Universitat Oberta de Catalunya

uoc.edu

Herramienta online para la creación y evaluación de especificaciones basadas en UML y OCL

DAVID EDUARDO DELGADO CAMACHO
Grado de Ingeniería Informática - Mención en Computación
Desarrollo web

Nombre Consultor/a: Vincenç Font Sagristà
Nombre Profesor/a responsable de la asignatura:
David García Solórzano | Santi Caballé LLobet

24 de Junio de 2023

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Herramienta online para la creación y evaluación de especificaciones basada en UML y OCL*
Nombre del autor:	*David Eduardo Delgado Camacho*
Nombre del consultor/a:	*Vicenç Font Sagristà*
Nombre del PRA:	*David García Solórzano* *Santi Caballé LLobet*
Fecha de entrega (mm/aaaa):	06/2023
Titulación:	*Grado de Ingeniería Informática*
Área del Trabajo Final:	*Desarrollo Web*
Idioma del trabajo:	*Castellano*
Palabras clave	*UML, OCL, USE*

Resumen del Trabajo:

Este proyecto consiste en una herramienta online para crear, evaluar y compartir especificaciones en UML. Se trata de un entorno colaborativo de especificación textual que por lo tanto permite: Modelar, obtener diagramas de clase, instanciar objetos, obtener diagramas de objetos, especificar la definición de atributos derivados en OCL, obtener diagramas de objetos con atributos derivados ya evaluados y finalmente poder especificar restricciones invariantes.

Aborda principalmente la utilización de varios DSLs (*domain specific languages*) ad-hoc diseñados cada uno con el propósito de cumplir una tarea de especificación concreta, para así finalmente integrarlos dentro de un editor de código web y sus conceptos análogos, como bien puede ser el *linting*, *syntax highlighting*, diagnóstico, auto-completado, etc.

Finalmente, el propósito de este proyecto es el de crear un prototipo útil y un punto de vista documentado sobre cómo debería abordarse un reto similar, tan complejo y tan amplio. Como por ejemplo: el trabajo con parsers usando editores de código, la presentación gráfica de los diagramas de clases de alta calidad, la integración de múltiples lenguajes mediante un modelo semántico, cómo abordar la evaluación de especificaciones en una web frente a la indisponibilidad de herramientas web similares y la definición de una arquitectura para este caso de uso concreto, comparado con otras opciones y el porqué.

Abstract:

This project is an online tool for conceptual modeling and specification in UML. It is a collaborative and textual specification environment that allows: Modeling, obtaining class diagrams, instantiating objects, obtaining object diagrams, specifying the definition of derived attributes in OCL, obtaining object diagrams with derived attributes already evaluated, and finally to also be able to specify invariant constraints.

This is a project that mainly deals with the use of several ad-hoc DSLs (domain specific languages) each one designed with the purpose of fulfilling a specific task, to be able to integrate them into a web code editor and its analogous concepts, such as linting, syntax highlighting, diagnostics, auto-completion, etc.

Finally, the purpose of this project is to create an extensive prototype and a documented point of view on how to approach such a complex and broad challenge. For example: how to deal with parsing for code editors, how to be able to release high-quality class diagrams, the integration of multiple languages through a semantic model, how to approach the evaluation of specifications in a web application while facing the unavailability of some similar web tools and the definition of an architecture and technologies for this specific use case compared to other options and why.

Índice

1. Introducción

1.1 Contexto y justificación del Trabajo

La especificación basada en modelos cubre una necesidad básica de la ingeniería del software. Destacan fundamentalmente cuando se discuten sistemas complejos, porque permite abstraer el diseño de los detalles, tal y como lo haría su homólogo en la ingeniería de lenguajes: el AST (*abstract syntax tree*). Son la base del MDE (*model driven engineering*) y del MBE (*model based engineering*). Por ello, en el proceso de formación de profesionales para su expresión mediante especificaciones y el trabajo en la industria con especificaciones formales, se requieren entornos específicos que sirvan para evaluar resultados concretos, como puede ser el solapamiento de restricciones, máquinas de estado, evaluación de definiciones, etc.

Dentro de este ámbito de las especificaciones y concretamente dentro del ámbito de la especificación mediante UML[1], existen herramientas que cumplen dicho propósito (con un enfoque concreto o más amplio), como bien puede ser USE-OCL[2], Papyrus. etc.

Todas estas soluciones presentes que comparte la comunidad investigadora de la ingeniería del software se constituyen como aplicaciones de escritorio y han cumplido de forma completa y precisa su propósito.

A pesar de ello, también resulta fundamental el tener la capacidad de no solo definir sino también de poder discutir sobre una idea concreta de una forma menos ambigua. Entonces, es lógico considerar que exista una forma de poder expresar estas ideas complejas con el resto del mundo. Incluso en un entorno educativo cuando se discuten ideas de diseño, análisis, etc. Y que tal mundo no sería más amplio si no es web. Pero no de cualquier manera: Debe resulta posible jugar con estas ideas complejas para demostrar su validez, contribuir en la solución, tener ideas derivadas... todo más allá de la mera comprensión de la solución.

Así nace este proyecto y propone que existe la necesidad de disponer de la capacidad de expresar especificaciones en la web, un *code playground* (como codepen[3] o jsfiddle[4]). En detalle:

- Poder utilizar OCL[5], conocerlo y usarlo como herramienta online, sin tener que instalar ningún software.

- Ser capaz de mostrar una especificación en cualquier lugar a través de un hipervínculo.

- Permitir a otras personas evaluar una especificación compartida.

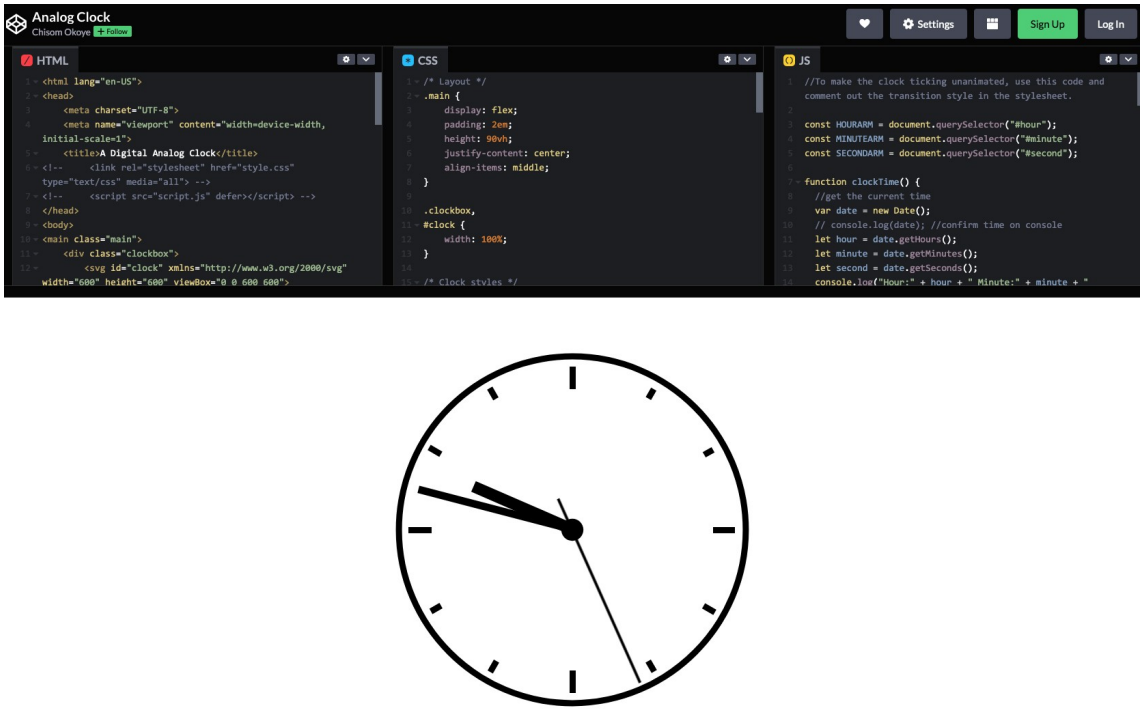


Figura 1: Ejemplo de reloj analógico en JS, HTML y CSS [6]

1.2 Objetivos del Trabajo

Este proyecto tiene como objetivo demostrar que es posible desarrollar una herramienta tan compleja y extensa como una herramienta de especificación de UML dentro de un tiempo limitado y muy corto. Por eso resulta especialmente descriptivo en relación a las soluciones y la justificación de las mismas. Y, así mismo, resulta su enfoque, como un prototipo que se desarrolla de forma extensiva, con el propósito de abarcar la funcionalidad completa y servir de punto de partida.

Principalmente se pretende abordar la especificación de modelos en diagramas de clases en una aplicación web, así como también los siguientes objetivos generales:

- Crear una especificación online.
- Compartirla con el resto del mundo.
- Visualizar y jugar con una especificación para comprender una idea.

Más específicamente, el objetivo del proyecto quedará definido por las capacidades de la aplicación para poder:

- Codificar modelos mediante diagramas de clases.
- Validar léxica, sintáctica y semánticamente la definición de los modelos.
- Visualizar estos diagramas gráficamente.
- Instanciar objetos de los modelos creados.
- Validar la instanciación en relación al modelo creado.
- Visualizar la instanciación de objetos a través de un diagrama de objetos.
- Crear restricciones invariantes en OCL.
- Crear atributos derivados del modelo en OCL.
- Validar la definición de los atributos derivados y los invariantes en función del modelo definido.
- Visualizar la evaluación de los atributos derivados definidos en un diagrama de objetos.
- Guardar estas especificaciones en la nube así como también poder compartirlas con otros usuarios de la aplicación.
- Exportar especificaciones completas a USE-OCL.

1.3 Enfoque y método seguido

Para la preparación de este proyecto ha existido un estudio previo de varias referencias bibliográficas señaladas en su apartado correspondiente. Específicamente, las referencias relacionadas con DSL han servido como base de las metodologías aplicadas para definición de las soluciones de los lenguajes, así como la comprensión de cómo abordar el trabajo con los resultados intermedios en distintas fases del proyecto.

Por otra parte, a nivel de producto, no habiendo la posibilidad de adaptar un producto existente web, se ha optado por la creación de un producto nuevo. Este producto nuevo, sin embargo, re-aprovecha productos de uso específicos ya existentes, como lo son los parsers específicos para el caso de uso, editores web, librerías *frontend*, para la programación y el aspecto web, así como también una herramienta de evaluación de especificaciones. Por ello, ha habido una fase de pruebas y evaluación de las capacidades de estos productos, lo cual se ha detallado en el contenido de esta memoria. Con el fin de que, independientemente de cómo se juzgue el resultado obtenido en este proyecto, esta otra parte de evaluación pueda servir a otras personas que tengan un interés de trabajar con editores web y lenguajes que no sean de propósito general.

La metodología de gestión seguida en este proyecto está basada en desarrollo el iterativo y dirigida por historias de usuario. Sin que ello implique que se haya usado Scrum, ya que no ha sido el caso. Se ha empleado un *story mapping* en 2 direcciones [7] para mantener una cohesión en el orden, priorización y gestión en general de las historias a desarrollar por iteración, ya que se trata de un desarrollo extensivo. También se ha utilizado un KanBan básico para poder tener una noción de las métricas mínimas para conocer el retraso, el avance y recalcular las historias en cada iteración. Por último, se ha utilizado simples logs para identificar errores, anticipación de problemas y propuesta de soluciones para el seguimiento del día a día.

La principal ventaja de la metodología de gestión escogida es que la misma resulte ágil y no suponga mucha carga de gestión y mantenimiento por tratarse de un *one-man project.* La otra ventaja y quizá la más importante es la disposición de un producto creado de forma incremental por cada iteración, lo cual permite re-definir las historias y adaptarlas al tiempo disponible. Así como también, poder anticipar con mayor antelación problemas y enfocar su solución mediante la propia redefinición de tareas o la adaptación.

Cabe destacar que la metodología de gestión es especialmente útil en todas las tareas relacionadas con el análisis semántico, editores, creación de

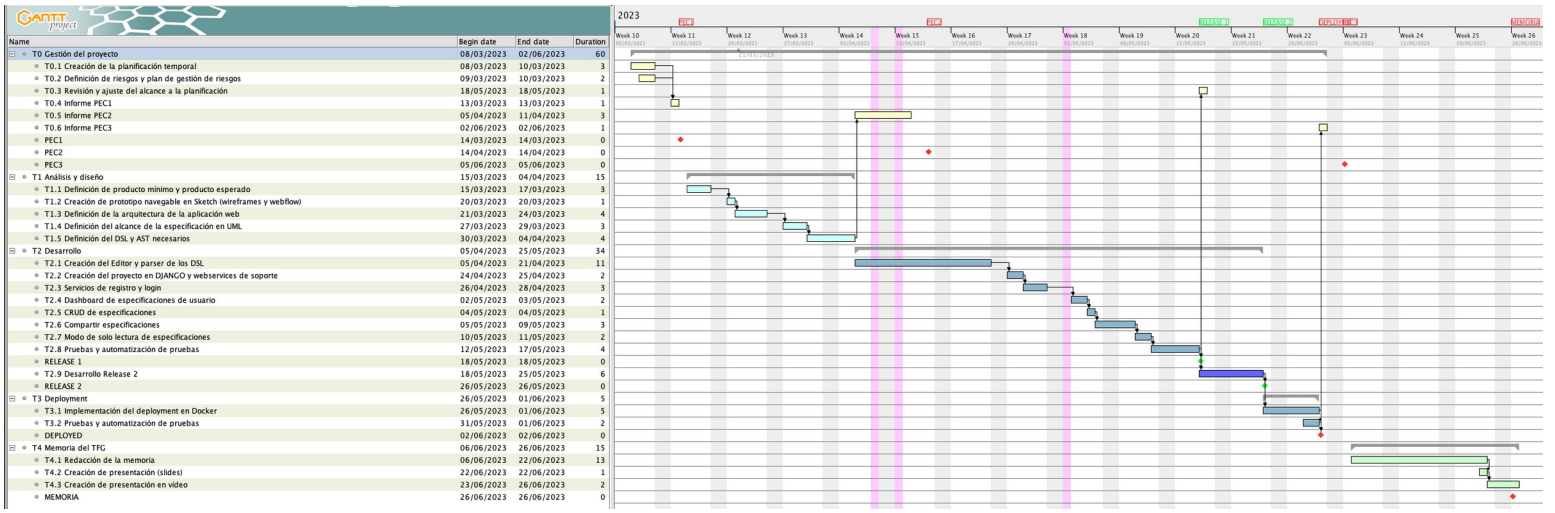
gramáticas, etc. Lo cual se aborda con una velocidad muchísimo inferior a la de otros aspectos de las aplicación, muchísimo menos complejos. Entonces, se puede estimar en jornadas ideales de 4 horas (y usar el valor escalar de la misma como puntos de historia) y así acotar por velocidad de desarrollo, para estimar tiempos más realistas. Tal y como propone Mike Cohn [8].

1.4 Planificación del Trabajo

Para la planificación se ha empleado un diagrama de Gantt con las siguientes características:

- Los días sábados, domingos (ambos en gris) y festivos (en rosa) de las jornadas no se contemplan en los cálculos de duración.

- Se señalan los hitos de la asignatura para poder relacionar los mismos con los diferentes hitos de intermedios del proyecto.



Name	Begin date	End date	Duration
T0 Gestión del proyecto	08/03/2023	02/06/2023	60
T0.1 Creación de la planificación temporal	08/03/2023	10/03/2023	3
T0.2 Definición de riesgos y plan de gestión de riesgos	09/03/2023	10/03/2023	2
T0.3 Revisión y ajuste del alcance a la planificación	18/05/2023	18/05/2023	1
T0.4 Informe PEC1	13/03/2023	13/03/2023	1
T0.5 Informe PEC2	05/04/2023	11/04/2023	3
T0.6 Informe PEC3	02/06/2023	02/06/2023	1
PEC1	14/03/2023	14/03/2023	0
PEC2	14/04/2023	14/04/2023	0
PEC3	05/06/2023	05/06/2023	0
T1 Análisis y diseño	15/03/2023	04/04/2023	15
T1.1 Definición de producto mínimo y producto esperado	15/03/2023	17/03/2023	3
T1.2 Creación de prototipo navegable en Sketch (wireframes y webflow)	20/03/2023	20/03/2023	1
T1.3 Definición de la arquitectura de la aplicación web	21/03/2023	24/03/2023	4
T1.4 Definición del alcance de la especificación en UML	27/03/2023	29/03/2023	3
T1.5 Definición del DSL y AST necesarios	30/03/2023	04/04/2023	4
T2 Desarrollo	05/04/2023	25/05/2023	34
T2.1 Creación del Editor y parser de los DSL	05/04/2023	21/04/2023	11
T2.2 Creación del proyecto en DJANGO y webservices de soporte	24/04/2023	25/04/2023	2
T2.3 Servicios de registro y login	26/04/2023	28/04/2023	3
T2.4 Dashboard de especificaciones de usuario	02/05/2023	03/05/2023	2
T2.5 CRUD de especificaciones	04/05/2023	04/05/2023	1
T2.6 Compartir especificaciones	05/05/2023	09/05/2023	3
T2.7 Modo de solo lectura de especificaciones	10/05/2023	11/05/2023	2
T2.8 Pruebas y automatización de pruebas	12/05/2023	17/05/2023	4
RELEASE 1	18/05/2023	18/05/2023	0
T2.9 Desarrollo Release 2	18/05/2023	25/05/2023	6
RELEASE 2	26/05/2023	26/05/2023	0
T3 Deployment	26/05/2023	01/06/2023	5
T3.1 Implementación del deployment en Docker	26/05/2023	01/06/2023	5
T3.2 Pruebas y automatización de pruebas	31/05/2023	01/06/2023	2
DEPLOYED	02/06/2023	02/06/2023	0
T4 Memoria del TFG	06/06/2023	26/06/2023	15
T4.1 Redacción de la memoria	06/06/2023	22/06/2023	13
T4.2 Creación de presentación (slides)	22/06/2023	22/06/2023	1
T4.3 Creación de presentación en vídeo	23/06/2023	26/06/2023	2
MEMORIA	26/06/2023	26/06/2023	0

Figura 2: Plan de trabajo

Así mismo, este proyecto se ha definido en las siguientes tareas de gestión:

- **T0**: Gestión del proyecto: Esta tarea será vital en este proyecto en virtud de ciertas tareas de desarrollo cuya estimación, alcance y complejidad son profesionalmente desconocidas. Como las fechas no pueden alterarse y deberían permanecer fijas en el plan, solo se podrá trabajar con el número de características. De modo que las características finales dependerán plenamente de una buena selección, priorización de acuerdo con los objetivos planificados.

    - **T0.1**: Creación de la planificación temporal inicial.

- **T0.2**: Definición de riesgos y plan de gestión de riesgos: Identificar los riesgos principalmente. Y definir un plan para abordar soluciones a los mismos.

- **T0.3**: Ejecución del plan: Seguimiento de la planificación, adaptación del plan de la RELEASE 2, selección y priorización de características (historias) del *backlog*, conforme al *story mapping*.

- **T0.4, T0.5 y T0.6**: Realización de informes periódicos de progreso: Los entregables de evaluación continua del TFG.

- **T1**: Análisis y diseño:

  - **T1.1**: Definición de Producto Mínimo y producto esperado: Conocer qué requisitos debe contener el producto mínimo que conformarán parte de la RELEASE 1. Definir y priorizar requisitos que se abordarían en parte en la RELEASE 1, pero sobre todo en la RELEASE 2.

    - Definición del *Story mapping* y las iteraciones que se realizarán en desarrollo, que se llevarán a cabo en 2 *releases*.

  - **T1.2**: Creación de prototipo navegable en Sketch (*wireframes* y *webflow*): Consiste en la creación del diseño UI/UX de la aplicación, considerando las pantallas más importantes y cómo se conectan entre ellas. Debe permitir una validación de concepto del producto final.

  - **T1.3**: Definición de la arquitectura de la aplicación web:

    - Diagrama de cómo se pretende realizar la solución del producto

    - Documentación de decisiones de arquitectura del software.

  - **T1.4**: Definición del alcance de la especificación en UML: El caso de uso de la aplicación no abarca todo el diseño de diagrama de clases de UML. Entonces, se debe definir el tipo de diagrama de clases de análisis, el subconjunto que comprenderá esta aplicación.

  - **T1.5**: Definición del DSL y AST necesarios: Diagramas de los Abstract Syntax Tree (AST) de los diferentes Domain Specific Language (DSL) utilizados.

- **T2**: Desarrollo:

- **T2.1**: Creación del editor IDE y parser de los DSL:
  - Diseño base mínimo de HTML y CSS.
  - Creación de las gramáticas especificadas para implementar los lenguajes definidos.
  - Implementación del coloreado de sintaxis en el editor, auto-completado básico e identificar los errores de sintaxis que cometa el usuario en la edición.
  - Generar lexers y parsers.
  - Características adicionales del editor, ventanas y navegación de la aplicación base.
  - Programación de listeners y estados. Dependerá de la elección de frameworks y de cómo se defina la arquitectura de la aplicación
- **T.2.2**: Creación del proyecto en DJANGO y webservices de soporte:
  - Programación de los servicios web, como por ejemplo:
    - Guardar especificación.
    - Compartir especificación.
  - Creación del modelo de datos.
  - Creación de los serializers.
- **T2.3**: Servicio de registro y login.
- **T2.4**: Dashboard de especificaciones de usuario.
- **T2.5**: CRUD de especificaciones: Cómo se modifican, renombran y borran especificaciones.
- **T2.6**: Compartir especificación: Generación del enlace para compartir la especificación.
- **T2.7**: Desarrollo Release 2: Desarrollo del conjunto de historias de usuario que permiten completar la solución. El alcance de esta tarea

e incluso su propia realización al completo, dependerá del resultado y conocimiento adquirido en las iteraciones anteriores de la Release 1.

- **T3**: Deployment: Tareas relacionadas con la publicación y gestión de entornos.
    - Implementación de deployment en servidor dedicado, con todo el despliegue de servicios requeridos (nginx, gunicorn, mysql server, etc.)
        - Parametrizar los entornos diferenciados de pruebas y producción.
        - Implementación de servicios y entornos.
        - Programación de scripts de inicialización desde cero, para la reproducción de la instalación por parte del consultor.
        - Publicación del proyecto.
- **T4**: Memoria del TFG
    - **T4.1**: Redacción de la memoria.
    - **T4.2**: Creación de la presentación en slides.
    - **T4.3**: Creación del vídeo de la presentación oral del proyecto.

1.5 Breve sumario de productos obtenidos

En este proyecto se obtienen los siguientes productos y subproductos:

- Aplicación web escrita en Python y django-res-framework:
    - Parsers de 3 lenguajes diferentes:
    - 3 Paquetes de lenguajes desarrollados para CodeMirror.
    - Entorno para la consulta de especificaciones, listado de especificaciones del usuario, opción de compartir y exportar a USE-OCL.

    - Servicio de evaluación de evaluación de especificaciones en UML y OCL.

- Adapter de USE-OCL.

1.7 Breve descripción de los otros capítulos de la memoria

- Capítulo 2, Análisis y diseño: Capítulo fundamentalmente descriptivo y de experimentación que explora y proporciona un análisis de cómo se piensa abordar el proyecto. Se discuten diferentes alternativas, se plantean diversas fórmulas de trabajo a seguir en la fase de desarrollo.

- Capítulo 3 - Desarrollo: Expone diferentes resultados obtenidos y describe aspectos relacionados con el desarrollo y plantea el conocimiento adquirido empíricamente. Comienza explorando los lenguajes y ciertos aspectos técnicos. Finalmente produce una descripción en retrospectiva del producto final. Como también plantea el alcance de ciertos aspectos fundamentales del proyecto.

- Capítulo 4 - Bibliografía: Relación con todos los aspectos estudiados que se han vinculado en la redacción de la memoria.

- Capítulo 5 - Glosario: Se describen algunos términos que podrían no haber quedado bien definidos dentro de la memoria. Ya que muchos términos se han intentado definir claramente en el mismo momento en que se mencionan

- Capítulo 5 - Anexos: En este capítulo se añade un manual que permite tener una experiencia visual con la aplicación sin abrirla. Sirve para exponer los aspectos fundamentales de la aplicación y describir cómo funcionan determinados procesos.

2. Análisis y diseño

Actores principales

Existe un enfoque más genérico de este proyecto y su uso en investigación de la ingeniería del software. Pero también existe un enfoque más concreto como el de la formación de estudiantes de esta disciplina en la ingeniería orientada a modelos. Este análisis se realizará desde esta perspectiva más concreta y no adoptando un enfoque genérico. Todo esto permite determinar un caso de uso también más claro de despliegue.

Adoptado el enfoque, se puede definir la aplicación como un posible servicio que pueda integrarse como una herramienta de formación en la Universidad. Estimando unos recursos limitados y costes mínimos.

En consecuencia, se consideran actores principales del proyecto una Universidad como la UOC (con un enfoque online) y sus estudiantes de ingeniería del software. Esta definición de actores implica que se debe abordar un lenguaje en la interfaz que puede resultar diferente del lenguaje específico de la comunidad de investigación, pero más adaptado al estudiante.

Tecnologías

Es necesario abordar las tecnologías que se emplearán para poder establecer el contexto que motiva la arquitectura de este proyecto:

- Selección del editor web y el parser que se emplearán en la aplicación. Específicamente en ese orden, ya que el editor determina (por compatibilidad) el tipo de parser y ambos tienen un impacto en el resultado gráfico del diagrama.

- Tecnología para diagramar en el navegador.

- Solución para la validación y evaluación de OCL como lenguaje, pese a que el peso de este proyecto recaiga en la especificación de modelos en diagramas de clases, su representación gráfica y el entorno para poder compartirlos.

2.1 Estado del arte

Este proyecto se basa en especificaciones en UML. Sin embargo, sería importante considerar que, aunque exista una especificación formal de UML y un *concrete syntax* de OCL, una especificación en UML siempre será semi-formal ya que OCL está definido semi-formalmente. Más concretamente, valdría la pena citar el *overview* de USE-OCL [2]:

> *"Due to the semi-formal definition of OCL there are some language constructs whose interpretation is ambiguous or unclear... we have presented a formalization of OCL which attempts to provide a solution for most of the problems".*

Esta situación dificulta que exista un compilador-intérprete de propósito general de especificación en UML con OCL desacoplado y en Javascript.

Por otra parte, existen diversas herramientas web que permiten realizar diagramas de clases en UML. Aunque la mayoría de ellas pone su foco de atención en la diagramación con casi total libertad antes que forzar un estándar de UML, validar y evaluar.

Y, pese a que sí existan entornos de modelado en UML, la mayoría de los mismos consisten en aplicaciones de escritorio y no son aplicaciones web. Además, no todas estas aplicaciones permiten la validación de la especificación durante el modelado o simplemente no tienen soporte para OCL. Dentro de ese subconjunto que sí soporta OCL, existe una herramienta (ya mencionada) de uso muy extendido en la comunidad de investigación de la Ingeniería del Software: USE-OCL, una aplicación de escritorio escrita en Java de la cual se discutirá en detalle en esta memoria.

Este proyecto tiene una similitud muy grande con USE. Es decir que esta herramienta se podría ver aproximadamente (y respetando las diferencias) como un USE en versión web. A pesar de ello, el proyecto discutido en esta memoria guarda un enfoque diferente, ya que se centra en la idea de un *code playground* y que, por lo tanto, se trata de una aplicación web con carácter social y una expresión textual.

2.2 Editor

La decisión del editor es una decisión de arquitectura, ya que el editor define las operaciones principales y su capacidad condiciona el funcionamiento de la aplicación.

Editores estudiados:

- VSCODE *for the web* [9]*:* Su *use case* es mucho más amplio y está completamente enfocado en su título: Visual Studio Code. Por eso tiene funcionalidades como la vista de comparación de diferencias entre documentos. Sin embargo, el soporte para la implementación de nuevos lenguajes es más complejo de adoptar. Especialmente cuando hablamos de pequeños DSL.

- Monaco [10]: Es el editor empleado en VSCODE *for the web* de Microsoft. Por eso ha cobrado bastante interés últimamente. Permite *syntax highlighting*, validaciones, etc. Pero como proyecto está enfocado en VSCODE, por lo que el soporte de múltiples lenguajes no forma parte del caso de uso. Solo se considera: css, html, json y typescript. Todo esto, en conclusión, dificulta la implementación de un nuevo lenguaje.

- ACE [11]: Es un editor maduro, cuyo enfoque es menos amplio que los 2 anteriores. Tiene un soporte muy amplio de diferentes lenguajes, con una muy buena capacidad de respuesta. Sin embargo, la extensión de lenguajes resulta compleja porque se debe seguir haciendo a mano toda la coloración (siguiendo la guía de los *mode*), auto-completado y diagnóstico de errores. No es una capacidad integrada, pese a que permita la implementación de parsers, como el propio ANTLR.

- CodeMirror [12]: Es un editor de texto maduro, como los anteriores. Su caso de uso es aún menos amplio que el de ACE, sin embargo, destaca por la capacidad de integrar nuevos lenguajes. Hasta el punto de tener un parser específico a juego: Lezer [13]. Lo cual directamente incluye la facilidad de crear paquetes de idiomas completos partiendo de la definición de la gramática.

Una vez expuestas algunas alternativas válidas de editores, se puede presentar el criterio de selección, el cual consiste en la simplicidad del servicio (y coste de mantenimiento) que tenga que dar soporte a la aplicación. De modo que, a los efectos de simplificar el mantenimiento, deberá utilizarse el navegador del usuario como editor y responsable de los lenguajes. Finalmente,

debe considerarse la facilidad de implementar nuevos lenguajes, puesto que los recursos y el tiempo de este proyecto son bastante limitados.

Siguiendo el criterio indicado, el editor que se ha sido elegido es CodeMirror. Principalmente porque la inclusión de nuevos idiomas resulta más práctica y porque presenta una solución que integra *syntax highlighting* y diagnóstico con un parser, lo cual permite trabajar directamente con gramáticas.

2.3 Parser

En este caso principalmente, se pretende evaluar la implementación de ANTLR como *parser* o la adopción alternativa de Lezer.

ANTLR [14] es un *parser* profesional y muy maduro que cuenta con una comunidad muy amplia y un buen soporte. Este *parser* se utiliza para lenguajes de diferentes propósitos, especialmente para trabajar con DSL. Incluso es el *parser* adoptado por proyectos como Xtext [15], que se centran exclusivamente en la creación de DSL.

Desde la versión 4, puede funcionar perfectamente en Javascript, debido al desacoplamiento de código en las gramáticas de los lenguajes en esta versión. Entonces, puede generarse el *parser* directamente en Javascript, así como también utilizar el programa de Java y generar el *parser* con *target* en Javascript.

Sin lugar a dudas, técnicamente y con mucho más tiempo disponible, resultaría conveniente trabajar en ANTLR. Las capacidades que tiene este *parser* son muchísimo más amplias para la ingeniería de lenguaje en comparación con las capacidades de Lezer. Sin embargo, además de considerar el aspecto del tiempo limitado de este proyecto, existen las siguientes consideraciones adicionales sobre ANTLR:

- No es un *parser* ***Error tolerant*** o más específicamente ***error insensitive***: Esta propiedad implica que el *parser* pueda continuar parseando un fichero a pesar de que se presente un error sintáctico o léxico. Esto significa que debe ser capaz de generar un PT partiendo de cualquier entrada, a pesar de los errores.

- No es un *parser* incremental [16]: La propiedad ***incremental*** indica que el *parser* produzca un PT de forma iterativa sin tener que procesar todo el fichero de entrada cada vez. En otras palabras, significa que se conservan los nodos del PT del fichero original y se parsean

exclusivamente los cambios. De modo que no hay motivo para re-parsear todo el fichero de entrada nuevamente, con todo el ahorro de consumo de recursos que implicaría.

Motivación de las propiedades mencionadas

Porque el objetivo de este proyecto es disponer de un editor web. De modo que resulte posible la especificación de modelos conceptuales. Para ello, se debe cumplir con unas funcionalidades básicas que tiene cualquier IDE moderno:

- Colorear las definiciones de la sintaxis para una mejor comprensión.
- Auto-completar código al escribir.
- Diagnosticar errores en el código.

Todo ello no resultaría posible si el *parse*r se detiene ante cualquier error y/o resulta extremadamente lento de usar. Porque el *parser* opera tras cada letra que se escriba en el editor de código. Entonces, no sería recomendable parsear todo el texto de entrada cada vez, porque el parseo se convertiría en una operación ineficiente.

Dándose el caso de que además resulta necesario implementar múltiples DSL en diferentes editores (IDE*)* en esta aplicación web. Entonces, se debe considerar una solución que optimice una operación tan recurrente en esta aplicación como lo será el *parsing*. Ello garantizaría obtener el menor tiempo de respuesta posible, aunque no se trate del *parser* más completo y mejor para la ingeniería de lenguajes.

Así que la decisión del *parser* está condicionada por el producto que se pretende desarrollar en este proyecto. De modo que, aunque CodeMirror no exija el uso de Lezer como *parser*, su integración resulta muchísimo más práctica: El propio *parser* resulta muchísimo más adaptado al caso de uso, porque además es *error tolerant* e *incremental*. Así que se debe optar por *Lezer* porque se considera que **debe mantenerse la ventaja de facilidad de integración y debe disponerse de un parser completamente adaptado al caso de uso de un IDE**.

2.4 Diagramación

2.4.1 Utilizar plantUML

Primero, ¿qué es *plantUML?*. *plantUML*[21] es una aplicación en Java que permite realizar diagramas de UML mediante texto, no resultando necesario producir los diagramas de clases manualmente. Es decir, que produce diagramas con un DSL propio con código del usuario como punto de partida.

```
@startuml
Class01 "1" *-- "many" Class02 : contains

Class03 o-- Class04 : aggregation

Class05 --> "1" Class06
@enduml
```

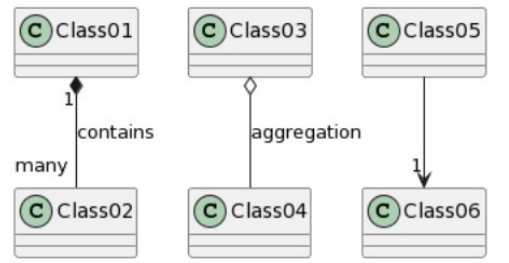


Figura 7: Código plantUML y diagrama resultante

Y la respuesta a la pregunta inicialmente planteada es: No.

No se puede utilizar *plantUML* por el simple hecho de que se estaría invalidando la arquitectura adoptada. Porque *plantUML* no se puede ejecutar en el navegador directamente (ya que está escrito en Java) y debería adaptarse a esta aplicación como un servicio web.

Considerando el código de diagramación como un producto que debe ser validado por los editores, este código tendrá que ser ejecutable o interpretado en el propio navegador del usuario, sin acudir a *webservice* alguno, en razón a que:

- Resultaría más costoso:
    - Si por cada tecla que presione el usuario hasta terminar de codificar, existe un *request-reply* de un servicio web (aunque se añada un *delay* por cada 2 ó 3 caracteres), es fácil deducir que el consumo ancho de banda será alto.
    - Consume mucho más proceso:
        - *plantUML* está pensado para diagramas de clases más amplios y con menos rigurosidad. Luego, tampoco sirve solamente para

diagramas de clase sino que soporta diagramas de actividad, diagramas de máquinas de estado, etc. Todo ello resulta completamente innecesario para este proyecto.

- *plantUML* está escrito en Java y decidiendo trabajar con *Python* y *DJANGO* como framework para el *backend*, la adopción de *plantUML* implicaría una ejecución externa de Java. O crear un nodo independiente que atienda estas peticiones. Lo cual añade una complejidad innecesaria o un proceso innecesario.

- plantUML requiere el código fuente completo. De poco sirve el trabajo del compilador instalado en el navegador con el cual se debe comprender el lenguaje para poder cumplir con los propósitos de un IDE en el propio navegador. Si, luego, se debe acudir a un *webservice*, se estaría duplicando el esfuerzo y tiempo nuevamente, resultando en una arquitectura ineficiente.

- Consume espacio: plantUML genera imágenes. Y el usuario recibe como resultado una imagen hipervinculada en el HTML. Por cada cambio del diagrama, se generaría una nueva imagen. Y cada una de estas imágenes por cada especificación de cada usuario se tendrían que ir almacenando, además de posiblemente gestionar sus versiones.

- Resulta mucho más lento, cuando se trabaja con editores y la velocidad es muy importante.

2.4.2 Utilizar Graphviz

Por otra parte, Graphviz[22] es un software *opensource* para la visualización de gráficos, el cual utiliza su propio DSL, conocido como DOT.

Además, *plantUML* precisamente utiliza Graphviz para la representación gráfica de los diagramas. De hecho, hay otros proyectos del listado de *Jordi Cabot* (en su blog[23]) que también utilizan Graphviz con el mismo propósito de representación gráfica de diagramas.

De modo que resulta completamente válido pensar en utilizar directamente Graphviz.

Sin embargo, utilizar Graphviz es mucho más complejo que utilizar *plantUML*, porque implica realizar una transformación *M2T (model to text)* [24] del DSL de diagramación al lenguaje DOT[25].

Para ello habría 2 alternativas:

- Utilizar Graphviz en el *backend*, en *python*, que podría suponerse más sencillo. Lo cual no resultaría válido, precisamente por los mismos motivos mencionados anteriormente para *plantUML:* Principalmente por el sobre coste innecesario de ancho de banda y espacio en el servidor. Porque al final, será mejor renderizar el resultado de la transpilación a DOT en imagen y mandarle como respuesta un simple hipervínculo al usuario, en lugar de transmitir el SVG resultante por cada petición.

- Utilizar Graphviz en el *frontend*: Se puede utilizar Javascript y adoptar la librería HPCC/WASM[26] de forma que el navegador se encargue de la renderización del DOT, que también se tendrá que transpilar en el propio navegador. Todo esto permite mantener la definición de la arquitectura inicialmente propuesta y obtener tiempos de respuesta instantáneos, emparejado con las modificaciones letra por letra.

Dicha todas las alternativas y observando la complejidad de la adopción de *Graphviz*, se puede apreciar el gran beneficio que supone para el proyecto.

2.5 Evaluación de especificaciones

En esta etapa se ha podido conocer la dificultad de la implementación de OCL como lenguaje, su evaluación, validación completa, etc. Por consiguiente, se considera necesario la reutilización de algún proyecto ya funcional que lo implemente, para que sea viable disponer de esta funcionalidad sin mayor dedicación de tiempo. En esta línea, se plantea la opción de utilizar USE-OCL.

Cabe comentar que la elección del USE para este proyecto resulta de la experiencia de uso, sin haber realizado un estudio de alternativas posibles. Simplemente se ha optado por una opción conocida y conveniente.

¿Qué es USE-OCL?

USE es un proyecto *open source* desarrollado originalmente por *Martin Gogolla*, *Mark Richters*, entre otros muchos más investigadores de ingeniería del software [17]. Se trata de una herramienta para la especificación en UML desarrollada en Java. Una herramienta para trabajar especificaciones en el escritorio.

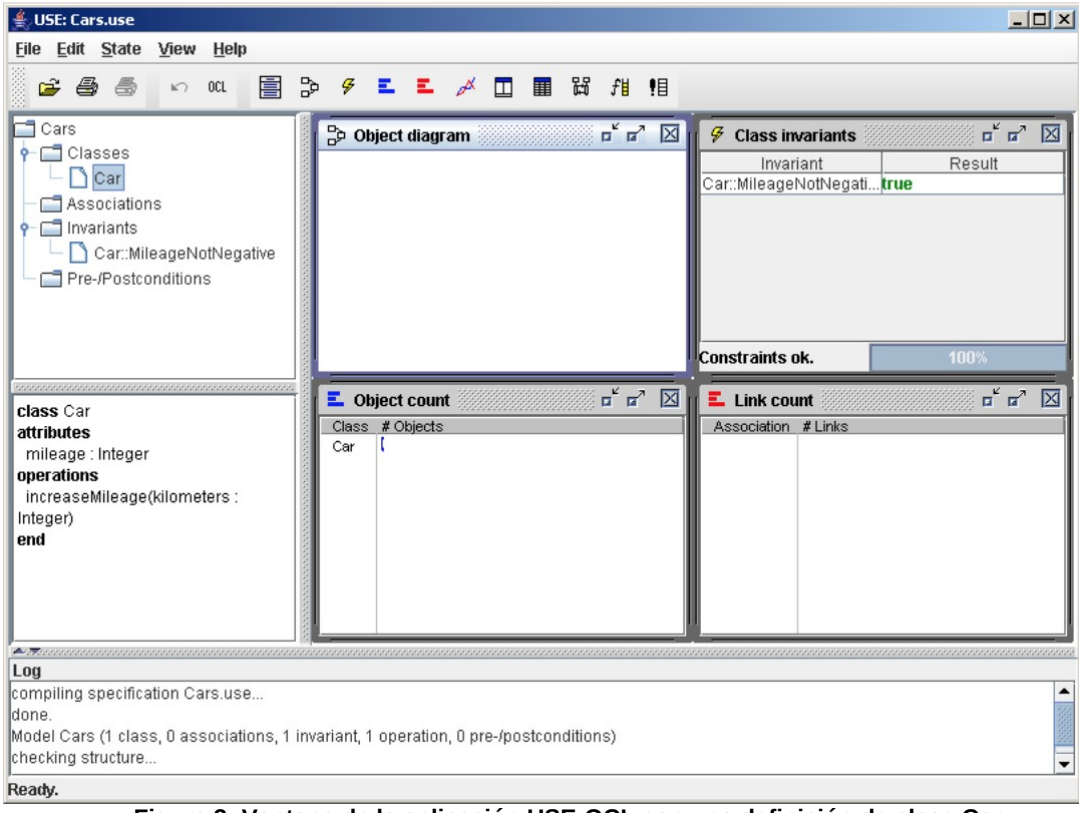


Figura 3: Ventana de la aplicación USE-OCL con una definición de clase Car

Sin embargo, MBSPEC (este proyecto) es un *code playground* para trabajar especificaciones en el navegador. De modo que se pueda compartir, listar,

trabajar en colaboración, etc. En conclusión, coinciden en unos aspectos aunque tienen propósitos muy diferentes.

Por lo tanto, USE permite tener un soporte para el lenguaje OCL, que no sería posible desarrollar en este proyecto.

Indisponibilidad de un compilador de OCL

No existe la posibilidad de usar el compilador de USE-OCL tal cual. Precisamente por este motivo se ha considerado exponer algunos patrones de implementación de lenguajes en este TFG, ya que se puede apreciar una diversidad de patrones que resulta adecuado señalar y que fundamentan las decisiones tomadas. Así, en esa misma linea, en USE-OCL se adoptaron los patrones (de Martin Fowler[18]) de *Syntax-Directed Translation* y *Embeded Translation*, con los cuales el *parser* quedó completamente acoplado a su caso de uso. Y, aunque esto sea lo habitual cuando se trabaja directamente con Lexer y Cup (hablando con las tecnologías propias de la Universitat Oberta de Catalunya de 2023), **el parser por sí mismo no se puede re-aprovechar.**

En el caso específico de USE se produce uno o varios AST que forma parte de los subsiguientes componentes del compilador. Todos los cuales no son útiles para MBSPEC.

En conclusión, la única forma de utilizar USE es creando un adaptador que permita adecuar el propósito del funcionamiento de tal compilador al caso de uso de este proyecto.

Sin embargo, el uso de USE no resulta del todo conveniente, tal y como se discutirá en detalle en las decisiones adoptadas sobre la arquitectura de este proyecto. Se tratará como algo accesorio, primero porque encarece innecesariamente el coste de desarrollo y (segundo y más importante) porque produce una ineficiencia del coste de mantenimiento del servicio, cuando siempre será mejor hacer la validación y la evaluación de OCL en el navegador del usuario y no a través de servicios web.

Por consiguiente, la adopción de USE será mínima y con fecha caducidad.

Finalmente, sobre USE, cabe expresar las limitaciones añadidas del soporte del lenguaje OCL, debido a esta adopción mínima y temporal. Aunque, para poder comprender estas limitaciones es necesario explicar cómo está estructurado USE, de forma muy general:

- USE trabaja con diferentes lenguajes y gramáticas. Entre ellas tenemos, la gramática de USE, OCL, SOIL y SHELL.

- La gramática de USE integra el diseño de clases con incrustaciones de lenguaje OCL para la expresión de atributos derivados y reglas invariantes de la especificación. Esto es lo que se encuentra habitualmente en los ficheros *.use*.

```
class Apple < Orange, Lemon
end

abstract class Orange
attributes
  juice : Boolean
end

class Lemon
operations
  squeeze(i : Integer) : Integer = i + 1
end

class Banana < Lemon
attributes
  flatware : Set(Sequence(Flatware))
operations
  peel() : String = 'abcd'
      pre: true
      post: 2 = 2
      post: result = 'theResult'
end

class Peach
attributes
operations
constraints
  inv: 3 > 2
  inv neverViolated: true
end
```

Figura 4: Ejemplos de fichero .use

- El lenguaje que USE utiliza para la instanciación es: SOIL *(Simple OCL-based Imperative Language)*.

```
!create smallCar : Car
!create bigCar : Car

!set smallCar.mileage := 2000
!set bigCar.mileage := -1500
```

Figura 5: Ejemplo de instanciación en USE-OCL

Dicho lo anterior, entonces es posible enumerar las limitaciones que añadimos al adoptar USE:

- No se puede definir derivaciones vía operaciones y uso de "body". Solo se puede hacer vía en atributo *derive* del lenguaje de USE (*derive* = expresión OCL booleana). Sin embargo, esto se debería poder hacer de acuerdo a Antoni Olivé [19].

- No se puede definir derivaciones directamente sino es a través del .use (no se consideran las que se puedan añadir por la interfaz gráfica,

porque al final se traducen en .use). No se puede usar una regla como: "*context EntityType:derivedAttribute: Boolean derived OCLExpression*". Ya que de esta forma solo es posible especificar *invariant constraints* al final del fichero .use.

- No es posible crear expresiones con el *keyword* "init" de OCL. No resulta necesario para su caso de uso.

- Con el *keyword* "context" solo se pueden crear *invariant constraints*.

Lo importante es considerar que **NO SE ABARCA TODO EL LENGUAJE OCL EN SU COMPLETA EXTENSIÓN**. Y que la limitación que MBSPEC tiene sobre lo que pueda abarcar de OCL está condicionada también por la implementación de USE, aunque sea de forma mínima y temporal.

2.6 Arquitectura de la aplicación

2.6.1 Definición de características

Antes de discutir la arquitectura de la aplicación, es conveniente hacer una pequeña introducción al patrón *language server* [20]. Este término, empleado por Microsoft (que vinculo en la bibliografía), hace mención a un protocolo específico de comunicación entre un editor de texto (IDE) y la implementación de lenguajes. Hablando el mismo protocolo, un editor de texto puede adquirir las habilidades de un lenguaje sin tener que programar una extensión específica del IDE para ello. En sí mismo, hace mención al desacoplamiento de la capacidad de reconocimiento y tratamiento de lenguajes del IDE, del propio editor, trasladando esta responsabilidad a un servidor (habitualmente local).

Esta medida resulta necesaria cuando las operaciones con el lenguaje resultan más extensas. Por ejemplo:

Además del tratamiento básico del lenguaje, se puede añadir muchas otras operaciones automáticas de refactorización de código. Se puede disponer de aplicaciones específicas de *intellisense*, complementar aspectos del IDE con modelos de inteligencia artificial, etc.

Si imaginamos todas estas funcionalidades ejecutándose en el mismo editor (o IDE), se puede intuir que el resultado será un proceso de edición bastante recargado, porque fácilmente se observa el alto consumo de recursos. Así mismo, se producen muchísimas complicaciones por la adecuación del entorno. Sin embargo, mediante *language server* se descarga al cliente-editor y se centraliza la responsabilidad en un servidor.

Seguidamente, considerando el inciso anterior, se pasa a discutir la arquitectura de la aplicación desarrollada en este proyecto:

La arquitectura propuesta en este proyecto se basa en la **separación** de la aplicación (la especificación en UML) del servicio (*code playground* y compartir especificaciones, colaborar...), considerando que se trata de un modelo de SASS (*software as a service*). En el sentido estricto de la especificación y presentación de resultados, el coste del funcionamiento de la aplicación debe correr a cargo del cliente. Es decir que **el funcionamiento de los editores y compiladores debe producirse en el navegador del cliente**.

Esto precisamente enlaza con el punto anterior del uso de USE:

No tiene sentido que parte de estas operaciones básicas corran a cargo del servidor. ¿Por qué? **¿Por qué no funciona un modelo *language server* en este caso?**

Porque, si corren a cargo del servidor, definitivamente aumenta el coste y la complejidad del servicio. Y esta aplicación tiene que encajar en la financiación de las Universidades, por lo que debe resultar económico su mantenimiento o cercano al cero-coste, con el propósito de difundir el uso de UML y especialmente de OCL para la especificación en la comunidad de estudiantes (en primer lugar), profesionales practicantes e investigadores. Dicho más detalladamente:

- Adoptar un modelo *language server* implica centralizar la responsabilidad de los DSL en un servidor y por consiguiente multiplicar el coste por nodo y el número de nodos para cumplir con garantías. E igualmente implica disponer de una arquitectura autoescalable y a demanda, con todo el sobrecoste que esto supone.

- Como el caso de uso de *language server* supone parsers incrementales, se tiene que conservar el árbol en memoria el nodo de servicio. Esto también tiene consideración en el coste.

- Implicaría la necesidad de utilizar un *websocket* bidireccional con el navegador, lo cual **está soportado solo por navegadores modernos**. Considerando que la alternativa ineficiente y compatible sería AJAX, con todo lo que en velocidad implicaría.

- El caso de uso de un *language server* es el de separar el proceso de análisis del lenguaje del editor de código. Porque el análisis semántico y otros tratamientos más abstractos del lenguaje implican sobrecargar el editor.

    - De acuerdo a este principio, el uso del editor requiere un tiempo de respuesta rápido.

    - Y que el uso del lenguaje no sea simplemente de análisis léxico sintáctico y semántico.

En todo caso, lo único que se debe pensar que pueda ser escalable en la aplicación es el servicio de disposición de compartir una especificación personal con cualquier persona, mirarlo en cualquier dispositivo, en cualquier parte y el alojamiento de este espacio, listado de especificaciones, etc. Siempre, considerando que no exista la dependencia con USE, que ya ha sido categorizada como temporal.

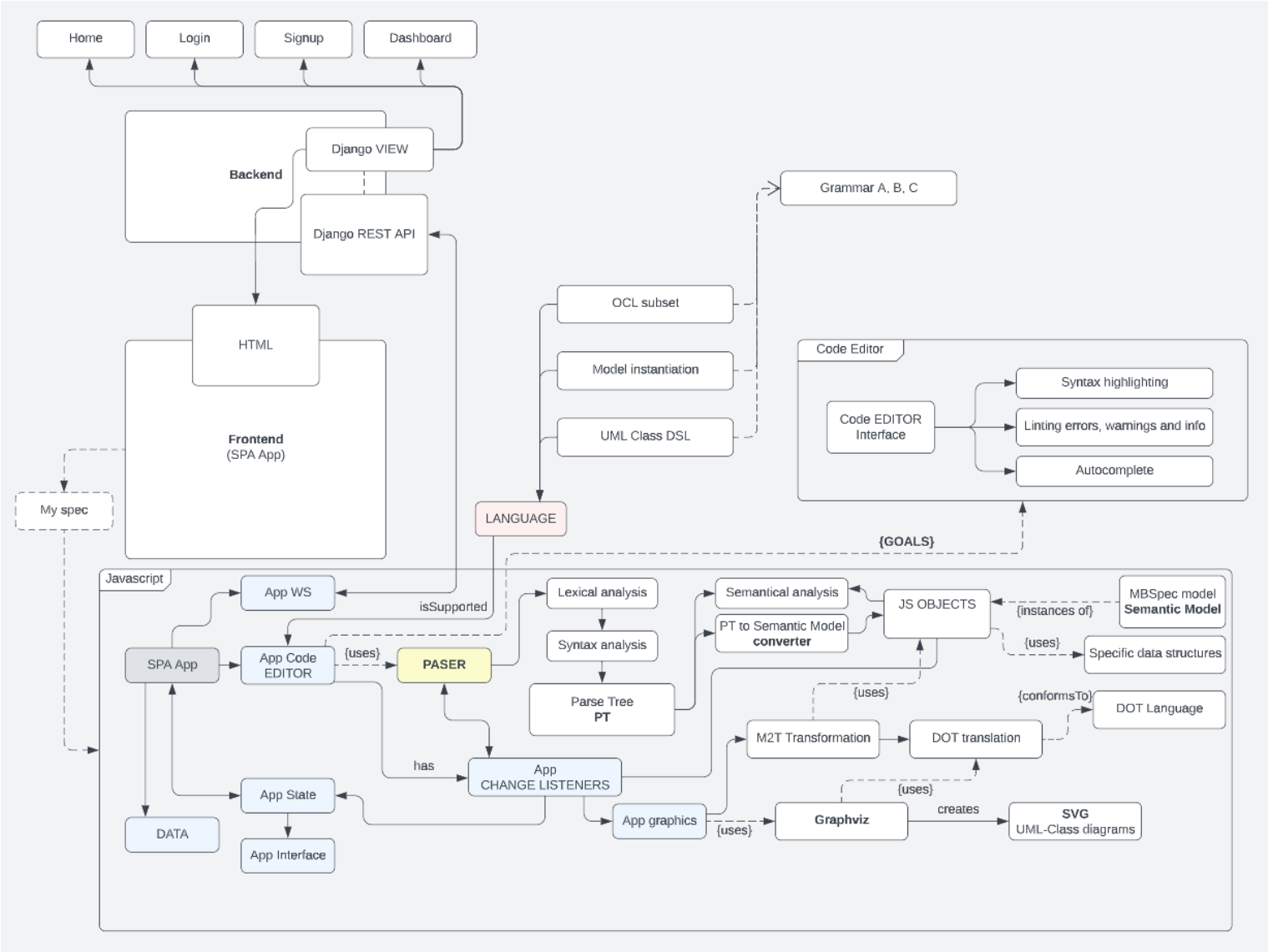


Figura 6: Arquitectura de aplicación desde la perspectiva de los servicios del producto

A nivel estructural de la memoria, la definición de la arquitectura se expone seguida de la adopción del editor, parser y método de diagramación. A pesar de ello, la arquitectura condiciona totalmente tales opciones, de modo que los principios básicos de la arquitectura se definieron con anterioridad. Posteriormente, la definición más técnica de la arquitectura ocurrió paralelamente a la definición de las mencionadas opciones.

Perspectiva de la arquitectura y la diagramación

Para finalizar, se muestra la arquitectura vinculada con la visualización de diagramas de clases:

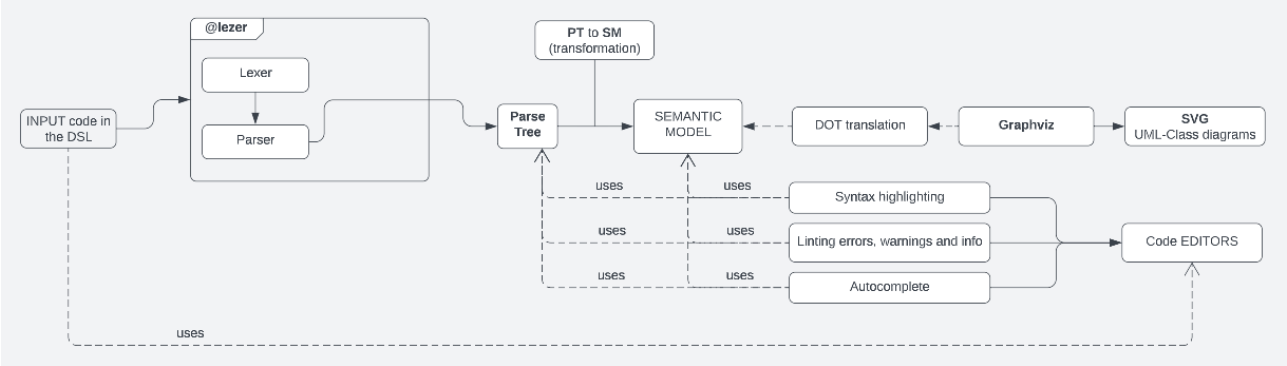


Figura 8: Arquitectura desde la perspectiva del lenguaje de diagrama de clases y su representación

Como puede observarse, en ambas vistas de la arquitectura, se aprecia el término *semantic model* o modelo semántico.

El *semantic model* es un concepto que se ha extraído del libro Martin Fowler[18]. Explica un resultado que se puede obtener tras parsear un DSL. Como tal, en este proyecto se observa este concepto en las clases específicas creadas para expresar el significado del código escrito por el usuario en el editor, que posteriormente sirve para hacer otra transformación M2T (*Model to text*) al DOT Language, especialmente para el tratamiento del lenguaje de diagrama de clases y el lenguaje de instanciación que se emplearán en la aplicación.

Este modelo semántico es el que se pasará a definir en los puntos siguientes.

2.6.2 Arquitectura 3-tier

Se obvia USE en la diagramación de la arquitectura porque se trata de una solución temporal adoptada para poder ofrecer resultados de validación de OCL. Sin embargo, USE no ocupará un proceso independiente o nodo la capa de negocio, ni será escalable de ninguna manera. Porque simplemente constará de un *adapter* programado específicamente para el caso de uso de este proyecto.

Se presentan 3 capas:

- La capa de presentación: Editor y lenguajes de la aplicación.

- La capa de negocio: Donde tendremos todos los servicios relacionados con los usuarios, compartir, listados de especificaciones, etc.
- La capa de datos: En la que se trabajará con MariaDB.

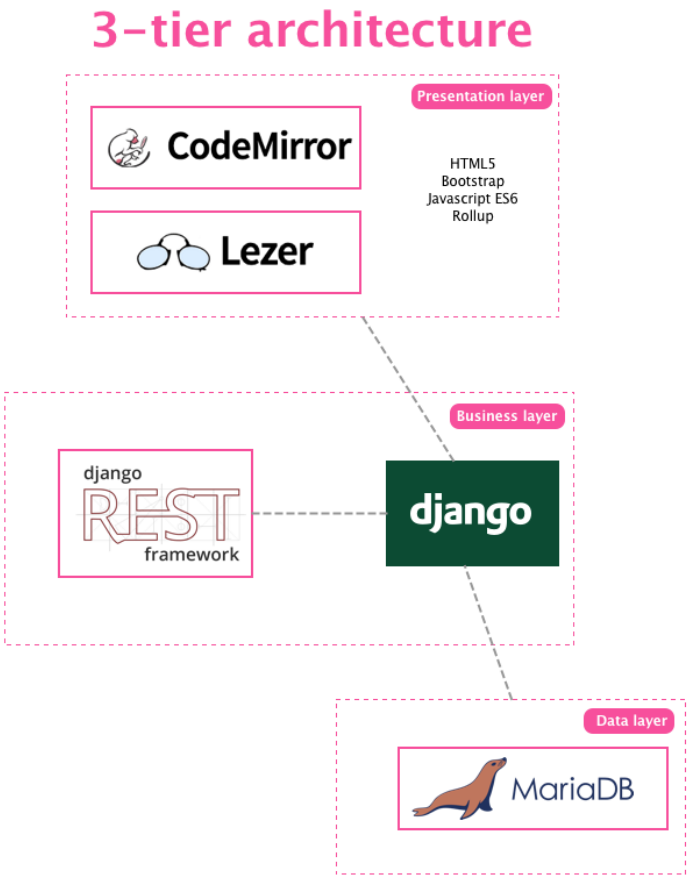


Figura 9: Arquitectura en capas de la aplicación

2.6.3 Patrones y definiciones del compilador

Terence Parr introduce el término *event method* [27]: Un patrón para evitar el acoplamiento tradicional entre la gramática de un *parser* y el código de la aplicación resultante. Este término se podría aproximar con el término *syntax directed definition* o SDD y así facilitar su comprensión. En consecuencia, la utilidad de los *event methods* es básicamente desacoplar el código específico de la aplicación de la gramática, rehuyendo de las también llamadas *embeded actions*.

Entrando en detalle, los *event methods,* que en ANTLR son el *Listener* y *Visitor*, permiten ganar legibilidad en la gramática del *parser*, ganar en capacidad de reutilización del *parser* resultante y producir un desacoplamiento entre el lenguaje de programación *target* y el *parser*.

En esa misma línea, conviene expresar que la solución *Lezer-CodeMirror* obliga a producir este desacoplamiento (siguiendo el mencionado patrón) y a la modularidad de los diferentes idiomas. Todo mediante los llamados *language-packages*, que permiten la reutilización de los lenguajes en diferentes editores CodeMirror con diferentes propósitos.

Sin embargo, conviene destacar:

- *Lezer* no dispone de estructuras para hacer transformaciones de PT a AST, aunque cuenta con un método para realizar un recorrido manual del PT, conocido por Cursos. Y también cuenta con la posibilidad de realizar un recorrido automático del grafo, mediante *Cursor Iterate*.

- Así, *Cursor Iterate* se puede ver como el *event method* propio de *Lezer*. Este estaría próximo al concepto del *Listener* de ANTLR pero dentro de un recorrido automático *Depth First Search* por todos los nodos del PT.

- Aunque ciertamente es posible modificar el recorrido de un *Cursor* manteniendo su estado y el DFS, específicamente sobre el mismo nivel del PT (mediante las operaciones *nextSibling* y *prevSibling*), que se usará para poder realizar la revisión semántica de las expresiones.

Tras introducir los patrones y las soluciones de la tecnología adoptada, se puede discutir los 2 métodos diferentes que se deben aplicar para la implementación de los lenguajes de esta aplicación:

- **Generar un *semantic model*:** Se trata de un caso de uso que señalado por Keith Cooper[28], el cual describe la construcción de objetos para proveerlos de atributos y valores. Primero, considerando que el PT es la IR (*intermediate representation*) válida. Y, seguidamente, operar esa IR para producir un modelo (o unas estructuras de datos) que resulte conveniente para el propósito de la aplicación. Igualmente, este método empleado está descrito por el patrón *Tree Construction* (de Martin Fowler[18]), en el que el modelo resultante es el conocidísimo: *Semantic Model*.

- **Generar un *AST*:** Para evaluar una especificación o realizar un análisis semántico más completo de OCL, no se puede utilizar un *semantic model* y resulta necesario trabajar con otra IR. Es decir, que se requiere la construcción de una abstracción del PT, que produzca una reducción de los detalles que sirvieron como soporte sintáctico, centrado en

operaciones, operadores y tipos. Generando, en consecuencia, lo que se conoce por AST. Por ejemplo:

Dentro del análisis semántico de OCL, para asegurar el cumplimiento de *types* y *return types* es necesario evaluar las sub-expresiones de OCL que compongan una única expresión OCL. Así, para realizar todas estas validaciones más complejas se debe partir de un AST que permita disponer de la evaluación del tipo de las sub-expresiones de OCL, las que servirán posteriormente como operadores de otras operaciones.

Entre los motivos para exponer en detalle la metodología arriba mencionada, quisiera destacar los siguientes:

- El hecho de que en este proyecto no se puede abordar el segundo método de implementación de los lenguajes. Ya que no se trabajará en base al *concrete syntax* ni el meta-modelo del AST propuesto en documento de la OMG[5].
    - Que por ese motivo se ha empleado USE para la evaluación de especificaciones.
- Y para poder indicar claramente e ilustrar el patrón empleado en este proyecto: El modelo semántico.

2.7 Modelo semántico

El modelo semántico general que se ha definido para este proyecto está condicionado por la dependencia de USE. Sin embargo, puede adaptarse fácilmente en el futuro para poder implementar un intérprete completo que valide y evalúe el lenguaje OCL.

Este modelo tiene la siguiente composición (que también se muestra en la figura):

- Reificación: Donde se produce la acción de diseño que convierte las ideas en un modelo.
- Reglas de OCL: Donde se definen las restricciones del modelo, operaciones, atributos y queries.
- Transacciones e instanciación: Donde se crean instancias o individuos del modelo reificado.

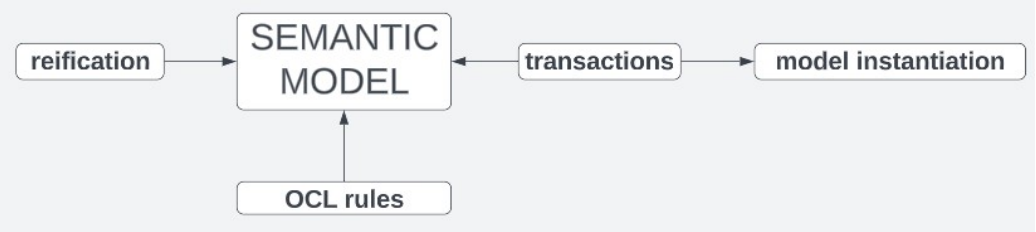


Figura 10: Vista del modelo semántico en relación a la aplicación y lenguajes usados

2.7.1 Modelo semántico de reificación

A continuación se expresa cada componente del modelo, comenzando por el modelo semántico de reificación, que sirve para expresar el significado del DSL de diagrama de clases. Por consiguiente, el propósito de este modelo semántico es el de servir de meta-modelo para la construcción de todas las clases definidas por el usuario.

En resumen, se deben considerar los siguientes detalles:

- **No tiene soporte para multi-herencia de clases:** Específicamente porque no lo soporta Java y porque el estilo del lenguaje que se creará

será más intuitivo para un desarrollador de Java. Además de que tampoco está soportado por USE.

- **Dispone de una definición específica para un atributo con valor:** Esta clase no tiene ninguna utilidad en el lenguaje de reificación; pero es indispensable en el modelo semántico de la instanciación de clases. De modo que este modelo es importante considerarlo como indispensable para la instanciación de clases. Sirve para definir las instancias y sus atributos con valores expresados en *ValueTypes*. Ya que no es lo mismo expresar el tipo *:String* que el valor de tipo *"Cadena de texto"*, expresado con comillas.

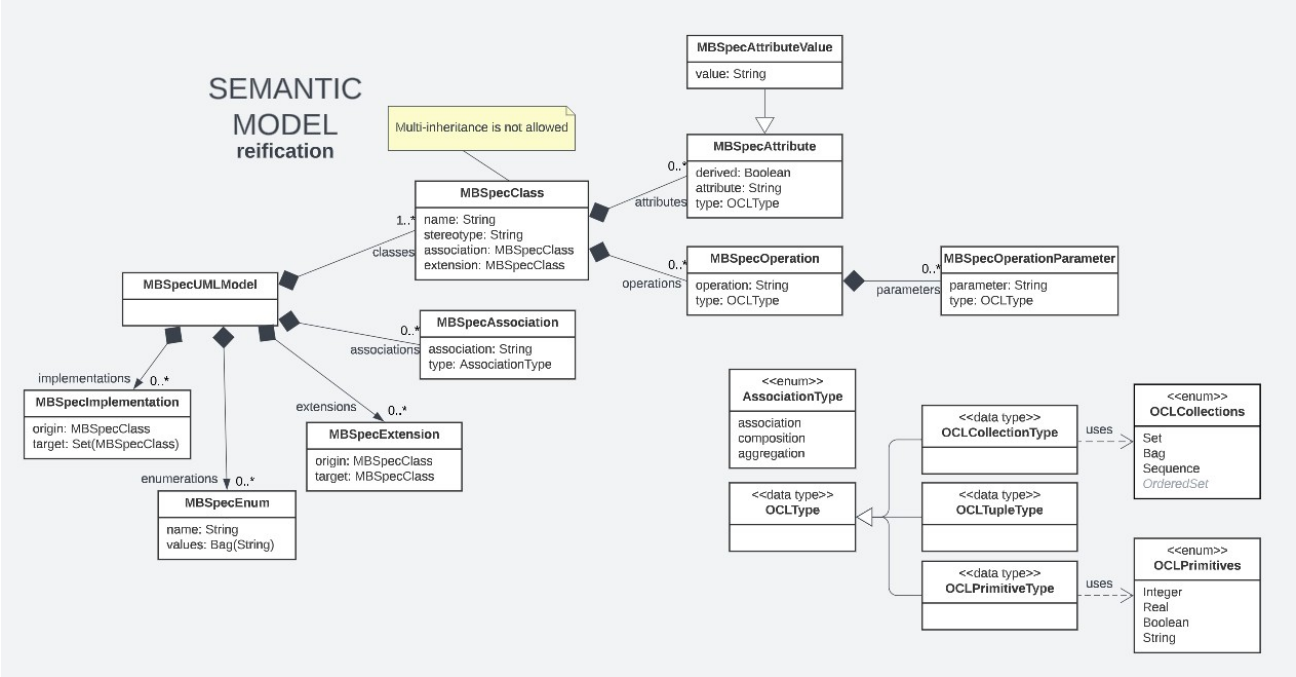


Figura 11: Meta-modelo del modelo de clases

2.7.2 Modelo semántico de reglas de OCL

Este modelo es el componente de una estructura de datos simple, una lista enlazada de las diferentes reglas de OCL.

Las expresiones de OCL se producirán de forma completamente independiente. A diferencia de la forma adoptada por USE, en la cual el OCL se incrusta en el diagrama de clases dentro del fichero *.use*. Esto abre la posibilidad a la implementación futura de expresiones que no son compatibles con USE.

Cabe destacar de este modelo que se planea soportar 2 tipos de reglas inicialmente:

- Los atributos derivados
- Y las reglas invariantes.

Podría darse el caso de que sea posible alcanzar el soporte a *queries* simples de la especificación. Pero, como no es seguro, no se expresa ese soporte en el modelo actual:

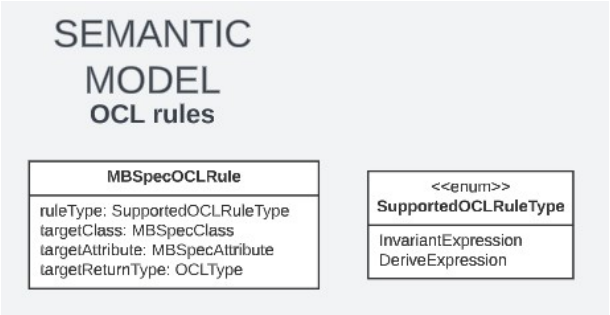


Figura 12: Vista del modelo de reglas

2.7.3 Modelo semántico de transacciones

Cuando se habla de la creación de un modelo de transacciones tras procesar un *PT*, puede intuirse que exista cierta relación del modelo con un *AST*. Lo cual efectivamente resulta cierto, ya que este modelo convierte el *DSL* de partida (ya parseado en un *PT*) en un conjunto de operaciones y operadores, al estilo de un típico *AST*, pero mediante una estructura diferente al grafo: Un modelo.

Este modelo en particular es el que permite posteriormente la construcción del otro modelo semántico de instanciación.

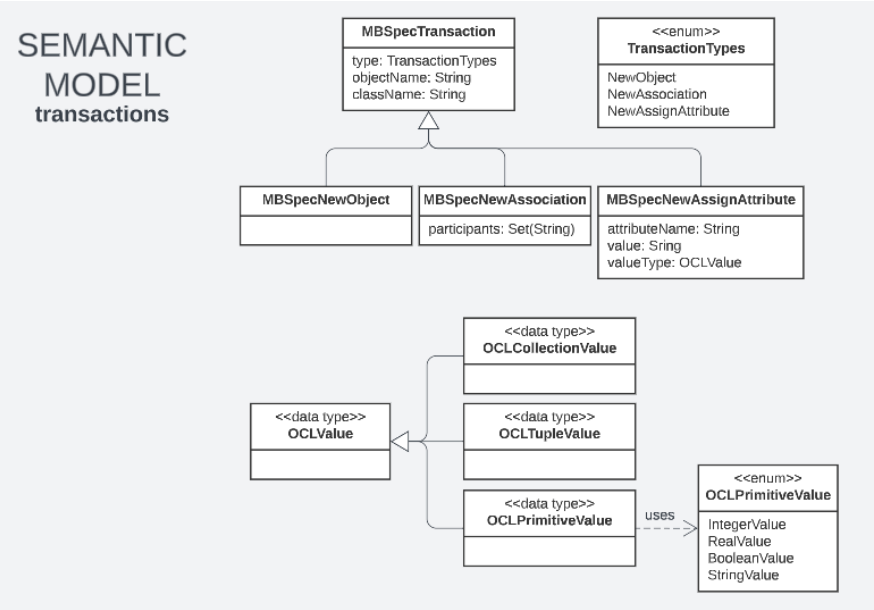


Figura 13: Vista del modelo de transacciones

Se puede destacar los siguientes motivos para hacer acopio de transacciones:

- Sirven como base para la transpilación a *SOIL* de *USE*. Operación para la cual no resulta útil el modelo semántico de instanciación (por eso el modelo de instanciación no formará parte del *request* de evaluación vía *webservice*).

- Sirven para la transpilación a DOT en combinación con el modelo semántico de instanciación.

Principalmente, por el primer motivo expuesto de la lista anterior, se considera en este proyecto las transacciones como un modelo semántico y no como una representación intermedia (*IR*).

2.7.4 Modelo semántico de instanciación

Este modelo se puede ver como el resultado de interpretar el modelo semántico de transacciones.

En el mismo se emplea la especificación de la clase atributo que sí permite guardar un valor con un *ValueType* en el modelo semántico. Modelo definitivo que sirve concretamente para la generación de la instancia del objeto.

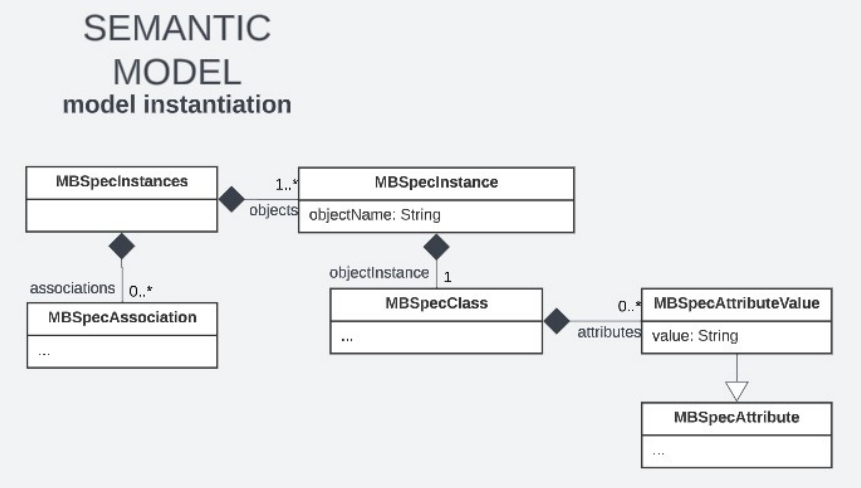


Figura 14: Vista del modelo de instanciación

2.8 Modelo relacional de base de datos

El modelo que se presenta a continuación es un modelo que permite compartir las especificaciones, almacenarlas, listarlas y visualizarlas posteriormente en cualquier dispositivo y desde cualquier lugar. También implementa usuarios grupos y permisos para poder incluirlos en los servicios.

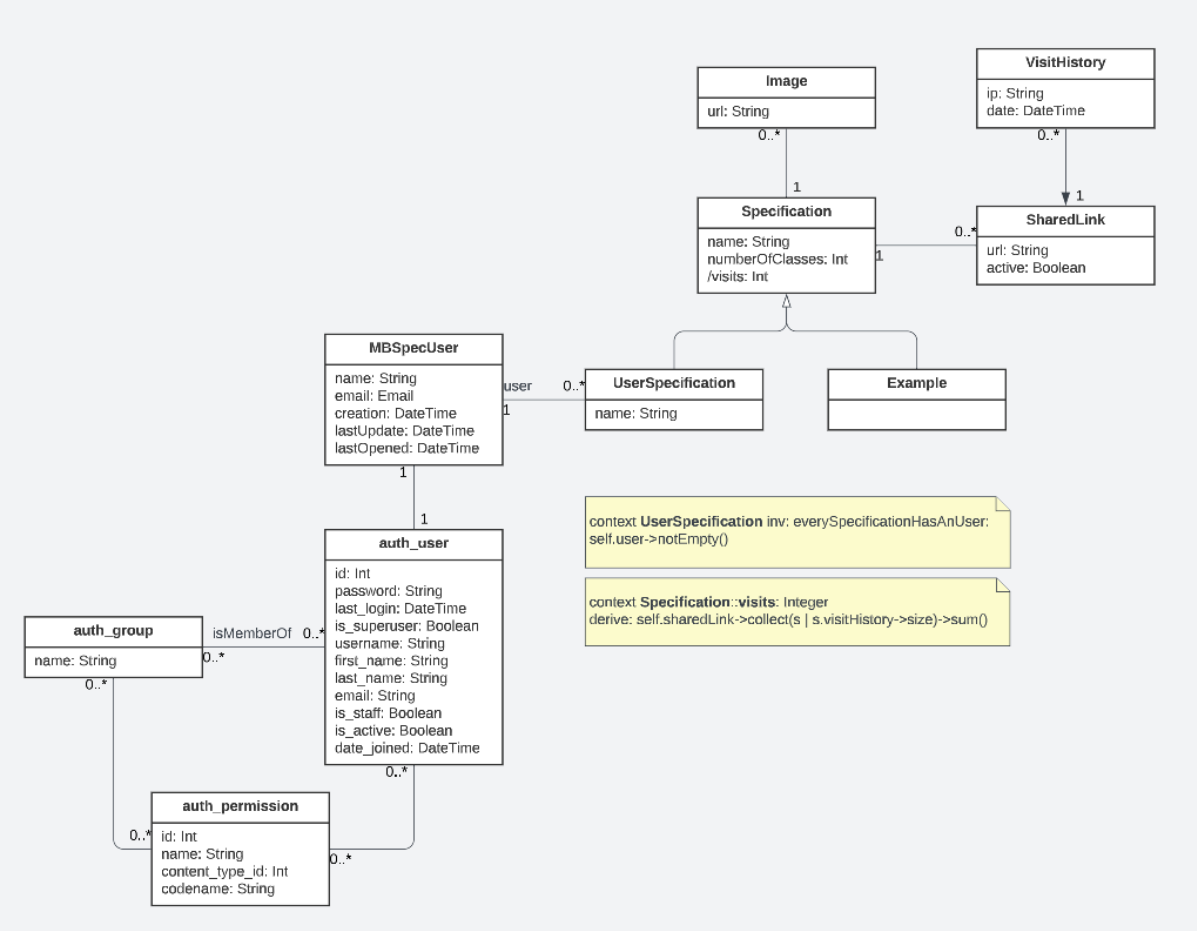


Figura 15: Diagrama de clases de la aplicación

Este pequeño modelo conceptual es el que directamente se pasa a implementar en el fichero *models.py* dentro del *framework* seleccionado (*DJANGO*). Este proceso de creación del models.py constituye la creación del modelo lógico de la base de datos. Lo cual supone una gran ventaja posteriormente, ya que automatiza la creación del diseño físico mediante *DJANGO Migrations*.

El cliente de la base de datos relacional se debe especificar con la parametrización de la aplicación del *backend*. Como también debe considerarse que tal cliente se deberá instalar en *Python*. Por lo tanto, tendrá que ser compatible con el servidor de base de datos relacional seleccionada (*MariaDB*).

2.9 Deployment

El *deployment* se hará en un *Host* con sistema operativo *Debian*. Específicamente, se trabajará con una infraestructura de VPS en OVH. Se utilizará Gunicorn como WSGI para Python. El stream de Gunicorn se procesa con X-Forwarded en NGINX. Sin embargo, los contenidos estáticos los debería servir NGINX directamente, ya que Gunicorn no es un servidor web sino un WSGI.

Finalmente, hay que disponer de un servicio de base de datos MySQL. En este caso se ha utilizado el servidor MariaDB con el puerto habitual. Sin embargo, las credenciales de acceso está expresado en variables de entorno del sistema para evitar que exista contenido sensible dentro de los ficheros de configuración de DJANGO.

Es posible desacoplar el *adapter* de USE del *backend* y escalar ese nodo. Aunque, como tantas veces se ha expresado en los puntos anteriores, USE no puede considerarse como un elemento perenne en esta aplicación web, por lo que no tiene mucho sentido invertir tiempo en ello.

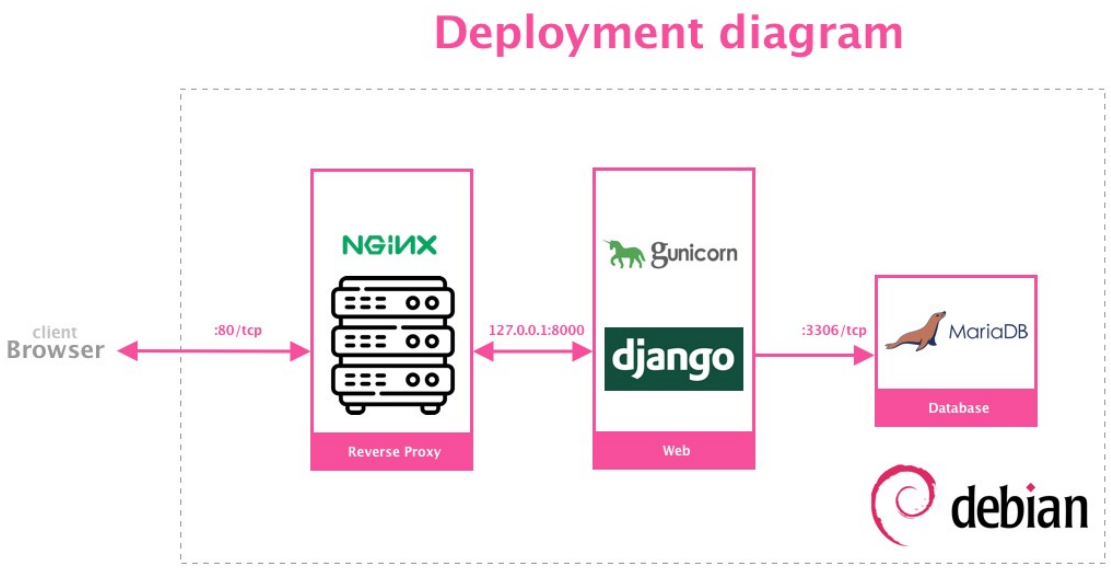


Figura 16: Diagrama de arquitectura de servicios de despliegue

2.10 Frontend

Como la adopción de diferentes tecnologías no debe entorpecer el futuro desarrollo la aplicación (más allá del alcance de este proyecto), se ha decidido trabajar de forma modular. Donde cada módulo ha sido concebido considerando responsabilidades concretas, siguiendo el principio de *Single Responsability* de *Robert. C.* Martin [29].

Por consiguiente, se utilizará un uso extensivo de EcmaScript6 (en adelante ES6) o EcmaScript 2015 (en adelante ES5). Decisión que tendría consecuencias en la compatibilidad de la aplicación con navegadores no-modernos, que no tienen la capacidad interna de transpilar ES6 a ES5 o que simplemente no comprenden ES6. Lo que precisamente ha motivado la decisión de emplear Rollup*:* Un compilador que permite trabajar con múltiples módulos ES6 distribuidos en múltiples ficheros (*separation of concerns*) y transpilarlos a ES5. Todo ello elimina por completo la barrera de incompatibilidad mencionada.

Se ha de remarcar que la decisión tecnológica descrita en el párrafo anterior también supone un paso añadido en el proceso de desarrollo del *frontend*. Es un paso muy similar al caso de uso de *Typescript* en el navegador (que también precisa de un transpilador a ES5)*.* De modo que, en detalle, el proceso pasará por transpilar ES6 a ES5 mediante Rollup y el *Javascript* compilado resultante será el que se hipervincule en el HTML. De esta forma, se dispondrá de ES5 nativo pese a trabajar completamente de forma completamente modular con ES6.

Adicionalmente, considerando el requisito de modularidad, el *frontend* debe abordarse con un principio reactivo, siguiendo los patrones de *listeners* y *state*.

En esa línea, se abre la consideración de *frameworks* de *frontend* existentes que se pudieran adaptar al caso de uso de este proyecto, que cumplan perfectamente con la elección de arquitectura y que ya adoptan patrones reactivos y orientados a estados. Tal es el caso React y Svelte. En detalle:

- React:
    - Es mucho más pesado.
    - Puede suponer un problema añadido en el aspecto de las compatibilidades y soporte a navegadores no-modernos.
- Svelte:
    - Se adapta mejor por su peso.

- Sin embargo, añade un riesgo de complejidad de integración con los editores y un acoplamiento a una tecnología menos flexible (para este proyecto), que puede resultar perjudicial. Especialmente, cuando parte del desarrollo del *frontend* se considerará temporal.

- En general (sobre cualquier otra alternativa de *framework* reactivo) se debe considerar que:

  - **Cambian el patrón de desarrollo en general:** Ya no sería un patrón MVC sino FLUX o similar. Lo cual no resulta conveniente por la adopción previa de *DJANGO*.

  - **Se orientaría todo a componentes:** Lo cual no corresponde con el caso de uso de DJANGO, porque se pierde todo lo que añade en peso la elección de *backend* y las ventajas que tiene ese framework. Por ejemplo, no tendría mucho sentido su *template engine*.

En consecuencia, en lugar de adoptar directamente un *framework* moderno que siga este principio, en este proyecto se ha decidido crearlos a mano, de forma sencilla, para poder adaptar, flexibilizar y evitar una complejidad adicional en el resultado.

A pesar de todo, tal decisión no implica que no adoptar un *framework* de *frontend* en este momento sea la mejor decisión. Simplemente se expresa la prioridad de la elección de *backend* porque existen unas ventajas de desarrollo que superan las ventajas de tener un mejor *framework* de *frontend*.

Finalmente, se usará Javascript puro en ES6 y se adoptará *jQuery*, considerando que siempre se podrá eliminar esa dependencia más adelante. El motivo de su adopción es el soporte a navegadores no-modernos. Considerando que ciertamente con *jQuery* se puede construir aplicaciones que bien podrían añadir cierta ineficiencia (y lentitud) por la manipulación directa que realiza del DOM. Del mismo modo que también se caracteriza por la producción de código espagueti diverso y fluido. Sin embargo, *jQuery* es una buena opción para un prototipo. Y si, además, se trabaja manualmente con un patrón reactivo y basado en estados, se produce un menor acoplamiento a esta tecnología. Lo cual permitirá eliminarla más adelante. Aunque siempre aceptando que esta elección en este proyecto inevitablemente añade un grado relativo de ineficiencia, aunque pueda considerarse infinitesimal.

2.11 Prototipos

El prototipo visual se ha creado con Sketch, una aplicación que permite la creación de prototipos navegables. Sin embargo, el resultado de Sketch no es incrustable en un PDF. De modo que, para disponer de una presentación de pantallas (visible) en este documento, se presentará primero un mapa con cada una de las pantallas, lo cual no sustituye pero sí ofrece una intuición del flujo de la navegación de las mismas.

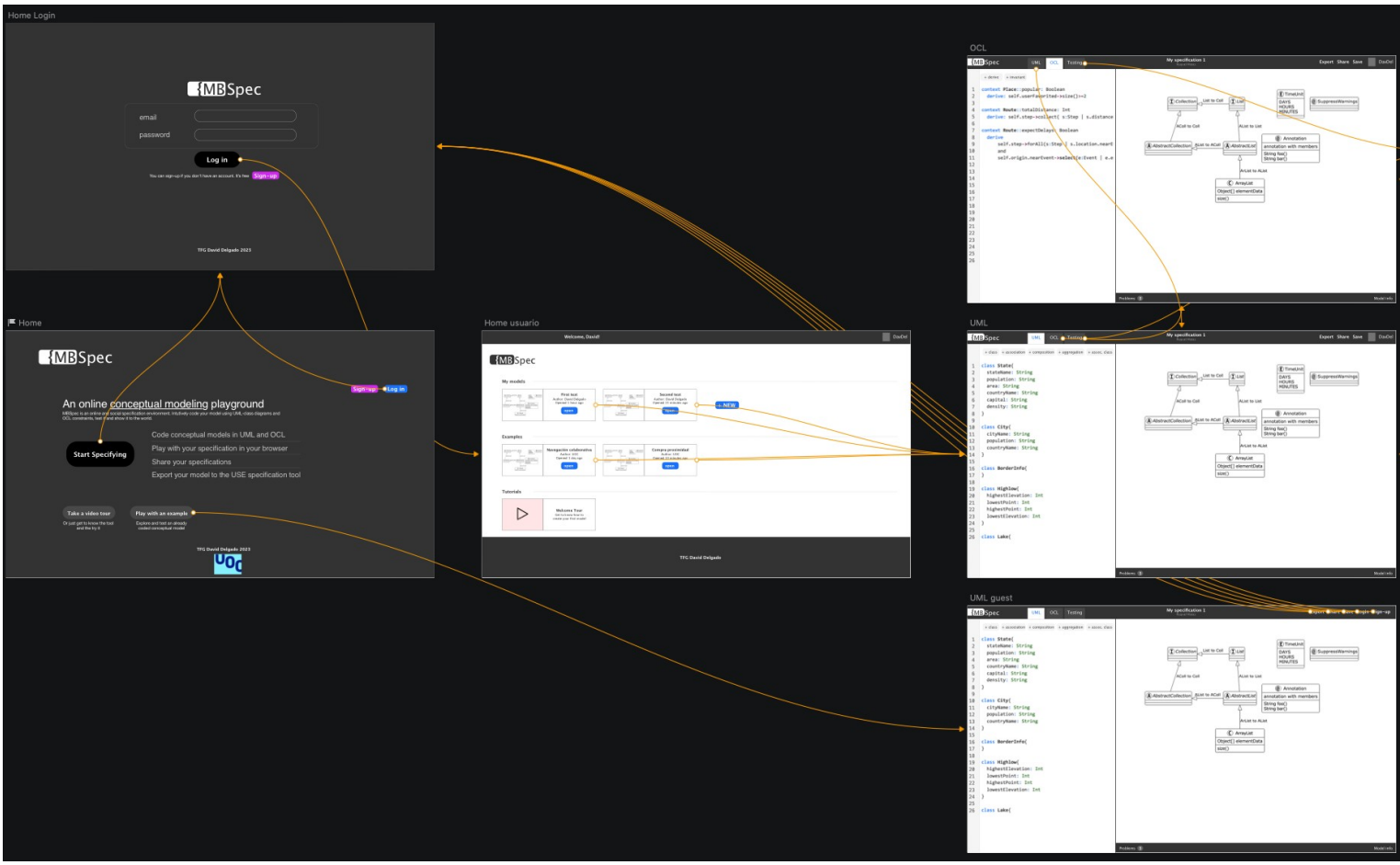


Figura 17: Muestra de pantallas y la versión del prototipo navegable con la aplicación Sketch

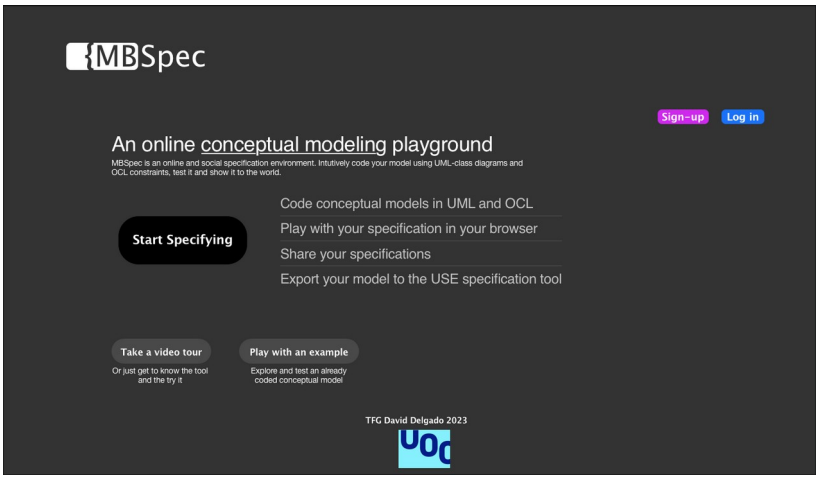


Figura 18: Prototipo de pantalla inicial de la aplicación

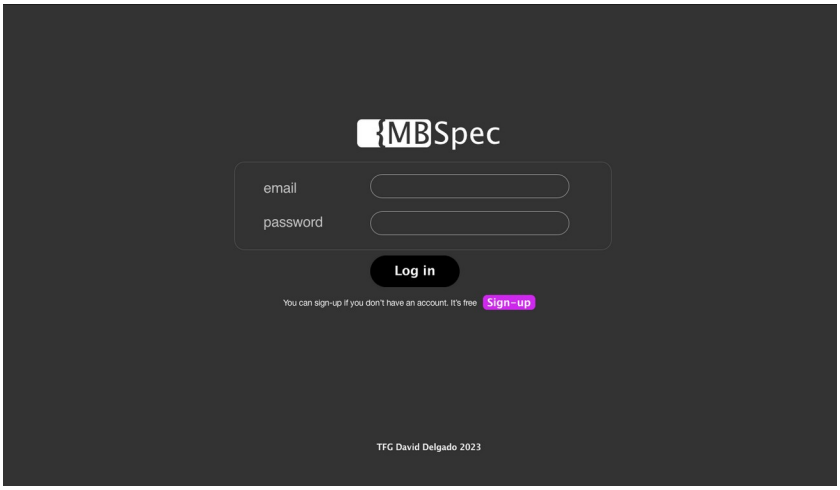


Figura 19: Prototipo de la pantalla de login

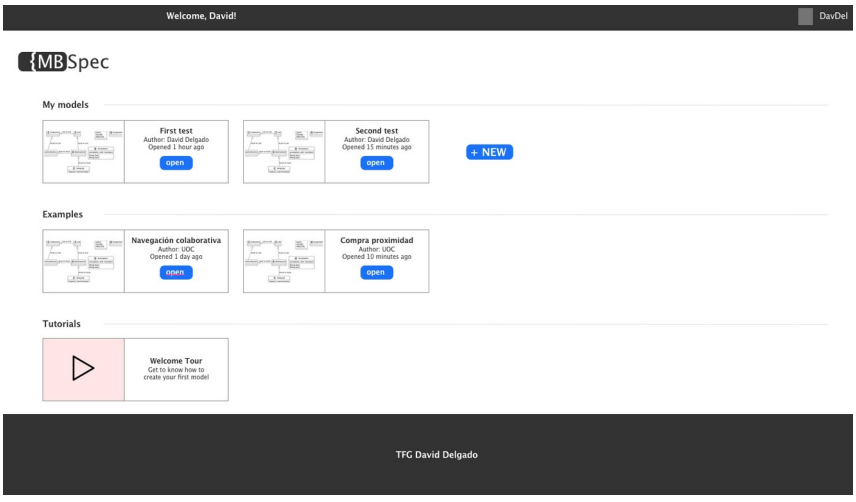


Figura 20: Prototipo del dashboard

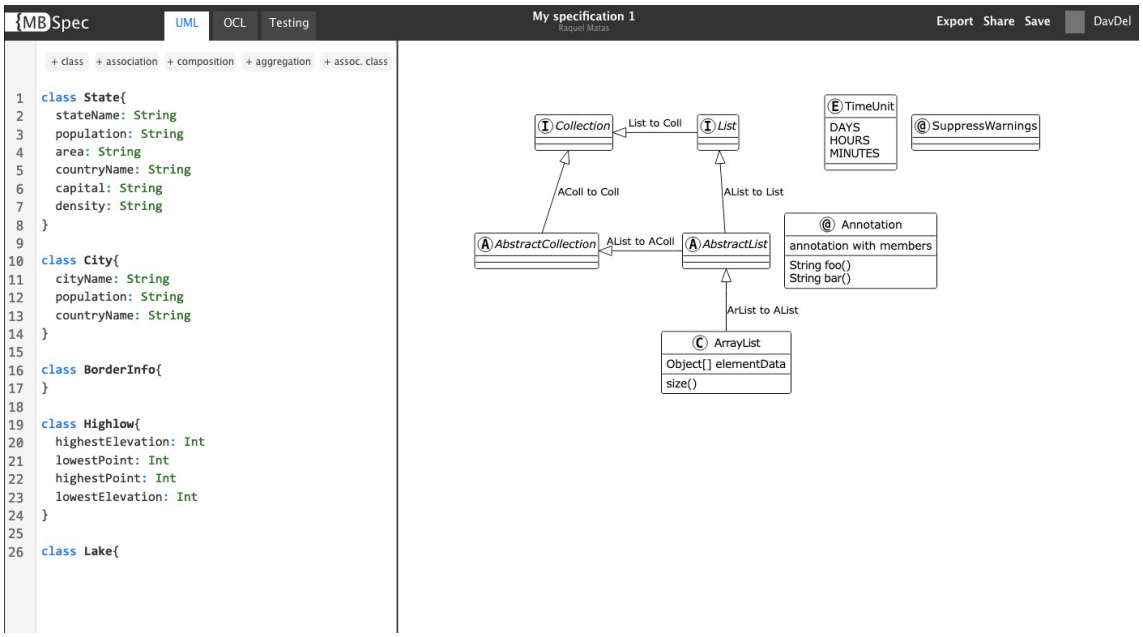


Figura 21: Prototipo de editor y visualizador de diagrama. *Nota:* No existe correspondencia entre el código y el diagrama.

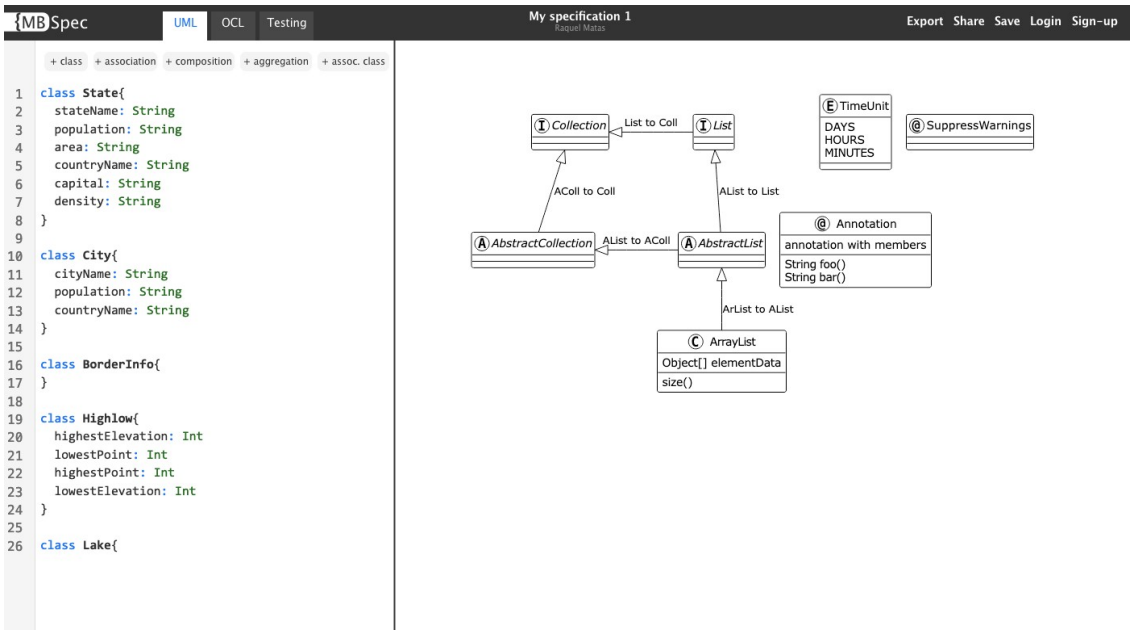


Figura 22: En esta última pantalla se puede apreciar cómo sería la vista aproximada de un usuario no registrado. Quien, al hacer click en cualquiera de las opciones de Exportar, Guardar o Compartir, tendrá que registrarse o iniciar sesión

2.12 Story map

Se muestra a continuación un *story map* de la aplicación, que sirve como detalle de las 2 *releases* pensadas en llevar a cabo para completar la aplicación, las cuales estarán divididas en las iteraciones y la velocidad por historia, aún por definir y no calculada.

Figura 23: Story Map inicial del proyecto

2.13 Resumen de herramientas

Con el propósito de describir el desarrollo, se enumeran aquí las tecnologías en las que se basa este proyecto:

- Python3.9, Anaconda, DJANGO, DJANGO-rest-framework.
- Bootstrap 5 (como módulo ES6).
- jQuery (como módulo ES6).
- Lezer: Compilador incremental y tolerante a errores de la gramática de los diferentes lenguajes empleados y para el análisis de la sintaxis, en virtud de los requerimientos del proyecto.
- CodeMirror: Editor web que utiliza los paquetes de lenguajes preparados con Lezer.
- Rollup para hacer el bundle de toda la aplicación en Javascript que ejecutará el usuario en su navegador. Por lo tanto: Transpila ES6 a ES5. Y así un navegador moderno no tendrá que perder tiempo en transpilar al cargar y un navegador no moderno tendrá soporte.
- hpcc-js y Graphviz para la generación de los diagramas de clase y de objeto en SVG mediante el lenguaje DOT en javascript y el navegador.

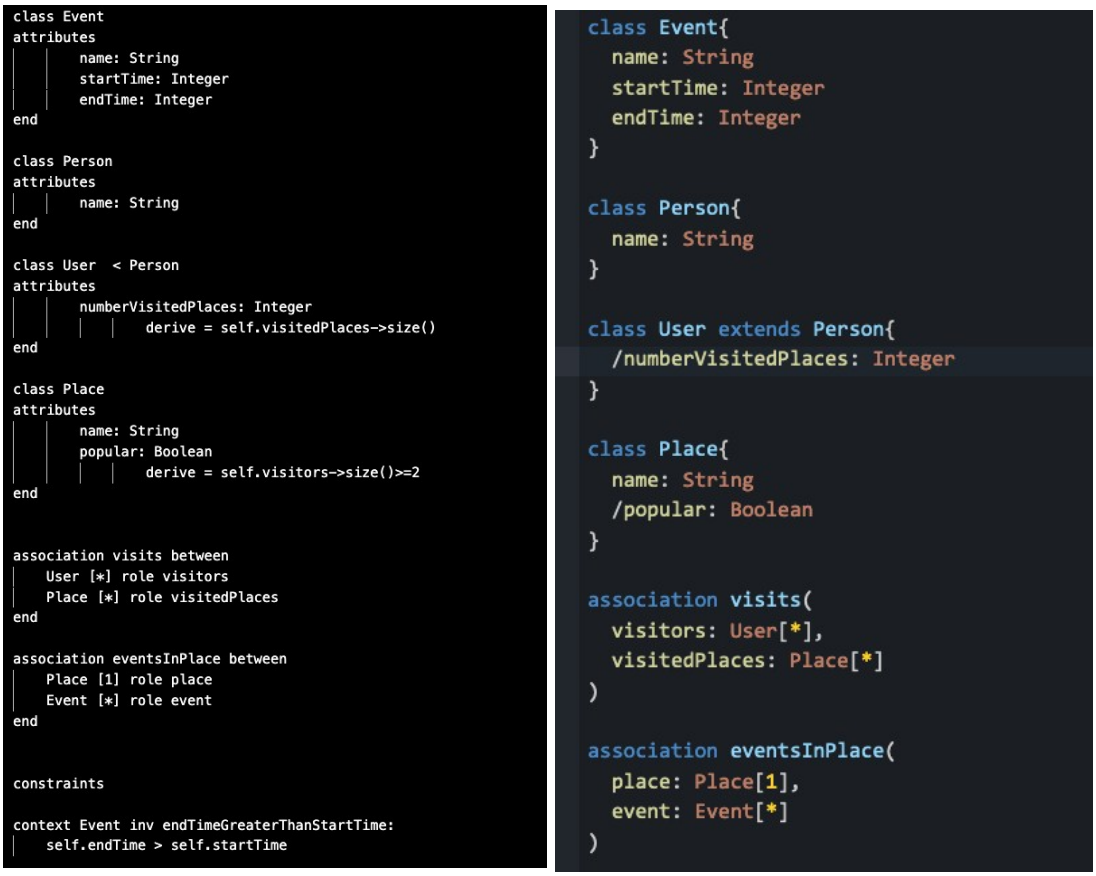
3. Desarrollo

3.1 DSL

Se han definido 3 lenguajes específicos para este proyecto. Uno de estos lenguajes es el OCL, que no se trata de un nuevo lenguaje y que por lo tanto no se discutirá como tal sino que se especificará su alcance. Los otros 2 son: El lenguaje de modelado y el de instanciación.

En relación al lenguaje de modelado, el mismo difiere considerablemente del lenguaje empleado con el mismo propósito en USE-OCL.

Para esta definición, es conveniente tener una vista en paralelo de ambos lenguajes



```
class Event
attributes
    name: String
    startTime: Integer
    endTime: Integer
end

class Person
attributes
    name: String
end

class User  < Person
attributes
    numberVisitedPlaces: Integer
        derive = self.visitedPlaces->size()
end

class Place
attributes
    name: String
    popular: Boolean
        derive = self.visitors->size()>=2
end


association visits between
    User [*] role visitors
    Place [*] role visitedPlaces
end

association eventsInPlace between
    Place [1] role place
    Event [*] role event
end


constraints

context Event inv endTimeGreaterThanStartTime:
    self.endTime > self.startTime
```

```
class Event{
  name: String
  startTime: Integer
  endTime: Integer
}

class Person{
  name: String
}

class User extends Person{
  /numberVisitedPlaces: Integer
}

class Place{
  name: String
  /popular: Boolean
}

association visits(
  visitors: User[*],
  visitedPlaces: Place[*]
)

association eventsInPlace(
  place: Place[1],
  event: Event[*]
)
```

Figura 24: Muestra del lenguaje de USE (izquierda) y de MBSPEC (derecha)

Diferencias del lenguaje de modelado de MBSPEC (este proyecto) con USE-OCL:

- La principal diferencia es que el lenguaje creado para MBSPEC es muy similar a Java. Por lo que puede resultar más familiar a un nuevo usuario (como un estudiante).

- Por eso utiliza también llaves en las clases en lugar de palabras como "end", "attributes", "between", etc.

- MBSPEC utiliza "extends" en lugar de "<", caso de USE.

- Luego en MBSPEC se utiliza implements, lo cual no está soportado por USE.

- Aunque en la definición de las asociaciones, se observa el uso de paréntesis con el propósito de identificar a los participantes como duplas, triplas, etc. Según corresponda.

- En USE-OCL se aprecia que directamente se definen los atributos derivados, incrustando en una gramática híbrida código OCL. Y en MBSPEC se utiliza solo OCL, como se verá más adelante.

- Pero también en las mismas asociaciones hay una diferencia en relación a la creación de clases asociativas. Las cuales, en el caso de USE-OCL, estas se deben crear como un nuevo tipo de estructura (con el token "associationclass"). En cambio, en MBSPEC se crean en el interior de una asociación para no crear otra estructura y tratar la asociación como un solo conjunto que incluya los participantes, roles, cardinalidades y la propia clase asociativa vinculada, si la hubiere. Dando una mayor importancia a la asociación y facilitando la comprensión de su instanciación.

El último punto de las diferencias, se puede apreciar mejor en los siguientes bloques de código:

Figura 25: Clase asociativa en USE (izquierda) y en MBSPEC (derecha)

Por otra parte, la instanciación es también un lenguaje muy diferente entre MBSPEC y USE. Principalmente, se aprecia que el lenguaje de instanciación de MBSPEC va completamente acorde con el GPL (Java), por lo que resulta más intuitivo. Las diferencias son más visibles si comparamos exactamente el mismo código en ambos lenguajes:

```
!new Place('house')
!new Place('office')
!house.name := 'My nice house'
!new User('onePerson')
!new User('anotherPerson')
!new User('david')
!insert(anotherPerson, house) into visits
!insert(david, house) into visits
!insert(anotherPerson, office) into visits
!insert(anotherPerson, house) into visits
```

```
house = new Place()
office = new Place()
house.name = "My nice house"

onePerson = new User()
anotherPerson = new User()
david = new User()

new visits (anotherPerson,house)
new visits (david,house)
new visits (anotherPerson,office)
new visits (anotherPerson,house)
```

Figura 26: Clase asociativa en USE (izquierda) y en MBSPEC (derecha)

También se puede apreciar las ventajas de trabajar con un lenguaje especifico para el modelado de clases en lugar de utilizar plantUML o DOT, lo cuales son lenguajes de diagramación propiamente (aunque realmente plantUML sea de más alto nivel). En esta línea, sería conveniente comparar plantUML con MBSPEC (figura 27).

```
@startuml

class UnaClase{
atributo: Integer
String atributoNuevo
}

class OtraClase{
atributo: Integer
}

UnaClase <|-- OtraClase

@enduml
```

```
class UnaClase{
  atributo: Integer
  atributoNuevo: String
}

class OtraClase extends UnaClase{
  atributo: Integer
}
```

Figura 27: Ambos códigos representan el mismo diagrama. Extensión de una clase en plantUML (izquierda) y en MBSPEC (derecha)

Se puede observar que en plantUML, se ha de crear la arista de la herencia (*extends*) aparte y que en MBSPEC (este proyecto) simplemente se utiliza "extends". Por consiguiente, lo mismo sucedería con "implements". Entonces, es posible observar que MBSPEC es de más alto nivel que plantUML.

Además (en la figura anterior) se ha intercambiado los tipos en la definición de una clase, lo cual para plantUML no representa ningún problema. Al contrario, en MBSPEC los atributos y los tipos sí tienen una sintaxis concreta. De otro modo, el código no pasa el diagnóstico y la aplicación no representa el diagrama.

En esta comparación con plantUML se evidencia la necesidad de utilizar un DSL en lugar de adoptar un lenguaje de diagramación específico. El lenguaje de modelado creado está muchísimo más próximo al usuario que crea una clase. Y se puede comprender claramente que el DSL colabora tanto en el análisis semántico como en la capacidad de expresión.

Lo mismo sucede con las clases asociativas, las cuales están constituidas como una unidad en MBSPEC y en cambio tanto en DOT como en plantUML son aristas, cuya representación se debe parametrizar para expresar cada vez un tipo de asociación o una extensión, etc.

Para concluir este apartado sobre lenguajes, conviene señalar la especificación de OCL en el entorno de MBSPEC. La cual se realiza expresando contextos. Sin embargo, en este proyecto se ha trabajado con un subconjunto de expresiones válidas de OCL en este sentido, primero porque USE-OCL actúa como límite y luego también por la limitación de tiempo. En definitiva, en MBSPEC no se soporta, por ejemplo, la declaración de operaciones, el uso de "body", etc. Solo se soportan exclusivamente 2 tipos de expresiones:

- Declaración de restricciones invariantes.
- Definición de atributos derivados.

```
context Event inv endTimeGreaterThanStartTime:
    self.endTime > self.startTime

context Place::popular: Boolean
    derive: self.visitors->size()>=2

context User::numberVisitedPlaces: Integer
    derive: self.visitedPlaces->size()
```

Figura 28: Se muestran diferentes expresiones en OCL válidas en MBSPEC

3.2 Características del resultado

Tratándose de un trabajo muy denso, es sería adecuado exponer algunas características destacables de la solución en la memoria

3.2.1 Herencia

En las especializaciones (herencia o extensión) de clase, las clases hijas deben tener una definición de atributos que incluyan aquellos atributos heredados de las clases padres. Por ejemplo (véase la figura siguiente):

Figura 29: La clase User extiende Person. Luego, Person define el atributo "name" (izquierda). Seguidamente, se puede instanciar User y definir david.name="David" (centro) y ver su resultado en el diagrama de objetos (derecha)

La primera nota va de acuerdo a la capacidad de poder instanciar un usuario con la propiedad heredada "name". Y la discusión de este punto, gira en torno a cómo comprueba y comprende la aplicación que "name" forma parte de User.

¿Cómo funciona esto en MBSPEC?

La instanciación del modelo semántico de MBSPEC se realiza con "copias" de los atributos heredados. Es decir (siguiendo con el ejemplo de la figura anterior), que la clase User sería una clase que incluye un atributo derivado y el atributo (heredado) "name".

Lo anterior tiene una importancia considerable cuando se realiza el análisis semántico del modelo. Ya que no se analiza cada clase y su herencia en el tiempo de diagnóstico (el momento en que se evalúa el código en el editor). En su lugar, en esta solución se ha preferido aumentar el consumo de espacio en memoria, para evitar la ineficiencia de resolución o comprobación de definiciones hereditarias por cada evaluación de diagnóstico del código. En

consecuencia, **se examina siempre una instancia del meta-modelo con los atributos heredados ya incluidos en copia**. Es decir, que una instancia de User incluye "name" como si así se hubiese codificado.

La mencionada transformación se puede visualizar como una refactorización del mismo modelo y que por lo tanto se trataría de una transformación endógena [24]. Lo cual además contribuye a explicar el motivo por el cual el modelo semántico de clases de instanciación propuestos en este proyecto no contemple la herencia en su diseño.

Y que por la descripción de situaciones como la herencia, debe resultar más fácil de comprender porqué se ha optado por un modelo semántico y porqué el modelo semántico es la representación intermedia de esta aplicación, en relación a la definición del intérprete.

3.2.2 Gramática

Lezer sigue un patrón *LL(K) top down parser* igual que ANTLR [30]. Y esto sucede así, aunque su tokenizer no tenga lookahead, debido a que se trata de un parser *error tolerant* e incremental. Entonces, ¿cómo puede ser *k>1*? Es decir, ¿por qué es un parser LL(k) _donde k debe ser mayor que uno_ y no un parser LL(1)? **¿Qué supone esto en el diseño de nuevos lenguajes?**

En la definición de la gramática, según Terence Parr [27], es recomendable definir todos los keywords como tokens, porque de esa forma se evita sobrecargar al parser con estas tareas. Todo ello significa que el tokenizer se encarga de los keywords.

Sin embargo, en Lezer esto no resulta posible: Hay que recordar que Lezer es "*error tolerant*", por lo tanto se produce una sobrecarga del tokenizer cuando se utiliza tokens que tengan patrones similares, ya que su tokenizer no tiene lookahead. Entonces, si se definen keywords con patrones similares, se producen overlappings (o solapamiento) que obligan a la gestión de precedencia de los tokens (o cual resulta inefectivo).

¿Por qué casi siempre se va a producir un solapamiento? Porque los keywords la mayoría de veces chocarían con el token ID (o identifier), el cual paso a ilustrar:

> Normalmente, es necesario utilizar un token definido como ID, con el cual se expresa cualquier cadena alfabética. Por ejemplo, se puede usar una regla gramática (MiClase = {"class" ID}) para expresar los nombres de las clases como "class MyClass", donde "MyClass" es el token ID.

Entonces, la duda que se plantea (el solapamiento) sucede cuando se debe diferenciar entre "class" y el nombre de clase "classmate" cuando no hay *lookahead* y no se puede ver más allá. Aunque "mate" siga despues de "class" y así se diferencien ambas palabras (*class* y *classmate*). Por lo cual, se hace necesario introducir una regla de precedencia del keyword "class" sobre ID. Que ya se ha adelantado que es una solución ineficiente.

Volviendo al problema, a diferencia de ANTLR (por poner un ejemplo), Lezer utiliza el parser para definir los keywords. Requiriendo que tales keywords se definan como reglas que expresen excepciones del token ID (o identifier).

Así, dicho en otras palabras, en la definición de los tokens no sería correcto definir el keyword "class" como token, como normalmente haríamos en un parser con *lookahead*. Porque se produciría en Lezer un solapamiento con un token identifier ([a-z-A-Z]), tal y como se ha visto. En consecuencia, se tiene que utilizar una *@specialize token rule* [31]. Evitando el solapamiento y evitando aumentar la complejidad del autómata.

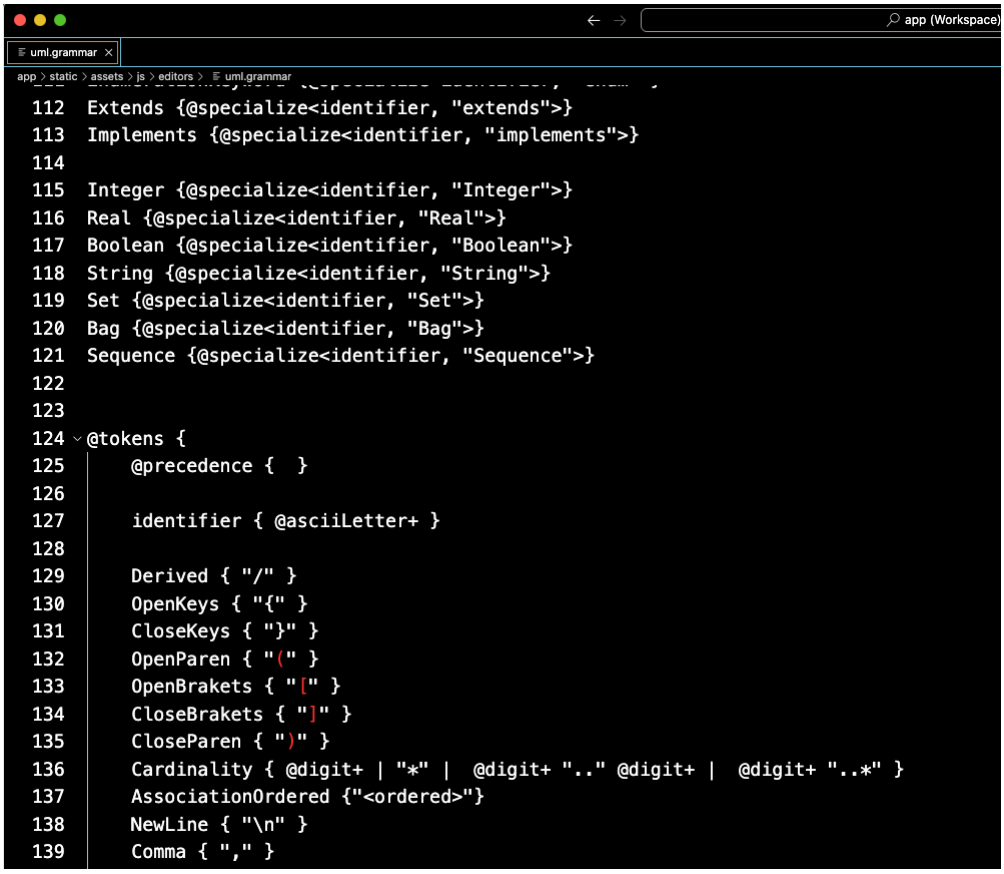


Figura 30: Gramática de UML donde se muestra el ID, definición de tokens y el uso de @specialize

En la figura anterior (figura 30), se muestra como "Integer" está definido con @specialize y el ID (identifier) como un token. Solución que no produce ningún solapamiento entre "Integer" e "Integrar", por ejemplo.

3.2.5 Análisis semántico

Es necesario realizar una descripción del alcance del análisis, ya que **el análisis semántico realizado por la aplicación es incompleto**. No obstante, resulta una tarea trivial completar el mismo en la mayoría de aspectos restantes, pero por razones obvias de tiempo, se ha optado por completar el prototipo de forma extensiva, en lugar de cerrar todos los detalles.

En el diagrama de clases:

- El nombre de las clases es único.
- El nombre de los atributos dentro de una misma clase es único.
- Los tipos válidos son los soportados por las especificaciones UML-OCL.
- Si se utiliza una clase de usuario, un Enum como tipo de dato o un <<data type>>, el mismo debe estar definido en el modelo.
- Las asociaciones contienen las partes necesarias. Su nombre debe ser único en todo el modelo.
- Los participantes de las asociaciones deben ser clases definidas en el modelo y deben especificarse con nombres de roles.
- El uso de tipos de colección (Sequence, Bag...) debe contener un tipo primitivo (Integer, String, Real...)

En el diagrama de objetos:

- No se puede instanciar clases inexistentes.
- No se puede definir valores de atributos inexistentes.
- No se puede instanciar asociaciones con nombres inexistentes.
- No se puede especificar valores de tipo diferentes a su definición como atributo de clase en el modelo.

En OCL:

- No se puede definir otra cosa que no sean restricciones invariantes y atributos derivados, de los cuales se debe respetar su estructura.
- No se puede hacer menciones de contextos de clases y propiedades que no existan previamente en el modelo de clases.

- No se puede especificar tipos de retorno de valores diferentes a la definición hecha en el modelo. Aunque no se comprueba la evaluación de la expresión en OCL directamente, ya que todo ello se lleva a cabo en USE-OCL.
- Se evalúan las expresiones de OCL de acuerdo a la gramática de USE-OCL para el lenguaje.

3.2.6 Editor

La selección del parser (en el punto 2.2 de esta memoria) permitió llegar a un muy buen resultado con el editor. Todo ello es debido a la identificación de las características que debía cumplir el parser. Sin embargo, tales características podían haber resultado un tanto abstractas cuando se enumeraron y describieron en esta memoria. Y, es por ello que quizá sea muy conveniente señalar cómo identificarlas. Para ello, se utilizará un ejemplo con las siguientes figuras 38 y 39.

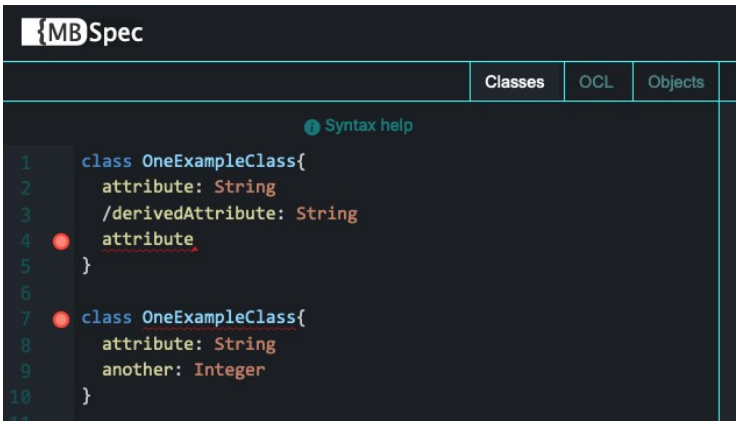


Figura 38: Código de modelado que muestra cómo pueden ocurrir 2 errores

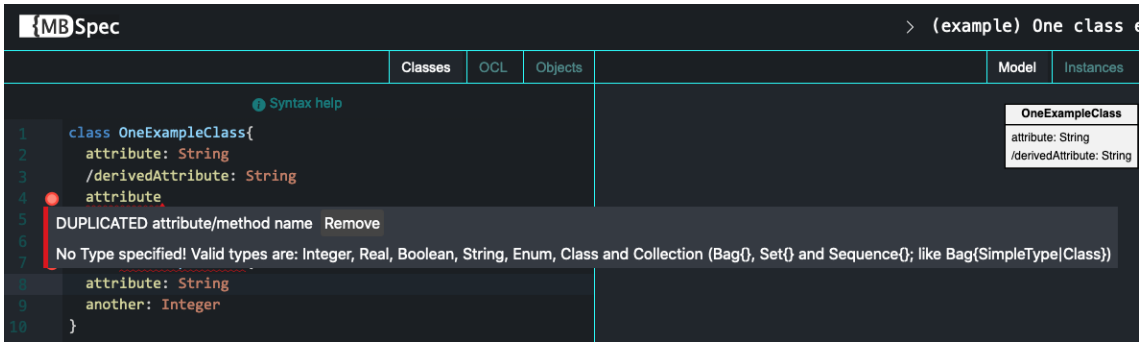


Figura 39: Conjunto de imágenes que muestra que cada error está identificado

Gracias a las imágenes, el concepto abstracto *error tolerant* se puede identificar claramente en el producto final. Por ejemplo, en la figura 38: Se

puede ver que la segunda palabra "attribute" (que no tiene un tipo definido y que no cumple con específica de los atributos) es reconocida por el parser como un atributo de la clase. Visualmente, solo hay que fijarse en que el color de la palabra "attribute" es amarillo, en que luego se ha coloreado todo correctamente a partir del error y en que es posible identificar un error semántico (atributo con nombre repetido) y sintáctico (falta del tipo del atributo) en la figura 39. Entonces, en cuanto se produce el error sintáctico, el parser no se detiene y genera un *Parse Tree* que permite posteriormente realizar el diagnóstico. No se ha interrumpido la coloración, ya que el parser ha seguido reconociendo otras palabras como parte del lenguaje. Aún más, es posible identificar un nuevo error semántico (una clase con nombre repetido "OneExampleClass") en la figura 38.

En conclusión, este proyecto exhibe una característica que es atípica (y muy específica) y que además se reconoce concretamente porqué resulta esencial para este caso de uso.

3.3 Adapter de USE-OCL

Primero, sería importante presentar una vista general de la implementación:

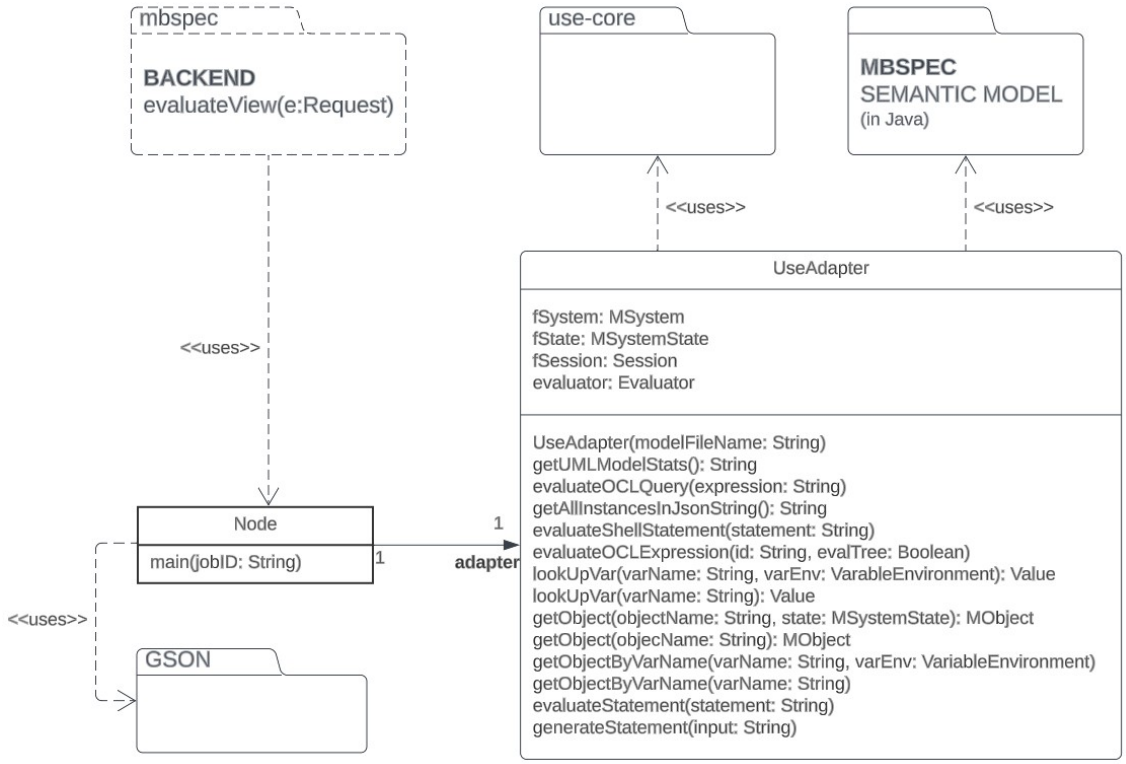


Figura 31: Solución adoptada para la modificación de USE-OCL

Básicamente, el desarrollo en Java se ha llevado a cabo pensando en disponer de un nodo que directamente produzca la evaluación partiendo de una serie de datos. Tal nodo sería consumido como servicio por el backend o se abriría al frontend (microservicio). Sin embargo, no se independizó el nodo para no aumentar la complejidad de la implementación del prototipo.

Igualmente, se puede evaluar cualquier expresión OCL (válida en USE-OCL) dado un modelo y una instanciación de objetos. De modo que estaría preparado para integrar *queries*. A pesar de esto, el trabajo realizado cumple el alcance exclusivo de validar el modelo y evaluar atributos derivados en función de la instanciación.

Finalmente es conveniente considerar cómo se trabaja con el Nodo de USE-OCL y su adaptador. Para ello se parte de la petición de evaluación lanzada desde el *frontend* de forma transparente y por cada modificación del modelo del código de OCL que realice el usuario. La petición de evaluación es procesada por una vista específica en el backend. A partir de aquí comienza el camino de la evaluación, que se puede ver más claramente en el siguiente diagrama:

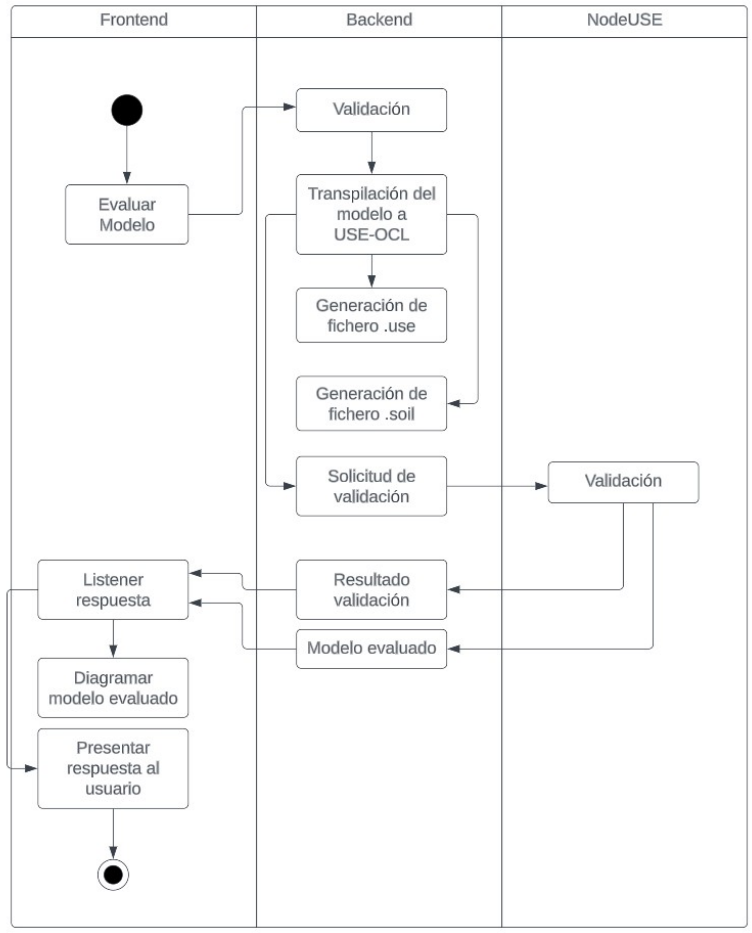


Figura 32: Gramática de UML donde se muestra el ID, definición de tokens y el uso de @specialize

Como se puede apreciar en el diagrama anterior, se generan 2 ficheros: un fichero .use y un fichero .soil. Estos ficheros se generan con un proceso de mapeo y plantillas, ya que la transpilación resulta trivial. Luego, se identifica el número de trabajo y se utiliza el nodo de USE para realizar la evaluación. El nodo accede directamente a los ficheros generados y produce un resultado en JSON con la evaluación del modelo. Este JSON está generado con el modelo semántico que ya se ilustró en el capítulo 2 de esta memoria. Aunque en Java simplemente se ha incorporado un subconjunto del mismo, como puede verse a continuación en la siguiente figura:

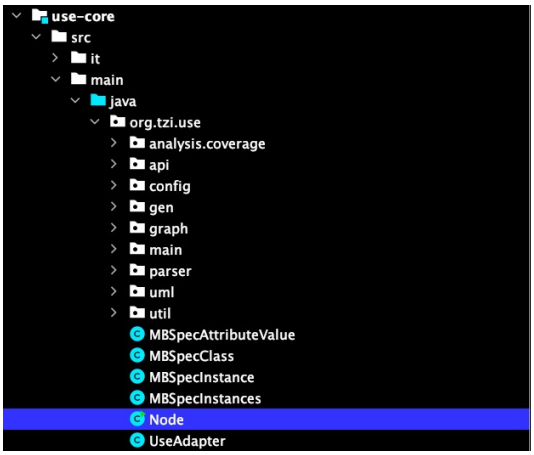


Figura 33: Árbol de directorios donde se puede observar las clases que componen el modelo semántico de MBSPEC generado en Java

3.4 Backend

Para definir esta implementación, se puede trabajar con un diagrama bastante simplificado sobre cómo trabajan las partes que componen el backend:

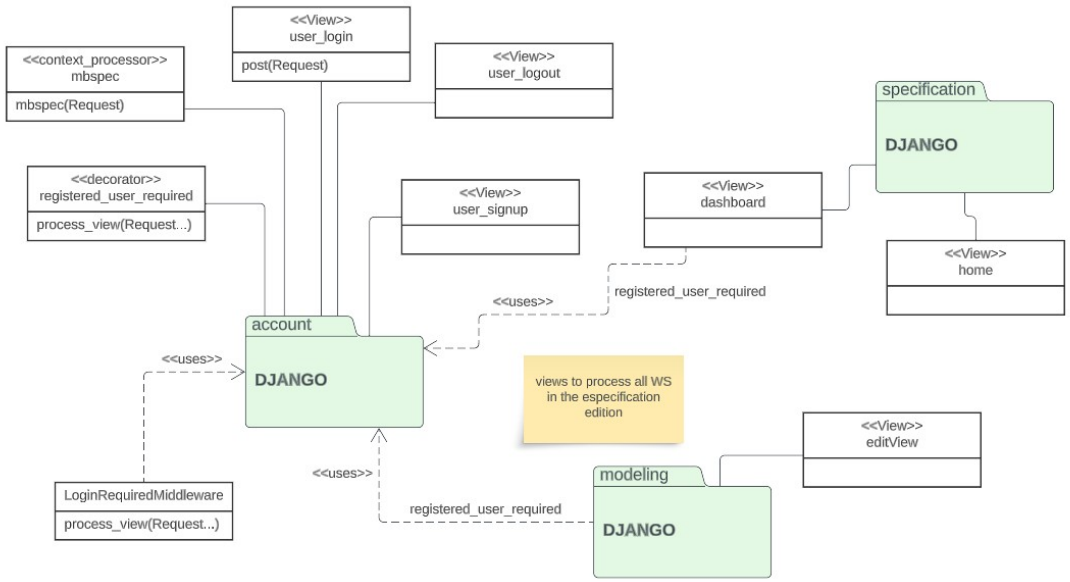


Figura 34: Diagrama simplificado de la implementación en DJANGO

Se puede observar 3 aplicaciones específicas (utilizando el argot de DJANGO): Una que constituye el pilar de los usuarios dentro del servicio (account). Otra donde se visualizan las distintas pantallas funcionales de la aplicación. Y, finalmente, la aplicación que sirve para la edición de las especificaciones, que contiene los *web services* más relacionados con la validación de la especificación, así como modificar atributos, transformar el modelo para descargar en USE, etc.

El decorator registered_user_required es el que se añade a todo los Views para producir una vista con requerimiento de usuario logueado específico. Luego, también está el LoginRequiredMiddleware que aplica la obligación de usuario logueado de forma general sobre aplicaciones, métodos de acceso, etc.

Finalmente se puede considerar el siguiente diagrama, utilizado para la validación de especificaciones.

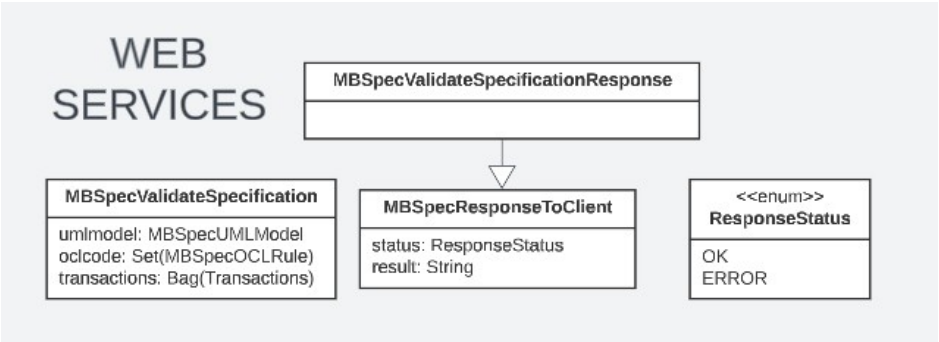


Figura 35: Diagrama de clases del modelo de servicios web

Igualmente que se hizo en Java y en Javascript, en DJANGO (python) también se ha tenido que replicar el modelo semántico, aunque añadiendo la clase de

respuesta del API que responde a un *request* realizado con el modelo de la figura anterior.

```python
class MBSpecApiResponse:
    def __init__(self, status, result):
        self.status = status
        self.result = result

class MBSpecAttribute:
    def __init__(self, attribute, type):
        self.attribute = attribute
        self.type = type

class MBSpecAttributeValue(MBSpecAttribute):
    def __init__(self, attribute, type, value):
        super(attribute, type)
        self.value = value

class MBSpecOperationParameter:
    def __init__(self, parameter, type):
        self.parameter = parameter
        self.type = type

class MBSpecOperation:
    def __init__(self, operation, type, parameters):
        self.operation = operation
        self.type = type
        self.parameters = parameters
```

Figura 36: Muestra de clases del modelo semántico y de las respuesta de la API en general

Es conveniente señalar que los *web services* trabajan con rest_framework y que por lo tanto se debe trabajar con Serializers para definir los tipos de datos esperados, para que sea posible una validación de los datos antes de la instanciación de los objetos del *request* como objetos nativos.

```python
class MBSpecAttributeSerializer(serializers.Serializer):
    attribute = serializers.CharField(max_length=200)
    type = serializers.CharField(max_length=200)

class MBSpecAttributeValueSerializer(MBSpecAttributeSerializer):
    value = serializers.CharField(max_length=200, allow_blank=True)

class MBSpecOperationParameterSerializer(serializers.Serializer):
    parameter = serializers.CharField(max_length=200)
    type = serializers.CharField(max_length=200)

class MBSpecOperationSerializer(serializers.Serializer):
    operation = serializers.CharField(max_length=200)
    type = serializers.CharField(max_length=200)
    parameters = serializers.ListField(
        child=MBSpecOperationParameterSerializer(),
        allow_empty=True
    )

class MBSpecClassSerializer(serializers.Serializer):
    name = serializers.CharField(max_length=200)
```

Figura 37: Muestra de Serializers de los datos del modelo semántico

3.5 Pruebas

Las pruebas del editor consistían en ficheros con el código necesario para reproducir un resultado esperado muy concreto. Así, cada característica, por ejemplo, en el caso del modelado, se observaba a través del resultado gráfico del diagrama de clases. En esa línea específica se probó lo siguiente:

- Se puede generar clases diferentes. Cada clase contiene los atributos definidos en el código así como sus tipos. Es posible informar de operaciones también, así como de sus parámetros.
- Se puede crear asociaciones. Las asociaciones tienen los participantes señalados y se pueden crear asociaciones no solo binarias sino que también ternarias, cuaternarias, etc.
- Se puede crear clases asociativas y se puede especificar atributos, operaciones y relaciones sobre este tipo de clases.
- Se puede especificar herencia y hacer implementaciones.

De la misma forma, a través de ficheros de textos, se realizaron las pruebas de los diagramas de objeto. Tanto en el caso de la instanciación de objetos como en la especificación en OCL. En ambos casos, se debe de ir generando un diagrama correcto para todos los casos observados con los diagramas de clase. Es decir que debe ser posible inicializar clases, asociaciones, clases asociativas, etc. Así como visualizar valores de atributos derivados o determinados errores forzados en USE-OCL.

La validación de OCL en sí mismo, consistía prácticamente en pruebas de integración de todos los lenguaje. En esta linea, la correcta evaluación de una especificación se realizó con 3 muestras de código, una para cada lenguaje. Por otra parte, hay una serie de ejemplos que contienen especificaciones completas que sirven para experimentar y se encuentran creados en la propia aplicación. Todos estos ejemplos, pueden servir para probar en funcionamiento del sistema al completo.

El sistema de pruebas ha sido exclusivamente unitario. Hay que considerar que no se ha invertido tiempo en un sistema de pruebas automático ya que todo ello implicaba experimentar sobre cómo probar con sistemas de pruebas en el desarrollo de lenguajes. Lo cual no resulta nada fácil de considerar, especialmente cuando también hay que probar otras funcionalidades del editor (*linting*, auto-completado...) difíciles de automatizar, ya que no es trivial observar esos comportamientos con Selenium, por ejemplo.

4. Conclusiones

Como se puede evidenciar en las diferentes entregas parciales de este proyecto, ya se anticipaba la dificultad de conseguir todos los objetivos. Aun así, la idea principal era la de obtener un prototipo completo en extensión pero no en detalle. De este modo, a pesar de la dificultad, se han cumplido todos los objetivos y con un alcance mayor, completando algunas tareas complejas que no se esperaban cumplir. Como sería el caso de la validación con USE-OCL y la evaluación de atributos derivados en el diagrama de objetos. Una tarea difícil y renunciable pero que muy destacable de este proyecto.

También es cierto que si no se hubiese centrado el trabajo en la especificación sino en el modelado, hubiese sido posible trabajar en otros aspectos interesantes, desde un punto de vista personal. Como lo sería la transformación de modelos con clases asociativas a modelos con las clases asociativas reificadas. También pudo haber estado centrado el trabajo más en la edición de código y completar aspectos de *linting*. Por ejemplo: Al instanciar una asociación, los participantes se deben informar en un orden específico y se podría fácilmente comprobar que un participante informado no es de la clase que corresponde e informar precisamente este hecho. Como este hay muchos detalles similares que son triviales pero que no se pueden abordar con un criterio extensivo de desarrollo.

La gestión del proyecto ha sido correcta. La metodología seguida también ha sido acertada. Se puede examinar que el producto es muy extenso y se cubren todos los aspectos para poder disponer de un prototipo funcional. No se consideran aspectos visuales, pero está claro que se ha realizado un trabajo correcto en relación a la interfaz de usuario. La planificación se ha llevado a cabo dentro de los hitos. Así, cualquier tarea restante que quedara por finalizar solo tenía cabida si se cumplía el límite o simplemente se modificaba el alcance de la misma a los efectos de abordarla en tiempo. Trabajando estos aspectos cada fin de semana para la programación de la siguiente iteración, permitía tener la visión del proyecto más clara. Especialmente porque se dedicó tiempo al *story-map* y las *releases*, que desvelaba desde antes de programar una sola línea de código, qué aspectos se podrían descartar para cumplir el recorrido completo de la historia.

Adicionalmente, conforme se fue comprendiendo la complejidad de ciertos aspectos desconocidos, se reaccionó cambiando de método (como por ejemplo, descartar utilizar el adapter de USE-OCL como microservicio y utilizar ficheros como parámetros) o renunciando a utilizar aspectos técnicos que se querían abordar (como por ejemplo, trabajar con un AST para abordar la evaluación de tipos de las operaciones en OCL). Algunos aspectos de

compilación, análisis semántico, interpretación, auto-completado con contextos (que requieren usar nuevamente el Parse Tree) se han tenido que dejar de lado. Sin embargo, pese a la complejidad de todo el proyecto en conjunto, se ha podido completar con éxito y se han cubierto todos los aspectos, lo cual es buen indicativo de una buena gestión.

Sin embargo, a pesar de lo dicho en el párrafo anterior, si se analiza el proyecto desde el punto de vista económico, no se puede engañar al lector de que efectivamente se ha invertido en el proyecto más horas que 300. Se trata de una situación que se conocía desde mucho antes de la planificación y se transmitió al tutor en términos de si se podía exceder esa cantidad de horas en total. Por eso sé que, si hay un mal resultado en la gestión del proyecto, este aspecto sería el punto de vista económico, el cual no es despreciable. Efectivamente, ha habido un esfuerzo muy grande para obtener un buen producto. Y quizá éste podría ser marcado como el defecto de la gestión. Sin extender los límites de la planificación, se ha trabajado dedicando más tiempo por jornada más específicamente en lo relacionado a los lenguajes, el parsing y los editores en el navegador.

Finalmente, cabe añadir que un *code playground* de especificaciones tiene muchísimas aplicaciones en la ingeniería del software o desde la perspectiva de la formación. Ciertamente, hay aspectos que se podrían cubrir con 1 mes más de trabajo, a nivel de *linting*. Pero hay otros aspectos que habría que abordar de lleno, más allá de completar, como por ejemplo:

- Disponer de un entorno de especificación colaborativo: En lugar de solo arrojar una copia de una especificación en UML a un amigo, sería interesante poder trabajar las especificaciones juntos. Algo que podría cubrirse fácilmente con GIT ya que se trata de código, pero ¿qué soluciones se podría plantear y cómo abordar las mismas en la interfaz dentro de un entorno de especificación?

- Seguridad de las operaciones: Hay soluciones inseguras, como por ejemplo la evaluación y la arquitectura del adapter del USE, que opera con ficheros. Pese a ello, se plantea en la memoria otra alternativa para tener un mejor sistema.

- El punto anterior, condiciona el registro de nuevos usuarios y el hecho de que esta característica no se deje finalmente abierta. Entonces, el sistema de usuarios debería completarse alcanzando un nivel de seguridad óptimo.

- Abordar el mantenimiento automático del servicio con tareas en el servidor (CRON) y Celery: Limpiar cada X tiempo los ficheros

intermedios SOIL de USE utilizados en las validaciones, limpieza de las sesiones en la base de datos (importante en DJANGO), etc.

- ¿Crear un intérprete de OCL en Javascript? Al menos con el criterio de OCL de USE. Sería una buena inversión crear USE-OCL en Javascript y poder usar el USE-BASE como librería.

- Despliegue de la aplicación: En un principio, se planteó la aplicación con un despliegue en Docker mediante docker-compose y la parametrización de múltiples servicios. Sin embargo, el *deployment* ha tenido una menor prioridad. Pero esta medida podría dotar a la aplicación de una instalación instantánea.

- Evaluación de invariantes: Obtener la misma información que USE sobre el estado de cumplimiento de variables.

- OCL queries: Disponer de la posibilidad de interrogar a la especificación y poder evaluar expresiones sin contexto que no estén vinculadas a atributos derivados ni restricciones. Como DQL para SQL.

- Evaluar el modelo por pasos (T0, T1, T3...), donde cada paso representa un evento de dominio o una colección de los mismos. Disponer de fotogramas que permitan observar la evolución del diagrama de objetos según pasa el tiempo (eventos de dominio).

- Por último y quizá más importante, disponer de tests automatizados de distintos aspectos. Deben abordar el aspecto unitario de cada bloque, como los lenguajes, la interfaz y el funcionamiento del editor, así como tests automáticos en relación al backend, etc. Seguramente integrando los mismos con Selenium.

5. Glosario

AST:	Abstract Syntax Tree. Un grafo específico mediante el cual se representa un lenguaje a más alto nivel que un Parse Tree.
DSL:	Domain Specific Language. Un lenguaje que cumple unos propósitos útiles y específicos en una aplicación.
DOT:	Gramática para definir diagramas en Graphviz
ES6	Javascript que cumple el stándard ECMAScript 2015. Que incluye soporte para módulos. No es nativo en todos los navegadores y habitualmente el navegador transpila a ES5.
ES5	Javascript que cumple el stándard ECMAScript 2009. Se trata del
GPL	General Purpose Languages. Como el caso de Java y Javscript.
ID	Hace referencia al identifier o identificador que habitualmente se expresa de forma alfabética
JS	Javascript. En este proyecto se utiliza JS para referirse a ES5 y ES6 transpilado a ES5.
Precedence	Regla que permite la priorización de tokens en un tokenizer
PT	Parse Tree. Un grafo que representa todos los tokens reconocidos por una gramática
WS	Web Service.

6. Bibliografía

[1] OMG Unified Modeling Language Specification, UML 2.5.1, Diciembre 2017. Visitada en Junio de 2023: https://www.omg.org/spec/UML/2.5.1/PDF

[2] UML-based Specification Environment (USE). Visitada en Junio de 2023: https://github.com/useocl/use

[3] CodePen: Online Code Editor and social development environment. Visitada en Junio de 2023: https://codepen.io/e

[4] JSFiddle: Code Playgraound, test your JS, CSS, HTML or CoffeeScript online. Visitada en Junio de 2023: https://jsfiddle.net/

[5] Object Management Group, OCL 2.4 Specification. Febrero 2014. Visitada en Junio de 2023: https://www.omg.org/spec/OCL/2.4

[6] Analog Clok in CodePen example. Visitada en Junio de 2023: https://codepen.io/trishachi/pen/ypdZgw

[7] Jeff Patton, *User Story Mapping.* O'Reilly, Sebastopol, 2014. Página 73.

[8] Mike Cohn, *User Stories Applied For Agile Software Development.* Addison-Wesley, Donnelley (Indiana) 2004.

[9] Vistual Studio Code for the Web. Visitada en Junio de 2023: https://code.visualstudio.com/docs/editor/vscode-web

[10] Monaco Editor. Visitada en Junio de 2023: https://microsoft.github.io/monaco-editor/

[11] ACE High performance code editor for the web. Visitada en Junio de 2023: https://ace.c9.io/

[12] CodeMirror Extensibe Code Editor for the web. Visitada en Junio de 2023: https://codemirror.net/

[13] The Lezer Parser System (suited for use in code editors). Visitada en Junio de 2023: https://lezer.codemirror.net/

[14] ANTLR Another Tool for Language Recognition. Visitada en Junio de 2023: https://www.antlr.org/

[15] Xtext Language Engineering for everyone. Visitada en Junio de 2023: https://www.eclipse.org/Xtext/

[16] Incremental parsing using ANTLR4 (respuesta de Terence Parr). Visitada en en Junio de 2023: https://github.com/antlr/antlr4/issues/1104

[17] Martin Gogolla, F. Büttner y Mark Richters, "*USE: A UML-based specification environment for validating UML and OCL*". Visitada en Junio de 2023: https://www.researchgate.net/publication/222594748_USE_A_UML-based_specification_environment_for_validating_UML_and_OCL

[18] Martin Fowler, Rebecca Parsons, *Doman-Specific Languages*. Addison-Wesley, New Jersey, 2011. Syntax-Directed Translations, página 201. Embeded Translations, página 219. Semantic model, página 159. Tree Construction, página 284.

[19] Antoni Olivé, *Conceptual Modeling of Information Systems.* Springer, Berlín, 2010. Página 164.

[20] Language Server Extension Guide. Visitada en Junio de 2023: https://code.visualstudio.com/api/language-extensions/language-server-extension-guide

[21] plantUML. Visitada en Junio de 2023: https://github.com/plantuml/plantuml

[22] Graphviz: Open source graph visualization software. Visitada en Junio de 2023: https://graphviz.org

[23] Test to UML and other "diagrams as code" tools - Fastest way to create your models. Visitada en Junio de 2023: https://modeling-languages.com/text-uml-tools-complete-list/

[24] Marco Brambilla, Jordi Cabot, Manuel Wimmer, *Model-Driven Software Engineering in practice. Second Editions*. Morgan & Claypool. taly, 2017. Model to text transformations (M2T), página 141. Transformaciones endógenas, página 124.

[25] DOT Language: Abstract grammar for defining Graphviz nodes, edges, graphs, subgraphs, and clusters.. Visitada en Junio de 2023: https://graphviz.org/doc/info/lang.html

[26] HPCC-JS/WASM V2 Useful libraris in Node JS now in Web Browser. Visitada en Junio de 2023: https://github.com/hpcc-systems/hpcc-js-wasm

[27] Terence Parr, *The Definitive ANTLR4 Reference*. The Pragmatic Programmers, US 2012. Event Method, página 111. Sobrecargar el parser, página 81.

[28] Keith D. Cooper & Linda Torczon, *Engineering a compiler*. Morgan Kaufmann, UK 2012. Second edition. Página 237.

[29] Robert C. Martin, Clean Architecture A Craftsman's Guide to Software Structure and Design. Pretince Hall, US 2018. Página 61.

[30] Terence Parr, *Language Implementation Patterns*. The Pragmatic Programmers, US 2011. LL(K), página 41.

[31] Lezer: Token Specialization. Visitada en Junio de 2023: https://lezer.codemirror.net/docs/guide/#token-specialization

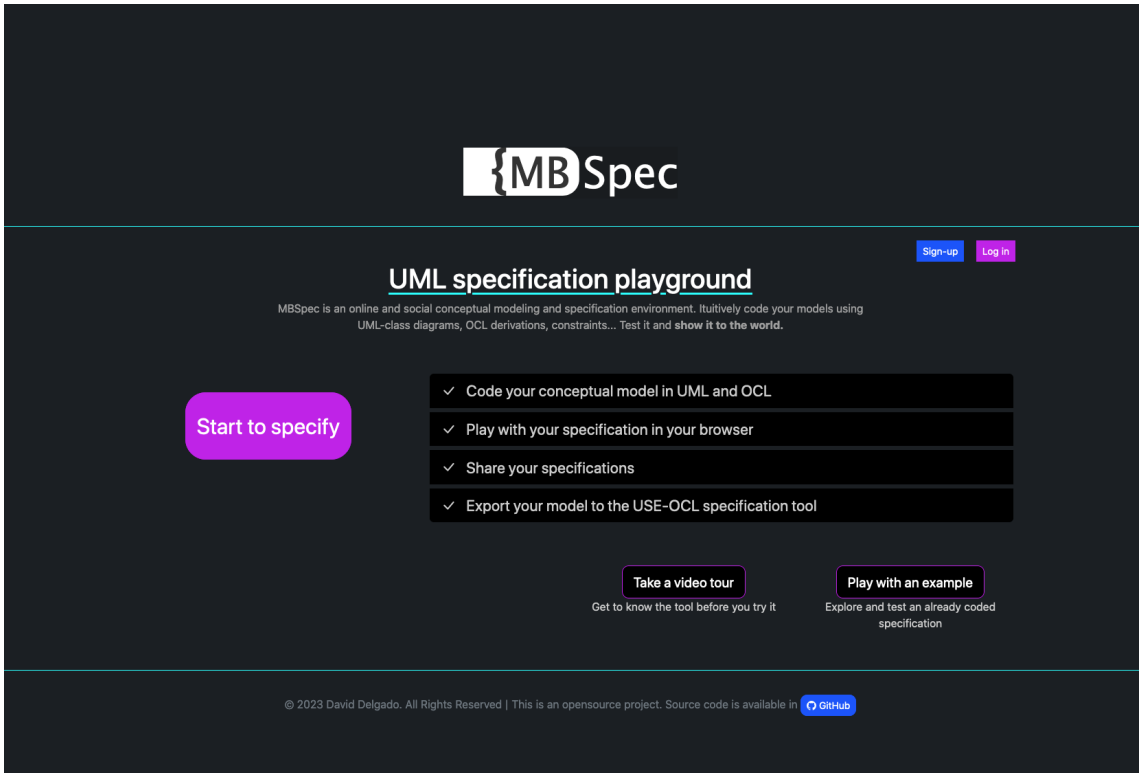
7. Anexos

7.1 Manual de usuario

7.1.1 Acceso

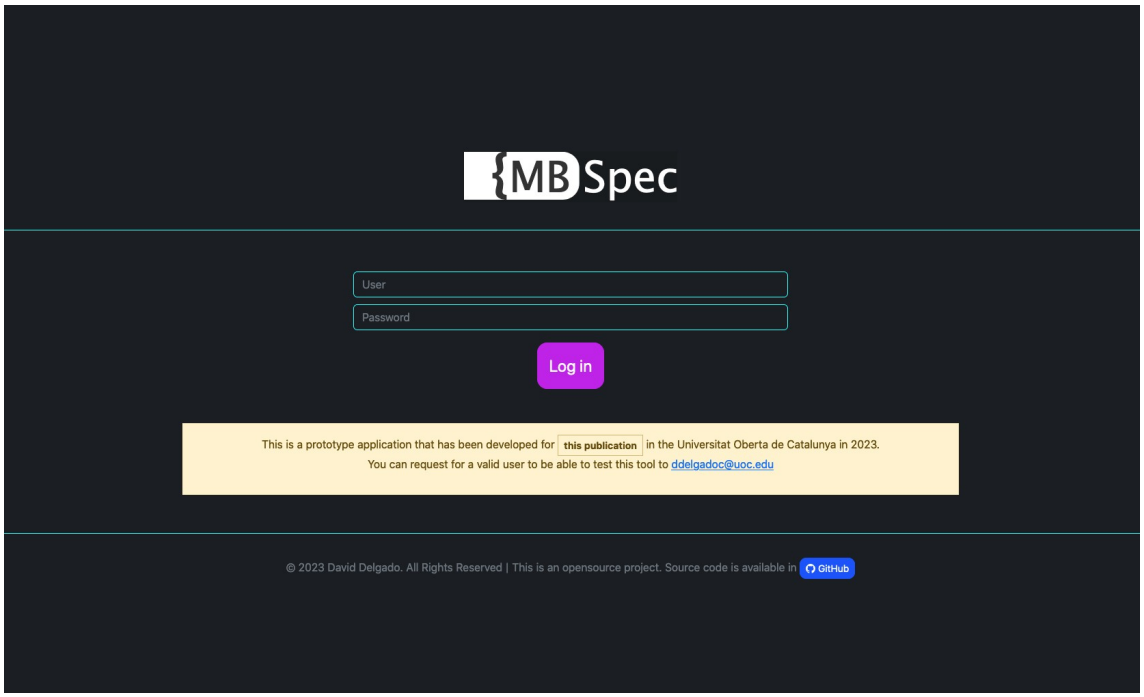
Para poder acceder a la aplicación se debe usar la siguiente URL (sin HTTPS):

- http://mbspec.delgado-rc.com



Se puede hacer directamente Log in o hacer click en Start to specify. Considerando que no se pueden crear usuarios, ya que esta opción está deshabilitada.

En cualquier caso, si no se ha iniciado previamente la sesión, se debe introducir usuario y contraseña:



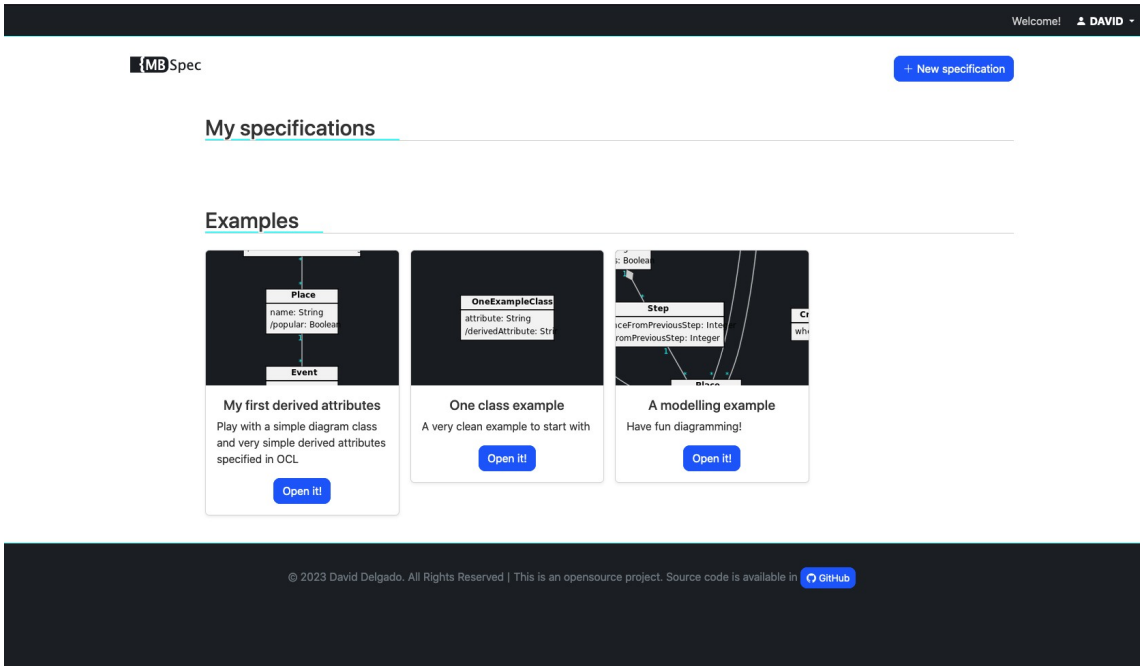
Se puede utilizar cualquiera de las credenciales de los siguientes usuarios:

User	Password
mbspectest	0
mbspectest2	_lH6fP7xN3uT2yE1e

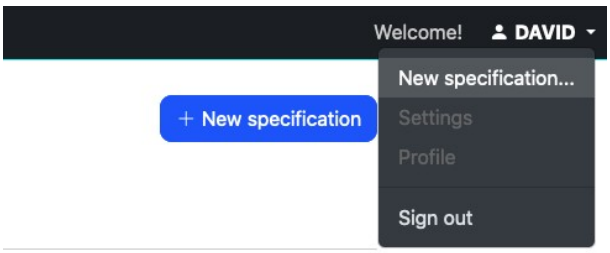
Nota: *Se habilitan 2 usuarios diferentes para que sea posible probar las opciones de compartir especificaciones entre usuarios. Aún así, también sería posible facilitar el acceso a DJANG-ADMIN y crear todos los usuarios que sean necesarios. Pero, no se permite la creación de nuevos usuarios ni se facilita los datos de acceso en esta memoria pública por seguridad en el entorno de producción.*

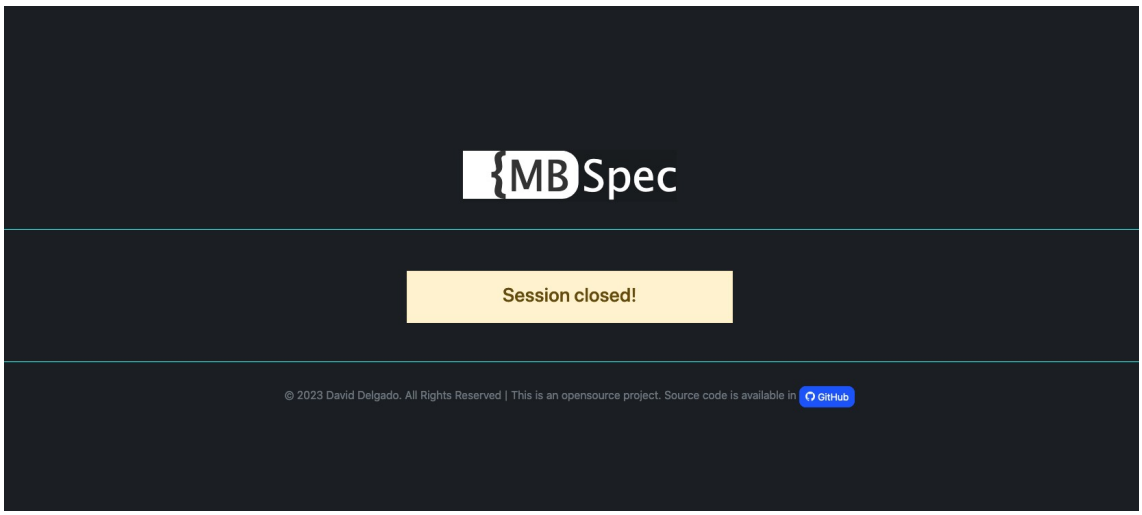
7.1.2 Dashboard y opciones comunes

A continuación, se puede observar el Dashboard, que es donde se almacenan todas las especificaciones que crea un usuario. En un principio, la pantalla estaría en blanco. Sin embargo, estarán disponibles una serie de ejemplos que facilitarían la prueba inicial del entorno y sobre todo ver cómo trabajan los lenguajes en equipo:

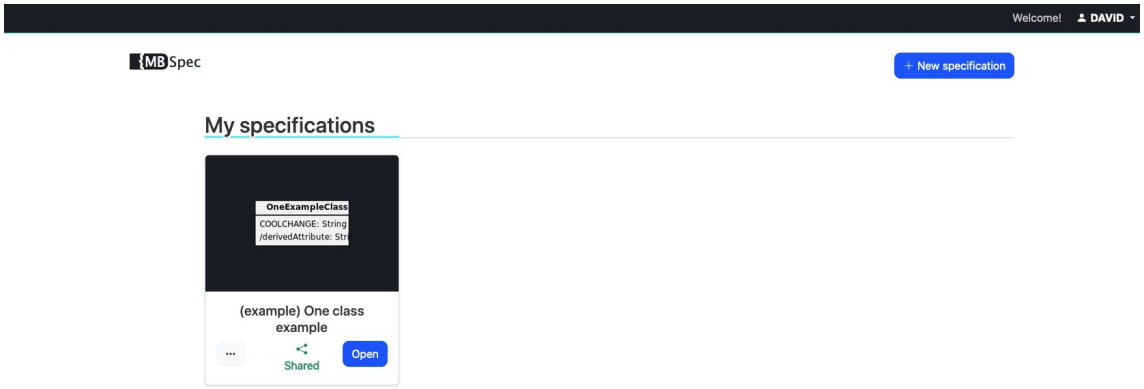


Desde esa misma pantalla se puede crear una nueva especificación vacía, copiar y abrir uno de los ejemplos o simplemente cerrar la sesión:

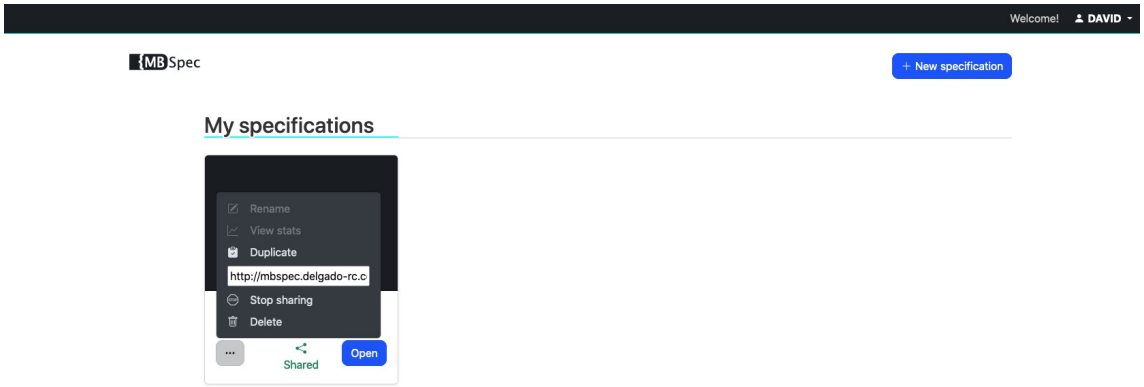


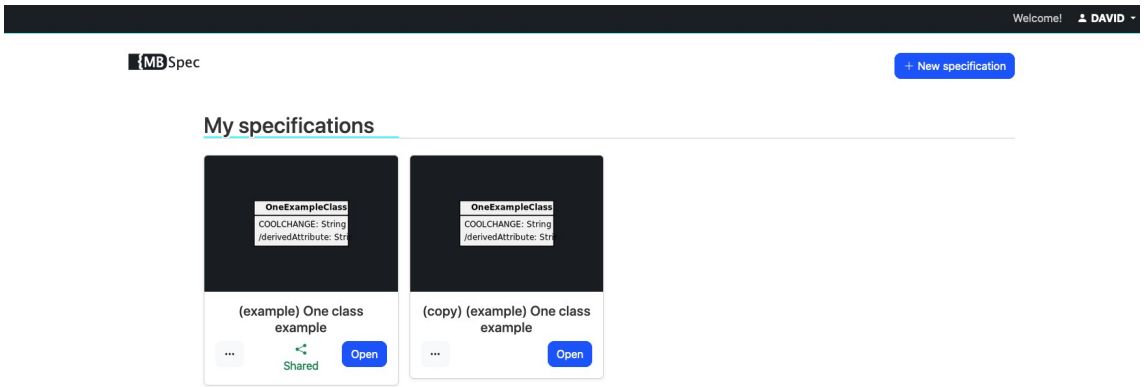


Se puede realizar diferentes acciones con cada especificación del usuario. Por ejemplo, tras copiar una especificación de ejemplo, compartirla y volver al Dashboard, se puede observar que se dispondría de la misma especificación de ejemplo, pero propia:

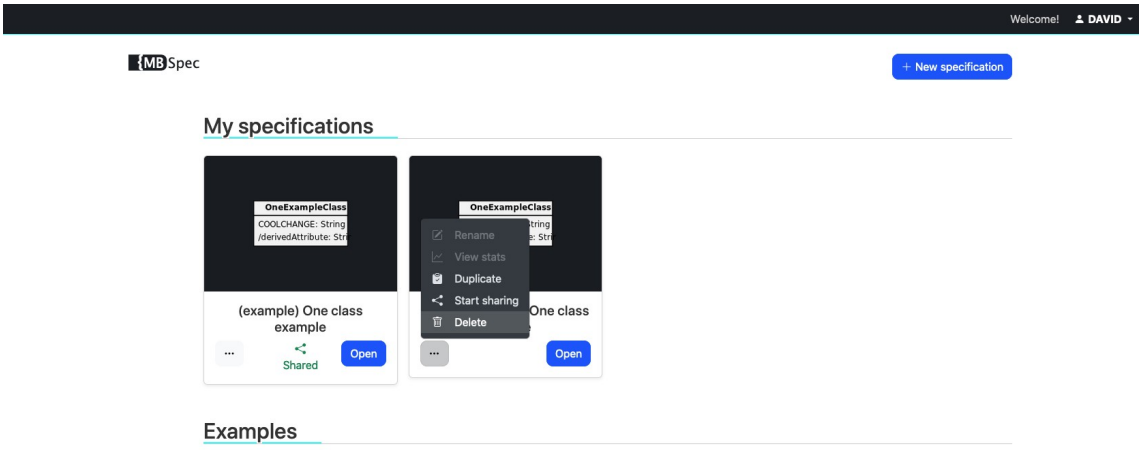


Entonces, se podría duplicar:

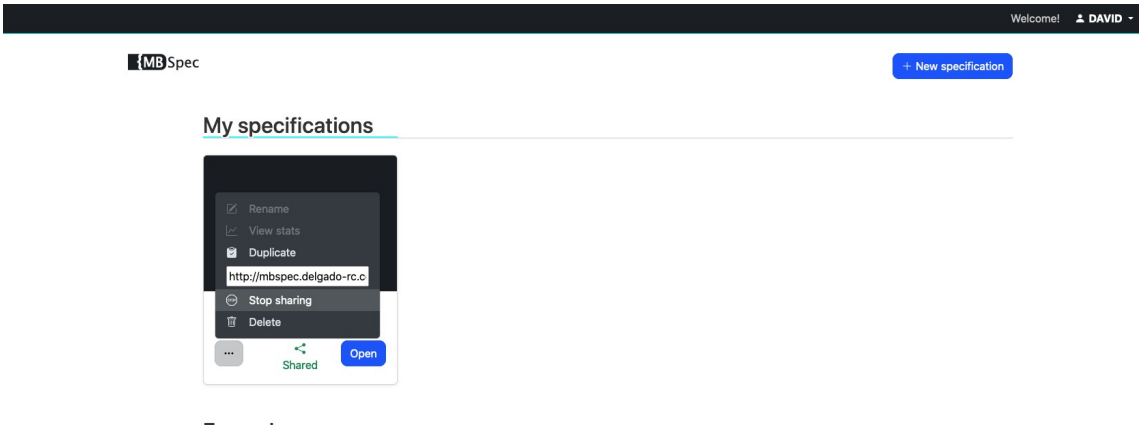


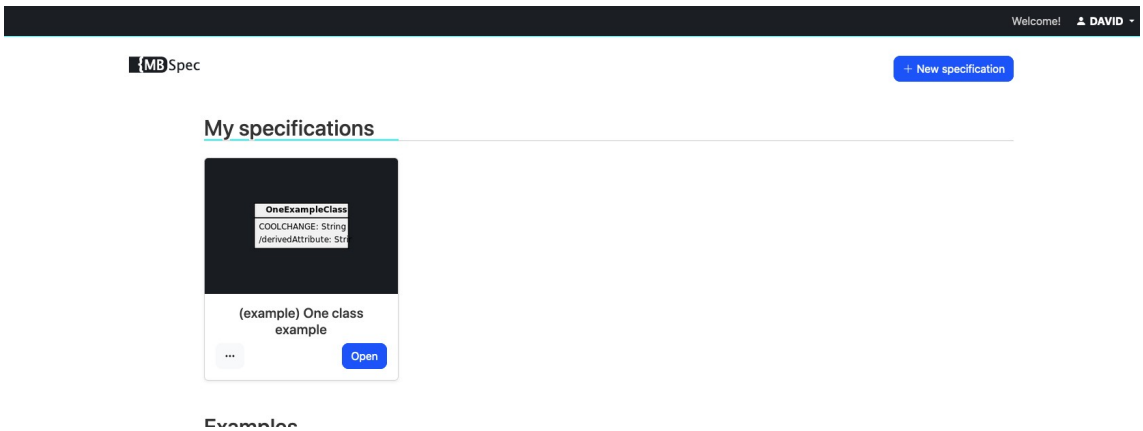


Borrar:



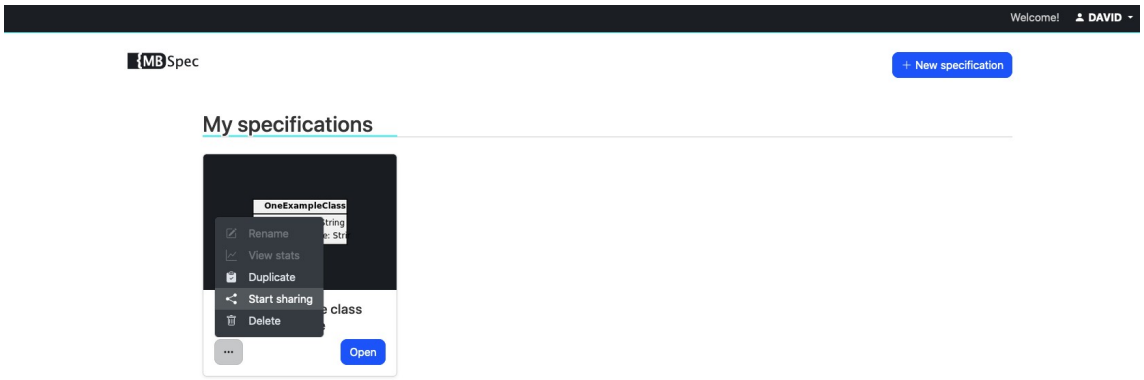
O dejar de compartir:

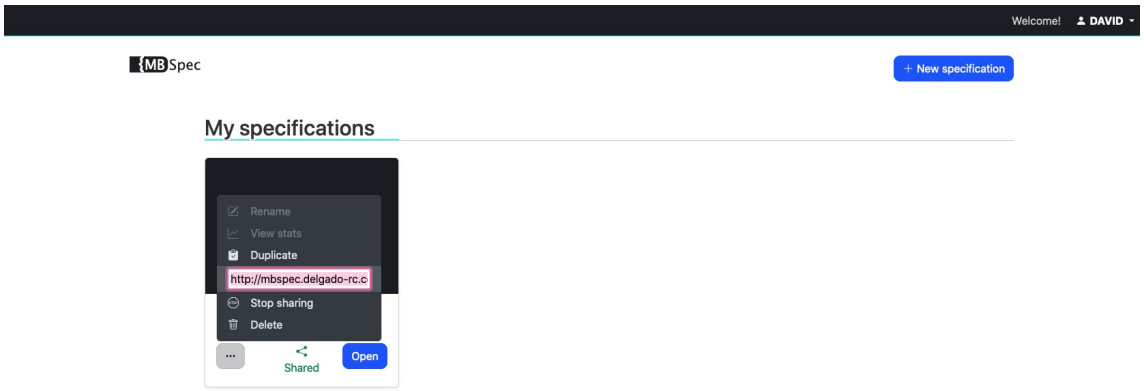




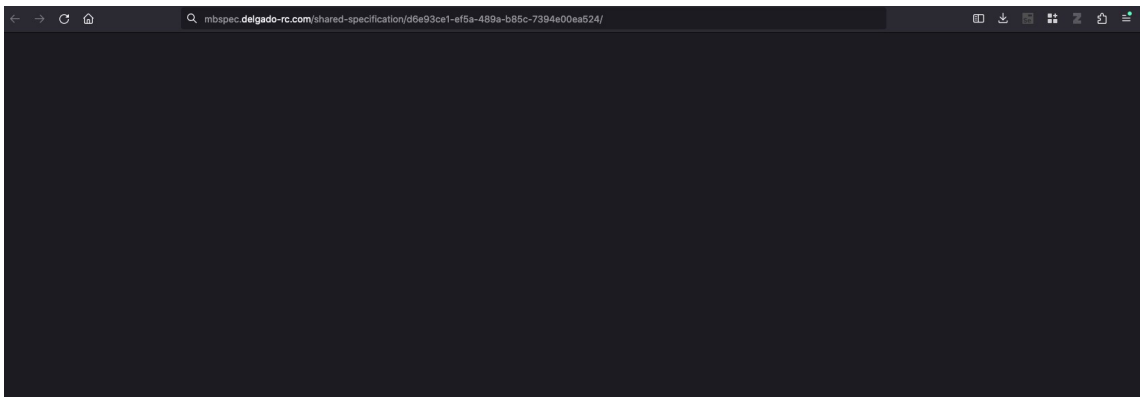
7.1.3 Cómo compartir una especificación desde el Dashboard

Para compartir una especificación simplemente se hace click en compartir y se copia el enlace generado para la especificación.

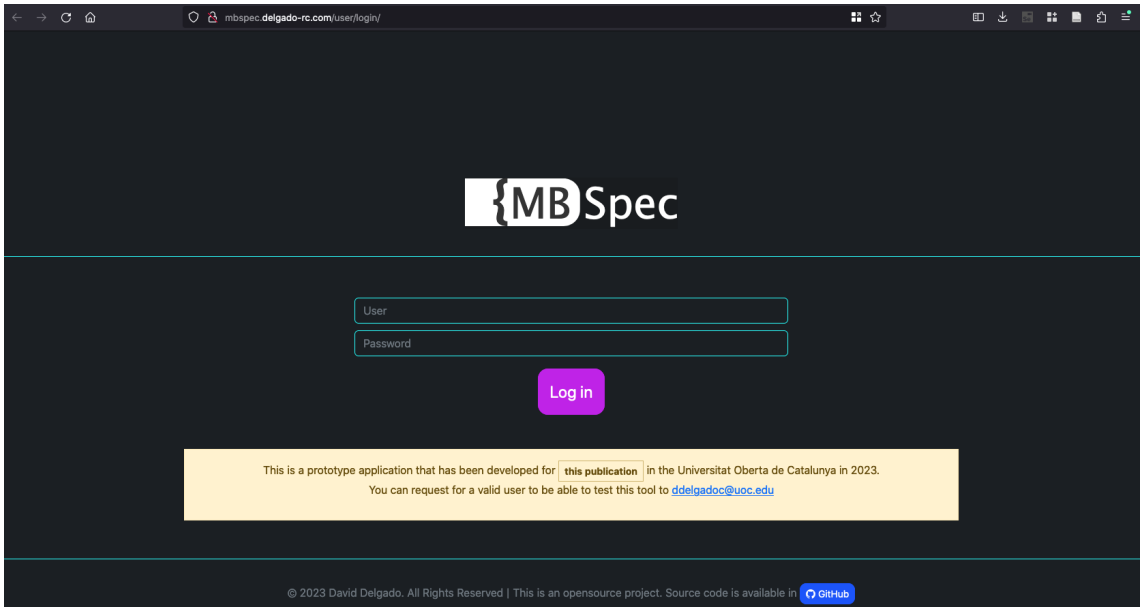




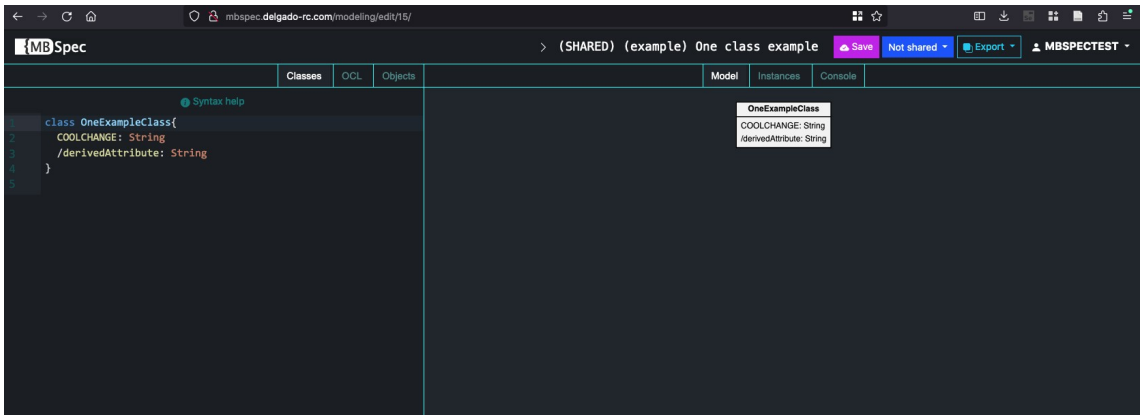
Con el enlace compartido, otro usuario abre su navegador y lo introduce:



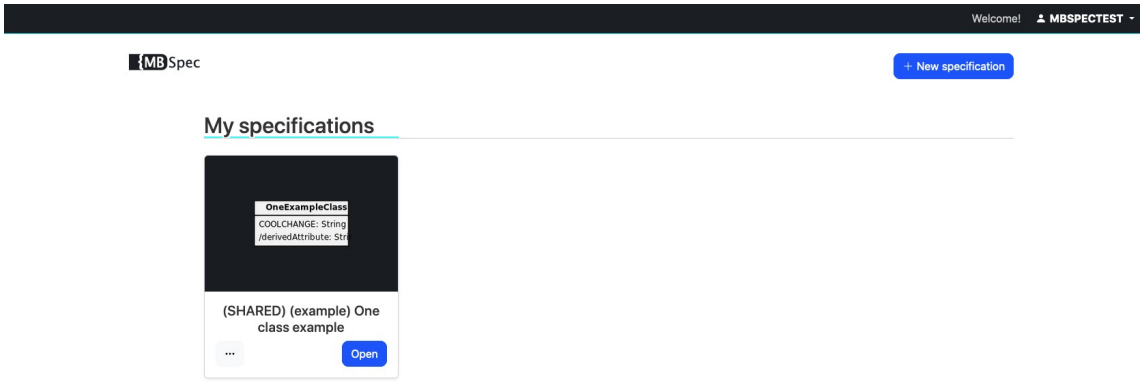
El sistema le solicita sus credenciales de acceso directamente:



Al introducir los datos correctamente del usuario, se abre la especificación compartida:



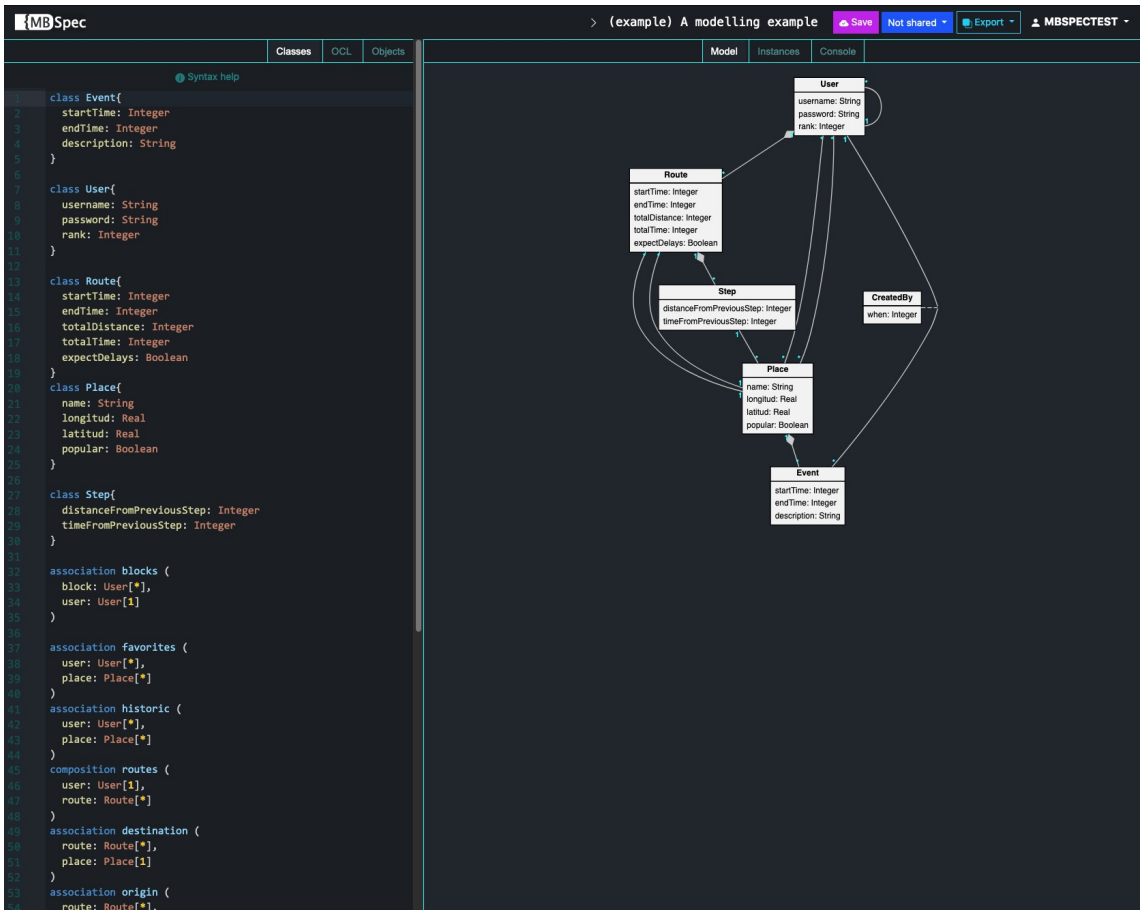
Se puede observar que el título indica SHARED. Y la realidad es que el título se modifica porque la especificación ha sido compartida pero no se trabaja de forma colaborativa. Simplemente, que con el enlace generado, se puede acceder a ella para consultar la última versión del diagrama. El cual se copia siempre en el listado de especificaciones del usuario, tal y como se muestra a continuación:



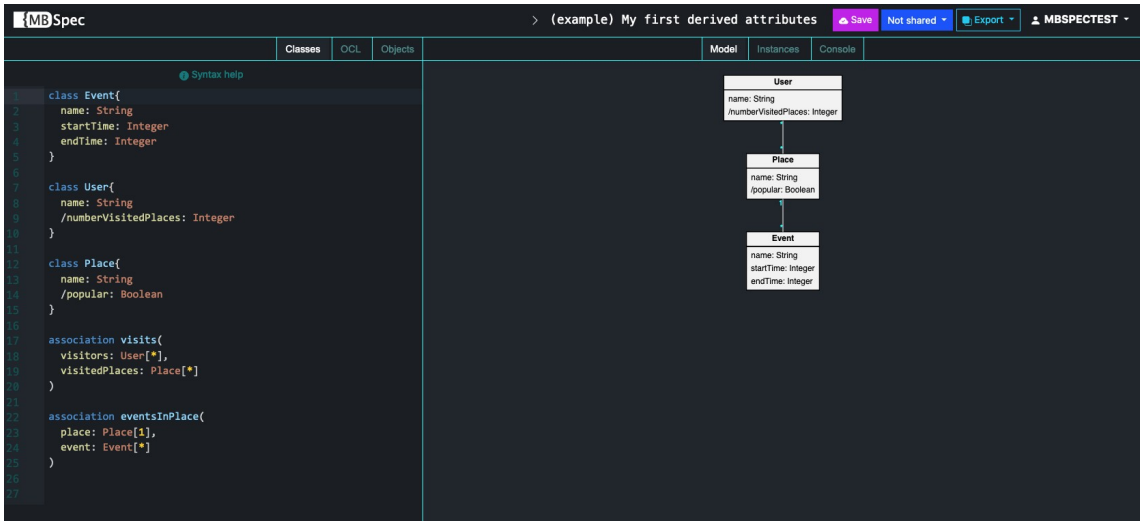
7.1.4 Especificaciones de ejemplo

Las especificaciones de ejemplo muestran casos con diferentes propósito, pero con un objetivo formativo-ilustrativo.

Se pueden observar casos más complejos de diagramación:



Y casos menos complejos, más centrados en la validación de la especificación propiamente dicha:

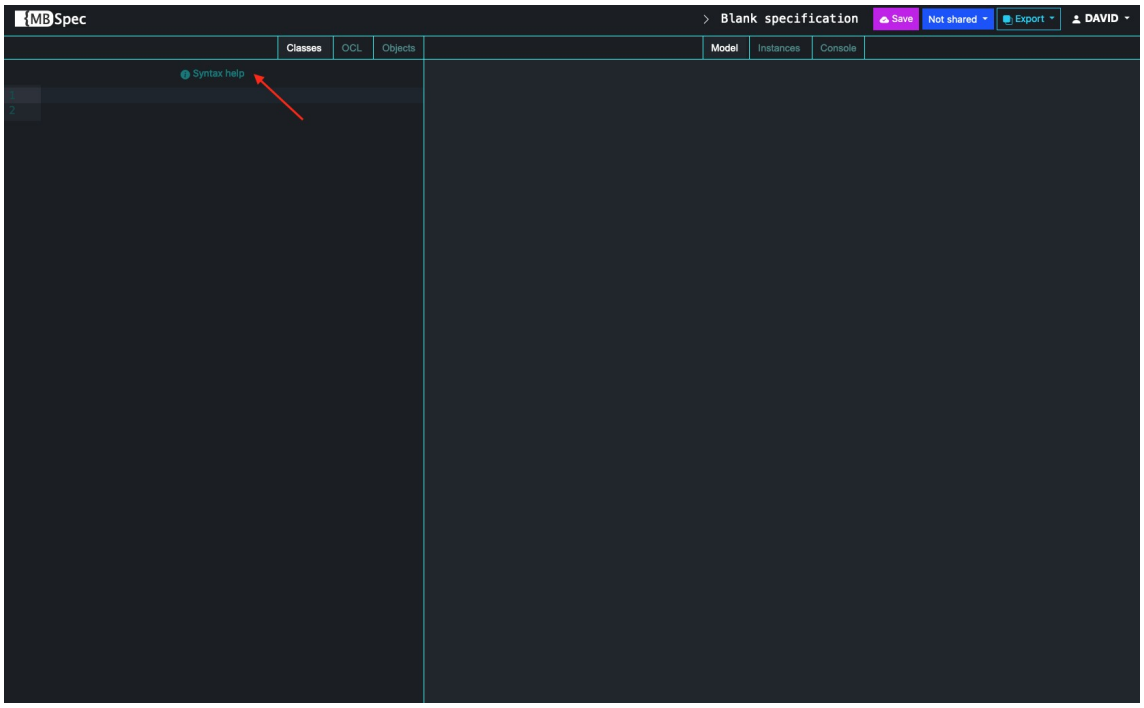


7.1.5 Entorno de especificación y editor de clases

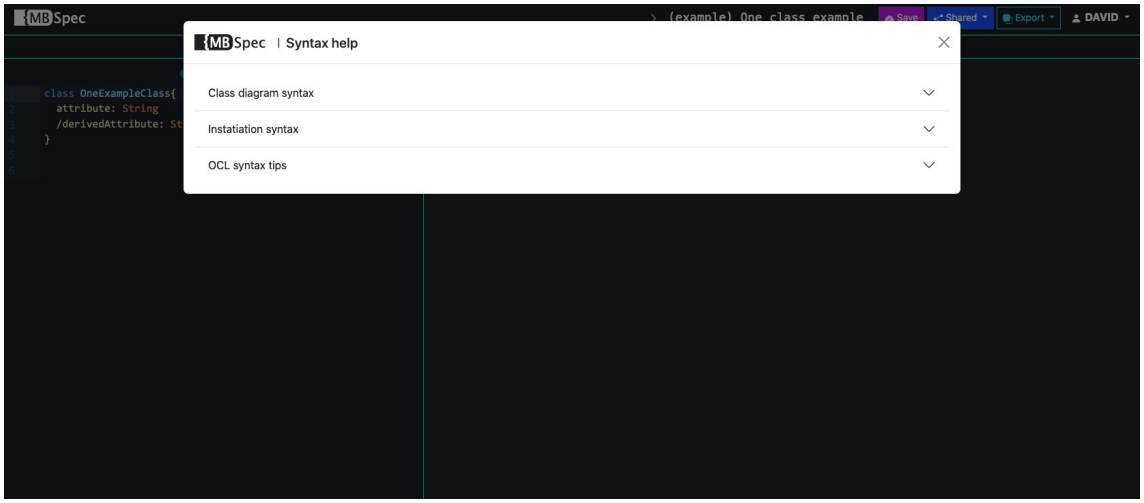
A continuación, se puede observar el Dashboard, que es donde se almacenan todas las especificaciones que crea un usuario. En un principio no se tendrá ninguna especificación de usuario creada (listado vacío). Sin embargo, existe una serie de ejemplos que permiten probar con especificaciones sin tener que crearlas desde cero.

Si se crea una nueva especificación vacía, se abre el entorno de trabajo, con una precarga inicial de la aplicación en el navegador.

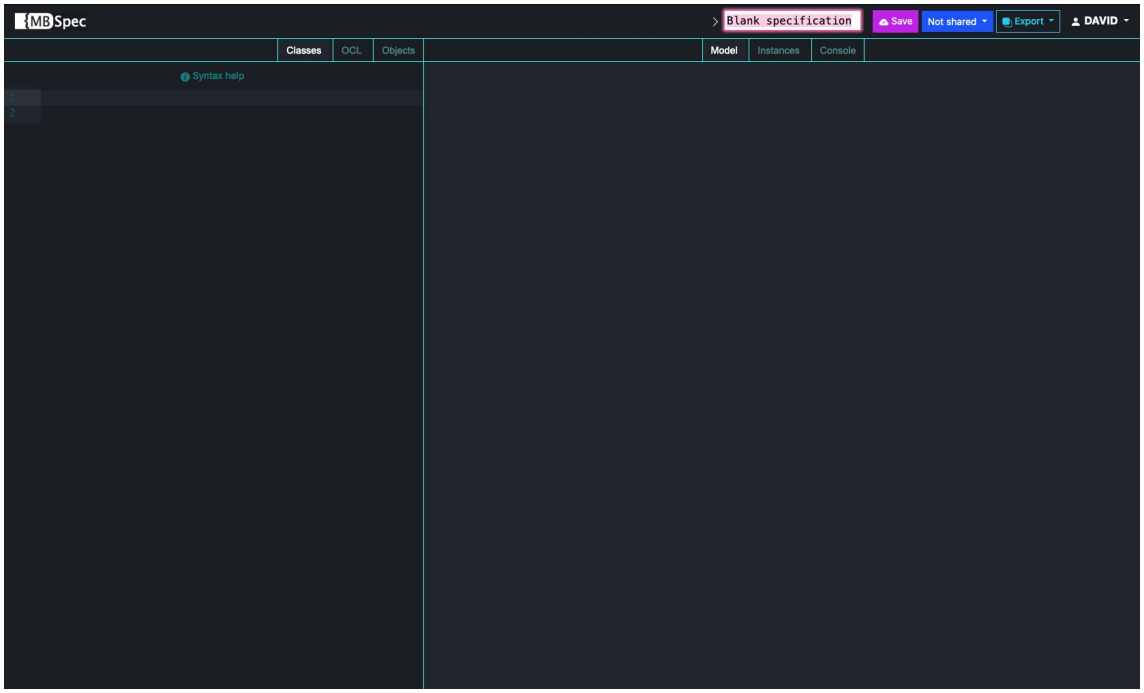
Posteriormente se desvela interfaz más importante de la aplicación. Que contiene, además, una ayuda (que se destaca con una flecha roja en esta captura) de sintaxis de los diferentes lenguajes utilizados en la aplicación:



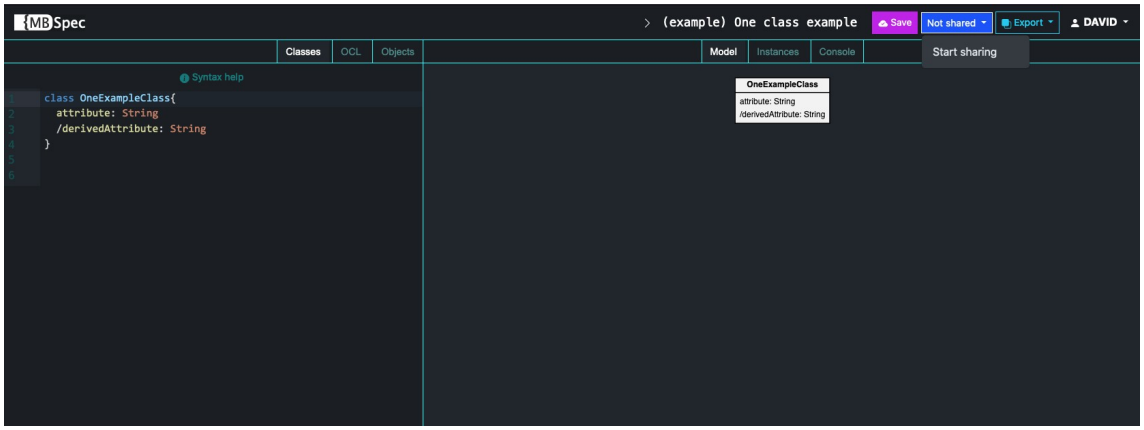
Si hacemos click sobre la ayuda, se puede desplegar diferentes tipos de ayuda en función del lenguaje, con diversas muestras de código y opciones posibles:

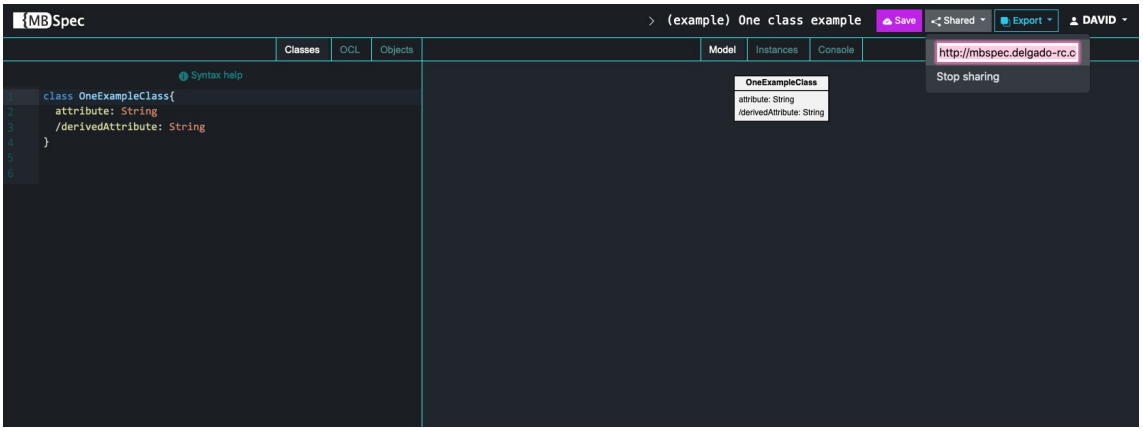


En esa misma pantalla, ya es posible directamente editar el nombre de la especificación, simplemente haciendo click sobre el nombre:

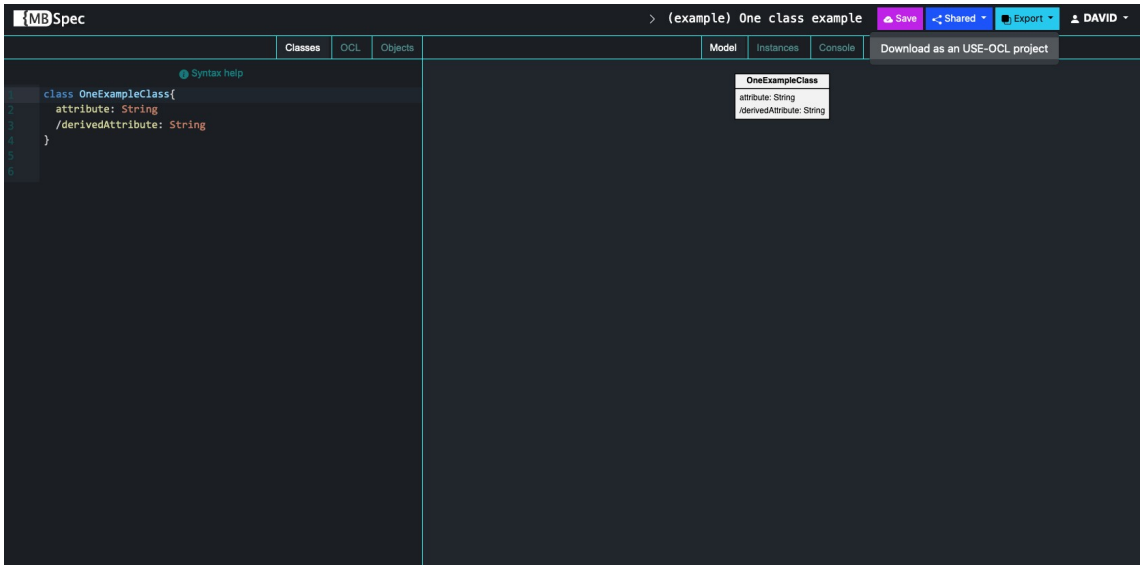


Compartir la especificación:



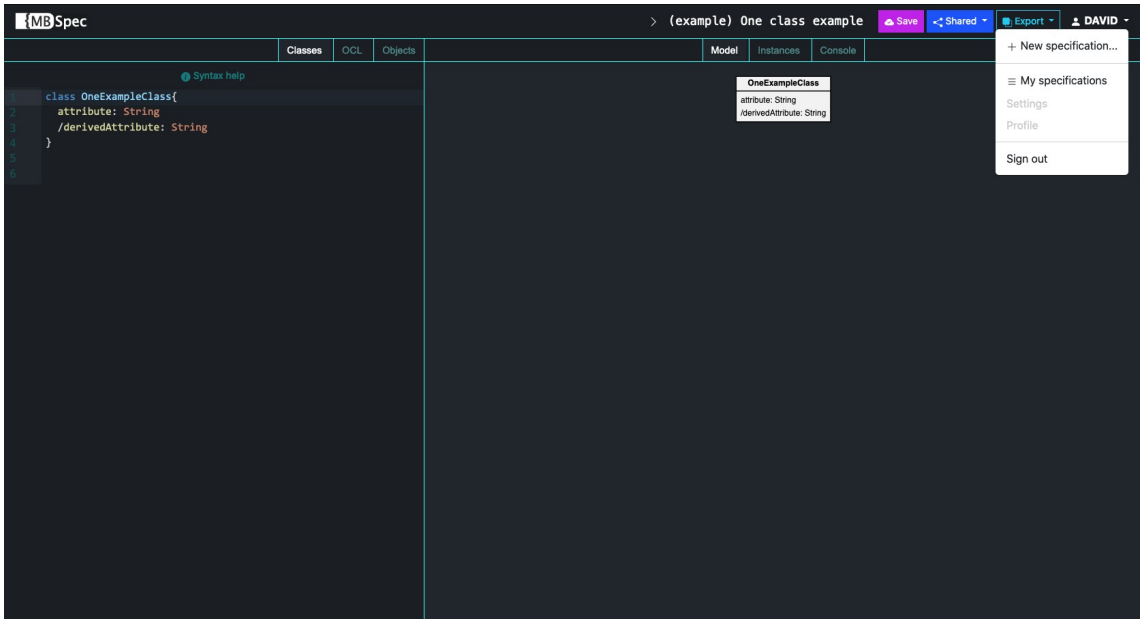


Así como realizar la exportación de la misma a USE-OCL:



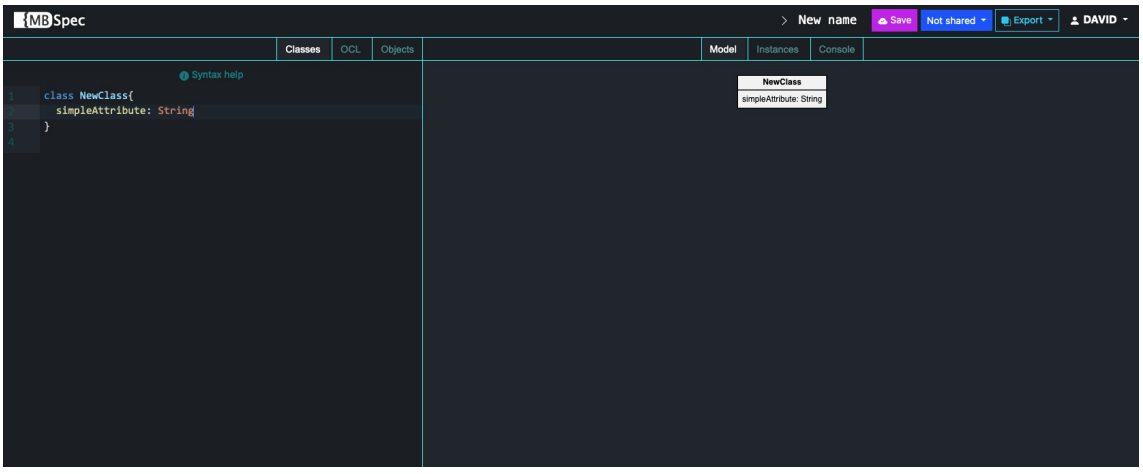
El procedimiento de exportación a USE-OCL se trabajará en detalle en los próximos apartados.

Igualmente, hay la opción de volver al Dashboard, tanto haciendo click sobre el logo como también haciendo click sobre My specifications.



7.1.6 Editor de clases

Lo más importante es que ya se puede comenzar a crear un diagrama de clases. Para ello se utiliza la navegación principal y las 2 vistas en las que se divide la pantalla. Considerando que, a la izquierda, se escribe primero el código del diagrama y a la derecha se visualiza el diagrama interpretado gráficamente:

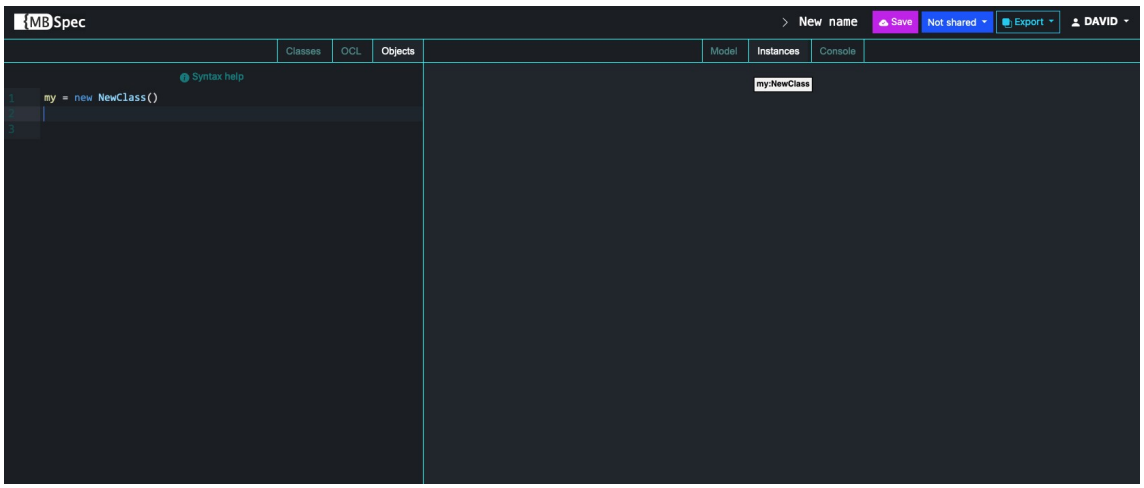


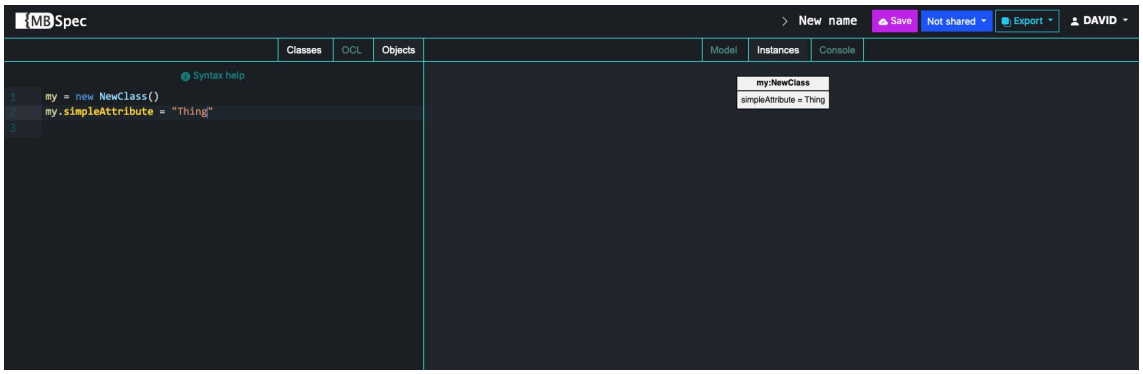
Al escribir, directamente se está haciendo una validación del código completo (léxica, sintáctica y semánticamente).

Aunque tal validación no se pretende realizar de forma exhaustiva, el caso de este Editor es el más completo en lo que a validación se refiere.

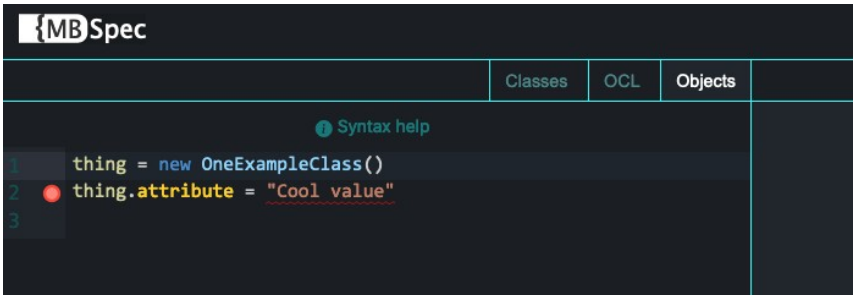
7.1.7 Editor de objetos (instanciación)

El menú superior izquierdo permite realizar la navegación entre los editores de los 3 lenguajes del entorno. El Lenguaje de Objetos es el empleado para la instanciación de los objetos del modelo que se haya definido en Classes. En la vista de la derecha (opción Instances) se puede observar la creación del diagrama de objetos de forma automática:



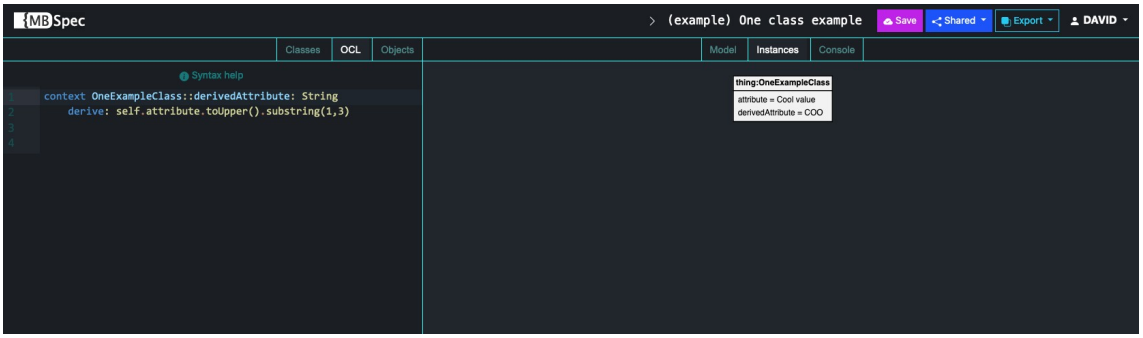


También se puede observar que se produce una validación del código introducido. Por ejemplo, si cambiamos el tipo de un atributo a Integer y lo instanciamos con un ValueType de tipo String:



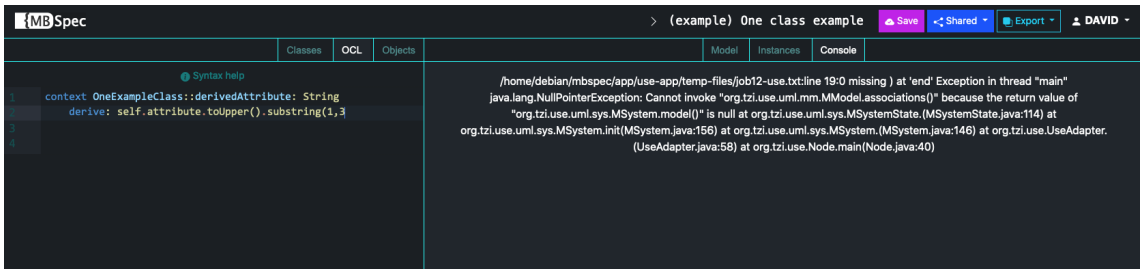
7.1.8 Editor de OCL

El menú OCL sirve para la especificación de las restricciones y atributos derivados del modelo previamente definido en Classes.



Hay que considerar que se utiliza la especificación completa para la definición de atributos derivados y que la misma es lo único que valida (léxica, sintáctica y semánticamente) la aplicación desde el navegador, ya que la validación del

contenido de las expresiones en OCL son validadas por un WebService. Por ejemplo, si introducimos un error en la expresión mostrada anteriormente, automáticamente se muestra una ventana con el error de la expresión evaluada por el webservice.



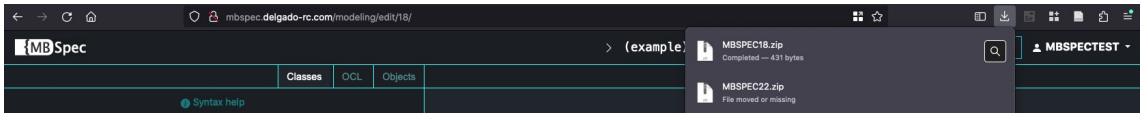
La ventana de error transmite exactamente la evaluación realizada por USE-OCL. Esto es así porque de otro modo habría que procesar el mensaje de error en el backend para mostrar solo los errores. Lo cual, para situaciones simples podría ser evidente y fácil, pero no para otras situaciones más complejas.

Así que no se trata de una información completamente útil, pero que sí guía de forma clara sobre la resolución del problema presente en el código evaluado por USE-OCL.

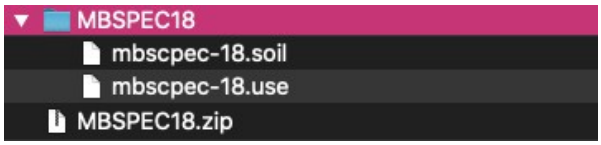
7.1.9 Exportación de especificación a USE-OCL

El primer paso simplemente será realizar la exportación. Hay que considerar que la misma solo podrá hacerse si existe definición de reglas de OCL e instanciaciones y además, se ha producido una validación correcta. Es decir, que no se pueden exportar diagramas de clases vacíos ni inválidos, ya que el propósito de la exportación es la evaluación en USE.

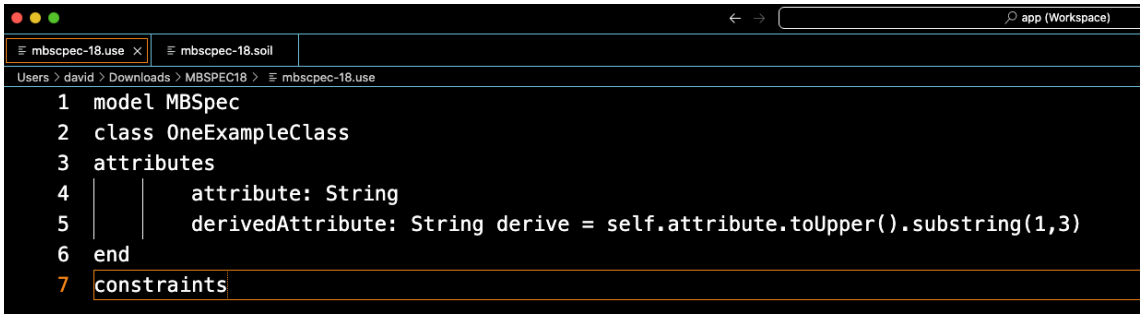
Al hacer click en Download se genera un fichero .zip con todos los ficheros de la especificación:



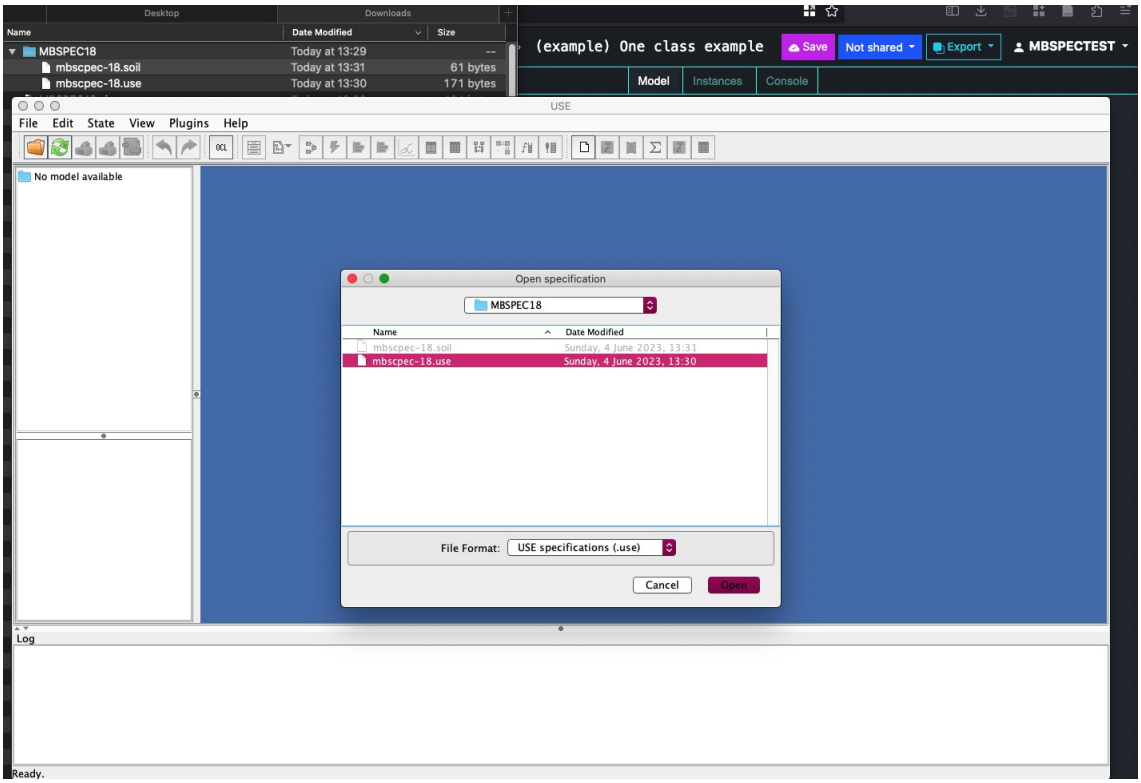
Se descomprime el mismo:

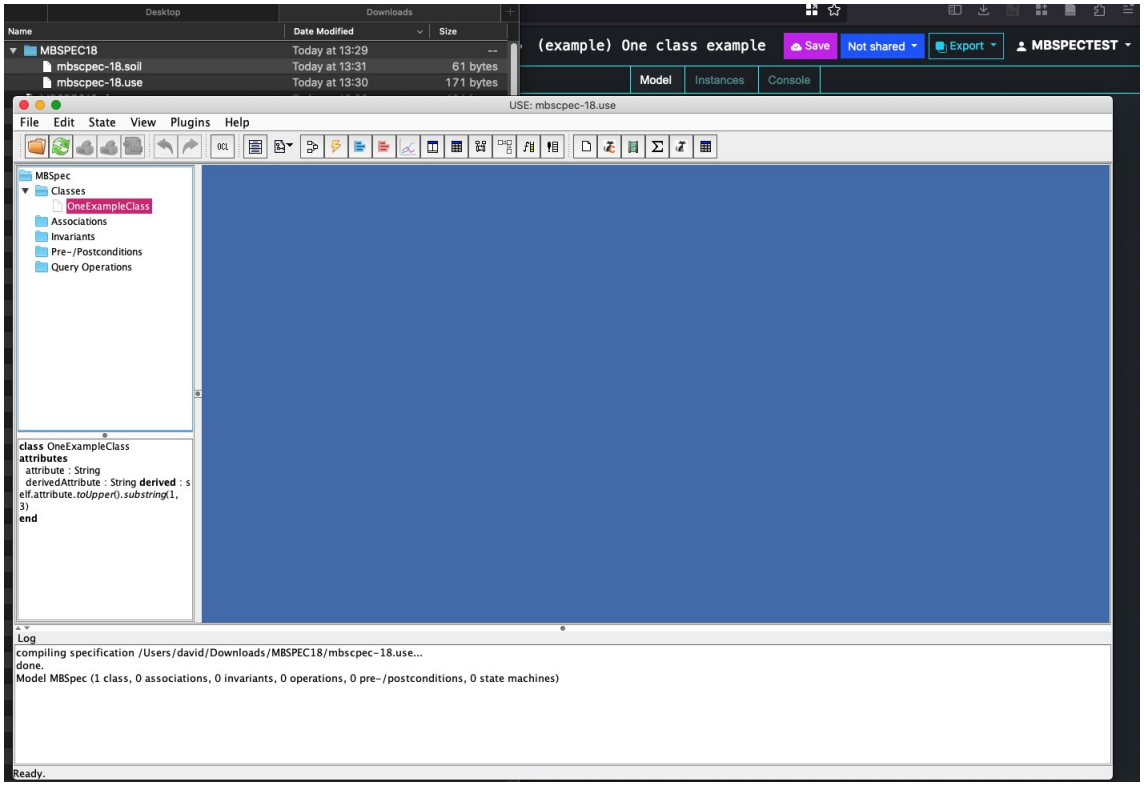


Si se abre el fichero .use, se puede observar que su contenido muestra un código en el lenguaje de la gramática de USE-OCL. Por lo tanto, contiene el código para la diagramación, los atributos con las expresiones OCL definidas y las restricciones invariantes al final del fichero (que no se observan en este caso porque no se ha definido ninguna).

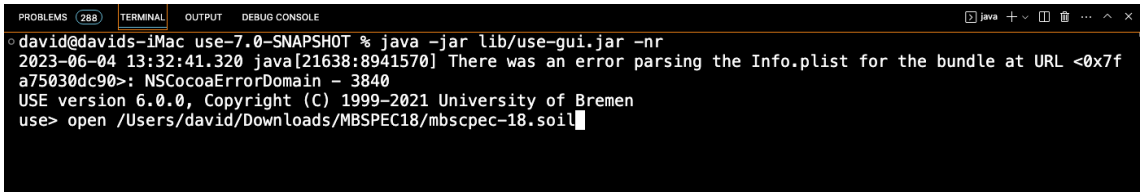


Ese mismo fichero .use se puede abrir directamente desde la propia aplicación USE-OCL:





Por último, se puede importa el fichero .soil, que precisamente está escrito en el lenguaje SOIL de USE-OCL y que permite generar las instanciaciones. Se importa, como se hace habitualmente en la herramienta:



Se observa que la importación es correcta:

```
mbscpec-18.soil>
use> open /Users/david/Downloads/MBSPEC18/mbscpec-18.soil
mbscpec-18.soil> !new OneExampleClass('thing')
mbscpec-18.soil> !thing.attribute := 'Cool value'
mbscpec-18.soil>
use>
```

A continuación, se muestra el resultado final: