizado para simular objetos para así poder realizar tests en aislamiento.

3.2.10. JaCoCo

JaCoCo [34] es una herramienta utilizada en el entorno de realizar casos de prueba la cual nos permite analizar la cobertura de los tests, pudiendo visualizar qué secciones del código

son ejecutadas al lanzar una batería de pruebas generando un informe con los resultados al final de la ejecución.

Estas secciones pueden realizarse de diversas formas, por clase, por función, por línea, por decisión y por condición.

Capítulo 4

Análisis

4.1. Análisis inicial del proyecto de partida

Como se ha indicado anteriormente, este TFG parte de un Trabajo previo realizado por Daniel Bazaco Velasco llamado "Definición y comprobación de estándares de calidad en la programación de IAMs en Vensim" [4].

El objetivo de dicho Trabajo fue la creación inicial de un *plugin* para la plataforma SonarQube que permitiese el control de calidad mediante la comprobación de reglas de programación y convenciones de nombres en los archivos de modelos generados por el software de simulación llamado Vensim.

Para comenzar este TFG, se debe iniciar realizando un análisis de la situación actual del proyecto, para ello se utiliza la memoria del TFG anterior y el código fuente desarrollado. Este análisis describe de forma resumida el estado del *plugin* al inicio de este TFG con el objetivo de poner en contexto y facilitar el entendimiento del resto de la memoria.

Al cierre del proyecto anterior, el *plugin* acabó con un total de 15 reglas de control, la capacidad de *parsear* las declaraciones de los símbolos del modelo y la posibilidad de inyección y recuperación de símbolos del diccionario de símbolos.

4.1.1. Parser y Visistors

Un *parser* permite extraer la estructura léxica y sintáctica de un archivo mediante una serie de reglas las cuales forman una gramática.

Este *plugin* utiliza la herramienta *ANTLR*, en su versión cuarta para la generación del parser para los modelos de Vensim.

Para generar ese parser, ANTLR necesita la creación de una gramática en específico para los archivos de Vensim.

Actualmente el *plugin* cuenta con un único archivo donde está agregada toda la gramática, llamado `Model.g4`.

Para continuar con la explicación es necesario comentar que los archivos de modelo de Vensim pueden ser divididos en cuatro secciones principales:

- Declaración de símbolos.
- Declaración de vistas.
- Declaración de gráficas.
- Declaración de meta-datos.

Actualmente el *plugin* es capaz de analizar y extraer un árbol de derivación de la primera sección. De aquí, utilizando un *visitor* llamado `RawSymbolVisitor` obtiene el nombre del símbolos, su tipo, las líneas en las que aparece, los posibles índices que pueda utilizar, las unidades, el comentario y las dependencias que tenga con el resto de símbolos del modelo.

Un *visitor* permite recorrer un árbol de derivaciones nodo a nodo estipulando qué eventos se deben realizar al llegar a un nodo de un tipo en concreto, por ejemplo `RawSymbolVisitor` tiene eventos al detectar ciertos nodos que le permiten extraer la información citada anteriormente.

En la versión de partida, el *plugin* no tiene forma de poder descubrir la categoría o el módulo primario y/o secundario de los símbolos.

4.1.2. Diccionario de datos del proyecto

En la situación actual del *plugin*, existe un sistema de comunicación con un diccionario de datos externo. Este sirve para poder almacenar y visualizar símbolos que se consideran válidos y así preservar los atributos de cada uno de ellos. El diccionario de datos del proyecto es un instrumento de compartición y control de la información entre los diferentes modeladores del proyecto, encargados de modelar y programar módulos diferentes.

En la versión de partida, la comunicación entre el plugin para SonarQube y el diccionario de datos del proyecto se realiza mediante dos *endpoints*:

- qaGetSymbolDefinition

  Este *endpoint* de la API permite recuperar símbolos considerados válidos.

  Su funcionamiento se basa hacer una petición POST con la lista de símbolos que se desea recuperar en formato JSON. Un ejemplo de la petición puede ser encontrado en el Fragmento de código 4.1.

```
{
  "symbols": [
    "symbolA",
    "symbolB"
  ]
}
```

Fragmento de código 4.1: Cuerpo de la petición a "qaGetSymbolDefinition"del plugin de partida.

La respuesta del diccionario es un objeto Json que contiene una lista con todos los símbolos validos existentes, teniendo en cuenta cuáles se habían requerido en el *request*. Un ejemplo de la respuesta puede ser encontrada en el Fragmento de código 4.2. Se puede ver que se manda información extra que actualmente no es utilizada por el *plugin* como pueden ser los modelos o las categorías.

```
{
  "symbols": [
    {
      "name": "symbolA",
      "definition": "definitionexampleforsymbolA",
      "unit": "Kg",
      "isindexed": "false",
      "indexes": [],
      "modules": {
        "main": "IAMNumberOne",
        "secondary": []
      },
      "category": "CategoryExampleTopLevel",
      "ProjectTypeOfValue": "Constant",
      "ProgrammingSymbolType": "Constant"
    }
  ],
  "modules": [...],
  "indexes": [...],
  "categories": [...]
}
```

Fragmento de código 4.2: Cuerpo de la respuesta de "qaGetSymbolDefinition"del plugin de partida.

- qaAddSymbolDefinition

  El segundo *endpoint* con el que el *plugin* se comunica con el diccionario de datos del proyecto, es el inverso a "qaGetSymbolDefinition", se utiliza para inyectar nuevos símbolos en el diccionario de símbolos.

  También se realiza con una petición POST la cual tiene un cuerpo similar al recibido del *endpoint* anterior, pero con algunos matices, la estructura puede ser encontrada en el Fragmento de código 4.3.

  Como se puede ver, los principales cambios son que el módulo es extraído de cada símbolo e inyectado a nivel global, esto quiere decir que cada petición que se envíe tiene que contener los símbolos del mismo módulo, además desaparece `ProjectTypeOfValue` ya que este parámetro es asignado de forma manual utilizando un cliente web del diccionario de datos del proyecto, una vez se han agregado los símbolos en el mismo.

```
{
  "symbols": [
    {
      "name": "symbolA",
      "definition": "definitionexampleforsymbolA",
      "unit": "Kg",
      "isindexed": "false",
      "category": "CategoryExampleTopLevel",
      "ProgrammingSymbolType": "Constant"
    }
  ],
  "indexes": [...],
  "module": "moduleName",
}
```

Fragmento de código 4.3: Cuerpo de la petición a "qaAddSymbolDefinition"del plugin de partida.

La respuesta del servidor estará vacía si todos los símbolos han podido ser inyectados de forma correcta, o devolverá un Json con el nombre de cada símbolo erróneo y el motivo de fallo.

Existe un tercer *endpoint* que se utiliza para realizar la autenticación con el servidor, envinado un usuario y contraseña válidos, se recibirá un token con el que se podrán realizar las futuras consultas. Este token se enviará al servidor en la cabecera de las peticiones con la siguiente estructura `Authorization: Bearer <Token>`.

4.1.3. Reglas

El principal objetivo del *plugin* es poder avisar de fallos en el modelo de Vensim, estos fallos no son relativos a una estructura errónea o a fallos sintácticos, si no que se refieren a fallos en seguir las convenciones y reglas de programación utilizadas por LOCOMOTION o por cualquier otro proyecto en el futuro.

Las 15 reglas actuales descritas en la introducción pueden ser agrupadas respecto a qué parte representan de estas convenciones y reglas.

Para comenzar, se encuentras 6 reglas de control de nombres de los símbolos, existe una por cada tipo distinto de símbolo que se revisa. Se encuentran reglas de nombrado para: Variables, Constantes, Lookups, RealityCheck, Subscript y SubscriptValue.

Por otro lado, existen otras 2 reglas que revisan que todos los símbolos, independientemente de su tipo, contengan un comentario y una unidad.

También existen 6 reglas relacionadas con el diccionario de datos del proyecto, 5 de éstas verifican que existe uniformidad entre los símbolos encontrados en el modelo y los símbolos válidos del diccionario y la otra regla comprueba que cada símbolo existe en el diccionario, ya que de no ser así es considerado no válido. La peculiaridad de este conjunto de reglas es que

si cuando se realiza el análisis con el *plugin* no se consigue una conexión con el diccionario de datos del proyecto, son desactivadas.

Por último, la regla número 15 hace una comprobación de números mágicos. Es decir, busca que no haya números en el modelo que se repitan demasiadas veces. Esta repetición de números mágicos es considerada una mala práctica y por ello se intenta evitar. Dependiendo del número de repeticiones del número se puede considerar como un fallo de tipo informativo o como un fallo grave.

Cuando una de estas 15 reglas detecta un fallo, este es almacenado en forma de una *issue* para posteriormente ser enviado a SonarQube.

4.1.4. Estructura y control del flujo

En la sección "8.9. Estructura general del código" de la memoria del TFG de Daniel Bazaco se encuentra una buena explicación de la estructura y flujo de actividad del plugin cuando este es ejecutado [4].

Esta estructura del *plugin* consiste en cinco paquetes:

- parser

  Donde se encuentran las clases relacionadas con la gramática y los modelos.

- plugin

  Donde se encuentra el núcleo de un *plugin* de Sonarqube

- rules

  Donde se encuentran todas las reglas

- service

  Donde se encuentran las clases responsables de la comunicación con el diccionario de símbolos

- utilities

  Donde se encuentran clases comunes como constantes o *helpers*.

4.1.5. Configuraciones del plugin

El *plugin* cuenta con diversos parámetros que pueden ser modificados para dar más flexibilidad al usuario final.

Para este *plugin* existen dos formas de poder configurarlo.

- Cambiando la configuración de SonarScanner

  SonarScanner utiliza un fichero de configuración llamado `sonar-project.properties`. En este se pueden añadir pares clave valor que después pueden ser recuperados por el *plugin*. Existen alguna claves por defecto utilizadas por SonarScanner, pero se pueden añadir más claves personalizadas. En el Fragmento 4.4 se pueden ver todas las claves personalizadas utilizadas por el *plugin*.

```
#Url de la API del diccionario de símbolos.
vensim.dictionaryService=<URL>
#Nombre del usuario del diccionario de símbolos.
vensim.dictionaryUsername=<Username>
#Contraseña del usuario del diccionario de símbolos.
vensim.dictionaryPassword=<Password>
#Propiedad de tipo boolean que dicta si se hace registro de las
    respuestas del diccionario de símbolos o no, por defecto es false.
vensim.logServerMessages=true
#Nombre del archivo de salida del registro de la ejecución, si no se añ
    ade esta propiedad, el registro se realizará por la salida estandar.
vensim.logFile=log.txt
```

Fragmento de código 4.4: Propiedades propias del plugin del fichero sonar-project.properties

- Modificando los Quality Profile de SonarQube

  Dentro de SonarQube existen una serie de perfiles de calidad. Éstos se utilizan para poder definir qué reglas se quieren utilizar en qué momentos y cuáles no y para qué lenguajes. Es una funcionalidad muy útil para cuando en un proyecto existen varios grupos trabajando sobre un mismo conjunto de archivos, pero cada uno sobre una temática diferente, podrían existir varios *Quality Profiles*, cada uno de ellos con las reglas pertinentes para cada trabajador.

  A parte de poder activar o desactivar reglas, también se pueden cambiar parámetros de ellas si el *plugin* lo soporta.

  En la versión del plugin de la que se parte, solo existe una regla que esté parametrizada, se trata de la regla de control de los números mágicos, la cual permite elegir un umbral que indique el número de repeticiones máximas que se pueden dar en un número antes de que este sea descrito como fallo.

4.2. Estructura de los archivos de salida generados

En el *plugin* original existe un archivo que se genera al acabar la ejecución de un análisis. En dicho archivo se encuentra toda la información relativa a los símbolos detectados en los modelos que se haya analizado. La estructura que tiene es la que se muestra en el Fragmento de código 4.5. Se puede ver cómo exporta todos los datos extraídos de cada símbolo como son el nombre, tipo, líneas de aparición, dependencias, unidades y comentarios.

```
[
  {
    "file": "WILIAM.mdl",
```

```
    "symbols": {
      "\"%_PHS_overcapacity_vs_potential\"": {
        "type": "VARIABLE",
        "lines": [
          2
        ],
        "dependencies": [
          "Available_FE_elec_stored_PHS_TWh",
        ],
        "units": "percent",
        "comment": "Overcapacity as a percentage of the total installed
    capacity over the \rmaximum potential. If no overcapacity, then =100%.",
      }
    }
]
```

Fragmento de código 4.5: Estructura original del archivo generado symbolTable.json

Este archivo generado recibe el nombre de `symbolTable.json`.

Este archivo se utilizó para validación con programadores Vensim que revisaran un modelo descrito en un .mdl y los datos extraídos por el plugin.

4.3. Análisis inicial de posibles modificaciones

Teniendo en cuenta los nuevos requisitos generados para este TFG que pueden ser encontrados en la Sección 2.1. Existen diversas modificaciones que tienes que ser realizadas al *plugin* actual.

A continuación se explicarán cuatros cambios de los más importantes. En esta sección se analizará qué cambios se deben hacer, pero no se explicará la forma de su implementación, para poder ver como se ha llevado a cabo la implementación puede leerse en el Capítulo 6.

4.3.1. Análisis de la gramática

Uno de los principales cambios en la gramática es extenderla para poder hacer el reconocimiento léxico y sintáctico de la sección donde se declaran las vistas. Para poder hacer este análisis, primero es necesario explicar cuál es la estructura léxica y sintáctica de las vistas.

La estructura de las vistas puede dividirse en cuatro bloques [59]:

- Separador
  Al inicio de la declaración de todas las vistas encontramos un separador, en la versión de Vensim utilizada en LOCOMOTION puede verse en el Fragmento de código 4.6

```
\\\---/// Sketch information - do not modify anything except names
V300  Do not put anything below this subsection - it will be ignored
```

Fragmento de código 4.6: Separador de las vistas en el modelo

- Nombre
  Cada vista tiene un nombre que la define. Este nombre no tiene por qué ser único, dos o más vistas pueden tener el mismo nombre. El nombre se escribe en una línea precedido del carácter "*" como se puede ver en el Fragmento de código 4.7.

```
1 *Nombre de la vista
```

Fragmento de código 4.7: Nombre de una vista en el modelo

- Configuración por defecto de la vista
  Se trata de una línea a continuación de la línea del nombre de la vista que contiene información general de la misma. En esta encontramos datos sobre los colores que tienen las relaciones y los símbolos, la relación de aspecto y el zoom entre otros. Esta línea siempre va precedida del símbolo "$" como se muestra en el Fragmento de código 4.8.

```
1     Ejemplo:
2 $192-192-192,0,Times New Roman
      |12||0-0-0|0-0-0|0-0-255|-1--1--1|-1--1--1|96,96,100,0
3 Descripción:
4 $iniarrow,n2,face|size|attributes|color|shape|arrow|fill|background|ppix,
      ppiy,zoom,tf
```

Fragmento de código 4.8: Configuración por defecto de una vista en el modelo

- Declaración de elementos de la vista
  El resto de la declaración de la vista define todos los elementos que existen en ella y sus relaciones, los distintos tipos de elementos que existen son: Flechas (id: 1), Variables (id: 10), Válvulas (id: 10), Comentarios (id: 12), Iconos (id: 30), Meta-archivos (id: 31) y objetos de entrada salida (id: 12) [60]. Cada uno de estos elementos tiene una id interna, la cual se utiliza para poder diferenciarlos. La estructura de la declaración para todos los símbolos a excepción de las flechas está expuesta en el Fragmento de código 4.9. Las flechas tienen una sintaxis diferente que se puede ver en 4.10

```
1 Ejemplo:
2     10,1,Population,275,69,40,20,3,3,0,0,0,0,0,0
3 Descripción:
4     n,id,name,x,y,w,h,sh,bits,hid,hasf,tpos,bw,nav1,nav2,box,fill,font
```

Fragmento de código 4.9: Declaración de una variable en una vista

```
1 Ejemplo:
2     1,3,5,1,4,0,0,22,0,0,0,-1--1--1,,1|(204,71)|
3 Descripción:
4     1,id,from,to,shape,hid,pol,thick,hasf,dtype,res,color,font,np|plist
```

Fragmento de código 4.10: Declaración de una flecha en una vista

Existe un caso especial con los comentarios y los objetos de entrada/salida, estos pueden utilizar una segunda línea para añadir información extra relevante, en el comentario por ejemplo sería el texto en sí como se puede ver en 4.11.

```
12,16,0,269,22,37,9,8,4,0,0,-1,0,0,0
Comment
```

Fragmento de código 4.11: Declaración de un comentario con texto en una vista

En el fragmento 4.12 se puede ver un ejemplo de la declaración de una vista entera.

```
\\\---/// Sketch information - do not modify anything except names
V300  Do not put anything below this subsection - it will be ignored
*View 1
$192-192-192,0,Arial
    |12||0-0-0|0-0-0|0-0-255|-1--1--1|-1--1--1|96,96,100,0
10,1,Population,275,69,40,20,3,3,0,0,0,0,0,0
12,2,48,91,71,8,8,0,3,0,0,-1,0,0,0
1,3,5,1,4,0,0,22,0,0,0,-1--1--1,,1|(204,71)|
1,4,5,2,100,0,0,22,0,0,0,-1--1--1,,1|(130,71)|
11,5,48,167,71,6,8,34,3,0,0,1,0,0,0
10,6,births,167,90,19,11,40,3,0,0,-1,0,0,0
12,7,48,487,70,8,8,0,3,0,0,-1,0,0,0
1,8,10,7,4,0,0,22,0,0,0,-1--1--1,,1|(441,70)|
1,9,10,1,100,0,0,22,0,0,0,-1--1--1,,1|(353,70)|
11,10,48,397,70,6,8,34,3,0,0,1,0,0,0
10,11,deaths,397,89,22,11,40,3,0,0,-1,0,0,0
10,12,birth rate,99,118,29,11,8,3,0,0,0,0,0,0
10,13,average lifetime,508,104,49,11,8,3,0,0,0,0,0,0
1,14,12,6,0,0,0,0,0,64,0,-1--1--1,,1|(130,104)|
1,15,13,11,0,0,0,0,0,64,0,-1--1--1,,1|(445,95)|
12,16,0,269,22,37,9,8,4,0,0,-1,0,0,0
Comment
1,17,1,5,1,0,0,0,0,64,0,-1--1--1,,1|(203,31)|
1,18,1,10,1,0,0,0,0,64,0,-1--1--1,,1|(349,27)|
```

Fragmento de código 4.12: Declaración de una vista del modelo

Una vez explicada la estructura de la declaración de una vista, se puede continuar con el análisis. En el plugin de partida, la gramática se encuentra unificada en solo un documento llamado `Model.g4`, aquí se encuentra tanto las reglas léxicas como las sintácticas para hacer el análisis de la sección donde se declaran los símbolos.

Entre la sección analizada actual y la nueva por analizar existen varias diferencias en la estructura que deben ser resueltas en la nueva gramática.

La primera diferencia observable es la forma de separar los elementos descritos, en la parte de símbolos, cada uno de ellos estaba compuesto de tres subsecciones: La declaración, las unidades y el comentario. Estas son separadas con el carácter “ ɀ entre los símbolos se diferencias con el carácter “|”. En la parte de vistas, se usa las cadena de caracteres “`\\\---///`”, dentro de la declaración de una vista se utiliza un salto de línea para diferenciar cada elemento, ya sean estos símbolos o relaciones entre símbolos. Esta forma de diferenciar utilizando el salto de línea es problemática con la gramática actual por el motivo de que esta omite todos los saltos de línea. Será necesario modificar la gramática para que los saltos de línea sí se tengan en cuenta.

La segunda diferencia es el uso de información semántica a la hora de condicionar la sintaxis. Esto es un problema grave que puede romper el análisis de un modelo si no se

realiza correctamente. Un ejemplo de este problema puede verse en la declaración de los comentarios de una vista. Éstos ocuparán una línea si su tercer campo es distinto de 0, en caso contrario se expandirán una línea extra en la cual aparecerá un texto arbitrario. Será necesario que la nueva gramática sea lo suficientemente flexible para poder gestionar los dos casos.

Otro cambio, en este caso dentro de la sección de declaraciones de símbolos es que, como se puede ver en la historia de usuario 4.3, existe el concepto de grupo los cuales actualmente no son utilizados en la gramática inicial por lo que será necesario analizarlos.

El resto de la sintaxis no contiene diferencias notables aparte de la nueva estructura en sí.

4.3.2. Comunicación con el diccionario de datos

El *plugin* en la versión de partida solo cuenta con dos formas de poder comunicarse con el diccionario de símbolos, `qaGetSymbolDefinition` y `qaAddSymbolDefinition` como se mencionó anteriormente.

Para el nuevo *plugin* se requieren más formas de comunicación, alguna de éstas superpuesta con las actuales, por lo que será necesario redefinirlas. Los detalles del diseño de la nueva estructura de las llamadas al API del diccionario de datos del proyecto se puede encontrar en el Anexo C.

4.3.3. Estructura, datos y control de flujo de la versión de partida

Respecto a la estructura del *plugin*, viendo los cinco paquetes que contiene actualmente el *plugin* y observado qué clases contiene cada uno de ellos, se prevé que se creará un nuevo paquete llamado `model` el cual contendrá todas las clases utilizadas para el almacenamiento de datos y así poder dar más modularidad a la estructura y desacoplar del parser los tipos de datos que gestiona el *plugin*.

Adicionalmente, se estima que será necesario crear nuevas clases contenedoras de datos para poder almacenar las nuevas entidades que existirán, como pueden ser las vistas, los acrónimos, etc...

Además, ahora que se dará importancia a las vistas y al módulo y categorización asociados con cada una de ellas es necesario hacer una explicación de estos conceptos desde el punto de vista de este *plugin*:

- Módulo

  Un módulo es una organización de programación. Esto permite eliminar complejidad al modelo integrado y poder subdividirlo en equipos de desarrollo cada uno de ellos centrado en un módulo. La gran mayoría de los símbolos deberán pertenecer principalmente

a una vista o *sketch* y por ende a un módulo en concreto (salvo algunas excepciones como pueden ser los índices que no tienen un módulo principal). El equipo de desarrollo de ese módulo tendrá la responsabilidad sobre todos los símbolos primarios que tenga. Por otra parte un símbolo también puede existir en otros módulos aunque en este caso sería utilizado para hacer consultas, esta relación es de tipo secundario o *shadow*.

- Categoría

  Una categoría es una manera de clasificar variables, es decir, una organización semántica, define un conjunto en el cual todos los símbolos incluidos estarían relacionados semánticamente. En LOCOMOTION además se permite un nivel extra de categorías llamadas subcategorías. Un símbolo solo puede tener una categoría y subcategoría, la cual debe pertenecer a su vez a la categoría. Un símbolo obtiene su categoría y subcategoría de la vista o *sketch* en el cual es principal. Las categorías y subcategorías de las vistas en las que el símbolo es secundario no son relevantes. Una categoría puede tener símbolos de varios módulos.

En la sección de diseño de Estructura del código y paquetes se pueden ver una serie de diagramas cin la nueva estructura del *plugin*.

Respecto al control del flujo, lo más seguro es que haya que hacer nuevos flujos de datos y ampliar los actuales para que se puedan realizar los añadidos que se requieren para este *plugin*.

4.3.4. Configuración del plugin

En la versión de partida no se puede parametrizar gran parte del *plugin*. Esto es uno de los requisitos propuestos para esta nueva versión. Para poder realizar las historias de usuario desde la 5.1 a la 5.6 es necesario implementar nuevas formas de configuración.

Todas las historias que hacen referencia a parametrizar reglas (las historias de usuario 2.6 y 2.6) pueden llevarse a cabo con propiedades de las reglas como se hace en el *plugin* inicial en la regla de comprobar números mágicos. Este tipo de propiedades pueden ser modificadas directamente desde la página web de SonarQube.

El resto de historias de usuario relacionadas con la generación de archivos y otro tipo de opciones (historias de usuario de la 2.6 a la 2.6) serán gestionadas utilizando el archivo de propiedades de sonar scanner `sonar-project.properties`. Esto es así al tratarse de configuraciones globales del plugin que no tienen que ver con una regla en concreto.

Para poder aportar la funcionalidad necesaria para cumplir todas las historias de usuario se crearán un conjunto de nuevas claves para este archivo de propiedades, todas las nuevas propiedades añadidas pueden ser vista en la Sección 5.3 del Capítulo de Diseño.

4.4. Estructura de los archivos de salida generados

Para poder cumplir con los nuevos requisitos será necesario extender los datos incluidos en el archivo `symbolTable.json` y además será necesario generar un archivo completamente nuevo que contenga las diferencias que se han hallado entre el análisis en local y lo almacenados en el diccionario de datos del proyecto.

Las estructuras de estos dos archivos pueden ser encontradas en la Sección 5.2 del Capítulo del Diseño.

Capítulo 5

Diseño

En este capítulo se realizarán los diseños estructurales de los elementos analizados que requieran de una modificación clara.

5.1. Estructura de las peticiones al diccionario de datos del proyecto

A continuación se describirán todos los *endpoints* de la API REST con los que tendrá que comunicarse el *plugin*, es importante remarcar que las respuestas que se van a definir solo contienen la información que es relevante para el *plugin*, puede que existan más campos que sean de utilidad en otros proyectos:

- authenticate

  Este servicio utilizado para la autenticación de usuarios se mantiene igual que en la versión actual. Su estructura de petición y respuesta se pueden encontrar en C.1 y C.2 respectivamente.

- qaGetSymbolsDefinition

  Este punto de conexión actualmente incluye información extra que no encaja con la definición de símbolo. Por ello, se eliminarán de este punto la obtención de información relativa a los módulos, índices y categorías que se mandaban aparte de los símbolos.

  Además, para cumplir con la historia de usuario 2.6 la cual tiene que ver con la relaciones que existen entre símbolos y referencias a tablas en Excel, se decide en conjunto con los stakeholders interesados en esta información la estructura que deberá tener dentro de cada símbolo los datos relacionados con los Excels.

  La estructura de la petición no ha sido modificada.

  Por lo tanto, realizando estos dos cambios, la nueva estructura de la respuesta de este *endpoint* puede encontrarse en el Fragmento de Código C.3.

- qaAddSymbolsDefinition

  Este *endpoint* también ha sido modificado. Se ha añadido la misma estructura relacionada con los Excels descrita anteriormente y, aparte, al trabajar ahora con módulos de una forma más predominante, se decide añadir el nombre del módulo dentro de la definición de cada símbolo y no de manera global en la petición, de este modo en una única petición se podrían enviar todos los símbolos a inyectar.

  De igual manera que en el caso de `qaGetSymbolDefinition` se han eliminado elementos en la petición que no se identificaban correctamente como símbolos. En este caso se ha eliminado el envío de índices por esta vía.

  La respuesta no se ha alterado.

  La nueva estructura de la petición a este *endpoint* puede ser vista en el Fragmento de Código C.4.

- qaGetIndexesDefinition

  Este es un nuevo *endpoint*. Como su nombre indica, este punto de conexión se utiliza para recibir información relativa a los índices guardados en el diccionario de datos del proyecto. En el *plugin* inicial esta información es enviada en la respuesta de `qaGetSymbolDefinition`. Al haber sido eliminada dicha información de la respuesta de que se obtiene de `qaGetSymbolDefinition`, es necesario la creación de este nuevo *endpoint*.

  La petición en este caso es una petición de tipo GET.

  La respuesta devuelve un *array* con todos los índices existentes en el diccionario de datos del proyecto. La estructura que tiene esta respuesta puede ser encontrada en el Fragmento de Código C.5.

- qaAddIndexesDefinition

  Se necesita un sistema para poder inyectar índices en el diccionario de datos del proyecto. Inicialmente estos eran enviados mediante la petición a `qaAddSymbolDefinition`. Sin embargo, al haber sido eliminado de ahí, ahora se necesita un nuevo `endpoint`.

  La estructura de la petición es similar a la de la respuesta de `qaGetIndexesDefinition` y se puede encontrar descrita en el Fragmento de Código C.6.

  La respuesta estará vacía si todo ha sido inyectado correctamente, o contendrá un mensaje de error por cada índice que sea erróneo.

- qaGetCategories

  Con este servicio sucede lo mismo que en el caso de `qaGetIndexesDefinition`. Existía en `qaGetSymbolDefinition` originalmente, pero ahora se ha extraído y generado un *endpoint* nuevo.

  La petición es de tipo GET, por lo que no tiene ninguna información extra sobre qué categorías se desea solicitar.

  La respuesta devuelve todas las categorías almacenadas en el diccionario de datos del proyecto. Su estructura puede verse en el Fragmento de Código C.7.

  En el proyecto LOCOMOTION solo se permite un nivel de jerarquía en las categorías, es por ello que este *plugin* solo permite categorías y subcategorías de esas categorías.

- qaAddCategories

  Con el nuevo *plugin* se podrán descubrir a partir del código del archivo .mdl las categorías que existen en los modelos analizados. Es por ello que ahora es necesario tener algún modo de poder inyectarlos en el diccionario de símbolos. Para ello se crea este *endpoint* el cual, usando una sintaxis idéntica a la utilizada en las respuestas de `qaGetCategories`, por ello su estructura se puede ver en el mismo Fragmento de Código de código C.7.

  Al igual que el resto de *endpoints* utilizados para la inyección de elementos, la respuesta estará vacía si está todo correcto, o una lista de errores de las categorías que no se han podido inyectar.

- qaGetModules

  Este *endpoint* es generado a causa de haber sido extraído la obtención de módulos de `qaGetSymbolDefinition`.

  La petición es de tipo GET si ninguna información extra.

  La estructura de la respuesta se puede encontrar en el Fragmento de Código C.8

- qaAddModules

  Al igual que con las categorías, el *plugin* final podrá extraer los módulos de las vistas analizadas de los modelos de Vensim. Por tanto, para mantener el máximo de información en el diccionario de símbolos, se creará una forma de inyectar estos nuevos módulos.

  La petición tiene la misma estructura que `qaGetModules` que puede ser encontrada en el Fragmento de Código C.8.

  La respuesta sigue la misma técnica que el resto, vacía si todo se ha inyectado correctamente, o una lista de errores de los módulos que no hayan podido ser inyectados.

- qaGetAcronyms

  Actualmente según las reglas de nombrado de LOCOMOTION, los símbolos de tipo variable no pueden tener mayúsculas en su nombre a excepción de cuando son acrónimos. Esta parte de reconocer acrónimos a la hora de verificar nombres será añadida nueva en el *plugin* y, por ende, es necesario una forma de recuperar del diccionario de datos cuáles son los acrónimos permitidos en el proyecto.

  Para ello se crea este *endpoint*, el cual, recibe una petición de tipo GET, y devuelve una lista completa de todos los acrónimos almacenados en el diccionario. La estructura de la respuesta se puede encontrar en el Fragmento de Código C.9.

  No existe un servicio para poder inyectar acrónimos automáticamente. Esto se debe a que se decidió que era preferible añadir los válidos manualmente en el diccionario de datos a tener que estar filtrando por todos los falsos positivos que se pudiesen dar si se automatizase.

- qaGetUnitSystem

  En LOCOMOTION existe un conjunto de unidades oficiales. A esto le llamamos el Sistema de Unidades del proyecto. Estas son las únicas unidades que puede tener cualquier símbolo del modelo. Para poder verificar esta condición, es necesario tener una

forma de recuperar cuáles son estas unidades válidas. Para ello se crea este punto de conexión, el cual con una petición de tipo GET, devuelve la lista de todas las unidades válidas que existen en el diccionario de datos del proyecto. La estructura de la respuesta puede verse en el Fragmento de Código C.10.

No existe una forma automatizada de inyectar estas unidades. Esto se debe a que este tipo de información no es obtenible en el modelo, si no que tiene que ser la dirección del proyecto quien decida cuáles son las unidades a usar y las inserten manualmente en el diccionario de datos mediante el cliente web.

En la figura 5.1 se puede ver un diagrama que representa la nueva API del diccionario de símbolos.

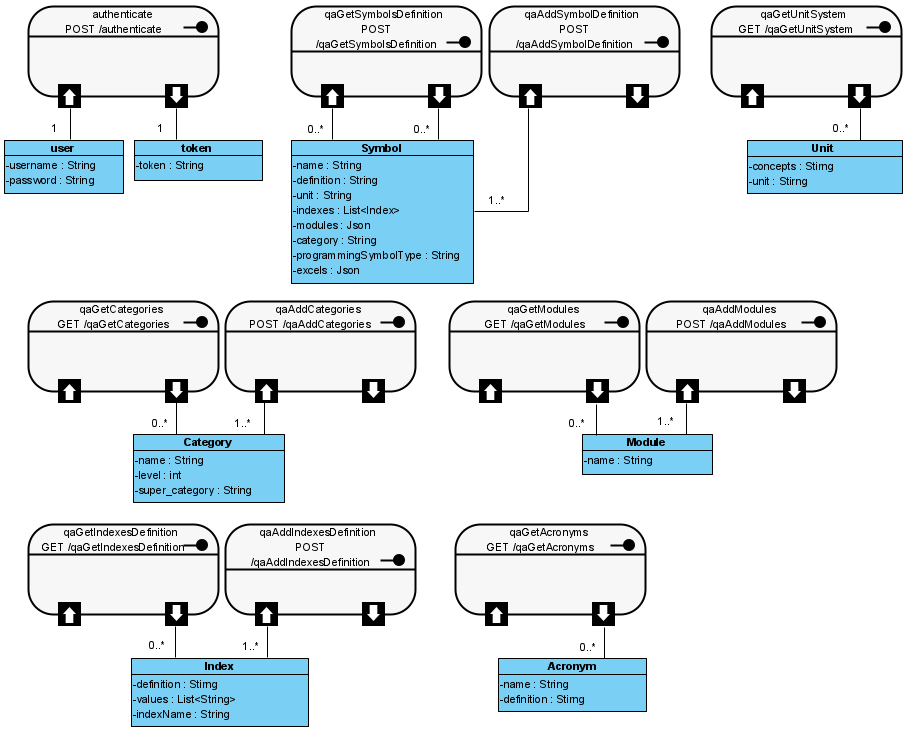

Figura 5.1: Diagrama de la API del diccionario de símbolos.

5.2. Estructura de los archivos de salida generados

Teniendo en cuenta todos los datos que se van a poder extraer gracias al nuevo *plugin*, se actualiza la estructura del archivo generado `symbolTable.json` para que pueda contener todos estos nuevos datos. En el Fragmento de Código 5.1 puede encontrarse esta nueva estructura.

Además, es necesario diseñar la estructura para el nuevo archivo que exponga las diferencias existentes ente el análisis en local y el diccionario de símbolos. En el Fragmento de Código 5.2 se puede encontrar la estructura de este nuevo archivo. Este archivo recibirá el nombre `dictionaryDiff.json`

```
[
  {
    "file": "WILIAM.mdl",
    "symbols": {
      "\"%_PHS_overcapacity_vs_potential\"": {
        "type": "VARIABLE",
        "lines": [
          2
        ],
        "dependencies": [
          "Available_FE_elec_stored_PHS_TWh",
        ],
        "primary": "energy-electricity-PHS_generation",
        "shadows": [
          "energy-electricity-PHS_generation"
        ],
        "units": "percent",
        "comment": "Overcapacity as a percentage of the total installed
    capacity over the \rmaximum potential. If no overcapacity, then =100%.",
        "group": "null",
        "notValidBecause": "ViewNameCheck",
        "isFiltered": false,
        "indexes": [
          "REGIONS_I"
        ]
      }
    },
    "views": [
      {
        "module": "land_and_water",
        "lines": [
          38316
        ],
        "category": "land",
        "subcategory": "RES_land_requirements"
      }
    ],
    "modules": [
      {
        "name": "energy.perception_PE_scarcity",
        "notValidBecause": "ViewNameCheck",
        "lines": [
          31039
        ]
      }
    ],
    "categories": [
      {
        "name": "COAL",
        "lines": [
          40357
        ],
        "level": 1,
        "super": null,
```

```
        "notValidBecause": "CategoryDuplicatedCheck"
      }
    ]
  }
]
```

Fragmento de código 5.1: Estructura nueva del archivo generado symbolTable.json

```
[
  {
    "file": "WILIAM.mdl",
    "symbols": {
      "missmatches": {},
      "not_found_in_DB": [],
      "not_found_in_local": []
    },
    "indexes": {
      "missmatches": {},
      "not_found_in_DB": [],
      "not_found_in_local": []
    },
    "indexes_values": {
      "missmatches": {},
      "not_found_in_DB": [],
      "not_found_in_local": []
    },
    "modules": {
      "not_found_in_DB": [],
      "not_found_in_local": []
    },
    "categories": {
      "missmatches": {},
      "not_found_in_DB": [],
      "not_found_in_local": []
    }
  }
]
```

Fragmento de código 5.2: Estructura nueva del archivo generado dictionaryDiff.json

5.3. Estructura del archivo de configuración

Para poder cumplir con los requisitos del proyecto es necesario generar un nuevo conjunto de propiedades en el archivo de configuración de SonarQube, la lista con las nuevas propiedades puede encontrarse en el Fragmento de Código 5.3.

```
vensim.dictionary.getDiff
vensim.dictionary.inject
vensim.dictionary.inject.modules
vensim.dictionary.inject.categories
vensim.dictionary.inject.symbols
vensim.view.module.name
vensim.view.module.separator
vensim.view.category.separator
```

```
9 vensim.auxiliaryFiles.directoryName
```

Fragmento de código 5.3: Nuevas propiedades añadidas a sonar scanner

Para una mayor facilidad a la hora de saber cómo usar las nuevas propiedades, se genera una documentación que explicar para qué sirve cada una de ellas y toda esta información se podrá utilizar como plantilla de este archivo de propiedades en el futuro. La plantilla de la configuración puede ser encontrada en el Fragmento de Código A.2 en el Anexo 1.

5.4. Estructura del código y paquetes

Para poder hacer la implementación de todos los nuevos requisitos de este proyecto es necesario crear nuevas clases y paquetes. La estructura que tendrá el nuevo *plugin* puede ser encontrada a continuación en esta sección y se presenta utilizando diagramas de clases y de paquetes.

Se utilizará un esquema de colores generado al final del proyecto para representar cuál es el estado de cada clase o paquete:

- Amarillo

  Clases o paquetes que ya existían en el *plugin* original y no han sido modificados.

- Naranja

  Clases o paquetes que ya existían en el *plugin* original y que han sido modificados pero de sin añadir funcionalidad nueva por si mismo. Se podría decir que son modificaciones leves o modificaciones que ocurren en consecuencia de otras.

- Azul

  Clases o paquetes que ya existían pero que han sido altamente modificados para añadir nuevas funcionalidades.

- Verde

  Clases o paquetes completamente nuevos.

En la figura 5.2 se muestra el diagrama de paquetes del *plugin*. Posteriormente se tienen los diagramas de clases de cada paquete, en la figura 5.3 el del paquete `model`, en la figura 5.4 el del `parser`, en la figura 5.5 el del `plugin`, en la figura 5.6 el del `rules`, en la figura 5.7 el del `service` y en la figura 5.8 el del `utilities`.

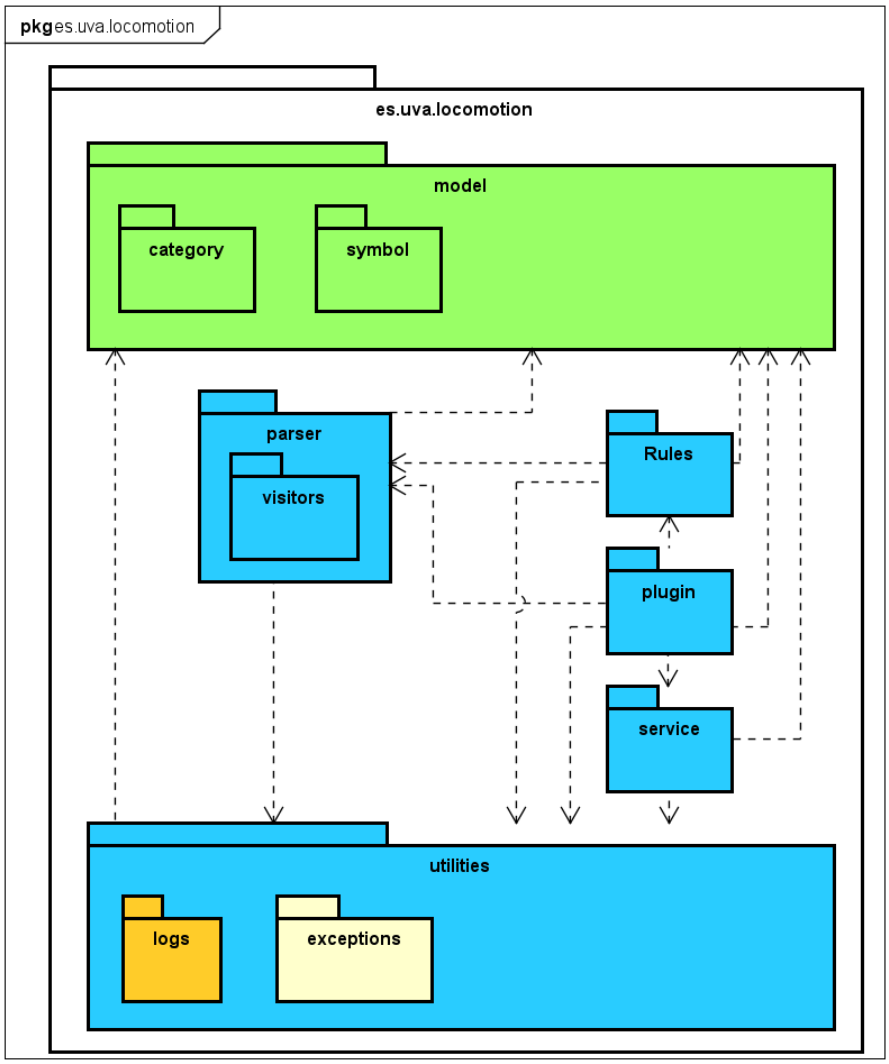

Figura 5.2: Diagrama de paquetes.

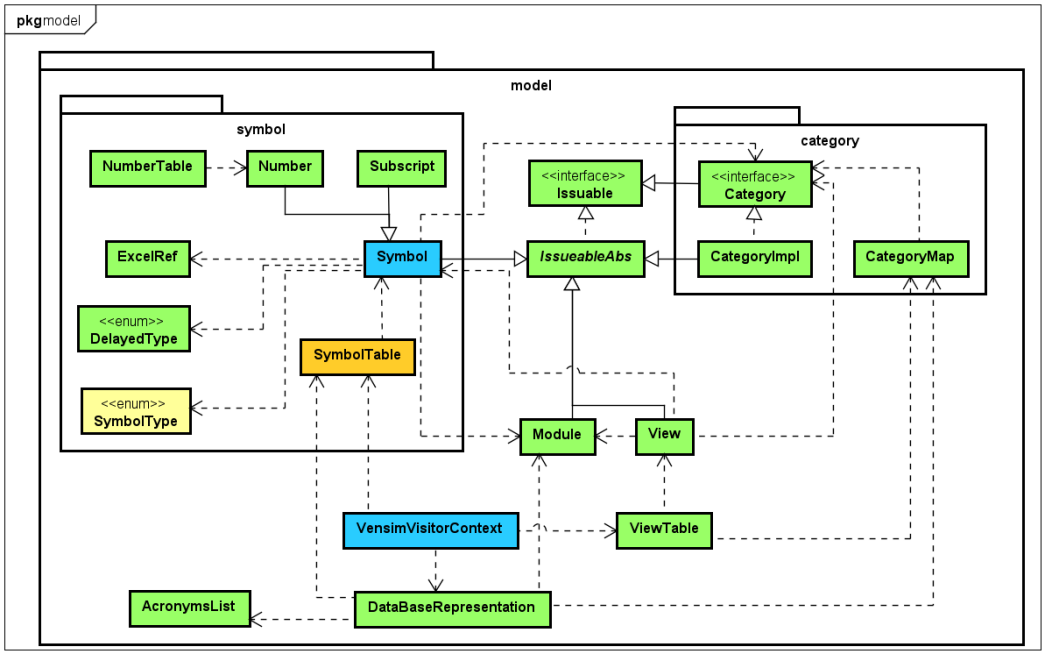

Figura 5.3: Diagrama de clases del paquete model.

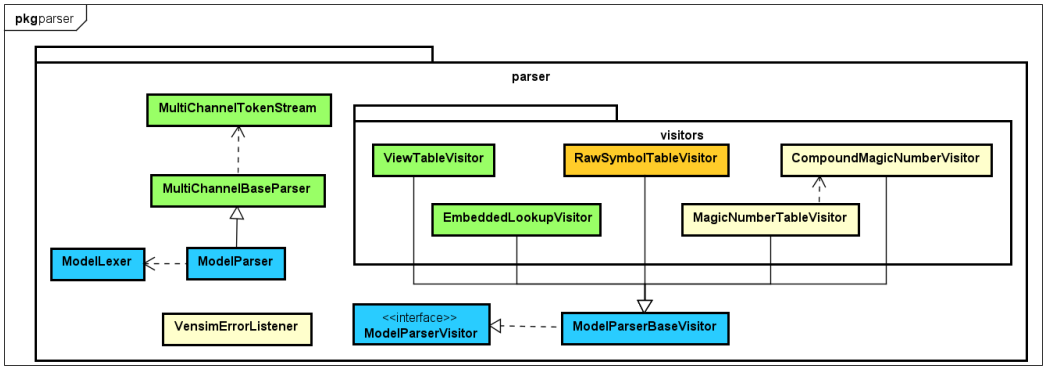

Figura 5.4: Diagrama de clases del paquete parser.

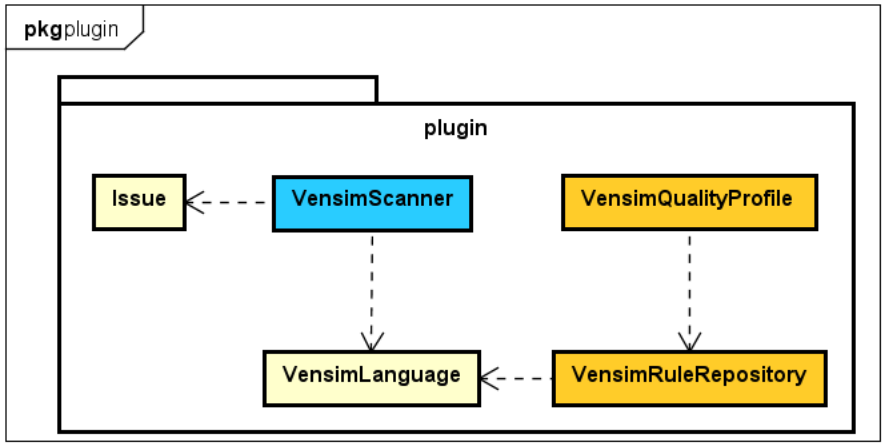

Figura 5.5: Diagrama de clases del paquete plugin.

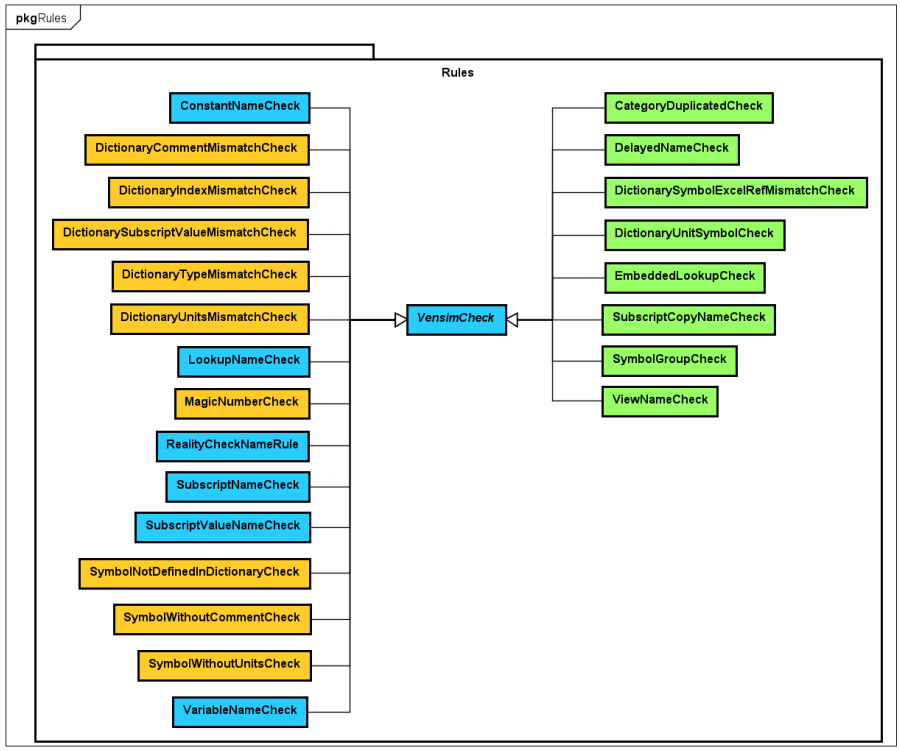

Figura 5.6: Diagrama de clases del paquete rules.

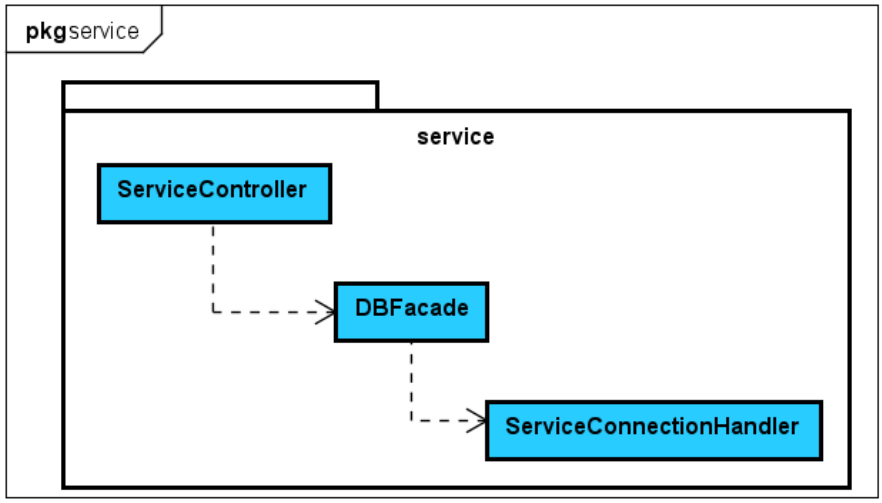

Figura 5.7: Diagrama de clases del paquete service.

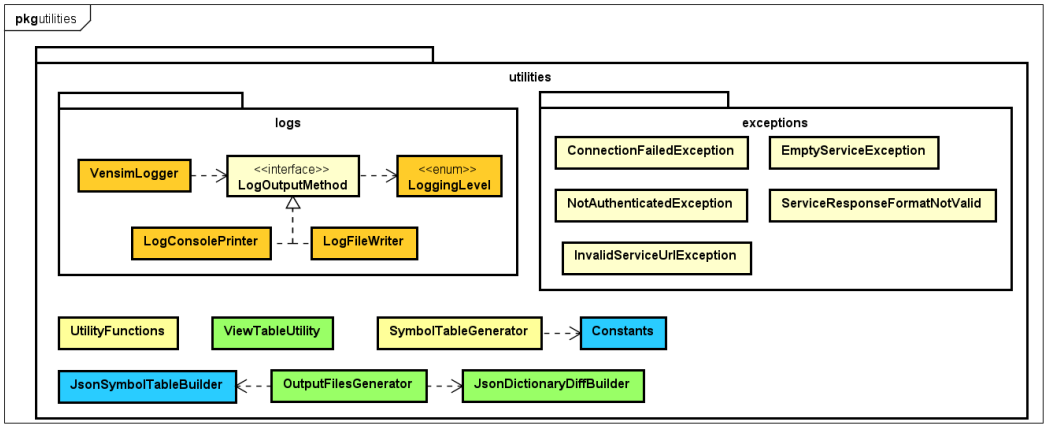

Figura 5.8: Diagrama de clases del paquete utilites.

Capítulo 6

Implementación y pruebas

En este capítulo se explicará la forma de implementación utilizada para el desarrollo del proyecto.

Se dividirá por las historias de usuario que se han generado en el *product backlog* que puede ser encontrado en la Tabla 2.6.

6.1. Historia de usuario 1

Historia: "*Como usuario quiero poder seguir realizando todas las acciones que se pueden realizar con el plugin en su estado inicial*".

Para conseguir el mantenimiento de todas las características con las que ya contaba el *plugin* inicialmente es necesario realizar tes de regresión. Es decir garantizar que todos los tests viejos siguen pasando bien. Es necesario actualizar los tests antiguos para adaptarlos por ejemplo a los nuevos formatos de petición y respuesta del diccionario de símbolos.

6.2. Historia de usuario 2

Historia: "*Como usuario quiero poder analizar la representación textual de la parte de las vistas de los modelos de Vensim almacenada en archivos* `.mdl` *para verificar que la sintaxis es correcta*".

Para llevar a cabo esta historia de usuario es necesario actualizar la gramática inicial. Para ello se utiliza el fichero con la gramática del *plugin* inicial y con la ayuda de un set de modelos de prueba para ir verificando el correcto parseo de lo modelos se fueron llevando a cabo las modificaciones.

Cabe decir que para la revisión de esta gramática se hizo una puesta en común con otro Trabajo de Fin de Grado que estaba en desarrollo a la vez y que además necesitaba una gramática completa del los archivos `.mdl` de Vensim. Este otro TFG es el de Pablo Martínez López llamado "Desarrollo de un plugin para SemanticMerge para facilitar la integración de versiones de archivos Vensim"[30].

A continuación se detalla una lista con todas las modificaciones realizadas en la gramática:

- Se modificó la regla de *sketches* para que se diferenciase entre *sketches*, *graphs* y*metadata* con la posibilidad de que cada uno de estos grupos individualmente no exista en el modelo.
- Se añade la gramática de los gráficos permitiendo poder ser utilizados por el plugin en versiones futuras.
- Se añade la gramática de las vistas.

La forma en la que están escritas las vistas hace que haya conflictos con la gramática inicial, esto se debe al uso de saltos de línea como limitadores, los cuales están eliminados de la gramática de partida por la sencillez que otorgaba hacer esto para realizar el parseo de las declaraciones de los símbolos.

Para resolver este conflicto se contemplan dos soluciones: Modificar la gramática para no eliminar los saltos de línea o eliminar solo los saltos de línea en la parte de las declaraciones de símbolos. Cada solución tiene sus pros y sus contras.

Por ejemplo, el modificar toda la gramática tiene como desventaja el tener que rehacer un trabajo ya hecho y ofuscar la forma en la que se realiza la extracción de información.

Por otro lado, el eliminar los saltos de línea solo en la zona de declaraciones de símbolos no es una característica existente en ANTLR y sería necesario buscar, si fuese posible, cómo llevarlo a cabo.

Al final se decidió optar por eliminar parcialmente los saltos de línea. Esto se debe a que se descubrió que es posible modificar la forma que utiliza ANTLR4 de declarar los lexemas. Para ello se utilizó el proyecto `antlr4multichannel` creado por Stefan Mandel [47]. En este proyecto es posible activar o desactivar dinámicamente los canales de lexemas existentes en ANTLR4. De esta forma es posible asignar los saltos de línea a un canal que es desactivado nada más empieza el análisis del modelo y, una vez se alcanza la sección donde se empiezan a declarar las vistas, este canal es activado para que los saltos de línea puedan ser utilizados en la gramática.

Para utilizar este proyecto, es necesario importar los dos archivos que existen en el: `MultiChannelBaseParser.Java` y `MultiChannelTokenStream.Java`. Una vez importados, es necesario especificar en la gramática que el parser generado por ANTLR4 tendrá como superclase a `MultiChannelBaseParser`. Por otra parte, en el momento de crear el árbol sintáctico de derivación, en vez de utilizar el *stream* por defecto de *tokens* (`CommonTokenStream`), se utiliza el nuevo.

En el Fragmento de código 6.1 se puede ver cómo declarar en la gramática de ANTLR4 la superclase del parser y en el Fragmento 6.2 se puede ver la nueva función que se utiliza para generar el árbol de derivaciones a partir del contenido de un fichero.

```
parser grammar ModelParser;

options {
  superClass=MultiChannelBaseParser;
}

[...]
```

Fragmento de código 6.1: Declaración de la superclase en la gramática

```
protected ModelParser.FileContext getParseTree(String fileContent) {
        ModelLexer lexer = new ModelLexer(CharStreams.fromString(fileContent))
    ;
        MultiChannelTokenStream tokens = new MultiChannelTokenStream(lexer);

        ModelParser parser = new ModelParser(tokens);
        parser.removeErrorListeners();
        parser.addErrorListener(new VensimErrorListener());

        return parser.file();
    }
```

Fragmento de código 6.2: Función para crear el árbol de derivaciones de un modelo

Un problema con el uso de canales de ANTLR4 es que estos solo pueden ser utilizados en una gramática pura de lexemas, y no en una gramática conjunta como la que se utiliza actualmente. Por ello, fue necesario dividir la gramática en dos, una gramática de lexemas "`ModelLexer.g4`" y una gramática de parseo "`ModelParser.g4`".

En el Fragmento de código 6.3 se puede ver cómo se declara el lexema para los saltos de línea en un canal especial y en el Fragmento 6.4 se pueden ver las reglas en las cuales se habilita y deshabilita este canal.

```
lexer grammar ModelLexer;

channels{
    VIEWS
}

NewLine: [\n\r]+ -> channel(VIEWS);
[...]
```

Fragmento de código 6.3: Declaración de un canal de ANTLR4 para los saltos de línea

```
parser grammar ModelParser;

[...]
model: {disable(ModelLexer.VIEWS);}( symbolWithDoc | macroDefinition | group)*
     sketchesGraphsAndMetadata;
[...]
sketches: {enable(ModelLexer.VIEWS);} viewInfo* sketchesDelimiter  {disable(
    ModelLexer.VIEWS);};
```

```
[...]
```

Fragmento de código 6.4: Reglas para habilitar o deshabilitar un canal de ANLTR4

Una vez realizados todos estos cambios en la gramática y comprobado que todos los modelos de prueba son parseados correctamente se pasa a la modificación del *visitor*. Para evitar una complejidad excesiva del *visitor* `RawSymbolTableVisitor`, responsable de extraer la información sobre las declaraciones de los símbolos, se decide crear uno nuevo llamado `ViewTableVisitor`. Este nuevo *visitor*, utilizando la nueva parte de la gramática es capaz de extraer el nombre, módulo, categoría y subcategoría de cada vista declarada en el modelo. Además, de poder extraer que símbolos pertenecen a qué vista, incluido su tipo de pertenencia, si es primario o secundario (shadow). Para poder extraer esta información es necesario saber cuáles son los caracteres utilizados como separadores entre el módulo, la categoría y la subcategoría. Esta información viene dada por el usuario en archivo de configuración de sonar-scanner llamado `sonar-project.properties`, en el fragmento 6.5 se encuentra la manera de poder leer desde el *plugin* las configuraciones asignadas. En este caso las claves utilizada en la configuración son: `vensim.view.module.separator` y `vensim.view.category.separator`.

No siempre serán asignados estos parámetros, por lo que hay que tener gestionado qué hacer en esos casos. Si falta el separador entre categoría y subcategoría, se asume que no existe una subcategoría y que todo lo que haya después del separador de módulo es categoría. Si no hay separador de módulo se asume que todo es módulo. En cualquiera de estos casos, se inhabilita la inyección en diccionario para evitar categorías y módulos posiblemente erróneos.

```
        String moduleSeparator = context.config().get(MODULE_SEPARATOR).orElse
    ("");
        String categorySeparator = context.config().get(CATEGORY_SEPARATOR).
    orElse("");
```

Fragmento de código 6.5: Lectura de la configuración de sonar-project.properties desde el plugin

Al acabar la extracción de información con este *visitor* nos encontramos con dos estructuras de datos, una almacena la información sobre los símbolos declarados y la otra sobre las vistas y los símbolos que pertenecen a ellas. Pero estas dos estructuras no son independientes. Mientras se va extrayendo la información sobre las vistas y qué símbolos pertenecen a ellas, también se van actualizando dichos símbolos para que reflejen cuál es su módulo y categoría principal y, adicionalmente, cuáles son sus módulos secundarios. Para poder saber cuáles son lo símbolos declarados y también para poder ser actualizados con sus respectivos módulos, es necesario que el *visitor* de *views* requiera la tabla de símbolos.

En el Fragmento de código 6.6 se puede ver cómo se llama al *visitor* teniendo en cuenta las posibles combinaciones de posibilidades que existen.

```
public static ViewTable getViewTable(SymbolTable table, ModelParser.
    FileContext context, String moduleSeparator, String categorySeparator) {
        ViewTableVisitor generator = ViewTableVisitor.createViewTableVisitor(
    moduleSeparator, categorySeparator);
        generator.setSymbolTable(table);
        return generator.getViewTable(context);
    }
```

```
6
7       public static ViewTable getViewTable(SymbolTable table,ModelParser.
        FileContext context, String moduleSeparator) {
8           ViewTableVisitor generator = ViewTableVisitor.createViewTableVisitor(
        moduleSeparator);
9           generator.setSymbolTable(table);
10
11          return generator.getViewTable(context);
12      }
13
14      public static ViewTable getViewTable(SymbolTable table,ModelParser.
        FileContext context) {
15          ViewTableVisitor generator = ViewTableVisitor.createViewTableVisitor()
        ;
16          generator.setSymbolTable(table);
17
18          return generator.getViewTable(context);
19      }
```

Fragmento de código 6.6: Funciones utilizadas para llamar al visitor de las views y crear la tabla de vistas

Por último, queda llamar al *visitor* desde el *plugin*. Esto se hará en la clase `VensimScanner`, acto seguido de crear la tabla de símbolos, como se puede ver en el Fragmento 6.7. La función `getViewTable` extrae del archivo de propiedades de sonar-scanner cuáles son los separadores de módulo y categoría especificados por el usuario y llama al método adecuado de entre los tres descritos en el Fragmento 6.6

```
1   public void scanFile(InputFile inputFile) {
2               [...]
3               ModelParser.FileContext root = getParseTree(content);
4               SymbolTable table = SymbolTableGenerator.getSymbolTable(root);
5
6               ViewTable viewTable = getViewTable(root, table);
7               [...]
8           }
```

Fragmento de código 6.7: Creación de la tabla de vistas

Llegado este punto se puede dar por implementada la historia de usuario 2.

6.3. Historia de usuario 3

Historia: "*Como usuario quiero poder filtrar qué símbolos generan issues mediante el módulo al que pertenecen*".

Una vez se ha acabado la historia de usuario 2 se tiene la información necesaria para poder realizar la implementación de esta historia de usuario.

Para empezar, si se quiere filtrar la lista de símbolos mediante el nombre de uno de los módulos es necesario que el usuario declare el nombre. Este se pondrá en el archivo de confi-

guración de sonar-scanner. La lista con todas las configuraciones posibles de este *plugin* puede encontrare en la Sección 4.3.4. En este caso se utilizará la clave `vensim.view.module.name`

Si el usuario ha seleccionado un módulo a filtrar, será necesario iterar por la lista de símbolos comprobando cuáles pertenecen a ese módulo, ya sean primarios o secundarios, y marcando cuáles deben ser filtrados. La función responsable de aplicar este filtro puede verse en el Fragmento 6.8.

```
public static void filterModule(SymbolTable table, String moduleName) {
        for (Symbol symbol : table.getSymbols()) {
            boolean filtered = true;
            for (Module module : symbol.getModules()) {
                if (module.getName().equals(moduleName)) {
                    filtered = false;
                    break;
                }
            }
            symbol.setFiltered(filtered);
        }
    }
```

Fragmento de código 6.8: Filtrado de símbolos por modulo

Se decide marcarlos como filtrados, y no borrarlos, por el hecho de que existen dependencias entre símbolos a la hora de comprobar las reglas de calidad, por lo que si en este momento los símbolos filtrados fuesen borrados, podría haber tanto falsos positivos como falsos negativos a la hora de comprobar las reglas.

Una vez se tienen marcados cuáles elementos están filtrados y cuáles no, es necesario realizar una comprobación a la hora de generar las *issues* que compruebe esta propiedad de los símbolos. Esto se lleva a cabo utilizando la interfaz existente que implementan todas las reglas de calidad llamada `VensimCheck`. Se añade una función llamada `addIssue` la cual es la responsable de añadir una *issue* a la lista de *issues* cuando sea necesario. Como se quiere añadir una función ya implementada, se transforma la interfaz y se convierte en una clase abstracta, esta clase puede ser encontrada en el Fragmento de código 6.9.

```
public abstract class VensimCheck {

    abstract void scan(VensimVisitorContext context);

    public void addIssue(VensimVisitorContext context, Issue issue, boolean
    isFiltered){
        if(!isFiltered){
            context.addIssue(issue);
        }

    }
}
```

Fragmento de código 6.9: Clase abstracta padre de todas las clases de reglas de control de calidad

Por otra parte, cuando existe un filtro en activo, es necesario también restringir los elementos que se inyectan en el diccionario de datos del proyecto. Solo se deben inyectar el

módulo filtrado y sus símbolos. Para llevar esto a cabo, se decide que la mejor opción es modificar la clase que almacena la información sobre las vistas `ViewTable` y simular que solo se ha detectado el módulo a filtrar. La manera de simularlo se puede encontrar en el Fragmento 6.10

```
1 if (!moduleName.isEmpty()) {
2             [...]
3             viewTable.setModules(Set.of(new Module(moduleName)));
4         }
```

Fragmento de código 6.10: Gestión del filtrado por módulo en la inyección

Esta simulación hará que solo se inyecten los símbolos y el módulo pertinente, la explicación de por qué esto es así se puede encontrar en la sección de las historias de usuarios 6.1 y 6.2.

Una vez añadida esta función y modificadas todas las reglas de control de calidad que deban llamarla para poder añadir *issues*, se da por concluida la historia de usuario 3.

6.4. Historia de usuario 4

6.4.1. Historia de usuario 4.1

Historia: "*Como usuario quiero que no se generen issues si una variable sigue el convenio de nombres, pero contiene en su nombre un acrónimo definido en el diccionario de datos del proyecto*".

Pasamos ahora a hacer la primera mejora de una de las reglas que ya existen en el *plugin* de inicio. Se trata de un problema que se da a la hora de comprobar los nombres de los símbolos de tipo variable. El nombre de estos siempre tiene que estar escrito en minúsculas, pero existe una excepción. Se trata del caso del uso de acrónimos. Se pueden añadir acrónimos que estén validados en el diccionario de datos del proyecto.

Esta excepción no está contemplada en el *plugin* de partida. Si, por ejemplo, tenemos un acrónimo valido con nombre `H2O` y una variable con nombre `H2O_delta_change`, en el *plugin* de inicio se marcaría como que no sigue el convenio de nombres cuando realmente sí que lo sigue.

Para llevar a cabo la implementación de esta historia de usuario, primero necesitamos recuperar del diccionario de datos del proyecto esa lista de acrónimos. Esta lista se puede recuperar del diccionario de datos gracias a la nueva incorporación de un *endpoint* en la API llamado `qaGetAcronyms`, descrito en la Sección 5.1. En el anexo C se puede encontrar la estructura detallada de todos los *endpoints* que tiene la API.

Como se prevé que se vayan a crear nuevas llamadas a futuros *endpoints* del diccionario de símbolos, se decide refactorizar la clase responsable de enviar las peticiones (`ServiceConnectionHandler`) de un modo que existan dos métodos (`sendPOSTRequest` y

`sendGETRequest`) que reciban la url de destino y los posibles datos a enviar, de esta forma crear nuevas llamadas a `endpoints` será solo añadir una nueva función con la url de destino y los datos necesarios.

Una vez se recupera esta lista de acrónimos, es necesario almacenarla de un modo que las reglas de control puedan acceder a ella. Actualmente estas reglas reciben como parámetro un objeto de tipo `VensimVisitorContext` el cual contiene información sobre la tabla de símbolos, tanto la local como la recibida del diccionario de datos, y el árbol de derivaciones generado por el *parser*.

En esta clase, teniendo en cuenta que en un futuro se recibirán más datos del diccionario de datos, se decide crear otra nueva clase, `DataBaseRepresentation` responsable de almacenar toda la información proveniente del diccionario de símbolos. Por lo tanto en la clase `VensimVisitorContext`, se sustituye la tabla de símbolos remota por esta nueva clase. `DataBaseRepresentation` actualmente contiene la tabla de símbolos remota y la tabla de acrónimos.

Teniendo la tabla de acrónimos ya en `VensimVisitorContext`, la regla de control de los nombres de las variables ya puede acceder a ella y, por lo tanto, comprobar la existencia de acrónimos. Para realizar esta comprobación se decide realizar un algoritmo que pase a minúsculas todas las ocurrencias de acrónimos en una copia del nombre de la variable antes de comprobar si sigue el convenio de nombre. Se decide pasar a minúsculas los acrónimos y no borrarlos directamente para evitar situaciones en las que se pueda generar dos barras bajas seguidas lo cual no seguiría la convención de nombre. Ej: current_H2O_stored

La función que contiene este algoritmo puede encontrarse en el Fragmento 6.11.

Una vez llamada esta función, la regla de control puede continuar con su flujo original con una ligera excepción: en el caso de que no se pueda conectar con el diccionario de símbolos sea por el motivo que sea, no se podrán recuperar los acrónimos válidos, dado este caso pueden surgir gran cantidad de falsos positivos. Por ese motivo, si el *plugin* no ha podido conectar y recuperar la lista de acrónimos del diccionario de símbolos, en cada *issue* de la regla de nombrado de variable que se genere se añadirá el siguiente comentario: "*WARNING: Could not connect with the acronyms database. This variable could be well written.*";

```
1  private  boolean checkIfVariableHaveAnAcronym(String name, List<String>
       acronyms){
2          String trimmedName = name;
3          for(String acr : acronyms){
4              if(trimmedName.matches(".*(^|_|\")"+acr+"($|_|\").*"))
5                  trimmedName = trimmedName.replace(acr, acr.toLowerCase());
6          }
7          return  checkVariableFollowsConvention(trimmedName);
8      }
```

Fragmento de código 6.11: Función para comprobar si existen acrónimos en los nombres de las variables

6.4.2. Historia de usuario 4.2

Historia: "*Como usuario quiero ser informado si existe algún símbolo de tipo lookup el cual tenga incrustada su declaración en el modelo*".

Para implementar esta regla primero es necesario entender que es un *lookup* incrustado.

Los *lookups* se pueden declarar principalmente de dos maneras, con una llamada a una función o embebiendo los datos. En el fragmento 6.12 se pueden ver las dos formas. Se puede apreciar que en el caso del *lookup* incrustado se tiene todos sus datos en forma de lista y hace que sea difícil de leer, de ahí el motivo de que se quieran evitar.

```
Incrustado:
"Net_coal_extraction_de_Castro_PhD_-_Scen_I"(
[(0,0)-(10,10)],(1985,1378.15),(1986,1422.43),(1987,1466.37),(1988,1509.97)
    ,(1989,1553.27\
),(1990,1596.27),(1991,1639),(1992,1681.45),(1993,1723.6),(1994,1765.43)
    ,(1995,1806.88\
),(1996,1847.87),(1997,1888.31),(1998,1928.08),(1999,1967.04),(2000,2005.02)
    ,(2001,\
2041.85),(2002,2077.35),(2003,2111.34),(2004,2143.61),(2005,2174.01)
    ,(2006,2202.36)\
,(2007,2228.55),(2008,2252.45),(2009,2274),(2010,2293.18),(2011,2310)
    ,(2012,2324.5)\
,(2081,925.587),(2082,898.96),(2083,871.837),(2084,844.332),(2085,816.56)
    ,(2086,788.637\
,(2087,760.675),(2088,732.784),(2089,705.067),(2090,677.624),(2091,650.543)
    ,(2092,\
623.906),(2093,597.787),(2094,572.25),(2095,547.351),(2096,523.135)
    ,(2097,499.642),\
(2098,476.902),(2099,454.936),(2100,433.76))
~ MToe/Year
~ |

Llamando a una función:

GDPpc_ANNUAL_GROWTH_SSP2_LT(
  GET_DIRECT_LOOKUPS('/model_parameters/economy/economy.xlsx', 'World', '
    time_index_GDPpc'\
    , 'SSP2_GDP'))
  ~ Dmnl
  ~   |
```

Fragmento de código 6.12: Diferencias entre un lookup incrustado y un lookup con una función

Para implementar esta historia de usuario se crea una nueva regla de control, llamada `EmbeddedLookupCheck` y será la responsable de generar issues en los símbolos de tipo *lookup* incrustados.

Se crea también un nuevo *visitor* que detecte dónde hay *lookups* incrustados. Éste es llamado por la propia regla de control. La forma de funcionar se basa en buscar en el árbol de derivaciones un bloque de tipo `numberList` o `lookupPointList` los cuales son las sintaxis utilizadas para incrustar los datos de un *lookup* y contar la cantidad de repeticiones que tiene,

esta son añadidas a una tabla junto al símbolo al que pertenecen. Al acabar se devuelve esta tabla a la regla.

Para tener en cuenta la posibilidad de filtrado por módulo, es necesario incluir la tabla de símbolos en el *visitor*. En el Fragmento 6.13 se puede ver como se marcan a filtrar los *lookups* incrustados.

```
@Override
public Void visitLhs(ModelParser.LhsContext ctx) {

    if (symbols == null) {
        logger.unique("Symbol table unassigned in EmbeddedLookupVisitor",
    LoggingLevel.INFO);
    }
    else if (!symbols.hasSymbol(ctx.Id().getText())) {
        logger.error("Found symbol \"" + ctx.Id().getText() + "\" that is not
    in the symbol table");
    }
    else {
        Symbol symbol = symbols.getSymbol(ctx.Id().getText());
        isSymbolFiltered = symbol.isFiltered();
    }
    return null;
}
```

Fragmento de código 6.13: Filtrado de lookups incrustados

Cuando la regla recibe la lista, itera sobre ella anotando *issues* en todos los símbolos en los que se ha detectado un conjunto de datos incrustado.

La gravedad de la *issue* depende del número de datos incrustados. La regla tiene un límite en el cual pasa de ser una incidencia de tipo `INFO` a una `MAJOR`, este límite se parametriza para poder ser modificado en un quality profile.

6.4.3. Historia de usuario 4.3

Historia: "*Como usuario quiero ser informado si existe algún símbolo dentro del grupo de control que no sea de control y viceversa*".

Esta historia de usuario usa el concepto de grupo [58]. Un grupo en *Vensim* permite realizar agrupaciones de símbolos. Por defecto existe el grupo de control el cual contiene las variables que existen en un modelo de *Vensim* por defecto. En el Fragmento de código 6.14 se pude ver la sintaxis de un grupo en un modelo. Un símbolo pertenece al grupo más próximo superior.

```
********************************
Control
********************************~
Variables relating to vensim control
|
```

Fragmento de código 6.14: Sintaxis de un grupo en un modelo

Para llevar a cabo la implementación, se amplia `RawSymbolVisitor` para que guarde los grupos en los símbolos. Para ello cada vez que se detecta un grupo en el modelo se guarda en el *visitor* para poder añadirlo en todos los símbolos próximos hasta encontrar el siguiente grupo que sobrescriba este grupo. Existe un caso especial al principio del modelo y es que puede haber símbolos que no pertenecen a ningún grupo. El grupo es guardado en un nuevo atributo de los símbolos.

Con esta información almacenada se puede crear la nueva clase para la regla (`SymbolGroupCheck`) la cual itera por toda la lista de símbolos comprobando si cada símbolo está en un grupo adecuado o no.

Los símbolos que deben aparecer exclusivamente en el grupo de control se parametrízan para que estos puedan ser modificados si en un futuro cambian. Por defecto son los siguientes:

- TIME
- TIME STEP
- INITIAL TIME
- FINAL TIME
- SAVEPER

La lógica que utiliza la regla es bastante directa, generar una *issue* en un símbolo cuando este está en el grupo de control, pero no es uno de los símbolos permitidos, o bien cuando el símbolo no aparece en el grupo de control, pero sí que es uno de los símbolos que debería.

6.4.4. Historia de usuario 4.4

Historia: "*Como usuario quiero ser informado si existe alguna vista que no siga las reglas de nombrado*".

Esta historia de usuario necesita la creación de una nueva regla, el nombre de la clase asignada nueva es `ViewNameCheck`.

Esta nueva regla utiliza la tabla de vista, que es iterada para comprobar las vistas una a una.

Las vistas tienen información sobre su módulo y su(s) categoría(s), por lo que para comprobar si el nombre es válido se comprueba si los nombres de los módulos y de las categorías son válidos. En el Fragmento 6.15 se puede ver la función para realizar la validación. Esta comprobación utiliza una expresión regular parametrizada.

```
1 private boolean generateIssue(View view) {
2     if (view.getModule() == null || !view.getModule().getName().matches(
      getRegexp()))
3         return true;
4
```

```
    if (view.getCategory() == null || !view.getCategory().getName().matches(
    getRegexp()))
        return true;
    return view.getSubcategory() != null && !view.getSubcategory().getName().
    matches(getRegexp());
}
```

Fragmento de código 6.15: Validación del nombre de las vistas

Si una vista no tiene un nombre correcto debe ser invalidada y todos sus símbolos también. No se debe invalidar automáticamente también el módulo o la categoría ya que estos pueden estar bien escritos. Para ello es necesario volver a comprobar los nombres antes de invalidarlos. En el Fragmento 6.16 se puede ver como cuando se invalida una vista, se invalidan automáticamente los símbolos y en el Fragmento 6.17 se puede ver cómo se comprueba si se puede invalidar el módulo o las categorías.

```
public class View extends IssuableAbs {
    [...]
    @Override
    public void setAsInvalid(String invalidReason) {
        super.setAsInvalid(invalidReason);
        primarySymbols.forEach(symbol -> symbol.setAsInvalid(invalidReason));
    }
}
```

Fragmento de código 6.16: Invalidación automática de los símbolos primarios en una vista

```
private void invalidateView(View view){
        view.setAsInvalid(this.getClass().getSimpleName());

        if (view.getModule() != null && !view.getModule().getName().matches(
    getRegexp())){
            view.getModule().setAsInvalid(this.getClass().getSimpleName());
        }
        if (view.getCategory() != null && !view.getCategory().getName().
    matches(getRegexp())){
            view.getCategory().setAsInvalid(this.getClass().getSimpleName());
        }
        if (view.getSubcategory() != null && !view.getSubcategory().getName().
    matches(getRegexp())){
            view.getSubcategory().setAsInvalid(this.getClass().getSimpleName()
    );
        }
    }
```

Fragmento de código 6.17: Comprobación de si invalidar el módulo y las categorías de una vista

En la descripción de las *issues* se desea añadir los separadores que se deberían usar, pero para ello es necesario conocer los separadores que existen entre el módulo y la categoría así como entre la categoría y la subcategoría. Para poder acceder a esta información es necesario modificar los datos que reciben las reglas almacenados en `VensimVisitorContext` y añadir el objeto que almacena las propiedades declaradas por el usuario en `sonar-project.properties`.

6.4.5. Historia de usuario 4.5

Historia: "*Como usuario quiero ser informado si existe algún símbolo cuya unidad no se encuentre en el sistema de unidades definido en el diccionario de datos del proyecto*".

Para implementar esta regla es necesario incluir una nueva llamada a la API que devuelva el conjunto de unidades válidas en el proyecto. Esta comunicación se realiza de manera similar a las ya existentes,

El conjunto de unidades válidas para el proyecto es almacenado en `DataBaseRepresentation` para que pueda ser accedido por las reglas.

Se crea la regla `DictionaryUnitSymbolCheck` la cual itera por toda la tabla de símbolos y va comprobando que las unidades de cada uno de ellos están contenidas en el conjunto de unidades válidas. Existe una excepción, los símbolos de tipo `function`, `subscript` y los `subscript_value` son ignorados debido a que son adimensionales.

6.4.6. Historia de usuario 4.6

Historia: "*Como usuario quiero ser informado si existe alguna copia de un símbolo de tipo subscript y no siga las reglas de nombrado*".

En esta historia de usuario se utiliza parte de la gramática que hasta ahora no se utilizaba, se trata de la copia de *subscripts*.

La información de si un *subscript* está definido como una copia de otro se almacena en una nueva clase que hereda de los símbolos.

Se modifica el *visitor* `RawSymbolVisitor` para que al detectar la regla de copiar un *subscript* genere uno nuevo pero con el *flag* de copia activo. Este *subscripts* se va a almacenar de la misma manera que el resto de símbolos en la tabla de símbolos.

Se crea una nueva regla llamada `SubscriptCopyNameCheck` que funciona de manera similar al resto de reglas de calidad que comprueban los nombres de los símbolos. Es decir, itera toda la lista de símbolos seleccionando solo los de tipo *subscript* que sean copias y comprueba mediante una expresión regular, que el nombre del símbolo cumpla la convención de nombres.

Una cualidad significativa de esta regla es que usa el polimorfismo existente en Java como se puede ver en el Fragmento 6.18

```
1 @Override
2 public void scan(VensimVisitorContext context) {
3     SymbolTable table = context.getParsedSymbolTable();
4
5     for (Symbol symbol : table.getSymbols()) {
6         if (symbol.getType() == SymbolType.SUBSCRIPT) {
7             Subscript subscript = (Subscript) symbol;
```

```
8               if (subscript.isCopy() && !checkSubscriptNameFollowsConvention(
    subscript.getToken())) {
9                   [...]
10              }
11          }
12      }
13  }
```

Fragmento de código 6.18: Utilización de polimorfismo en la regla SubscriptCopyNameCheck

6.4.7. Historia de usuario 4.7

Historia: "*Como usuario quiero ser informado si existe alguna discrepancia entre las referencias a tablas Excel encontradas en local y las referencias guardadas en el diccionario de datos del proyecto*".

Esta historia de usuario utiliza la información que algunos símbolos tienen sobre las referencias a tablas Excel externas que existen, por lo que es necesario extraer esta información. En la Sección de la historia de usuario 6.5 se explica estos datos que se quieren extraer y cómo se extraen.

Para poder realizar esta comprobación es necesario recuperar esta información de la API del diccionario de datos, por lo que la estructura de `qaGetSymbolDefinition` es modificada para que contenga la información sobre los Excels. La estructura de esta llamada se puede ver en el Anexo C.

Se modifican las clases y funciones responsables de generar la tabla de símbolos a partir de los datos recibidos del diccionario de datos del proyecto para que añadan la información referente a los Excel en los símbolos en los que sea necesario. Una vez ya se tiene la información de los Excels del diccionario, se crea una nueva clase para la regla de control de calidad llamada `DictionarySymbolExcelRefMismatchCheck`.

Esta regla funciona como el resto de reglas que comprueba desigualdades entre el análisis local y la información almacenada en el diccionario de datos del proyecto. Es decir, solo se comprueban si existe la tabla de símbolos del diccionario y cuando es llamada itera la tabla de símbolos buscando por cada uno de ellos su idéntico en la tabla recuperada del diccionario de datos. Si encuentra el símbolo entonces comprueba la igualdad entre las dos instancias de `ExcelRef`, si no fuesen iguales se genera una *issue*.

6.4.8. Historia de usuario 4.8

Historia: "*Como usuario quiero ser informado si existe alguna subcategoría cuyo nombre no sea único en el conjunto de categorías y subcategorías*".

Esta historia de usuario es requerida por una limitación que existe en el diccionario de datos, el cual no permite que existan dos categorías o subcategorías con el mismo nombre.

Esto se debe a que en el diccionario de datos las categorías son guardadas en una única tabla independientemente de su jerarquía. Si una categoría es subcategoría de otra simplemente tendrá una referencia a ella. En esta tabla está especificado que los nombres sean únicos, es por esto que hay que evitar la duplicación de nombre de categorías independientemente de su nivel jerárquico.

Toda la información necesaria para implementar esta regla ya está almacenada en la clase `VensimVisitorContext` que reciben todas las reglas como parámetro a la hora de analizar.

Se crea la clase `CategoryDuplicatedCheck` para implementar esta regla.

La lógica de esta regla es más compleja que el resto por todas las situaciones que se pueden dar. A continuación se describen todas las posibles situaciones detectadas y cómo se han gestionado:

- Dos categorías con el mismo nombre:

  Se trata de la misma categoría, no es una duplicación.

- Dos subcategorías con el mismo nombre de la misma categoría:

  Se trata de la misma subcategoría, no es una duplicación.

- Dos subcategorías con el mismo nombre de distintas categorías:

  Se trata de una duplicación, es necesario marcar una de las dos como inválida y generar una *issue*. Para elegir cuál de las dos invalidar, primero se comprueba si hay alguna ya guardada en el diccionario de datos del proyecto. Si una de las dos está ya almacenada, se invalida la otra. En caso de que ninguna de las dos esté en el diccionario de datos, se invalida la primera en aparecer en la lista de subcategorías.

- Una categoría y una subcategoría con el mismo nombre:

  Se trata de una duplicación, es necesario marcar una de las dos como inválida y generar una *issue*. Para elegir cuál de las dos invalidar primero se comprueba si hay alguna ya guardada en el diccionario de datos del proyecto. Si una de las dos está ya almacenada, se invalida la otra. En caso de que ninguna de las dos esté en el diccionario de datos, se invalida la subcategoría.

La implementación consta de dos bucles unos primero para las subcategorías y luego otro para las categorías. En el Fragmento 6.19 se pueden ver estos dos bucles.

```
1  while (!subcategoryList.isEmpty()) {
2              Category currentSubcategory = subcategoryList.remove(0);
3              if (dbSubcategoryList.contains(currentSubcategory)) {
4                  alreadyInDB.add(currentSubcategory);
5              } else {
6                  if (generateIssue(subcategoryList, categoryList,
       dbCategoryList, alreadyInDB, currentSubcategory)) {
7                      [...]
8                  }
9              }
10         }
```

```
        while (!categoryList.isEmpty()){
            Category currentCategory = categoryList.remove(0);
            if(dbSubcategoryList.stream().anyMatch(subcategory -> subcategory.
    getName().equalsIgnoreCase(currentCategory.getName()))){
                [...]
            }
        }
```

Fragmento de código 6.19: Implementación de la comprobación de categorías duplicadas

6.4.9. Historia de usuario 4.9

Historia: "*Como usuario quiero ser informado si existe alguna variable de tipo delayed que no cumpla la convención de nombres*".

Para implementar esta historia de usuario se necesita primero poder detectar cuáles son las variables de tipo *delayed*.

Las variable *delayed* son aquellas que a la hora de realizar su declaración usan una de las siguientes funciones llamadas funciones dinámicas: `DELAY BATCH`, `DELAY CONVEYOR`, `DELAY FIXED`, `DELAY INFORMATION`, `DELAY MATRIAL`, `DELAY N`, `DELAY PROFILE`, `DELAYP`, `DELAY1`, `DELAY3`, `SMOOTH`, `SMOOTH3` o `SMOOTH N`.

La regla que se quiere generar tiene una distinción entre dos tipos de variables *delayed* según si la variable está relacionada con el símbolo `TIME STEP` o no. Todas las funciones de la lista anterior tienen como segundo parámetro el *delay* con las que se van a definir. Es en este segundo parámetro donde puede encontrarse el símbolo `TIME STEP` o no.

Se empieza modificando el *visitor* `RawSymbolVisitor` para que al detectar el uso de una de estas funciones se marque el símbolo que está siendo declarado como *delayed*. En el Fragmento 6.20 se ve como cuando se detecta una llamada a una función, se comprueba si es a una función dinámica.

```
@Override
    public List<Symbol> visitCall(ModelParser.CallContext ctx) {
        [...]
        if (VENSIM_DYNAMIC_FUNCTIONS.contains(token)) {

            actualSymbol.setDelayed(DelayedType.DELAYED);

            ModelParser.ExprContext secondArgument = ctx.exprList().expr(1);
            if (secondArgument instanceof ModelParser.SignExprContext) { //
    Only if the delay is TIME STEP, the delay type is TIME_STEP_DELAYED.
                ModelParser.ExprAllowSignContext signedType = ((ModelParser.
    SignExprContext) secondArgument).exprAllowSign();
                if (signedType instanceof ModelParser.VarContext) {
                    String tokenOfArgument = ((ModelParser.VarContext)
    signedType).Id().getSymbol().getText();
                    if(TIME_STEP_TOKEN.contains(tokenOfArgument)){
                        actualSymbol.setDelayed(DelayedType.TIME_STEP_DELAYED)
    ;
                    }
```

```
                }
            }
        }
        [...]
    }
```

Fragmento de código 6.20: Detección de funciones dinámicas en la declaración de una variable

Al tener dos tipos de variables *delayed*, se utiliza un enumerado (`DelayedType`) para contener esas dos posiblidades: `DELAYED` y `TIME_STEP_DELAYED`. Si una variable no es *delayed* ese campo se deja en nulo.

Con esta información se puede generar una nueva regla que itere por todas las variables de tipo *delayed* y compruebe que sigue la convención de nombres.

6.5. Historia de usuario 5

6.5.1. Historia de usuario 5.1

Historia: "*Como usuario quiero poder configurar la regla de nombrado utilizada en cada regla que contenga una*".

Esta historia de usuario tiene como objetivo modificar todas las reglas que tengan una expresión regular y transformarlas para que pasen de tener incrustada la expresión en el código a que puedan ser parametrizadas utilizando los perfiles de SonarQube.

En el Fragmento 6.21 se puede ver como es la sintaxis del *framework* de SonarQube para poder declarar una variable como parametrizada y a la vez al tratarse de una expresión regular, el como se gestiona que un usuario añada una errónea. Este fragmento es añadido en todas las reglas que tienen una expresión regular.

```
public static final String DEFAULT_REGEXP = "([a-z0-9]+_)*[a-z0-9]+";
@RuleProperty(
        key = "variable-name-regexp",
        defaultValue = DEFAULT_REGEXP,
        description = "The regexp definition of a variable symbol name.")
public final String regexp = DEFAULT_REGEXP;

private String getRegexp() {
    try {
        Pattern.compile(regexp);
        return regexp;
    } catch (PatternSyntaxException exception) {
        logger.unique("The rule " + NAME + " has an invalid configuration: The
     selected regexp is invalid. Error: " + exception.getDescription(),
                    LoggingLevel.ERROR);
        return DEFAULT_REGEXP;
    }
}
```

Fragmento de código 6.21: Declaración de una expresión regular parametrizada desde SonarQube

6.5.2. Historia de usuario 5.2

Historia: "*Como usuario quiero poder configurar lo límites numéricos en cada regla que contenga uno*".

Esta historia de usuario es muy similar a la anterior. Al tratarse de un número no hace falta realizar la función adicional para verificar que la expresión regular sea correcta.

Este cambio afecta únicamente a las reglas `MagicNumberCheck` y `EmbeddedLookupCheck`.

6.5.3. Historia de usuario 5.3

Historia: "*Como usuario quiero poder decidir cuáles serán los separadores utilizados en el nombre de una vista para diferenciar módulo, categoría y subcategoría*".

La selección de separadores entre módulo-categoría y categoría-subcategoría se realizará mediante el uso del archivo de configuración `sonar-project.properties`. Los nombres de las propiedades nuevas son:

- `vensim.view.module.separator` y
- `vensim.view.category.separator`

Las instrucciones de uso de estos dos parámetros se pueden encontrar en la plantilla que se muestra en el Anexo A.2 junto con el resto de configuración disponible en el *plugin*.

Para acceder a esta información desde el **plugin**, el *framework* de SonarQube de Java contiene una clase llamada `SensorContext` la cual se pasa como argumento a la función de ejecución del **plugin**. En el Fragmento 6.5 se puede ver como se implementa la recuperación de los separadores, aunque se puede extrapolar a cualquier tipo de propiedad.

6.5.4. Historia de usuario 5.4

Historia: "*Como usuario quiero poder decidir si se inyectan nuevos elementos al diccionario de símbolos del proyecto o no*".

Para poder seleccionar por parte del usuario cuándo y qué poder inyectar, se generan cuatro nuevas propiedades en el archivo de configuración de SonarQube:

- `vensim.dictionary.inject`,
- `vensim.dictionary.inject.modules`
- `vensim.dictionary.inject.categories` y
- `vensim.dictionary.inject.symbols`

En la plantilla que se muestra en el Anexo A.2 se puede encontrar la forma de utilización de todos ellos.

Su implementación consiste en comprobar el valor booleano de cada una de las propiedades antes de llamar al método responsable de hacer la inyección de módulos, categorías y símbolos.

6.5.5. Historia de usuario 5.5

Historia: "*Como usuario quiero poder decidir el nombre de la carpeta donde se generarán los documentos auxiliares*".

Para realizar esta historia de usuario es necesario que el usuario pueda proporcionar el nombre de la carpeta que quiere utilizar, esta podrá ser añadida en el archivo de configuración de la misma manera que las dos historias de usuario anteriores. El nombre de esta nueva propiedad es `vensim.auxiliaryFiles.directoryName`.

Es necesario añadir en la clase responsable de generar los archivos de salida (`OutputFilesGenerator`) un parámetro con la ruta de la carpeta de destino. También hay que guardar en esta carpeta el archivo de *log* si se ha decidido generar.

La implementación desarrollada permite crear la carpeta de destino por parte del *plugin*.

6.5.6. Historia de usuario 5.6

Historia: "*Como usuario quiero poder decidir si se genera el archivo auxiliar con las diferencias encontradas entre local y el diccionario de símbolos*".

Esta historia sigue el mismo flujo que las últimas tres historias de usuario: se crea una nueva propiedad en el archivo de configuración llamada `vensim.dictionary.getDiff` y luego esta propiedad es leída en el *plugin*, la lectura se pasa a `OutputFilesGenerator` que, dependiendo de si el valor es verdadero o falso, genera o no el archivo de diferencias entre el diccionario de símbolos y el análisis local.

Como extra en esta historia de usuario se analiza que pueden ocurrir configuraciones del archivo de propiedades que no sean válidas, por ejemplo querer generar la diferencia con el diccionario, pero que no exista la forma de conectar con el mismo. Para estas situaciones se avisa por el *log* de que no se ha podido generar el archivo de diferencias por no haber podido recuperar la información necesaria del diccionario.

6.6. Historia de usuario 6

6.6.1. Historia de usuario 6.1

Historia: "*Como usuario quiero que los módulos nuevos detectados sean inyectados al diccionario de símbolos del proyecto*".

Para poder llevar a cabo esta historia de usuario es necesario cumplir una serie de requisitos:

- Obtener una lista con todos los módulos detectados en el análisis.

  Esta parte ya ha sido implementada en la historia de usuario 2 la extrae la información de las vistas del modelo. Esta información es almacenada en la clase `ViewTable`, la cual contiene tres listas almacenadas, una lista de vistas, una lista de categorías y una lista de módulos. De aquí podemos conseguir la lista de módulos.

- Obtener una lista de todos los módulos guardados en el diccionario de símbolos para así evitar inyectar módulos ya presentes en el diccionario.

  Para obtener la lista de módulos del registrados en el diccionario de datos es necesario utilizar una de los nuevos servicios que ofrece la API. En este caso me refiero a `qaGetModules`, para ello se genera una estructura de llamadas similar a la que ya existía en el *plugin* actual que se utilizaba para hacer llamadas a `qaGetSymbolsDefinition`.

  Esta se basa en una estructura de tres capas representadas por tres clases distintas. La primera `ServiceController` es la responsable de identificar qué datos son los que hay que mandar exactamente. Esto lo puede hacer mediante comprobaciones lógicas y filtrado. La segunda capa `DBFacade` transforma los datos a enviar de su forma de objeto Java a una estructura JSON y viceversa. La última capa, `ServiceConnectionHandler` es la responsable de mandar las petición al diccionario de datos.

  En el caso de los módulos se necesita una llamada de tipo `GET` que devuelve una lista con todos los módulos almacenados en el diccionario. Esta lista es finalmente almacenada en la clase `DataBaseRepresentation`, responsable de mantener todos los datos del diccionario de datos del proyecto en el *plugin*.

- Tener una manera de inyectar módulos al diccionario de datos.

  Igual que el anterior punto, se realiza utilizando la estructura existente del *plugin* para las llamada que hace a `qaAddSymbolDefinition`, en este caso se harán peticiones al nuevo servicio de la API llamado `qaAddModules`.

  Para realizar el filtrado de que módulos es necesario mandar, se necesitan pasar como parámetros las dos listas de módulos previamente explicadas, con ellas se filtran los símbolos de la manera que se puede ver en el Fragmento 6.22.

```
1 public void injectNewModules(Set<Module> modulesList, Set<Module>
      dbModules) {
2         [...]
3         List<String> newModules = modulesList.stream()
4                 .filter(module -> !dbModules.contains(module))
```

```
5                 .filter(Module::isValid)
6                 .map(Module::getName)
7                 .collect(Collectors.toList());
8
9         if (!newModules.isEmpty()) {
10                [...]
11                DBFacade.injectModules(dictionaryService, newModules,
    token);
12                [...]
13        } else {
14            logger.info("No new modules to inject");
15        }
16    }
```

Fragmento de código 6.22: Implementación del filtrado en el método para inyectar módulos de la clase ServiceController

Una vez se han conseguido cumplir los requisitos, se puede añadir en el flujo del *plugin* la inyección de nuevos módulos. Esto se lleva a cabo en la clase responsable de organizar las llamadas a los diversos bloques funcionales del *plugin* (`VensimScanner`).

6.6.2. Historia de usuario 6.2

Historia: "*Como usuario quiero que los símbolos inyectados contengan el módulo al que pertenecen*".

Para llevar a cabo esta historia de usuario, hay que modificar la forma que se tenía de inyectar los símbolos en el diccionario de símbolos utilizando el servicio `qaAddSymbolDefinition`. El *plugin* inicial al no tener forma de detectar módulos enviaba todos los símbolos con un módulo *placeholder*. En la actualidad, como se tiene la información de a qué modulo pertenece cada símbolos, este módulo inventado ya no es necesario.

Antes de implementar hay que saber que cualquier símbolo que se suba al diccionario debe de tener todos sus campos correctamente añadidos. Es decir, no se puede inyectar un símbolo si el mismo o su categoría o módulo han sido marcados como inválidos.

Pasando ahora a la implementación, el hecho de añadir el módulo en los símbolos se hace directamente añadiendo un nuevo campo que contenga el módulo primario del símbolo al el objeto JSON mandado al diccionario. Esta estructura se puede encontrar en el Anexo C.

Ahora solo falta filtrar cuáles son los símbolos que deben ser inyectados. Este filtrado se realiza de una forma similar a la inyección de módulos de la historia anterior. En el Fragmento 6.23 se puede ver la implementación de este filtrado. Cabe destacar que la función requiere una lista con los módulos detectados. Esto se debe a que durante la gran mayoría del desarrollo, la estructura que el diccionario de símbolos utilizaba para el servicio en `qaAddSymbolDefinition` necesitaba un módulo global el cual era al que pertenecían todos los símbolos de la petición. A pesar de que esta estructura de la petición fue modificada, se mantiene este parámetro para así poder filtrar símbolos que no tengan un módulo, como los

índices. Estos son mandados mediante otro servicio llamado `qaAddIndexDefinition` que no contiene información sobre el módulo. Esta inyección de índices puede ser vista en la sección dedicada a la historia de usuario 6.6.

```
1     public void injectNewSymbols(List<Symbol> foundSymbols, List<Module>
      modules, SymbolTable dbSymbolTable) {
2         [..]
3         List<Module> validModules = modules.stream().filter(Module::isValid).
      collect(Collectors.toList());
4
5         List<Symbol> newSymbols = foundSymbols.stream()
6                 .filter(symbol -> !dbSymbolTable.hasSymbol(symbol.getToken().
      trim()) && hasToFetchSymbolFromDB(symbol))
7                 .filter(Symbol::isValid)
8                 .filter(Predicate.not(Symbol::isFiltered))
9                 .filter(symbol -> symbol.getPrimaryModule() != null &&
      validModules.contains(symbol.getPrimaryModule()))
10                .collect(Collectors.toList());
11
12        if (!newSymbols.isEmpty()) {
13            inyectSymbols(validModules, newSymbols);
14        } else {
15            logger.info("No new symbols to inject");
16        }
17
18        inyectNewIndexes(foundSymbols, dbSymbolTable);
19
20    }
```

Fragmento de código 6.23: Implementación del filtrado en el método para inyectar símbolos de la clase ServiceController

6.6.3. Historia de usuario 6.3

Historia: "*Como usuario quiero que las categorías y subcategorías nuevas detectadas sean inyectadas al diccionario de símbolos del proyecto*".

Esta historia de usuario tiene el mismo flujo de trabajo que la historia de usuario 6.1. Tanto la lista de categorías local como la lista de categorías del diccionario de datos se consiguen de la misma forma. La llamada al diccionario de datos en este caso se realiza mediante el servicio `qaGetCategories`.

Existe una diferencia crucial aun así. Esta es la jerarquía de las categorías, aunque todas sean implementadas por la misma clase. Algunas pueden ser subcategorías de otras, por lo que tienen que estar relacionadas. Los atributos de la clase `Category` se pueden ver en el Fragmento 6.24.

```
1 public class CategoryImpl extends IssuableAbs implements Comparable<Category>,
       Category {
2     private Category superCategory;
3     private final String name;
4     private Set<Category> subcategories;
5
```

```
    [...]
    }
```

Fragmento de código 6.24: Atributos de la clase Category

Como se mencionó anteriormente, por limitaciones en el diccionario de datos, los nombres de todas las categorías y subcategorías deben de ser único.

Para la gestión de la jerarquía, la comunicación con el diccionario de datos se realiza mediante una lista de categorías, cada una de ellas con información sobre su nombre, su nivel en la jerarquía (0 significa que no tiene supercategoría) y su supercategoría, si la tuviese. La estructura exacta se puede ver en el Anexo C. Esto implica que es necesario un mecanismo de aplanamiento de las jerarquías que tienen las categorías y convertirlas en una lista plana. Esta transformación de una estructura jerárquica de categorías a una estructura en lista se puede ver en el Fragmento 6.25. El aplanamiento solo está pensado para cuando existe un único nivel de jerarquía, requisito que existe en las categorías de LOCOMOTION.

```

public class ViewTable {
    [...]
    private final CategoryMap categoriesList;

    [...]

    public List<Category> getCategories() {
        return categoriesList.getCategories();
    }

    public List<Category> getSubcategories() {
        return categoriesList.getCategories().stream().flatMap(cat -> cat.
    getSubcategories().stream()).collect(Collectors.toList());
    }

    public List<Category> getCategoriesAndSubcategories() {
        return Stream.concat(getCategories().stream(), getSubcategories().
    stream())
                .collect(Collectors.toList());
    }
    [...]
}
```

Fragmento de código 6.25: Aplanamiento de la jerarquía de las categorías

La gestión de los nombres de categorías duplicados ya se realiza en la historia de usuario 4.8 por lo que ahora en la inyección solo hay que preocuparse de si una categoría o subcategoría es válida.

Una vez que se tiene la lista plana de categorías y subcategorías tanto del análisis local como del diccionario de datos, se puede realizar la llamada al servicio de la API `qaAddCategories`, enviando únicamente las categorías válidas y que no están ya en el diccionario de datos del proyecto.

6.6.4. Historia de usuario 6.4

Historia: "*Como usuario quiero que los símbolos inyectados contengan la categoría y subcategoría a la que pertenecen*".

Para llevar a cabo esta historia de usuario se trata simplemente de añadir el campo `category` en el JSON de cada símbolo que se envía en el servicio `qaAddSymbolDefinition`. En la historia de usuario 6.2 está explicado como se realiza esta adición, sustituyendo la idea de módulo por la de la categoría.

Un punto a tener en cuenta es que el diccionario de datos está esperando como dato la categoría más inferior en la jerarquía. Esto se consigue haciendo que cada símbolo guarde la categorías de más bajo nivel que la representa.

6.6.5. Historia de usuario 6.5

Historia: "*Como usuario quiero que los símbolos inyectados contengan las referencias a tablas excel externas cuando existan en dicho símbolo*".

Esta historia de usuario utiliza la información que algunos símbolos tienen sobre las referencias a tablas Excel externas que existen, por lo que es necesario extraer esta información.

Primero una descripción de qué se va a extraer y de dónde.

En LOCOMOTION se utilizan varias funciones que sirven para llamar a tablas externas, estas son:

- GET_DIRECT_CONSTANTS
- GET_DIRECT_LOOKUPS
- GET_DIRECT_DATA

Hay que tener en cuenta que en Vensim existen aun más funciones que se utilizan para esta finalidad, pero las cuales no son utilizadas en LOCOMOTION y por eso no nos centramos en ellas en esta versión del *plugin*.

Los argumentos de las tres son los mismos, a excepción de `GET_DIRECT_LOOKUPS` y `GET_DIRECT_DATA` que tienen uno mas.

A continuación se explican los parámetros en común teniendo de ejemplo la función `GET_DIRECT_CONSTANTS` que se puede ver en el Fragmento 6.26.

```
1 "'a'_demand_projection_minerals_Rest"[materials]=
2   GET_DIRECT_CONSTANTS('model_parameters/materials/materials.xlsx', 'World', '
      a_demand_proyection_minerals_rest*')
3   ~
4   ~   |
```

Fragmento de código 6.26: Declaración de un símbolo con la función GET_DIRECT_CONSTANTS.

Los datos que nos interesan de la función son:

- Ruta al archivo Excel referenciado: `model_parameters/materials/materials.xlsx`
- Nombre de hoja en el archivo Excel: `World`
- Nombre del rango de celdas que contienen los datos: `a_demand_proyection_minerals_rest*`

Por otra parte las funciones `GET_DIRECT_LOOKUPS` y `GET_DIRECT_DATA` tienen los parámetros que se pueden ver en el Fragmento 6.27.

```
1 afforestation_program_2020_W:INTERPOLATE::=
2   GET_DIRECT_DATA('model_parameters/land_and_water/land_and_water.xlsx', '
      World_land',
3   'time_afforestation_index', 'afforestation_program')
4   ~
5   ~       |
```

Fragmento de código 6.27: Declaración de un símbolo con la función GET_DIRECT_DATA

En estas dos funciones se cuenta con un parámetro más que define el nombre del rango de celdas que contiene la información sobre la serie de coordenadas, en el caso del ejemplo sería `time_afforestation_index`.

Para poder almacenar este tipo de información se crea una nueva clase de datos llamada `ExcelRef`. Cada instancia de esta clase almacena el nombre de la ruta al archivo Excel y también un conjunto de *tripletas* donde se guarda el nombre de la hoja, el rango de celdas donde están los datos, y si existe, el rango de celdas donde se encuentra la serie de coordenadas.

Se permite que en una instancia puede haber varias *tripletas* debido a que en *Vensim* se puede definir múltiples referencias como se puede ver en el Fragmento 6.28.

```
1 CF_ini_RES_elec[hydro]:INTERPOLATE::=
2   GET_DIRECT_DATA('model_parameters/energy/energy.xlsx', 'World', '
      time_RES_nuclear_index'\
3     , 'CF_ini_hydro') ~~|
4 CF_ini_RES_elec[geot_elec]=
5   GET_DIRECT_CONSTANTS('model_parameters/energy/energy.xlsx', 'World', '
      CF_ini_geot_elec'\
6     ) ~~|
7 CF_ini_RES_elec[solid_bioE_elec]=
8   GET_DIRECT_CONSTANTS('model_parameters/energy/energy.xlsx', 'World', '
      CF_ini_bioE_elec'\
9     ) ~~|
```

Fragmento de código 6.28: Multiples llamadas a archivos excel externos

Una vez se tiene la clase que define las instancia que guardan los datos, se modifica el *visitor* `RawSymbolVisitor` para que, cuando en una declaración de símbolo detecte el uso de una de estas tres funciones, extraiga la información y se asocie con los símbolos pertinentes.

Por último, se modifica la llamada a `qaAddSymbolVisitor` para que se pueda añadir esta información en los símbolos en los que se ha detectado.

6.6.6. Historia de usuario 6.6

Historia: "*Como usuario quiero que los índices se inyecten y se recuperen de forma independiente al resto de símbolos*".

Para hacer una inyección y recuperación independiente de los índices, es necesario crear dos nuevas peticiones al diccionario de datos. En este caso se definen `qaGetIndexesDefinition` y `qaAddIndexesDefinition`, en el Anexo C se puede encontrar la estructura de sendos servicios.

Al no querer inyectar elementos duplicados y como cada índice tiene un nombre y una lista de valores posibles, es necesario comprobar si cada índice detectado existe ya en el diccionario y de ser así comprobar si todos los valores del índice detectado están en el diccionario, si alguno de los valores detectados no está en el diccionario de símbolos, es necesario inyectar todo el índice. En el Fragmento 6.29 se muestra cómo se realiza esta lógica de elección de que índices inyectar.

```
for (Symbol index : filteredindexes) {
        if (dbSymbolTable.hasSymbol(index.getToken().trim())) {
            Symbol dbIndex = dbSymbolTable.getSymbol(index.getToken());

            List<Symbol> localDependencies = index.getDependencies().stream().
    sorted().collect(Collectors.toList());
            List<Symbol> dbDependencies = dbIndex.getDependencies().stream().
    sorted().collect(Collectors.toList());
            Boolean toSend = false;
            int i = 0;
            while (i < localDependencies.size() && !toSend) {
                if (!localDependencies.get(i).dbEquals(dbDependencies.get(i)))
     {
                    indexesToSend.add(index);
                    toSend = true;
                }
                i++;
            }

        } else {
            indexesToSend.add(index);
        }
    }
```

Fragmento de código 6.29: Lógica para seleccionar que índices inyectar

6.7. Historia de usuario 7

Historia: "*Como usuario quiero que al ejecutar un escáner se generen documentos auxiliares que contengan información relativa a los elementos que contiene el modelo*".

El *plugin* original realiza la generación de un archivo llamado *symbolTable.json* el cual contiene información (nombre, unidades, dependencias, comentario, tipo y líneas de aparición) de cada símbolo de cada modelo, su estructura se puede ver en el fragmento 4.5.

Para realizar esta historia de usuario es necesario modificar la estructura de salida de este archivo y, además, crear un archivo nuevo que contenga las diferencias encontradas entre el análisis local y el diccionario de datos del proyecto. El nuevo archivo será llamado `dictionaryDiff.json`. Las estructuras de estos nuevos archivos pueden ser encontradas en 5.1 para `symbolTable.json` y en 5.2 para `dictionaryDiff.json`.

Por lo tanto hay tres pasos a realizar:

- Modificar la clase responsable de generar `symbolTable.json`

  `JsonSymbolTableBuilder` es la clase responsable de generar el archivo de salida con los símbolos detectados. Las modificaciones en esta clase se basan en ampliar la generación del JSON de salida, replicando las estructuras originales de esta clase. Iterando primero por la tablas de símbolos de los cuales se extraen todo sus atributos y se transforman en un objeto JSON. Y después por la tabla de las vistas en la cual se extrae también información de los atributos como pueden ser su módulo y categoría.

- Crear la clase responsable de generar `dictionaryDiff.json`

  `JsonDictionaryDiffBuilder` es el nombre de la nueva clase, funciona de forma similar a la clase anterior, con la diferencia de que ésta necesita comprobar qué elementos son diferentes entre el análisis local y el diccionario de datos del proyecto. Se comprueba qué elementos solo existen en el análisis local y qué elementos solo existen en el diccionario de datos. Se considera que dos símbolos son el mismo si tienen el mismo nombre y estos serian iguales si el resto de sus propiedades son iguales. En el Fragmento de Código 6.30 se puede ver el algoritmo que clasifica los símbolos en estos tres grupos.

```
1  JsonObjectBuilder missmatchBuilder = Json.createObjectBuilder();
2  JsonArrayBuilder missingLocalBuilder = Json.createArrayBuilder();
3  JsonArrayBuilder missingDBBuilder = Json.createArrayBuilder();
4
5  for (Symbol symbol : localSymbols) {
6
7      if (dbTable.hasSymbol(symbol.getToken())) {
8          Symbol dbSymbol = dbTable.getSymbol(symbol.getToken());
9
10         if (!symbol.dbEquals(dbSymbol)) {
11             missmatchBuilder.add(symbol.getToken(), symbolDiffToJson(
       symbol, dbSymbol));
12         }
13         dbTable.removeSymbol(symbol.getToken());
14     } else {
15         if(symbol.isValid())
```

```
16                 missingDBBuilder.add(symbol.getToken());
17             }
18         }
19
20         List<Symbol> dbSymbols = dbTable.getSymbols().stream().filter(symbol -> !
               ignoreTypes.contains(symbol.getType())).sorted(Comparator.comparing(
               Symbol::getToken)).collect(Collectors.toList());
21
22         for (Symbol dbsymbol : dbSymbols) {
23             missingLocalBuilder.add(dbsymbol.getToken());
24         }
```

Fragmento de código 6.30: Algoritmo para clasificar los símbolos al generar el archivo de diferencias.

- Crear clase responsable de gestionar la generación de archivos.

  `OutputFilesGenerator` es la nueva clase responsable de esta gestión. En el *plugin* original esta responsabilidad recaía sobre la clase que creaba el único archivo que se generaba, pero en la actualidad al existir dos distintos es recomendable que se encargue otra clase. De esta forma, además, se facilita el poder crear nuevos archivos en el futuro.

  Esta clase se encarga de recibir las tablas necesarias para generar todos los archivos de salida y llama a las clases responsables de esta generación dándoles a cada una de ellas las tablas que necesita para poder crear su archivo.

Con estas tres tareas completadas, solo queda que una vez a finalizado el análisis del *plugin* se pasen las tablas de símbolos y vistas a `OutputFilesGenerator` y mandar la creación de archivos externos.

6.8. Refactorización y calidad del código

A lo largo del desarrollo se han llevado a cabo modificaciones en la calidad tanto del código original como del código nuevo desarrollado. En esta sección se hablará de las tres más interesantes desde el punto de vista del estudiante.

6.8.1. Envío de peticiones al diccionario de datos

`ServiceConnectionHandler` es la clase responsable de enviar peticiones al diccionario de datos. En el *plugin* original solo tenia tres métodos, autenticarse, recibir símbolos del diccionario y enviar símbolos al diccionario. Entre el segundo y tercer método existía un duplicidad en el código. Esta duplicidad se vería muy agravada si se empezasen a añadir los nuevos métodos necesarios para realizar las peticiones al resto de nuevos servicios que se ha necesitado crear.

Por esta razón se decidió abstraer la lógica del envío de peticiones, tanto GET como POST a dos métodos privados de la clase, con incluso un tercer método en común a estos dos que recibe la petición generada y la manda.

Gracias a esta refactorización, al crear un nuevo método destinado a comunicarse con el diccionario de datos se pasó de necesitar más de 40 líneas de código mayoritariamente duplicado, a la llamada de una función que contiene el nombre de servicio de destino y los posibles datos. En el Fragmento de Código 6.31 se puede ver una llamada GET y una llamada POST utilizando los nuevos métodos.

```
sendPOSTRequest(serviceUrl, "qaAddSymbolsDefinition", symbols, token);
sendGETRequest(serviceUrl, "qaGetModules", token);
```

Fragmento de código 6.31: Llamadas utilizadas para realizar una petición al diccionario de datos.

6.8.2. Gestión de creación de categorías

Antes de explicar la mejora de calidad del código y refactorización es necesario explicar el contexto en el que estamos.

Al principio cuando se crearon las categorías, no existía ningún tipo de comprobaciones. Cualquier categorías podía tener cualquier otra supercategoría, por lo que se generarían jerarquías de más de un nivel que no están permitidas en LOCOMOTION y también podrían haber jerarquías cíclicas que no pueden existir. También se podía generar varias veces la misma categoría, creando duplicados de ella. Esto generó problemas a la hora de poder invalidar símbolos de categorías erróneas a la vez de complicar el código con necesidad de realizar comprobaciones en muchas partes del código sin estar centralizadas en una.

Para mejorar estos problemas se realizaron varios cambios.

Primero, se restringió la forma de crear relaciones entre categorías, se cambia para que solo se pueda dictaminar que una categoría es hija de otra. En esa llamada se hará la referencia inversa y se harán comprobaciones para evitar referenciarse a sí mismo o crear jerarquías de más de un nivel de profundidad. En el Fragmento 6.32 se encuentra la implementación de esta función. Con esta función se evitan todos los problemas con la jerarquía.

```
public void addSubcategory(CategoryImpl subcategory) {
    if (this.getSuperCategory() != null) {
        throw new IllegalStateException("Subcategories can't have new
    subcategories");
    }

    if (subcategory.getSubcategories() != null) {
        throw new IllegalStateException("Supercategories can't be added as
    subcategories");
    }
    if (subcategory.equals(this)) {
        throw new IllegalArgumentException("Category can have himself as
    subcategory.");
    }

    if (subcategory.getSuperCategory() != null) {
        throw new IllegalStateException("Category " + subcategory.getName() +
    " already have a supercategory: " + subcategory.getSuperCategory().getName
    ());
```

```
15      }
16
17      subcategory.setSuperCategory(this);
18      if (subcategories == null) {
19          subcategories = new HashSet<>();
20      }
21      this.subcategories.add(subcategory);
22  }
```

Fragmento de código 6.32: Función responsable de añadir a una categoría una subcategoría.

Para evitar los problemas de duplicidad, se genera una factoría de categorías la cual gestiona que categorías y subcategorías se han creado ya dentro de un contexto. Esta factoría está implementada dentro de la clase `CategoryMap`. La única manera de poder añadir nuevas categorías en esta estructura es mediante el uso de una función destinada a crear categorías. Esta función antes de crear una nueva comprueba si ya tiene almacenada una con el mismo nombre. Si es así, devuelve una referencia a la ya creada. Para poder añadir subcategorías en estas nuevas categorías es necesario utilizar otra función de `CategoryMap` la cual utiliza una lógica similar a la primera pero esta vez para la creación de subcategorías.

Para obligar a que haya que usar estas dos funciones a la hora de gestionar las categorías se crea una interfaz (`Category`), la cual no da acceso a la adición de nuevas subcategorías. La implementación de estas dos funciones se puede encontrar en el Fragmento 6.33.

```
1  public Category createOrSelectCategory(String categoryName) {
2      if (map.containsKey(categoryName)) {
3          return map.get(categoryName);
4      } else {
5          CategoryImpl c = new CategoryImpl(categoryName);
6          c.setSubcategories(new HashSet<>());
7          map.put(categoryName, c);
8          return c;
9      }
10 }
11 public Category addSubcategoryTo(String category, String subcategory) {
12     CategoryImpl categoryImpl;
13     if (getCategory(category) == null) {
14         throw new IllegalArgumentException("Category not found");
15     } else {
16         categoryImpl = (CategoryImpl) getCategory(category);
17     }
18     CategoryImpl c = new CategoryImpl(subcategory);
19     categoryImpl.addSubcategory(c);
20     return c;
21 }
```

Fragmento de código 6.33: Funciones responsables de crear una categoría y una subcategoría.

6.8.3. Gestión de invalidar elementos dependientes de uno inválido

La ultima modificación a comentar es la que se realiza a la forma de invalidar los elementos ya sean símbolos, módulos, categorías, etc.

Al inicio, si una regla quería invalidar una categoría por tener mal su nombre, también tenía que invalidar a todos los símbolos pertenecientes a esta categoría, haciendo que existiesen dependencias innecesarias entre clases.

Para arreglar esto se modifica la forma de invalidar cada elemento para que estos tenga la responsabilidad de invalidar a otros elementos que los tengan como dependencia. Para poder hacer esto es necesario primero tener una interfaz en este caso llamada `Issuable` que define un método para invalidar. Todos los elementos que pueden ser invalidados implementan esta interfaz.

Una vez se tienen todas las clases de los elementos que pueden invalidarse implementando esta nueva interfaz, cualquiera puede llamar a los métodos de invalidación del resto sin tener que saber su clase concreta.

En el Fragmento 6.34 se puede ver un ejemplo de lo que sucede si se invalida una categoría que debe invalidar a todos los símbolos que existen en ella.

```
@Override
public void setAsInvalid(String invalidReason) {
    super.setAsInvalid(invalidReason);
    if(subcategories != null) subcategories.forEach(symbol -> symbol.
    setAsInvalid("Supercategory inheritance: " + invalidReason));
}
```

Fragmento de código 6.34: Función responsable de invalidar una categoría.

Esta refactorización se corresponde con la combinación de dos muy conocidas del Catálogo de Refactorizaciones de Fowler [29] llamada *Extract interface* y *Use supertype where possible*.

6.8.4. Limpieza de malas prácticas

Para la calidad del código se usaron herramientas de análisis estático de código. Concretamente se utilizaron dos herramientas, el análisis de código integrado en IntelliJ IDEA y SonarQube para Java.

Con la utilización de estas herramientas se pasó de tener un proyecto con más de 200 *code smells* y más de 10 vulnerabilidades y reducirlos en un 90 % hasta tener menos de 30 *code smells* con cero bugs detectados y dos vulnerabilidades únicas.

Estos últimos no se corrigieron debido a que la reestructuración que habría que hacer para evitar alguno de ellos no compensa teniendo en cuneta su gravedad, por ejemplo, existen bloques de entre 6 a 10 líneas que aparecen duplicados los cuales son marcados como *issues*. Otro motivo son métodos que existen y devuelven resultados que actualmente no se utilizan en el *plugin*, pero que pueden ser de interés en futuras versiones, estos métodos y sus respuestas son también marcados como *issues*.

6.9. Pruebas y cobertura del código

Durante todo el desarrollo del proyecto se fueron creando y actualizando las baterías de pruebas que comprueban que las funcionalidades implementadas funcionan correctamente.

Para realizar estos tests se utilizaron las clases auxiliares desarrolladas por Daniel Bazaco en el TFG de partida, actualizándolas debidamente para poder incorporar las nuevas funcionalidades que se fueron agregando durante el desarrollo del proyecto.

En la Tabla 6.1 se puede ver un resumen por paquetes de los test creados y las coberturas por línea conseguidas. Cabe mencionar que las clases incluidas en el paquete `src/main/java/es/uva/locomotion`, son clases de configuración para el *framework* de SonarQube. Este es el motivo de que no tengan pruebas.

Paquete	**Nº tests directos**	**% cobertura línea**
es.uva.locomotion	0	0
es.uva.locomotion.utilities.logs	0	48.1
es.uva.locomotion.plugin	3	57.7
es.uva.locomotion.utilities	86	67.3
es.uva.locomotion.model.symbol	7	72.5
es.uva.locomotion.parser	31	75.0
es.uva.locomotion.rules	252	78.3
es.uva.locomotion.service	236	79.9
es.uva.locomotion.model	30	85.6
es.uva.locomotion.parser.visitors	245	89.6
es.uva.locomotion.utilities.exceptions	0	93.3
es.uva.locomotion.model.category	22	97.7
Total	912	78.8

Tabla 6.1: Cobertura obtenida en los tests

Capítulo 7

Seguimiento del proyecto

7.1. Introducción

A continuación se describirá el trabajo realizado en cada uno de los *Sprints* ejecutados en este proyecto. Se empezará con el Sprint 0 el cual empezó el 14 de septiembre del 2020. Cada Sprint tendrá una duración de dos semanas.

7.2. Sprint 0 (14/9/2020-27/9/2020)

La Tabla 7.1 tiene un resumen de las tareas realizadas durante este Sprint.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Lectura y estudio del TFG de Daniel	20h	14h 30m	Completado
Configuración del entorno de trabajo	2h 30m	2h 40m	Completado
Instalación del programa Vensim	1h	1h 20m	Completado
Total	**23h 30m**	**28h 30m**	**100 % Completado**

Tabla 7.1: Tareas del Sprint 0

Este primer Sprint se dedico principalmente a la lectura y comprensión del *plugin* de partida, llevando a cabo las tareas de lectura de la documentación y estudio del código desarrollado para familiarizarse con su estructura y ubicación de las clases.

También se preparó el entorno de trabajo para poder desarrollar el *plugin*. Como *Vensim* es un programa en exclusiva de *Windows* se decidió realizar el desarrollo del *plugin* en este sistema operativo. Para ello se instalaron las dependencias requeridas del *plugin* original como sonar-scanner y Java, y también los programas con los que trabaja el *plugin* como puede ser SonarQube. Para poder realizar pruebas contra la API del diccionario de datos se necesita instalar dyson para poder simular las repuestas que generaría la API a diversas llamadas, para poder ejecutar dyson es necesario instalar también node.[35]

Una vez se obtuvo un entorno de trabajo estable se realizaron pruebas con el código para analizarlo y también se cargaron modelos en *Vensim* para poder entender su interfaz y comprender de una mejor forma la estructura de los archivos `.mdl`.

7.3. Sprint 1 (1/10/2020-14/10/2020)

La Tabla 7.2 tiene un resumen de las tareas realizadas durante este Sprint.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Estudio de ANTLR	7h 30m	6h 30m	Completado
Ampliar gramática para implementar las views	10h	18h 40m	Completado
Adaptación de los símbolos para que contengan sus vistas	1h	1h 20m	Completado
Creación del visitor, view y tabla de views	4h	3h 20m	Aplazado
Volcar el módulo de cada símbolo extraido de las vistas	1h	1h 15m	Aplazado
Total	**23h 30m**	**31h 5m**	**60 % Completado**

Tabla 7.2: Tareas del Sprint 1

El primer *Sprint* se empezó aprendiendo a utilizar *ANLTR*. Esta tecnología de creación de gramáticas no se había utilizado nunca por parte del alumno, por ello se necesitó asignar un periodo de tiempo para estudiarla y familiarizarse con la forma de crear gramáticas.

Una vez se tuvo la confianza suficiente en el uso de *ANTLR* se empezó con las modificaciones de la gramática para poder analizar sintácticamente la sección de vistas que incluyen los archivos de tipo `.mdl`, surgieron varios problemas a la hora de definirla, los cuales ralentizaron el desarrollo de la gramática. La explicación de estos problemas puede encontrarse en la sección de implementación de la historia de usuario 2.

Cuando ya se consiguió una gramática estable que generaba el árbol de derivación de las

vista de forma correcta, se pasó a implementar el *visitor* que recorriese esta nueva parte de la gramática, para poder almacenar esta nueva información se crearon nuevas clases llamadas `View` y `ViewTable`. Además, como se requiere extraer el módulo de cada vista se genera una nueva propiedad en el fichero de configuración de sonar-scanner la cual contenga la definición del separador que se va a utilizar para separar el módulo en el nombre de la vista. Esta propiedad será `vensim.view.module.separator`.

Debido al tiempo extra necesario para desarrollar la gramática, la creación del *visitor* y el volcado de esta información en los símbolos no puedo ser acabada y se aplazó para el siguiente *Sprint*.

7.4. Sprint 2 (15/10/2020-28/10/2020)

La Tabla 7.3 tiene un resumen de las tareas realizadas durante este Sprint.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Creación del visitor, view y tabla de views	4h	6h 30m	Completado
Volcar la información de views de cada símbolo	2h 30m	1h 20m	Completado
Generar el filtro de símbolos por modulo	6h	5h 10m	Completado
Crear pruebas para el visitor de las views	4h	2h 40m	Completado
Arreglar CI/CD	4h	1h 20m	Completado
Bug: existen casos donde la gramática falla	2h	2h 15m	Completado
Total	**22h 30m**	**18h 15m**	**100 % Completado**

Tabla 7.3: Tareas del Sprint 2

El *Sprint* empieza retomando las dos tareas pendientes del *Sprint* anterior, como se comprobó que estas tareas eran más complejas de lo esperado, se decide ampliar las estimaciones de tiempo a cada una de ellas.

Una vez se completan las tareas del *Sprint* anterior se puede pasar a las nuevas tareas, la primera será usar esta nueva información sacada de los modelos y utilizarla para poder filtrar los símbolos por el modulo al que pertenecen. La forma de llevar a cabo la implementación se puede encontrar en la sección de implementación de la historia de usuario 3. A la vez que se va implementando el filtro por módulo se va creando también una batería de prueba que verifique que no existen fallos en la implementación.

Para poder tener el *plugin* ejecutándose de manera pública para así poder ser testado

y usado durante el desarrollo como ya ocurría con el proyecto del que parte este TFG, se actualiza el archivo de *gitlab* responsable de gestionar el CI/CD.

A mitad del *Sprint* en la reunión *weekly* se informa por parte de la tutora que existen casos en los cuales la gramática nueva genera excepciones y no realiza un análisis correcto. Un ejemplo de uno de estos casos es la existencia de un número impar de comillas en los comentarios de una vista. A partir de esta reunión se decide dar por cerradas las tareas del *Sprint* actual y generar dos nuevas, se podría ver como el cierre temprano del *Sprint* actual y el inicio de uno nuevo, pero se decide mantenerlo en el mismo para tener consistencia en el tiempo de los *Sprints* a lo largo del proyecto.

Durante la segunda semana del *Sprint* se corrigen los fallos detectados en la gramática y se crean nuevos modelos de prueba que tengan en consideración los casos en los cuales la gramática falló.

7.5. Sprint 3 29/10/2020-11/11/2020)

La Tabla 7.4 tiene un resumen de las tareas realizadas durante este Sprint.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Despliegue versión 1.1	2h	45m	Completado
Recuperación de los acrónimos del diccionario de datos	10h	9h 35m	Completado
Documentación	5h	4h 10m	Completado
Total	**17h**	**14h 30m**	**100 % Completado**

Tabla 7.4: Tareas del Sprint 3

Para empezar este *Sprint* se decide publicar la primera *release* del nuevo *plugin*, en este caso la versión 1.1 que por ahora solo tiene como nueva funcionalidad el filtrado por módulos.

La gestión del despliegue en el servidor oficial de LOCOMOTION donde se encuentra alojado una instancia en ejecución de SonarQube (`gitlab-locomotion.infor.uva.es`) es de la tutora, por lo que por mi parte mis responsabilidades son asegurar que la versión a publicar es estable y sin fallos detectados. Cuando se tiene la confianza de que la versión está lista, se hace un *merge* a la rama *master* del repositorio remoto y se avisa a la tutora de que ya está disponible la versión 1.1 para despliegue.

Una vez acabados los preparativos para el despliegue se pasa a realizar la siguiente tarea del *Sprint*, la recuperación de acrónimos del diccionario de datos.

Para poder realizar esta lectura de acrónimos desde el diccionario de símbolos es necesario generar un nuevo punto de conexión con la API que actualmente no existe. Al final se decide crear un nuevo punto llamado `qaGetAcronyms` para así mantener la consistencia de nombre de los puntos que existen actualmente (`qaGetSymbolDefinition` y `qaAddSymbolDefinition`). En el apéndice C se encuentra la estructura de todos los *end points* de la API al final del desarrollo.

Una vez definidos el nombre y la estructura se puede pasar a implementar la recuperación. Para empezar se crea un *mock* de este *end point* utilizando *dyson*. De este modo se pueden hacer ya pruebas reales pese a que todavía no esté implementado el servicio en la API oficial del diccionario de datos. En la sección de implementación de la historia de usuario 4.1 se pueden encontrar consideraciones que se tuvieron en cuenta a la hora de implementar esta recuperación de acrónimos.

Por último, en este *Sprint* se empieza a escribir esta memoria. Se está utilizando LaTeXpara escribirla. En el momento de este *Sprint* mis conocimiento sobre LaTeXson escasos por lo que será necesario destinar un fragmento del tiempo planificado para documentar en aprender como funciona LaTeX.

7.6. Sprint 4 12/11/2020-25/11/2020)

La Tabla 7.5 tiene un resumen de las tareas realizadas durante este Sprint.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Regla de nombrado de variable consciente de los acrónimos	7h 30m	10h 45m	Completado
Bug: Las *issues* de los números mágicos no son filtradas	3h 15m	3h 35m	Completado
Parametrización de las expresiones regulares de las reglas.	3h	3h 50m	Completado
Documentación.	2h	1h	Completado
Total	**15h 45m**	**19h 10m**	**100 % Completado**

Tabla 7.5: Tareas del Sprint 4

Por incompatibilidades de agenda con la hora a la que se realizaban las reuniones, se decide cambiar la hora de las reuniones y, por ende, la fecha de inicio y fin de los *Sprints*, esta cambiará de ser los lunes a ser los jueves.

Para empezar el *Sprint* se va a actualizar la regla de comprobación de nombre de variables

para que tenga en cuenta a los acrónimos. Tal y como está en la actualidad el paso de las tablas de símbolos tanto la local como la del diccionario de datos a las reglas, implica que extender los datos a pasar no sea sencillo, por ello antes de modificar la regla se decide mejorar ese paso de datos. Para ello se crean las clases: `DataBaseRepresentation` y `AcronymList` y se modifica la clase `VensimVisitorContext` para que las utilice. La explicación más detallada de esta modificación puede ser encontrada en la sección de implementación de la historia de usuario 4.1.

Una vez se tienen los datos de los acrónimos a disposición de la regla de nombrado de variables ya se puede implementar el cambio que también puede encontrarse detallado en la implementación de la historia de usuario 4.1.

Cuando se acabó de implementar el cambio en la regla, se llegó a la conclusión de que la estructura de paquetes actual del *plugin* no iba a soportar la creación de nuevas clases almacenadoras de datos como pueden ser las dos nuevas creadas en este *Sprint*. Por ello se decide crear un nuevo paquete llamado `model` en el cual se almacenarán todas las clases cuya responsabilidad sea el almacenamiento de datos. A este nuevo paquete son incorporadas las clases: `Symbol`, `SymbolTable`, `SymbolType`, `View`, `ViewTable`, `DataBaseRepresentation`, `AcronymList` y `VensimVisitorContext`. Estas clases estaban anteriormente en el paquete de `parser`.

Una vez hecho este cambio se pasa a arreglar el *bug* de los números mágicos. Cuando se realizó el filtro no se tuvo en cuenta que los número mágicos tienen su propio *visitor* que es llamado directamente desde la regla de calidad que los comprueba. Por ello están completamente desacoplados de sus respectivos símbolos. Una vez detectado el problema, se modifica la implementación de la regla para que tenga en cuenta el filtrado. La explicación detallada de la modificación puede ser encontrada en la sección de la implementación de la historia de usuario 3.

Por último en el *Sprint* se quiere modificar la forma de cambiar las convenciones de nombrado que siguen las reglas de calidad, las cuales dentro del *plugin* están implementadas como expresiones regulares. Actualmente esta expresiones regulares están *hard coded* en el código, por lo que se prefiere utilizar la capacidad de parametrización que posee *SonarQube*. Una explicación detallada de cómo se ha llevado a cabo esta implementación puede ser encontrada en la sección implementación de la historia de usuario 5.1.

7.7. Sprint 5 10/12/2020-23/12/2020)

La Tabla 7.6 tiene un resumen de las tareas realizadas durante este Sprint.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Extracción de las categorías de los nombre de las views	5h	5h 50m	Completado
Inyección y reciprocación de módulos y categorías del diccionario de datos	11h	10h 45m	Completado
Inyección en el diccionario de las relaciones de los símbolos con los módulos y las categorías	8h	4h 50m	Completado
Documentación.	2h	1h 40m	Completado
Total	**26h**	**23h 5m**	**100 % Completado**

Tabla 7.6: Tareas del Sprint 5

Este *Sprint* comienza ampliando los datos que se extraen de las vistas, ahora también se extraerá la categoría y subcategoría. Para ello primero es necesario saber cual es el separador que se utilizará entre la categoría y la subcategoría por lo que se crea una nueva propiedad en el fichero de configuración de sonar-scanner llamada `vensim.view.category.separator`.

Con la inclusión de esta nueva propiedad pueden darse configuraciones que no tengan sentido, como por ejemplo que se defina el separador entre categoría y subcategoría, pero no el de módulo y categoría. Por ello se decide crear un nuevo nivel de aviso en el *logger* del *plugin* el cual sea `WARNING` este se utilizará para avisar de problemas que no rompan el *plugin*, pero afecten a su funcionalidad.

Ahora que se puede obtener el separador de categoría/subcategoría se realizan las modificaciones necesarias en la clase `ViewTableVisitor` para que extraiga las categorías. La explicación de la implementación puede ser encontrada en la sección de implementación de la historia de usuario 2.

Para evitar trabajar con cadenas de texto que no tienen un valor semántico útil, se decide crear dos nuevas clases: `Category` y `Module` las cuales contendrán toda la información relativa a las categorías y los módulos respectivamente. Sendas clases serán creadas en el paquete `model`.

Una vez extraídas correctamente las categorías se comienza a trabajar en los nuevos servicios que tendrá que brindar la API del diccionario de datos para que se puedan realizar las inyecciones de módulos y categorías, actualmente solo existe `qaAddSymbolDefinition` que no permite una inyección de módulos y categorías independiente de símbolos. Para la recuperación de estos datos solo existen `qaGetSymbolDefinition` el cual devuelve una gran cantidad de datos combinados en un único Json. Estos datos no tienen una relación pura semántica con los símbolos. Al igual que el servicio de inyección, también está vinculado a los símbolos que se ha pedido al diccionario que devuelva.

Al final, para modularizar más la forma de comunicarse con la API y a su vez mejorar el significado semántico de las peticiones se deciden crear cuatro nuevos puntos de conexión, dos para los módulos y poder realizar inyección y recuperación de datos, y otros dos para las categorías con la misma funcionalidad que los descritos para los módulos.

Para mantener la consistencia de nombrado con las conexiones de la API existentes se deciden los siguientes nombres de los *end points*:

- `qaGetModules`
- `qaAddModules`
- `qaGetCategories`
- `qaAddCategories`

La estructura de las peticiones y de las respuestas puede ser encontrada en el Anexo C.

La implementación para realizar esta inyección y recuperación puede ser vista en la sección de implementación de la historia de usuario 6.1 para los módulos y en la sección implementación de la historia de usuario 6.3 para las categorías.

Una vez se tiene ya la conexión bidireccional de módulos y categorías con el diccionario de datos se continua adaptando los puntos de conexión iniciales `qaGetSymbolDefinition` y `qaAddSymbolDefinition` para que permitan realizar la comunicación de cual es el módulo y la categoría de cada símbolo independientemente del resto.

Una vez más la estructura de estas dos peticiones puede ser encontrada en el Anexo C y la implementación de este cambio en las secciones de las historias de usuario 6.2 y 6.4 para los módulos y las categorías respectivamente.

7.8. Navidades

Oficialmente durante las navidades no se realiza ningún *Sprint*. Así se realizó la planificación del proyecto debido a que estas fechas son muy próximas a los exámenes de convocatoria ordinaria de las asignaturas del primer cuatrimestre y se prefiere destinarlas a estudiar.

No obstante, se dedica algo de tiempo a realizar una refactorización ligera y limpieza del código.

Esta refactorización y limpieza utilizan el apoyo de los analizadores de código. Es decir, programas que hacen lo mismo que la finalidad de este *plugin* pero para el lenguaje Java. Los detalles de estas modificaciones se pueden encontrar en la sección de Refactorización y calidad del código.

Se dedica también tiempo a continuar escribiendo la memoria.

7.9. Sprint 6 21/01/2021-3/2/2021

La Tabla 7.7 tiene un resumen de las tareas realizadas durante este Sprint.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Verificación correcto funcionamiento de los nuevos servicios de la API	30m	25m	Completado
Crear regla para evitar lookups incrustados	10h	19h	Completado
Crear regla para evitar declaraciones de símbolos en el fragmento de control de forma errónea	8h	11h 20m	Completado
Documentación.	2h	0h	Aplazado
Total	**20h 30m**	**30h 45m**	**75 % Completado**

Tabla 7.7: Tareas del Sprint 6

Durante el evento de *Sprint Planning* de este *Sprint* se comunica por parte de la tutora que ya están implementados los nuevos servicios en el diccionario de datos, por lo que es necesario hacer una batería de pruebas para verificar su correcto funcionamiento, tanto desde el *plugin* como también de manera aislada.

Para ello, dentro del *plugin* se crea un nueva batería de pruebas contra estos *end points*. Para hacer las pruebas en aislamiento se utiliza el programa *PostMan* el cual es un programa que sirve para poder realizar peticiones customizadas mediante HTTP.

Al hacer las pruebas se reportan los fallos encontrados a los responsables del diccionario de datos y estos son arreglados durante el desarrollo de este *Sprint*.

Después de realizar las pruebas a los nuevos servicios de la API se pasa a crear la nueva regla de calidad que comprueba que no haya lookups incrustados en el código.

Al empezar a realizar esta nueva regla, se llega a la conclusión de que actualmente no existe ningún mecanismo que permita contar la cantidad de datos incrustados en cada símbolo de tipo *lookup*. Además, se analiza que el poder contar con estos datos supone una gran cantidad diferencias con cualquiera de los *visitors* existentes actualmente. Por estas dos razones se decide crear un nuevo *visitor* con la responsabilidad de contar estos datos incrustados en los *lookups*. El nombre de este nuevo *visitor* es `EmbeddedLookupVisitor`.

También se necesita crear una nueva clase que almacene la lógica de la nueva regla de control que recibe el nombre de `EmbeddedLookupCheck`. Una explicación detallada de la implementación puede ser encontrada en la sección de la historia de usuario 4.2.

Inicialmente esta regla generaba una *issue* en cada dato incrustado, durante la *weekly* se llegó a la conclusión de que esto puede provocar una gran cantidad de *issues* de manera innecesaria, por lo que se actualiza que se solo se genera una *issue* en la línea donde se declara el símbolo de tipo lookup que tiene sus datos incrustados.

Después de realizar este cambio en la regla se da por terminada su implementación.

Se continua el *Sprint* realizando la implementación de la otra regla que se tiene como objetivo.

Esta regla usa el concepto de grupo en el contexto de los archivos de extensión `.mdl`. Actualmente estos grupos no son contemplados en el *plugin* por lo que será necesario modificar la gramática y los *visitors* para que se pueda detectar. La implementación de estos cambios se encuentra en la sección de la historia de usuario 2.

Una vez se tiene una forma de detectar los grupos en los *visitors* es necesario trasladar esa información a los símbolos. Para ello es necesario extender las propiedades de los símbolos para que almacenen el grupo al que pertenecen.

Además, se necesita saber que símbolos pueden aparecer en este grupo de control, estos vienen definidos por Vensim y son los siguiente:

- TIME
- TIME STEP
- INITIAL TIME
- FINAL TIME
- SAVEPER

Estos cinco símbolos son utilizados por Vensim para gestionar los avances de la simulación.

Una vez que los símbolos tienen ya su grupo y se tienen definidos qué símbolos deben pertenecer en exclusiva al grupo de control, el resto es crear una nueva regla, en este caso la clase se llama `SymbolGroupCheck`. La implementación de esta explicación puede ser encontrada en la sección de la historia de usuario 4.3.

Por último, este Sprint también tenía como objetivo la continuación de la memoria, por un mal calculo de tiempo no se ha podido completar satisfactoriamente.

7.10. Sprint 7 04/02/2021-17/02/2021

La Tabla 7.8 tiene un resumen de las tareas realizadas durante este Sprint.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Crear regla para comprobar que el nombre de las vistas sigue el convenio	6h 30m	6h 20m	Completado
Bug: El diccionario no acepta dos subcategorías con el mismo nombre	10h	11h	Completado
Despliegue versión 1.2	1h	40m	Completado
Documentación.	4h	3h 45m	Completado
Total	**21h 30m**	**21h 45m**	**100 % Completado**

Tabla 7.8: Tareas del Sprint 7

Este *Sprint* comienza con la escritura de una nueva regla. En este caso la regla de calidad responsable de que la vistas tengan un nombre correcto.

Esta es la primera regla que es independiente de los símbolos. Esto hace que haya que modificar la forma en la que se comprobaba si una *issue* es guardada o no. La clase abstracta *VensimCheck* inicialmente tenia como atributo un símbolo. Esto es sustituido para que sea un booleano y así pueda funcionar también con vistas.

La clase de la nueva regla se llama `ViewNameCheck` y la explicación de su implementación se encuentra en la sección de la historia de usuario 4.4.

Una vez implementada la regla se pasa a la siguiente tarea del *Sprint* que tiene que ver con un fallo que se ha encontrado a la hora de inyectar categorías en el diccionario de datos del proyecto. El problema está dado por como funciona la estructura interna del diccionario de datos. Es imposible almacenar dos subcategorías con el mismo nombre aunque estas pertenezcan a categorías distintas. Este problema es elevado al equipo responsable de LOCOMOTION. Al final en la *weekly* de este *Sprint* se recibe el veredicto de como afrontar este problema. Se decide que esa restricción se mantendrá en pie debido a los problemas que daría tener que modificarlo en la implementación del diccionario de datos. Por ello es necesario que el *plugin* sea complaciente con este nuevo requisito.

Para llevar esto a cabo se genera una nueva regla de control, llamada `CategoryDuplicatedCheck`, que detecte cuando existen dos subcategorías con el mismo nombre, invalidando siempre una de las dos. Si existiese una de las subcategorías duplicadas ya en el diccionario de datos, se le daría prioridad a la almacenada ya en el diccionario de datos y se generaría la *issue* en la otra duplicada. Si ninguna de las dos existe en el diccionario, la *issue* se generará en la primera en aparecer en el modelo. La implementación de esta regla puede ser encontrada en la sección de la historia de usuario 6.3.

Una vez realizada toda la implementación de este *Sprint* se realiza un *merge* a la rama *master* con el código preparado para el despliegue de la versión del *plugin* 1.2

Por último, en este *Sprint* se recupera el tiempo de documentación perdido en el *Sprint* anterior.

7.11. Sprint 8 18/02/2021-03/03/2021

La Tabla 7.9 tiene un resumen de las tareas realizadas durante este Sprint.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Crear regla para comprobar que las unidades de los símbolos son las estipuladas en el convenio	9h 30m	8h 50m	Completado
Bug: Inyección de símbolos no pertenecientes a ningún módulo	3h	4h 15m	Completado
Actualización del archivo con los elementos encontrados en el análisis	3h	1h	Completado
Generación de un archivo con las diferencias entre los elementos en local y los elementos en el diccionario de datos	4h	3h 20m	Completado
Despliegue versión 1.3	20m	20m	Completado
Documentación.	2h	2h 10m	Completado
Total	**21h 50m**	**19h 55m**	**100 % Completado**

Tabla 7.9: Tareas del Sprint 8

El *Sprint* se inicia con la creación de una nueva regla, `DictionaryUnitSymbolCheck`, la cual se encargará de comprobar que los símbolos del modelo analizado tengan como unidad, una de las oficiales elegidas en el marco común de modelado definido en el proyecto.

Como estas unidades pueden ser modificadas en un futuro no pueden ser escritas en el código. En el diccionario de datos existe la lista con estas unidades válidas, por lo que es necesario generar un nuevo punto de conexión de la API para que pueda dar servicio de obtener las unidades.

Se decide que el nuevo servicio se llamará `qaGetUnitSystem`. No se crea un servicio de inyección por el motivo de que esta lista, o sistema de unidades, es fruto de un análisis y un acuerdo para el desarrollo del modelo y no tendría sentido que se pudiese modificar.

La estructura de este nuevo servicio puede ser encontrada en el Anexo C.

Con esta nueva lista de unidades válidas se puede pasar a crear la nueva regla de control. La explicación detallada de la implementación se puede ver en la sección de la historia de usuario 4.5.

Una vez se tiene la regla implementada se pasa a arreglar un fallo importante, pero que no había sido detectado por la situación temprana en la que se encuentra el modelo. Se trata de que actualmente los símbolos que no pertenecen a ningún módulo no son inyectados, esto se debe a las comprobaciones que se hacen sobre el filtrado de módulos que se realizan, este fallo afecta en especial a los índices, los cuales por la forma en la que se declaran, no pertenecen de forma principal a ningún módulo.

Al trabajar en arreglar el error, se decide que será más conveniente si se divide el servicio de la API `qa[Get-Add]SymbolDefinition` para que por un lado se suban los símbolos estándar que por lo general tienen su declaración en un módulo en concreto y por otro lado los índices que actúan a través de todos los módulos sin pertenecer a ninguno en concreto. Así se consigue que todos los servicios de la API solo utilicen una lista y no una lista de listas como era el caso de `qa[Get-Add]SymbolDefinition`.

Por lo tanto se decide crear dos nuevos servicios de la API: `qaGetIndexesDefinition` y `qaAddIndexesDefinition`.

La estructura de las peticiones y de las respuestas puede ser encontrada en el Anexo C y la implementación en la sección historia de usuario 6.2.

Una vez arreglado el fallo, se puede continuar con el desarrollo del *Sprint*, la siguiente tarea es un tema nuevo respecto a todo lo que se lleva hecho de TFG. Se trata de actualizar el archivo que se genera al realizar un análisis de un modelo. En este archivo aparece información sobre todos los símbolos detectados.

Actualmente la información que presenta es escasa y, si tenemos en cuenta todos los nuevos datos que se obtienen del modelo, faltan muchos datos por volcar, ya no solo de cada símbolo, si no secciones enteras como puede ser la información sobre las vistas o las categorías.

Por lo tanto, se actualiza la clase responsable de generar este documento (`JsonSymbolTableBuilder`)para que incorpore mucha más información. La explicación detallada de la implementación puede ser encontrada en la sección de la historia de usuario 7. Además, a partir de ahora, toda adición que se realice ya sea en los símbolos o en cualquier otro elemento será expuesta en este archivo.

También se requiere crear un nuevo archivo que, en vez de tener toda la información de los elementos detectados en el modelo analizado, contendrá todas las diferencias encontradas entre el diccionario de datos y el análisis local del modelo programado en el .mdl. Contendrá información relativa a diferencias encontradas en el mismo elemento como también elementos que solo estén en el diccionario de datos o solo en el análisis en local. La clase responsable tiene el nombre `JsonDictoinaryDiffBuilder`.

Como este archivo no es importante en todas las situaciones, se crea una nueva propiedad en el documento `sonar-project.properties` llamada `vensim.dictionary.getDiff`. Con

esta propiedad se puede elegir si se genera este archivo con las diferencias o no.

Para poder permitir la existencia de múltiples generadores de documentos de una forma extensible en el código, se crea una nueva clase responsable de dicha tarea, esta será la que llame a cada constructor de ficheros con la información necesaria para que pueda generarse de forma correcta y así el *plugin* solo tendrá que hacer una llamada a esta clase. Esta clase recibe el nombre de `OutputFilesGenerator`.

La implementación de la clase responsable de generar el fichero de diferencias y la clase responsable de llamar a los constructores de archivos puede ser encontrada en la sección de la historia de usuario 7.

Para rematar este *Sprint* se requiere generar un nuevo despliegue. Esto se debe a la necesidad de arreglar el fallo detectado y por otra parte poder poner en uso público los nuevos ficheros generados. Por lo tanto, se prepara el código para que se pueda realizar el despliegue de la versión 1.3

Se continua realizando la memoria.

7.12. Sprint 9 4/03/2021-17/03/2021

La Tabla 7.10 tiene un resumen de las tareas realizadas durante este Sprint.

A partir de este *Sprint* puedo dedicar más tiempo a la realización del TFG. Por ello y además unido con la familiaridad ya cogida con el código y su estructura, se decide ampliar la carga que tienen los *Sprints*.

Para empezar se trabaja en que se pueda elegir el nombre de la carpeta donde se generarán los archivos que se crean al realizar un análisis. Para ello se añade una nueva propiedad en el archivo `vensim.dictionary.getDiff` que se llama `vensim.auxiliaryFiles.directoryName`. Por defecto el nombre que se de a la carpeta será `auxiliary_files`, aunque este se puede cambiar al que quiera el usuario. La implementación de como se ha llevado a cabo se puede encontrar en la sección de la historia de usuario 5.5.

Para poder hacer que sea elegible inyectar o no en el diccionario de datos se va a crear también otra propiedad, en este caso su nombre será: `vensim.dictionary.inject`. Por defecto no se inyectará al diccionario. Esto se decide para evitar inyectar una gran cantidad de símbolos durante toda la primera parte del desarrollo del modelo que puede que vayan cambiando de nombre o dejen de existir. La implementación se encuentra en la sección de la historia de usuario 5.4.

Una vez se tienen implementadas las dos tareas anteriores se pasa a extraer la información relacionada con las tablas externas almacenadas en archivos Excel para posteriormente guardarla en los símbolos pertinentes. La implementación de esta extracción puede ser encontrada en la sección de la historia de usuario 2. Esta historia de usuario fue requerida por el "Centro de Investigación Ecológica y Aplicaciones Forestales (CREAF)"[10] a traves

Tarea	Tiempo estimado	Tiempo invertido	Estado
Parametrización nombre carpeta contenedora de archivos generados en el análisis	40m	1h	Completado
Selección por parte del usuario de inyectar o no en el diccionario de símbolos	1h 10m	50m	Completado
Extracción de información sobre las relaciones con datos externos en archivos excel en los símbolos que proceda	4h	3h 20m	Completado
Inyección de información sobre las relaciones con datos externos en archivos excel en los símbolos que proceda	2h	10m	Aplazado
Refactorización referencias referentes a los símbolos	15h	29h 30m	Completado
Crear regla que compruebe que las copias de los *subscripts* siguen el convenio de nombres	3h 30m	0m	Aplazado
Generar documentación para el archivo sonar-properties	40m	0m	Aplazado
Documentación.	2h	0m	Aplazado
Total	**29h**	**34h 50m**	**50 % Completado**

Tabla 7.10: Tareas del Sprint 9

del servicio de *issue tracker* del Gitlab de LOCOMOTION, se estuvo debatiendo la forma de poder implementar esta nueva funcionalidad y de como guardar la información sobre esta referencias, en esta conversación también estuvo Panos Stratis del equipo "Centre for Renewable Energy Sources and Saving (CRES) [11]"quien es el responsable de la implementación del diccionario de símbolos con el fin de que una vez se dictaminase cual sería la estructura de datos de las referencias a tablas excel, pudiese implementarlo en el diccionario de símbolos.

En este punto del *Sprint* se decide que es momento de afrontar la refactorización que se debe hacer de como se gestionan las relación entre los símbolos y sus modelos, categorías, vistas, etc.

Se decide hacerlo en este punto debido a que cuando se creó la clase `ExcelRef` se detectó una gran falta de calidad semántica debido a que existía un segmento de las relaciones que existían en formato `String` y usos sobrecargados de algunas clases.

Los cambios principales son: la forma de invalidar elementos anidados, la generación de categorías y módulos y la sustitución de identificadores de tipo String a punteros. Algunos detalles de la implementación de la refactorización pueden ser ser vistos en la sección de refactorizaciones en el capítulo de implementación y pruebas.

La refactorización llevó bastante más tiempo del esperado, sobre todo la parte de crear la factoría de las categorías. Es por este motivo que el resto de tareas del Sprint se quedaron sin completar y han tenido que ser aplazadas al siguiente *Sprint*.

En retrospectiva se tuvo que haber hecho un análisis más a consciencia de la magnitud que iba a tener la refactorización que se había propuesto.

7.13. Sprint 10 18/03/2021-31/03/2021

La Tabla 7.11 tiene un resumen de las tareas realizadas durante este Sprint.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Inyección de información sobre las relaciones con datos externos en archivos Excel en los símbolos que proceda	1h 30m	1h 50m	Completado
Crear regla que compruebe que las copias de los *subscripts* siguen el convenio de nombres	3h 30m	4h	Completado
Generar documentación para el archivo sonar-properties	40m	1h 10m	Completado
Parametrizar expresión regulares de `ViewNameCheck`	20m	15m	Completado
Crear rule para detectar diferencias entre el diccionario y el análisis local respecto a los datos de los excels	3h	1h 20m	Completado
Documentación.	4h	3h 20m	Completado
Total	**13h**	**11h 55m**	**100 % Completado**

Tabla 7.11: Tareas del Sprint 10

Una vez acabada la refactorización en el *Sprint* anterior, se empieza este *Sprint* con las tareas que quedaron pendientes empezando por la inyección y recuperación de los datos relativos a los archivos excel contra el diccionario de símbolos.

Para llevar a cabo esta tarea no es necesario crear un nuevo servicio en la API debido a que esta información siempre acompaña a un símbolo en concreto. Por este motivo se actualizan los servicios `qaGetSymbolDefinition` y `qaAddSymbolDefinition` para que contengan una nueva propiedad relacionada con los datos de los Excels.

Se necesita las opción de recuperación para poder realizar más adelante una comprobación

con dicha información. En la sección de la historia de usuario 6.5 se encuentra una explicación sobre como ha sido implementada esta tarea.

La siguiente tarea a llevar a cabo es la creación de una nueva regla que compruebe que las copias de los subscripts siguen su convenio de nombre. Estas copias de subscripts son un nuevo concepto para el *plugin* pero por suerte, la gramática antigua ya tenía en consideración este tipo de sintaxis, por lo que no es necesario modificar nada de la gramática.

Actualmente no existe una forma de almacenar que un *subscript* es una copia de otro, para ello se decide crear un nuevo tipo de dato que extienda de un símbolo estándar. La nueva clase se llama `Subscript` y lo único que tiene de especial es un nuevo atributo para comprobar si es una copia o no.

Una vez se tiene esta nueva clase, se modifica el *visitor* `RawSymbolVisitor` para que en todos los instante en los que se iba a crear un símbolo de tipo *subscript* ahora se cree directamente una instancia de esa clase.

Ahora una vez modificado el *visitor* de crear la nueva clase de la regla llamada `SubscriptCopyNameCheck`, la cual es la responsable de comprobar que el nombre de las copias de los *subscripts* sigue la convención de nombres. La descripción de la implementación puede ser encontrada en la sección de la historia de usuario 4.6.

Para acabar las tareas que quedaban aplazadas del *Sprint* anterior, se genera la documentación para el archivo de configuración `sonar-project.properties`. Mientras se realiza la documentación se detecta que hay diferencias sintácticas entre las propiedades, por lo que se decide modificar el siguiente conjunto de propiedades:

- vensim.dictionaryService
- vensim.dictionaryUsername
- vensim.dictionaryPassword
- vensim.logServerMessages
- vensim.logFile

Se modularizan más los nombres, por lo que al final quedarían de la siguiente forma:

- vensim.dictionary.service
- vensim.dictionary.username
- vensim.dictionary.password
- vensim.log.serverMessages
- vensim.log.file

La documentación del archivo `sonar-project.properties` puede ser encontrada en el Fragmento de código A.2.

Una vez resueltas todas las tareas aplazadas se puede pasar a realizar las nuevas tareas de este *Sprint*.

Primero se realiza un pequeño cambio a la clase `ViewNameCheck` para que ahora tenga la posibilidad de tener distintos convenios de nombres para los módulos y para las categorías. Adicionalmente, estos son parametrizados como el resto de reglas para que puedan ser modificados mediante perfiles desde la interfaz web de SonarQube.

Después se crea una nueva regla de control que se responsabiliza de comprobar las posibles diferencias que existan entre la información recopilada por el análisis local sobre las relaciones que existen entre algunos símbolos y archivos Excel, y las que existen en el diccionario de datos.

Esta regla funciona de forma parecida al resto de reglas ya existentes que comprueban diferencias entre local y el diccionario.

La explicación detallada de la implementación puede ser encontrada en la sección de la historia de usuario 4.7.

Durante la realización de este *Sprint* se continuó escribiendo esta memoria recuperando además el tiempo no empleado en el *Sprint* anterior.

7.14. Sprint 11 08/04/2021- 21/04/2021

La Tabla 7.12 tiene un resumen de las tareas realizadas durante este Sprint.

Este es oficialmente el último *Sprint* del proyecto. Empieza después de las vacaciones de semana santa en las cuales se decidió no invertir horas en el TFG.

Para empezar se implementa la última regla de control que se requiere agregar al *plugin*. Esta es responsable de comprobar la convención de nombres en las variables de tipo *delayed*, esta reglas se caracterizan por usar una de las siguientes funciones en su declaración: `DELAY BATCH`, `DELAY CONVEYOR`, `DELAY FIXED`, `DELAY INFORMATION`, `DELAY MATRIAL`, `DELAY N`, `DELAY PROFILE`, `DELAYP`, `DELAY1`, `DELAY3`, `SMOOTH`, `SMOOTH3` o `SMOOTH N`.

Esto es una información que no se guarda actualmente en el *plugin* por lo que se utiliza, para poder recuperar dicha información, una técnica similar a la utilizada para recuperar la información sobre el uso de ficheros Excel del *Sprint* 9. Es necesario modificar además la clase de los símbolos para que almacene este valor. Como este nuevo valor puede tener varios estados dependiendo de si el *delay* de la variable depende de `TIME STEP` o no, se decide crear una nueva enumeración llamada `DelayedType`.

Tarea	Tiempo estimado	Tiempo invertido	Estado
Crear regla de control que compruebe que las variable de tipo *delayed* cumple con la convención de nombres	6h	7h 50m	Completado
Revisar las descripciones de todas las reglas de calidad	2h	1h 40m	Completado
Limpieza de *code-smells* e incrementar de la cobertura de los tests del código	13h	11h	Completado
Despliegue versión 1.4	20m	15m	Completado
Documentación.	2h	1h 30m	Completado
Total	**23h 20m**	**21h 45m**	**100 % Completado**

Tabla 7.12: Tareas del Sprint 11

Se decide modificar directamente la clase símbolo y no generar una nueva que herede de ella por la posibilidad de que existen múltiples tipos de símbolos que puedan ser de tipo *delayed*. En retrospectiva puede que hubiese sido más recomendable haber generado una nueva clase.

Una vez se tiene la nueva información se puede crear la nueva clase de la regla llamada `DelayedNameCheck`. La implementación de esta regla puede ser encontrada en la sección de la historia de usuario 4.8.

A la vez que se va generando la regla anterior, se van dedicando pequeños fragmentos de tiempo a ir leyendo todas las descripciones de las reglas para ver si hay alguna que se haya quedado desactualizada por los cambios realizados en el *plugin*. La principal diferencia que hay que añadir a la mayoría es la nueva posibilidad de modificar la expresión regular de cada regla mediante los perfiles que dispone SonarQube.

Por último, se dedica la segunda semana del *Sprint* a corregir *code smells* e incrementar la cobertura de los *tests* del código. Para ello se vuelve a utilizar como se hizo en Navidad las herramientas de análisis de código. En esta ocasión además se emplea la herramienta JaCoCo (Java Code Coverage). En la sección Pruebas y su cobertura del código se puede encontrar como se han realizado estas pruebas y los resultados finales obtenidos de cobertura.

Al final de *Sprint* se prepara el código para hacer el último despliegue funcional, la versión 1.4 del *plugin*.

Se continua con la escritura de la memoria a lo largo del *Sprint*.

7.15. Sprints finales 22/04/2021 - 02/06/2021

A partir de aquí nos encontramos ya fuera de *Sprints* empleados para incrementar la funcionalidad del *plugin*. No obstante, se continuará manteniendo y corrigiendo posibles fallos que sean reportados por parte del equipo de LOCOMOTION.

Los *Sprints* seguirán siendo de dos semanas, con la tarea principal de continuar escribiendo la memoria. Se estima un tiempo de dedicación de 20 horas por *Sprints*

En total se realizaron tres *Sprints* más en esta etapa del TFG, .

Durante este periodo con el *plugin* en producción se detectan problemas con la recuperación del módulo y la categoría de una vista. Esto se debe a que pueden existir nombres de vistas erróneos semánticamente, pero que la estructura sintáctica sea compatible con el convenio de nombrado. Esto es un problema por el motivo de que se podría llegar el diccionario de símbolos de módulos y categorías erróneos.

Para mitigar este problema, sobretodo en las fases iniciales de desarrollo del modelo, se habilita la posibilidad de poder elegir qué elementos se inyectan. Para conseguir esto, se generan tres nuevas propiedades en el archivo de configuración `sonar-project.properties`:

- vensim.dictionary.inject.modules
- vensim.dictionary.inject.categories
- vensim.dictionary.inject.symbols

Así se puede prevenir la subida si de antemano se prevé que habrá nombres de vistas erróneos. A parte se pide al equipo de LOCOMOTION que establezcan los nombres de sus vistas lo antes posible para evitar el problema.

Aparte de este fallo, el resto del tiempo se dedicó a continuar con la escritura de la memoria. En total durante estos tres *Sprints* se ha dedicado un total de 63 horas de trabajo contando el arreglar el fallo encontrado y en continuar con la documentación.

7.16. Resumen de la ejecución del proyecto

7.16.1. Resumen final de tareas y tiempos

Al final el tiempo estimado en *Sprints* fue sobrepasado en gran cantidad, de los 10 *Sprints* estimados a desarrollar al principio, han acabado siendo 12. Este incremento se debe a, entre otros, a la mutabilidad que se ha encontrado en algunos de los requisitos y en su creación y a la carga externa que ha tenido el alumno en algunos momentos durante el proyecto que lo han limitado a continuar con el desarrollo en algunos momentos.

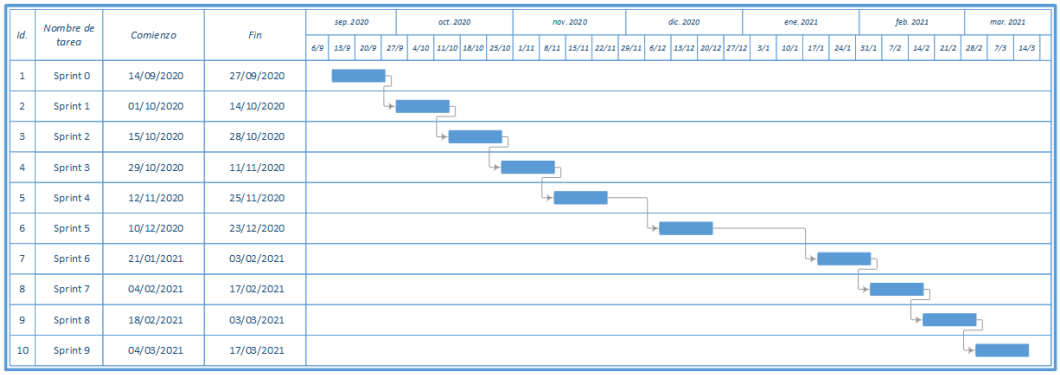

Figura 7.1: Planificación con las reuniones los jueves.

Por lo tanto la planificación inicial de *Sprints* ha cambiado considerablemente respecto al diagrama de Gantt inicial de la Figura 2.1.

El primer cambio surgió al iniciar el curso el día 28 de Septiembre de 2020, por causa de las horas de clase, no pudo ser posible seguir reuniéndose los lunes, todas las reuniones fueron trasladadas a los jueves. Para mantener la consistencia de empezar el *Sprint* a la vez que se hace la reunión de *Sprint Planning*, se decidió modificar el inicio de todos los *Sprints* a los jueves, esto provocó entre el *Sprint* 0 y 1 hubiese un espacio de 3 días sin *Sprint*.

En la Figura 7.1 puede encontrarse la actualización de la planificación teniendo en cuenta el cambio de día de las reuniones.

Por último, el segundo cambio fue al final del *Sprint* 9 del proyecto se requirieron una gran cantidad de modificaciones extras al *plugin* las cuales se estimó que no daría tiempo a realizarlas en el tiempo planificado restante. Por ello, se decidió ampliar la beca un mes extra hasta el día 14 de Abril de 2021 y así poder contar con dos *Sprint* extra para realizar las implementaciones necesarias. En este periodo ya se empezó con la escritura de la memoria. También se tuvieron que añadir los tres *Sprints* finales de documentación y supervisión del plugin en producción para reaccionar ante cualquier incidencia.

En la Figura 7.2 puede encontrarse la actualización de la planificación una vez se amplió el plazo de beca y de documentación.

Una vez se han definido todos los *Sprints* en la Tabla 7.13 se puede encontrar un resumen por *Sprint* del tiempo estimado e invertido.

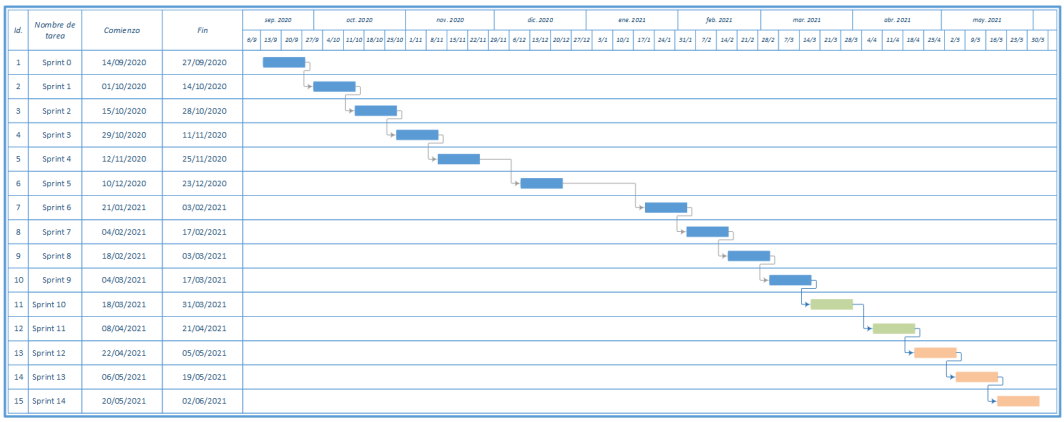

Figura 7.2: Planificación final del TFG.

Sprint	Tiempo estimado	Tiempo invertido
Sprint 0	23h 30m	28h 30m
Sprint 1	23h 30m	31h 5m
Sprint 2	22h 30m	18h 15m
Sprint 3	17h	14h 30m
Sprint 4	15h 45m	19h 10m
Sprint 5	27h	23h 5m
Sprint 6	20h 30m	30h 45m
Sprint 7	21h 30m	21h 45m
Sprint 8	21h 50m	19h 55m
Sprint 9	29h	34h 50m
Sprint 10	13h	11h 55m
Sprint 11	23h 20m	21h 45m
Sprint 12-14	60h	63h
Total	**318h 25m**	**338h 30m**

Tabla 7.13: Resumen tiempo estimado e invertido por Sprint

7.16.2. Análisis de los riesgos

A continuación se hará mención a todos los riesgos que se han detectado a los largo del desarrollo del TFG. La gran mayoría de ellos habían sido recogidos en la lista de posibles riesgos generada al inicio del proyecto.

Los riesgos que hubiesen sido detectados inicialmente estarán acompañados de su número de riesgo que tienen en la sección 2.5. Los riesgos que no hubiesen sido detectados se marcarán debidamente como nuevos.

- **Riesgo 2:** Flexibilidad horaria del equipo de trabajo

  Este riesgo es uno de los poco detectados inicialmente que era beneficioso para el proyecto, se consiguió reconocer la asignatura del segundo cuatrimestre por lo que esto permitió poder trabajar en el TFG a la mejor hora posible y además poder modificar ese horario si fuese necesario.

- **Riesgo 3:** Dificultad entendimiento código de partida

  Este riesgo se materializó a la hora de tener que modificar la gramática de ANTLR. La unión de una gramática compleja y la falta de experiencia del usuario en el uso de ANTLR hizo que las primeras semanas de desarrollo fuesen más lentas de los previsto. Al final se consiguió obtener el conocimiento sobre el código y ANTLR como para poder realizar modificaciones sin utilizar demasiado tiempo.

- **Riesgo 7:** Mutabilidad de las historias de usuario

  Al tratarse de un proyecto ágil, la mayoría de los requisitos que se tienen al final del proyecto han ido siendo decididos durante el desarrollo del mismo. Esto hace que no se pueda prever en gran medida cuáles serán los cambios en el proyecto en el futuro. Aun así, el uso de un marco de trabajo ágil ha facilitado en gran medida esta variabilidad en la modificación de requisitos.

  Ninguno de los requisitos ya implementados en un *Sprint* recibió un cambio drástico en *Sprints* posteriores.

- **Riesgo 8:** Falta de organización en el equipo de trabajo

  Una carencia en la organización en alguno de los *Sprints* (principalmente el *Sprint* 9) agravó la previsión de tiempos. Esto se debe a que en situaciones en las que se hizo una mala estimación de tiempos para las tareas, no se llevó a cabo un cambio en la forma de trabajar en el *Sprint* para intentar mitigar lo máximo posible la estimación errónea.

  Esto llevó a algunos *Sprints* a tener, como el 9, hasta el 50 % de la tareas aplazadas, o como el 1 y el 6, con el 60 % o el 75 %.

- **Riesgo 10:** Incompatibilidad horaria en el equipo de trabajo

  En el *Sprint* 4 fue necesario realizar un cambio en el horario de todas las reuniones por una incompatibilidad horaria.

  Este problema fue resuelto rápido, modificando el día de las reuniones de los lunes a los jueves. Por esto hubo un periodo de 3 días exento de un *Sprint* oficialmente.

- **Riesgo 13:** Falta de conocimientos del *stack* de desarrollo

  Este riesgo está relacionado con lo descrito en el riesgo 3, una carencia de conocimientos mayor a la esperada en ANTLR hizo que el desarrollo y actualización de la gramática llevase más tiempo del estimado.

- **Riesgo no detectado:** Necesidad de una comprensión semántica para la validación de alguna regla.

  Este es un riesgo no detectado al inicio del proyecto. Este riesgo se refiere a la posible situación de que alguna de las nuevas reglas previstas para el *plugin* no pueda ser analizada sintácticamente solo y se necesita un contexto y una comprensión semántica. Este tipo de reglas pueden catalogarse como muy complejas de implementar debido a que el *plugin* solo tiene capacidades sintácticas de análisis.

  Este riesgo se materializó a la hora de invalidar o no los nombres de las vistas. Los nombres de las vistas deben seguir la estructura dada por una expresión regular, el problema es que existen nombre que puedan seguir la estructura de la expresión regular, pero estén mal.

  Por ejemplo, si tenemos la expresión regular:
  `([a-zA-Z0-9_]+)*[a-zA-Z0-9]+(-[a-zA-Z0-9]+(\.[a-zA-Z0-9]+)?)?`
  la cual podría ser una forma válida del nombre de una vista que tiene "-" como separador entre módulo y categoría y "." como separador entre categoría y subcategoría. El nombre `energy-consumption.land` sería un nombre correcto tanto sintácticamente como semánticamente el cual tendría `energy` como módulo, `consumption` como categoría y `land` como subcategoría. Pero `time-saving_aplications_in_development` sería un nombre válido sintácticamente, pero inválido semánticamente ya que esto detectaría como módulo a `time` y como categoría a `saving_aplications_in_development`. En el proyecto no se define un módulo `time`.

  Para combatir este problema se realizaron dos procedimientos, el primero fue implementar un nuevo parámetro el cual permitiese poder deshabilitar la inyección de elementos al diccionario de datos para así poder prevenir la inyección de estos elementos erróneos sobretodo en versiones tempranas del modelo. Segundo, se avisó al equipo que implementa el modelo que intenten tener como prioridad máxima los nombres de las vistas correctos semánticamente.

7.16.3. Gestión de tareas con Gitlab issue tracker

Al haber utilizado las herramientas de gestión de incidencias de GitLab y además con el uso de etiquetas y *milestones* se puede realizar un resumen de cuántas tareas se han ejecutado por tipo de incidencia y cuántas tareas ha habido en cada *release*.

En la Tabla 7.14 se puede ver el resumen de las incidencias agrupadas por tipo, y en la Tabla 7.15 se pueden ver agrupadas por *release*.

Etiqueta	Nº de incidencias
Data dictionary	14
Documentation	4
File generation	3
Grammar	5
Refactor	2
Release	4
Rule added	7
Rule modification	7
Sin etiqueta	4
Total	50

Tabla 7.14: Agrupación de las incidencias por etiqueta

Existe un grupo de incidencias que no tienen etiqueta, esto se debe a que pertenecían a conceptos específicos de los cuales no se ha generado etiqueta en sí. Como por ejemplo incidencias del CI/CD que en este caso solo hay una.

Versión	Nº de incidencias	Fecha de despliegue
v1.1	7	29/10/2020
v1.2	17	16/02/2021
v1.3	6	03/03/2021
v1.4	20	21/04/2021
Total	50	

Tabla 7.15: Agrupación de las incidencias por release

Toda las modificaciones realizadas a partir del despliegue de la versión 1.4 fueron añadidas a esta propia versión.

7.16.4. Coste simulado

Todos los valores incluidos en este coste se toman del presupuesto estimado en la Sección 2.7.1 en el cual se hizo una previsión de costes del proyecto con una supuesta duración de 300 horas.

El proyecto final ha supuesto una duración de 338h 30m. Teniendo en cuenta que el salario del desarrollador es de 11,53 € a la hora implica un coste en salario de 3.902,9 €.

Los costes de hardware y de software no han sufrido variaciones a lo estimado inicialmente.

Por lo tanto el coste final del proyecto puede ser visto en la Tabla 7.16.

Sueldo	3.902,9 €
Hardware	106,25 €
Software	89,40 €
Total	**4.098,55 €**

Tabla 7.16: Coste simulado.

Teniendo en cuenta que en el presupuesto original se hizo una estimación de 3.654,65 € con un margen de contingencias del 12 % el cual ampliaba el presupuesto total a 4.093,21 €. Se puede ver que la relación entre el presupuesto y el coste es prácticamente idéntica. Esto verifica la importancia de establecer un buen margen de contingencias.

7.16.5. Coste real

Para llevar a cabo el coste real se utilizan los valores obtenidos en el presupuesto real el cual se encuentra en la Sección 2.7.2.

Teniendo en cuenta que el coste mensual de la beca para la Funge es de 348,41 € y, además, que la beca fue ampliada un mes para un total de siete meses. Se tiene un coste en salario de $348,41x7 = 2438,87$ €.

La amortización del ordenador se mantiene igual en 106,25 €.

Por lo tanto el coste total del proyecto ha sido de 2545.12 €.

Capítulo 8

Conclusiones

Una vez terminado el proyecto, se puede decir que se han cumplido satisfactoriamente todos los objetivos del proyecto dando como resultado un *plugin* utilizado actualmente en producción, con un *feedback* muy positivo del equipo de LOCOMOTION.

Además, se ha publicado el software obtenido en Gitlab, pasando a ser código libre. La flexibilidad de configuración del *plugin* permite poder adaptarlo a diferentes contextos de otros proyectos de una manera sencilla, además, esta publicación permitirá que el *plugin* pueda ser ampliado y mejorado por la comunidad.

Por otra parte, el uso de nuevas tecnologías como ANTLR o el *framework* de SonarQube requirió de una adaptación inicial por parte del alumno la cual en algunos momentos fue tediosa como puede ser a la hora de desarrollar la gramática y los problemas de incompatibilidades encontrados respecto a la gramática original. Aún así, la gramática resultante funciona en todas las situaciones en las que se ha probado. Respecto al uso del *framework*, se han necesitado muchas horas para su comprensión y entendimiento del funcionamiento del flujo, comparando como se creó la primera nueva regla de control de calidad a la última se nota una gran mejora en la comprensión de este *framework*.

Respecto a la gestión del proyecto, la reuniones con la tutora, ya fuesen *weeklys* o *sprint planning, review o retrospective*, fueron de gran ayuda para la organización y poder abstraerse a un punto de vista de más alto nivel sobre la situación actual del *plugin*. El desarrollo del proyecto ha sido óptimo, se han necesitado aun así más semanas debido a un incremento continuado del alcance del proyecto.

En cuanto a la memoria, esta ha sido desarrollada en paralelo al desarrollo, no obstante se han realizado mejoras sustanciales a posteriori conociendo el alcance total del proyecto y el resultado del mismo. Debido a ser el primer proyecto de esta envergadura de desarrollo por el alumno, las estimaciones de tiempo variaron ligeramente respecto a la planificación inicial.

Sobre la limpieza del código se ha intentando mantener en lo posible un bajo acoplamiento

y utilizar técnicas de diseño que faciliten la comprensión y mantenibilidad del código en el futuro. La gran mayoría de las clases y paquetes están organizados de una manera coherente y no se suele encontrar nada fuera de lugar.

Como experiencia personal, este ha sido sin duda el proyecto más grande al que me he enfrentado, con el cual he podido ver mejor cuáles son mis fortalezas y mis debilidades. Con ello puedo aprender de mis debilidades y que en el futuro debo tener más en cuenta el realizar una estimación de tiempos más realista y no intentar doblar horas cuando no es posible, gestionar mejor mi organización, estabilizar el trabajo diario a un número estándar de horas y priorizar la escritura de la documentación para evitar tener que hacer gran parte de ella al final. Por otra parte, he aprendido que puedo afrontar la gestión de una base de código con una complejidad significativa y que puedo adaptarme a nuevas tecnologías en un plazo de tiempo relativamente pequeño aunque intenso.

8.1. Aplicaciones reales del plugin en LOCOMOTION

Desde la versión 1.1 del *plugin*, este ha sido usado en el desarrollo de LOCOMOTION para mantener el control de calidad del modelo.

Actualmente, en el servidor SonarQube del proyecto se encuentran almacenados los análisis de 21 proyectos de SonarQube distintos.Cada uno de ellos con sus propios parámetros de análisis concretos como puede ser el filtrado por módulo, otros para analizar modelos que son subproyectos o proyectos colaterales, o bien para realizar pruebas.

En ellos se puede ver como se han realizado múltiples análisis y como ha ido reduciendo el número de *issues* detectadas a lo largo de estos análisis. Como ejemplo se pondrá el proyecto `demography` el cual se utiliza para filtrar, analizando solamente los símbolos del módulo `demography`.

Como se puede ver en la Figura 8.1, en el primer análisis el *plugin* detecto casi 5000 *issues* y al final estas han bajado a 411. Cabe decir que el primer análisis se realizó en la versión 1.0 del *plugin* en la cual todavía no existía una forma de filtrar por módulo, de ahí el número tan grande en comparación. El primer análisis realizado ya con el filtrado por módulo fue en noviembre con un total de 528 *issues*. Se puede ver también el uso activo que se está realizando del *plugin* y como la cantidad *issues* va fluctuando en función de las ampliaciones del modelo que se van llevando a cabo. Por últimos, la mitad de las *issues* reportadas avisan de que los símbolos no existen en el diccionario de datos, esto se debe a que el diccionario actualmente tiene los símbolos almacenados, pero no los tiene como validos por los que la API no los muestra, cuando se empiecen a validar, el número de *issues* bajará drásticamente.

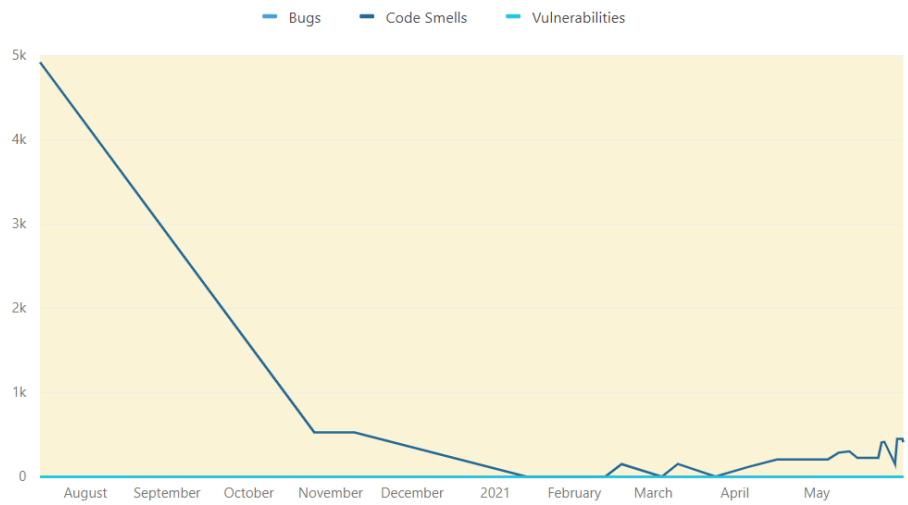

Figura 8.1: Transcurso del uso del plugin en el desarrollo de LOCOMOTION.

También durante los análisis de LOCOMOTION, el *plugin* ha estado comunicándose con el diccionario de datos del proyecto. Desde el inicio, se han inyectado un total de:

- 464 símbolos de todos los tipos.
- 158 categorías y subcategorías.
- 9 módulos válidos.

8.2. Publicación en GitHub

La publicación pública del *plugin* en Github está realizada bajo el nombre `VensimPlugin4SonarQube`, en este traspaso del repositorio central interno de la escuela en Gitlab a este nuevo se ha mantenido todo los *commits* realizados con los responsables pertinentes, de este modo, Daniel Bazaco, el alumno que empezó este *plugin* en su TFG aparecerá como contribuidor de todos sus *commits*.

La URL de este proyecto puede ser encontrada en el anexo B.

8.3. Líneas de trabajo futuras

Como líneas de trabajo futuras primero se expondrán las detectadas por Daniel Bazaco en su TFG las cuales no han sido contempladas en este proyecto. La descripción detallada de estas líneas puede ser encontrada en la Sección 10.1 de su memoria [4].

- Crear una interfaz gráfica
- Inyección de símbolos en condiciones no ideales
- Diferenciar constantes de *switches*
- Inyección de símbolos ya existentes
- Parsear las macros y los alcances
- Soporte para submodelos

A parte de estas líneas futuras, a continuación se sugieren nuevas líneas.

Generar clases para cada tipo de símbolo en vez de utilizar un enumerado: Cuanto más crezca el *plugin*, más técnica y precisa será la información que se almacene de cada tipo de símbolo. Esta información puede no ser compartida entre distintos tipos de símbolos y para ello será necesario disponer de varias clases.

Utilización de los warnings de SonarQube: SonarQube tiene un sistema de *warnings* para avisar de posibles problemas que han existido a la hora de realizar el análisis, actualmente ese sistema no es utilizado y todos los fallos o avisos se envían por el log.

Mejorar las issues: Actualmente las *issues* que gestiona el *plugin* no utilizan todas las funcionalidades que existen en las *issues* de SonarQube. Por ejemplo no se pueden seleccionar rangos de líneas donde generar una *issue* o hacer referencias a otras líneas.

Apéndice A

Manuales

A.1. Manual de despliegue e instalación y Manual de mantenimiento

Los Manuales de despliegue e instalación y el Manual de mantenimiento no han sido modificados en todo el proceso de desarrollo del nuevo *plugin*. Se puede seguir utilizando el elaborado por Daniel Bazaco en el Anexo 1 de su memoria [4].

A.2. Manual de usuario

Como en los manuales anteriores, el manual de usuario se mantiene prácticamente igual al elaborado por Daniel Bazaco en el Anexo 1 de su memoria [4]. No obstante hay que hacer dos aclaraciones nuevas.

En la Sección A3.1, la forma de ejecutar un análisis es la misma, pero se han modificado los parámetro a utilizar. Actualmente se recomienda no utilizar parámetros en la línea de comando (ejemplo en el Fragmento A.1) sino generar el archivo de configuración a parte llamado `sonar-project.properties`.

```
sonar-scanner\
    -Dsonar.projectKey=mi-proyecto\
    -Dsonar.sources=.\
    -Dsonar.host.url=http://localhost:9000\
    -Dsonar.login=a271df371c4996c978a55746103f7bb11693d91d\
    -Dvensim.dictionaryService=http://localhost:9999/
    [...]
```

Fragmento de código A.1: Ejemplo de llamar al análisis con parámetros inline. No recomentado actualmente.

El conjunto de parámetros que hay que exponer en este proyecto han aumentado considerablemente. En el Fragmento A.2 se encuentra una plantilla documentada de este archivo con todos los parámetros de los que dispone actualmente el *plugin*.

```
1
2  ##########################################################################
3  # Sonar scanner default properties                                       #
4  ##########################################################################
5  #-Set the URL of the SonarQube instance.
6  sonar.host.url=https://SonarQube-locomotion.infor.uva.es
7
8  #-Set the credentials for the SonarQube instance.
9  sonar.login=
10
11 #-Set the name of the project that will apear in sonaqube.
12 sonar.projectKey=Project Name
13
14 #-Set the folder where sonar scanner will search for models.
15 sonar.sources=.
16
17 ##########################################################################
18 # Dictionary properties                                                  #
19 ##########################################################################
20 #-Set the URL of the dictionary service.
21 #vensim.dictionary.service= http://www.cres.gr/LOCOMOTION_services/
       webresources/
22
23 #-Set the username of the dictionary service.
24 #vensim.dictionary.username=testingservice
25
26 #-Set the password of the username.
27 #vensim.dictionary.password=testing_DD_2020
28
29 #-If true, outputs a file containing all the differences between local symbols
       and dictionary symbols.
30 #-(Default: false)
31 #vensim.dictionary.getDiff=true
32
33 #-If true, inject all valid and unfiltered symbols, indexes, modules and
       categories to the dictionary.
34 #-(Default: false)
35 #vensim.dictionary.inject=true
36
37 #-If false, prevents injection of new modules (and it's primary symbols).
38 #-(Default: true)
39 #vensim.dictionary.inject.modules=false
40
41 #-If false, prevents injection of new categories (and it's symbols).
42 #-(Default: true)
43 #vensim.dictionary.inject.categories=false
44
45 #-If false, prevents injection of new symbols.
46 #-(Default: true)
47 #vensim.dictionary.inject.symbols=false
48
49 ##########################################################################
50 # Views and filter properties                                            #
51 ##########################################################################
52 #-Set the name of the module to filter with. Unset for unfiltered validation.
```

```
#vensim.view.module.name=materials

#-Set the separator between module and category in a view name.
#-Example: land_and_water.land ("." would be the separator)
#vensim.view.module.separator=.

#-Set the separator between category and subcategory in a view name.
#-Example: land_and_water.land-forest ("-" would be the separator)
#vensim.view.category.separator=-

#########################################################################
# Log and output files properties                                       #
#########################################################################
#-If true, all communications with the dictionary would be logged.
#-(Default: false)
#vensim.log.serverMessages=true

#-Set the name of the file where is saves all the log generated.
#-Is unset, log will be outputted in the terminal.
#vensim.log.file=log.txt

#-Set the folder name where all generated files will be stored.
#-(Default: auxiliary_files)
#vensim.auxiliaryFiles.directoryName=auxiliary_files
```

Fragmento de código A.2: Plantilla del archivo de configuración sonar-project.properties

Adicionalmente es necesario generar una nueva sección en el manual, explicando la forma de poder modificar las variables parametrizadas de las reglas de control de calidad.

A.2.1. Modificación de los parámetros de las reglas de calidad

Actualmente el *plugin* cuenta con un conjunto de parámetros que pueden ser modificados para alterar la forma que tienen las reglas de decidir si generan o no una *issue*. Estos parámetros alteran las expresiones regulares que se usarán, valores por defecto o límites numéricos.

Para poder modificar estos parámetros primero es necesario acceder a la sección de *Quality Profiles* en la página web de SonarQube como se puede ver en la Figura A.1.

Después en la sección de Vensim hay que seleccionar la configuración del perfil por defecto y seleccionar *Extend* como se puede ver en A.2. Se nos requerirá un nombre para el nuevo perfil a generar.

Una vez hecho este paso, si vamos a la sección de reglas y seleccionamos alguna que contenga un parámetro nos dará la opción de editarlo, como se puede ver en las Figuras A.3 y A.4. Actualmente las reglas que tienen parámetros son todas las de convención de nombres y las reglas de control de números mágicos, control de *lookups* embebidos y la regla de control de grupos.

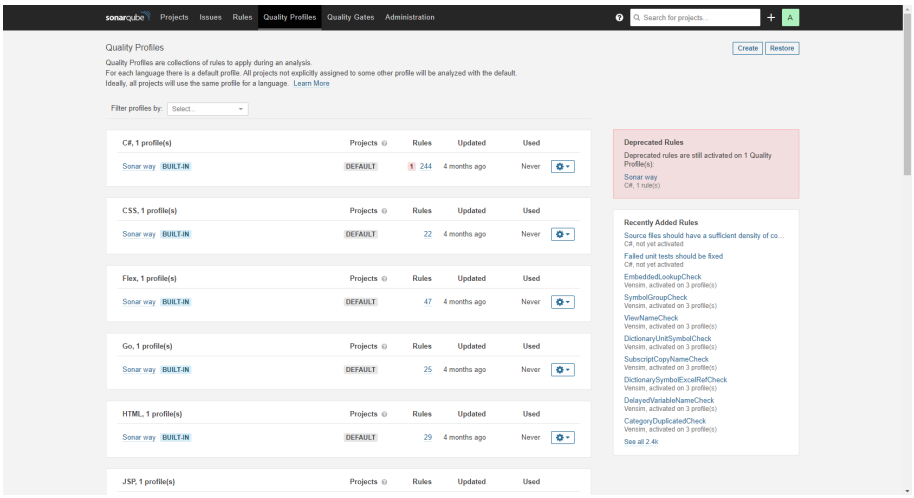

Figura A.1: Selección de la sección de perfiles de calidad en SonarQube.

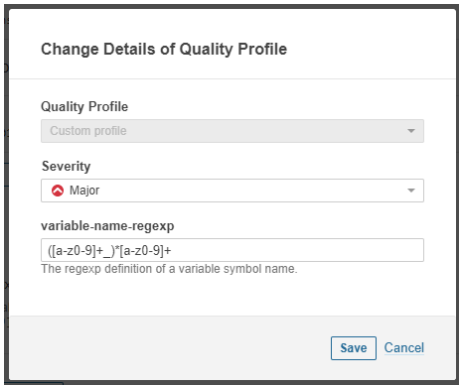

Figura A.4: Modificación del parámetro de una regla.

Por último para poder seleccionar el nuevo perfil creado, hay que ir a la sección de proyectos y escoger en el que se quiera modificar el perfil. Una vez seleccionado se accede a sus ajustes como se puede ver en A.5 y se selecciona la opción de *Quality Profiles*. Para finalizar se escoge el perfil que se desee utilizar y se guarda.

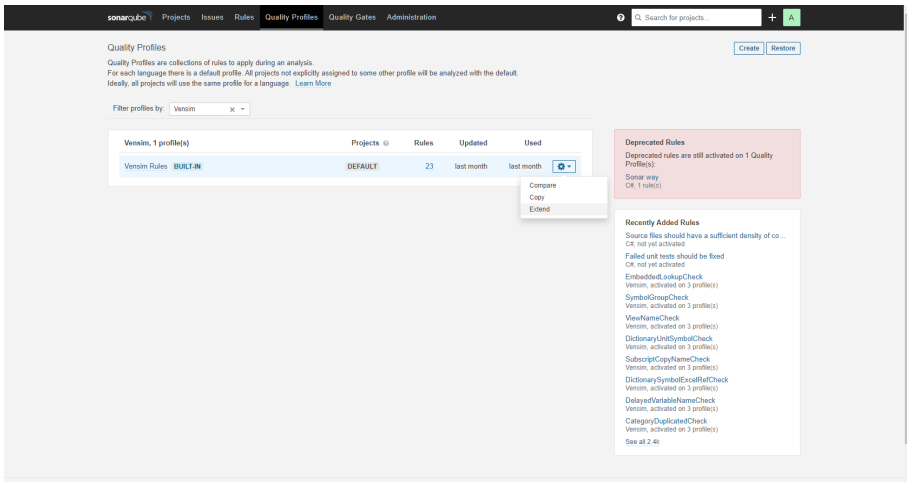

Figura A.2: Extensión de un perfil de calidad de SonarQube.

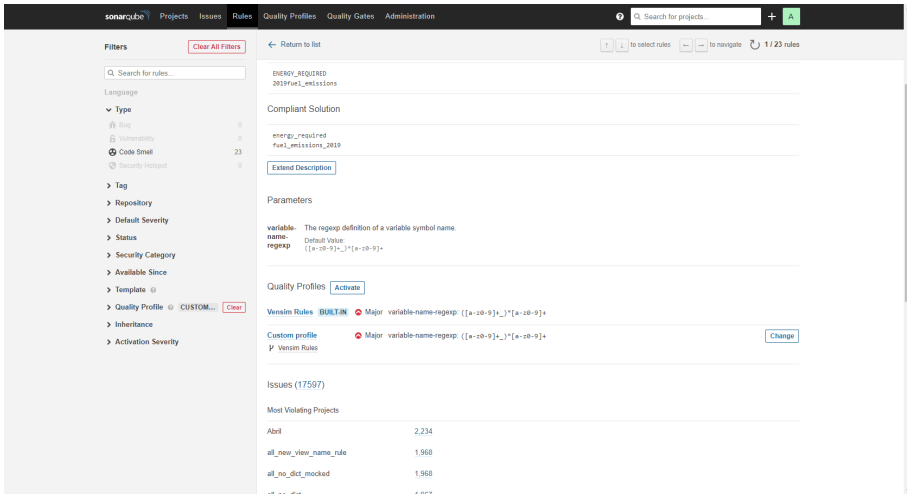

Figura A.3: Selección de regla para poder modificar sus parámetros.

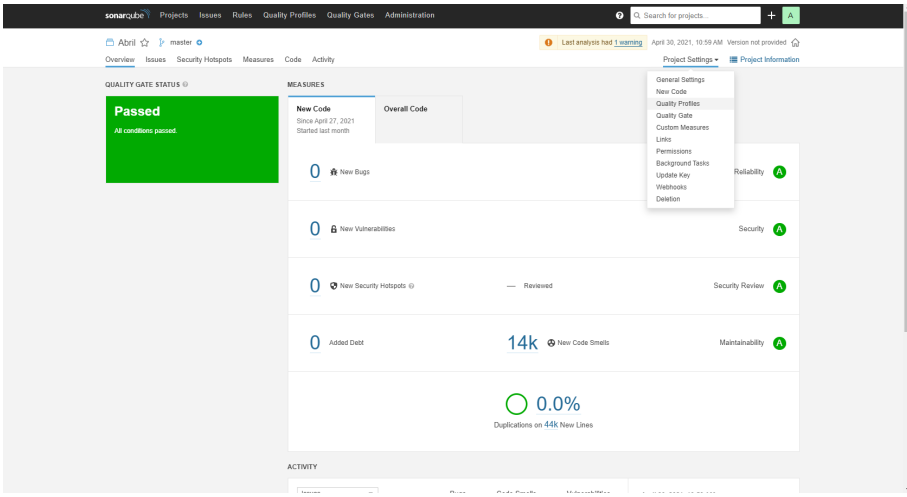

Figura A.5: Modificación del perfil de un proyecto SonarQube.

Apéndice B

Resumen de enlaces adicionales

Los enlaces útiles de interés en este Trabajo Fin de Grado son:

- Repositorio del código: `https://gitlab.inf.uva.es/juaherr/tfg-sonarvensim`.
- Repositorio público del código: `https://github.com/herruzo99/VensimPlugin4SonarQube`.
- Diccionario de datos: `http://www.cres.gr/LOCOMOTION_client/`.
- Plugin desplegado en instancia de SonarQube para dar soporte al desarrollo del IAM del proyecto Locomotion: `https://sonarqube-locomotion.infor.uva.es/`

Apéndice C

Estructura de los endpoints del diccionario de datos.

En este apéndice se pueden encontrar las estructuras en formato JSON que utiliza el diccionario de datos en los *endpoints* que utiliza el plugin en su versión final.

```
{
  "username": "usernameExample1",
  "password": "passwordExample1"
}
```

Fragmento de código C.1: Estructura de la petición al endpoint authenticate de la API

```
2u48urobe55eanjdm2p7mo4mmd
```

Fragmento de código C.2: Estructura de la respuesta del endpoint authenticate de la API

```
{
  "symbols": [
    {
      "name": "SymbolExample1",
      "definition": "definition example for symbol 1 example",
      "unit": "Kg",
      "isIndexed": "true",
      "indexes": [
        "index1",
        "index2"
      ],
      "modules": {
        "main": "IAM Number One",
        "secondary": []
      },
      "category": "CategoryExampleTopLevel",
      "projectTypeOfValue": "Constant",
      "programmingSymbolType": "Constant",
      "excels": [
        {
```

```
          "filename": "filename1",
          "info": [
            {
              "cellrange": "range1",
              "indexes": [
                "index1"
              ],
              "series": "series1"
            }
          ],
          "sheet": "shhet1"
        }
      ]
    }
  ]
}
```

Fragmento de código C.3: Estructura de la respuesta del endpoint qaGetSymbolDefinition de la API

```
{
  "symbols": [
    {
      "name": "test",
      "unit": "units",
      "definition": "CO2",
      "isIndexed": "false",
      "category": "",
      "programmingsymboltype": "Variable",
      "moduleName": "energy",
      "excels": [
        {
          "sheet": "World",
          "filename": "/model_parameters/energy/energy.xlsx",
          "info": [
            {
              "indexes": [
                "hydro"
              ],
              "cellrange": "CF_ini_hydro",
              "series": "time_RES_nuclear_index"
            },
            {
              "indexes": [
                "geot_elec"
              ],
              "cellrange": "CF_ini_geot_elec"
            }
          ]
        }
      ]
    }
  ]
}
```

Fragmento de código C.4: Estructura de la petición al endpoint qaAddSymbolsDefinition de la API

```
[
  {
    "definition": "Regions of the model, only 1 global region in this early
    version.",
    "indexName": "REGIONS_I",
    "values": [
      "EASOC",
      "EU27",
      "UK",
      "LATAM",
      "LROW",
    ]
  }
]
```

Fragmento de código C.5: Estructura de la respuesta del endpoint qaGetIndexesDefinition de la API

```
{
  "indexes": [
    {
      "indexName": "SUBSCRIPT_NOT_IN_DB_I",
      "values": [
        "VALUE_1",
        "VALUE_2"
      ]
    }
  ]
}
```

Fragmento de código C.6: Estructura de la petición al endpoint qaAddIndexesDefinition de la API

```
[
  {
    "name": "Category",
    "level": 0,
    "super_category": "null"
  },
  {
    "name": "Subcategory",
    "level": 1,
    "super_category": "Category"
  }
]
```

Fragmento de código C.7: Estructura de la petición y respuesta del endpoint qaGetCategories y qaAddCategories de la API

```
{
  "modules": [
    "module_1",
    "module_2"
  ]
}
```

Fragmento de código C.8: Estructura de la petición y respuesta del endpoint qaGetModules y qaAddModules de la API

```
[
  {
    "id": 1,
    "name": "RES",
    "definition": "Renewable Energy Source"
  },
  {
    "id": 2,
    "name": "2GEN",
    "definition": "Second generation biofuels"
  }
]
```

Fragmento de código C.9: Estructura de la petición al endpoint qaGetAcronyms de la API

```
[
  {
    "id": 1,
    "name": "RES",
    "definition": "Renewable Energy Source"
  },
  {
    "id": 2,
    "name": "2GEN",
    "definition": "Second generation biofuels"
  }
]
```

Fragmento de código C.10: Estructura de la petición al endpoint qaGetUnitSystem de la API

Bibliografía

[1] 8x8, Inc. Página principal de jitsi meet. https://meet.jit.si/. Accessed: 2021-2-4.

[2] Agencia Tributaria. Tabla de coeficientes de amortización lineal. https://www.agenciatributaria.es/AEAT.internet/Inicio/_Segmentos_/Empresas_y_profesionales/Empresas/Impuesto_sobre_Sociedades/Periodos_impositivos_a_partir_de_1_1_2015/Base_imponible/Amortizacion/Tabla_de_coeficientes_de_amortizacion_lineal_.shtml. Accessed: 2021-4-10.

[3] ANTLR. Página principal de antlr. https://www.antlr.org/. Accessed: 2021-2-4.

[4] Bazaco Velasco, Daniel. Definición y comprobación de estándares de calidad en la programación de iams en vensim. http://uvadoc.uva.es/handle/10324/44115?show=full. Accessed: 2021-1-19.

[5] Bill Hare Robert Brecha Michiel Schaeffer . Integrated assessment models: what are they and how do they arrive at their conclusions? https://climateanalytics.org/media/climate_analytics_iams_briefing_oct2018.pdf. Accessed: 2021-2-4.

[6] ChangeVision Inc. Página principal de astah. https://astah.net/. Accessed: 2021-6-5.

[7] Cisco. Página principal de webex. https://www.webex.com/es/video-conferencing.html. Accessed: 2021-2-4.

[8] Cleventy. ¿qué es git flow y cómo funciona? https://cleventy.com/que-es-git-flow-y-como-funciona/. Accessed: 2021-2-4.

[9] Climate interactive. C-roads website. https://www.climateinteractive.org/tools/c-roads/. Accessed: 2021-2-4.

[10] CREAF. Centro de Investigación Ecológica y Aplicaciones Forestales Website. http://www.creaf.cat/es. Accessed: 2021-6-5.

[11] CRES. Centre for Renewable Energy Sources and Saving. http://www.cres.gr/kape/index_eng.htm. Accessed: 2021-6-5.

[12] EU Research and Innovation. Horizon 2020 website. https://ec.europa.eu/programmes/horizon2020/en. Accessed: 2021-1-19.

[13] GEEDS. Geeds website. https://geeds.es/. Accessed: 2021-1-19.

[14] git-scm. Página principal de git. `https://git-scm.com/`. Accessed: 2021-2-4.

[15] GitHub, Inc. Página principal de github. `https://github.com/`. Accessed: 2021-2-4.

[16] GitLab. Página principal de gitlab. `https://about.gitlab.com/`. Accessed: 2021-2-4.

[17] Gómez Pedriza, David. Una arquitectura de microservicios python como soporte a un diccionario de datos en un sistema de aseguramiento de la calidad y asistencia al desarrollo de iams por grandes equipos globalmente distribuidos. ://uva-doc.uva.es/handle/10324/44278.

[18] IEA-ETSAP. Times website. `https://iea-etsap.org/index.php/etsap-tools/model-generators/times`. Accessed: 2021-2-4.

[19] Javier Santos Pascualena. ¿Cuánto cuesta contratar un trabajador? `https://www.infoautonomos.com/blog/cuanto-cuesta-contratar-un-trabajador/`. Accessed: 2021-4-10.

[20] JetBrains s.r.o. Free Educational Licenses. `https://www.jetbrains.com/community/education/#students`. Accessed: 2021-4-10.

[21] JetBrains s.r.o. IntelliJ IDEA Subscription options & Pricing. `https://www.jetbrains.com/idea/buy/#personal?billing=monthly`. Accessed: 2021-4-10.

[22] JetBrains s.r.o. Página principal de intellij idea. `https://www.jetbrains.com/es-es/idea/`. Accessed: 2021-2-4.

[23] Jordi Cabot. La diferencia entre transpilación y compilación. `https://ingenieriadesoftware.es/diferencia-transpilacion-compilacion/`. Accessed: 2021-6-3.

[24] Josep Lluís Monte Galiano. Implantar scrum con éxito, 2017.

[25] Lars Kappert. Página principal de dyson. `https://www.npmjs.com/package/dyson`. Accessed: 2021-2-4.

[26] LOCOMOTION. Diccionario de símbolos. `http://www.cres.gr/LOCOMOTION_client/`. Accessed: 2021-6-3.

[27] LOCOMOTION. Locomotion website. `https://www.locomotion-h2020.eu/`. Accessed: 2021-1-19.

[28] LOCOMOTION. Report of the common modeling framework. `https://www.locomotion-h2020.eu/download/d9-1-report-of-the-common-modeling-framework/`. Accessed: 2021-6-3.

[29] Martin Fowler, with Kent Beck. Refactoring, Improving the Design of Existing Code. `https://martinfowler.com/books/refactoring.html`, 2018. Accessed: 2021-6-3.

[30] Martínez López, Pablo. Desarrollo de un plugin para semanticmerge para facilitar la integración de versiones de archivos vensim, 2021. Accessed: 2021-6-16.

[31] MEDEAS. Medeas website. `https://www.medeas.eu/#home`. Accessed: 2021-1-19.

[32] Ministerio de Empleo y Seguridad Social, Gobierno de España. XVII Convenio colectivo estatal de empresas de consultoría y estudios de mercado y de la opinión pública. https://www.boe.es/boe/dias/2018/03/06/pdfs/BOE-A-2018-3156.pdf. Accessed: 2021-4-10.

[33] Mockito. Página principal de mockito. https://site.mockito.org/. Accessed: 2021-2-4.

[34] Mountainminds GmbH & Co. KG. Página principal de jacoco. https://www.jacoco.org/jacoco/trunk/index.html. Accessed: 2021-2-4.

[35] OpenJS Foundation. Node.js. https://nodejs.org/en/. Accessed: 2020-11-6.

[36] Overleaf. Página principal de overleaf. https://www.overleaf.com/. Accessed: 2021-6-5.

[37] Deemer Pete, Benefiel Gabrielle, Larman Craig, and Vodde Bas. The scrum primer v2.0, 2012.

[38] PMI. Pmbok® guide – sixth edition (2017).

[39] Postman, Inc. Página principal de postman. https://www.postman.com/. Accessed: 2021-2-4.

[40] Pralit PatelKate Calvin. Gcam website. http://www.globalchange.umd.edu/gcam/. Accessed: 2021-2-4.

[41] Red Hat. ¿Qué son la integración/distribución continuas (CI/CD)? https://www.redhat.com/es/topics/devops/what-is-ci-cd. Accessed: 2021-6-5.

[42] Rocket.chat. Página principal de rocket.chat. https://rocket.chat/es. Accessed: 2021-2-4.

[43] Ken Schwaber and Jeff Sutherland. Scrum Guide. https://scrumguides.org/scrum-guide.html. Accessed: 2021-6-3.

[44] SimRess. Simress website. https://simress.de/en. Accessed: 2021-2-4.

[45] SonarQube. Sonarqube website. https://www.SonarQube.org/. Accessed: 2021-2-4.

[46] SonarQube. Sonarscanner. https://docs.SonarQube.org/latest/analysis/scan/sonarscanner/. Accessed: 2021-2-4.

[47] Stefan Mandel. Gestor de canales dinámicos en antlr4. https://github.com/almondtools/antlr4multichannel. Accessed: 2021-3-14.

[48] Stockholm Environment Institute. Leap website. https://leap.sei.org/default.asp. Accessed: 2021-2-4.

[49] Stratis, Panos. Documentación de la API del diccionario de símbolos. http://gitlab-locomotion.infor.uva.es/locomotion/data-dictionary/uploads/610e2a98ff2f71172eb198ef30c1236d/LOCOMOTION_Web_Services.docx. Accessed: 2021-6-3.

[50] The Apache Software Foundation. Welcome to apache maven. `https://maven.apache.org/`. Accessed: 2021-2-4.

[51] The JUnit Team. Página principal de junit 5. `https://junit.org/junit5/`. Accessed: 2021-2-4.

[52] Toggl. Página principal de toggl. `https://toggl.com/`. Accessed: 2021-2-4.

[53] Universidad de Valladolid. Preguntas frecuentes másteres oficiales uva. `https://www.uva.es/export/sites/uva/2.docencia/2.02.mastersoficiales/2.02.13.preguntasfrecuentes/index.html`. Accessed: 2021-2-5.

[54] Universidad de Valladolid. Proyecto docente del trabajo de fin de grado 2020-2021 (mención ingeniería de software). `https://albergueweb1.uva.es/guia_docente/uploads/2020/545/46976/1/Documento.pdf`. Accessed: 2021-2-5.

[55] Venstana systems inc. Vensim website. `https://vensim.com/`. Accessed: 2021-2-4.

[56] Ventana Systems, Inc. Defined and Shadow Variables. `https://www.vensim.com/documentation/22890.html`. Accessed: 2021-4-8.

[57] Ventana Systems, Inc. Vensim Control Parameters. `https://www.vensim.com/documentation/ref_sim_control_params.html`. Accessed: 2021-4-8.

[58] Ventana Systems, Inc. Vensim groups. `https://www.vensim.com/documentation/groups.html`. Accessed: 2021-4-8.

[59] Ventana Systems, Inc. Vensim sketch Information. `https://www.vensim.com/documentation/ref_sketch_format.html`. Accessed: 2021-4-8.

[60] Ventana Systems, Inc. Vensim sketch Objects. `https://www.vensim.com/documentation/24305.html`. Accessed: 2021-4-8.

[61] Ventana Systems, Inc. Vensim variable types. `https://www.vensim.com/documentation/ref_variable_types.html`. Accessed: 2021-4-8.

[62] Visual Paradigm. Página principal de visual paradigm. `https://www.visual-paradigm.com/`. Accessed: 2021-6-5.