Asistente de corrección y validación de ejercicios

Laura Hernando Serrano
Daniel Rossetto Bermejo

Director: Manuel Montenegro Montes
Codirector: Santiago Saavedra

UNIVERSIDAD
COMPLUTENSE
MADRID

Trabajo de Fin de Grado del Grado en Ingeniería Informática

Facultad de Informática
Universidad Complutense de Madrid

Agradecimientos

Agradecemos a Manuel Montenegro su disposición a ayudarnos y su dedicación durante el proyecto y a lo largo de la carrera.

Índice

Agradecimientos **3**

Índice **5**

Resumen **7**

Palabras clave **7**

Abstract **8**

Keywords **8**

1. Introducción **9**
1. Motivación 9
2. Objetivos 10
3. Plan de trabajo 10

2. Introduction **13**
1. Motivation 13
2. Objectives 14
3. Work Plan 14

3. Selección de herramientas y tecnologías **17**

4. Lenguaje de especificación de requisitos **23**

5. Implementación del comprobador **27**
1. PoVALE-core 27
    1.1. Estructura de clases 27
2. PoVALE-reader 33
    2.1. Lenguaje de especificación mediante XML 33
    2. 2. Estructura de clases 37

6. Implementación de la interfaz gráfica **39**
1. Manual de usuario 39
    1.1. Datos de entrada 40
    1.2. Requisitos 42
    1.3. Validación 43
2. Detalles de implementación 43

7. Implementación del generador de especificaciones de entrega **47**
1. Manual de usuario 47
2. Detalles de implementación 49

8. Conclusiones **53**
1. Valoración de los resultados 53
2. Trabajo futuro 54

9. Conclusions **57**
1. Evaluation of the results 57
2. Future work 58

10. Apéndice: Contribución de los integrantes **61**
1. Contribución de Laura Hernando Serrano 61
2. Contribución de Daniel Rossetto Bermejo 63

11. Apéndice: PoVALE-plugin-files **65**

12. Apéndice: PoVALE-plugin-lines **67**

13. Apéndice: ¿Cómo crear un plugin en PoVALE? **69**

14. Apéndice: PoVALE-core: Assertion **73**

15. Apéndice: PoVALE-specification: PluginSpec **75**

16. Apéndice: PoVALE-specification: TermRep **77**

Bibliografía **79**

Resumen

La implantación del Espacio Europeo de Educación Superior ha supuesto modificar las metodologías docentes para hacer mayor hincapié en evaluar el esfuerzo del alumno mediante mecanismos de evaluación continua. Este sistema potencia la capacidad de los alumnos de cursar con éxito las asignaturas. Sin embargo, el proceso de evaluación continua aumenta la carga de trabajo del profesor de manera significativa y resulta poco factible en asignaturas cursadas por un gran número de alumnos.

Nuestro objetivo es desarrollar una serie de herramientas que faciliten y agilicen la corrección de ejercicios entregados por medio digital para favorecer el uso de la evaluación continua en grupos de alumnos numerosos. Para cumplir dicho propósito, nos planteamos los objetivos de implementar una herramienta que compruebe que un ejercicio respeta unas especificaciones de entrega, desarrollar la interfaz gráfica de esta herramienta para su uso por parte de los alumnos, y por último, implementar una herramienta para que el profesor pueda generar documentos de especificación.

El resultado obtenido ha cumplido con las expectativas y objetivos marcados. Las herramientas desarrolladas en `PoVALE`, son extensibles e intuitivas, podrían ser continuadas y mejoradas, ya que tienen potencial para ser usadas en las nuevas titulaciones para ayudar a los profesores en la corrección de ejercicios.

Palabras clave

Validador de ejercicios, especificación de requisitos, generador de especificaciones, evaluación continua, lenguaje de especificaciones, entrega de ejercicios, EEES.

Abstract

The establishment of the European Higher Education Area has encompassed modifying the teaching methodologies to place greater emphasis on assessing the student's effort through a continuous assessment system. This system heightens the ability of students to succeed in their subjects. However, the continuous assessment system increases the teacher's workload significantly and is unfeasible in subjects taken by a large number of students.

Our goal is to develop a series of tools that facilitate and speed up the evaluation of exercises submitted digitally to encourage the use of continuous assessment in large groups of students. To fulfill this purpose, we set ourselves the objectives of, firstly, developing an application that verifies that an exercise respects a specification, in other words, a set of requirements. Secondly, to develop the graphical interface of this tool for the use of students, and finally, develop an application that allows a teacher to generate specification documents.

The results obtained have met the expectations and objectives set. The PoVALE applications developed are extensible and intuitive, and could be continued and improved, since they have potential to be used in the new degrees to help teachers in the correction of exercises.

Keywords

Exercise validator, specification of requirements, specifications generator, continuous assessment, specification language, exercise submission, EHEA.

1. Introducción

En este capítulo explicaremos la motivación para desarrollar un sistema de evaluación de ejercicios, así como los objetivos a lograr en el desarrollo de dicho sistema. Por último, concretaremos el plan de trabajo que hemos seguido, detallando qué objetivos se han abordado en cada una de las fases de desarrollo.

1. Motivación

A consecuencia del Proceso de Bolonia, en el que veintinueve países de la UE acuerdan reformar los sistemas de Educación Superior para crear un marco unificado de enseñanza, surge el Espacio Europeo de Educación Superior (EEES) [14]. Su implantación ha supuesto modificar las metodologías docentes para hacer mayor hincapié en evaluar el esfuerzo del alumno mediante mecanismos de evaluación continua.

El sistema de evaluación continua consiste en la realización de ejercicios de manera regular, su evaluación y la subsecuente retroalimentación proporcionada a los alumnos. Dicho sistema favorece un aprendizaje en el que los alumnos desarrollan y adquieren competencias de forma progresiva, potenciando su capacidad de cursar con éxito las asignaturas. No obstante, el proceso de evaluación continua aumenta la carga de trabajo del profesor, en tiempo y esfuerzo, de manera significativa y proporcionalmente al número de alumnos. A consecuencia, esta metodología resulta poco factible en asignaturas cursadas por un gran número de alumnos, donde la carga de trabajo imposibilita poder dar resultados detallados sobre los ejercicios a los alumnos con la suficiente antelación para que tengan tiempo para corregir y aprender de los fallos cometidos.

El sistema de evaluación de ejercicios `PoVALE` tiene como objetivo desarrollar un conjunto de herramientas que faciliten la corrección de ejercicios entregados por medio digital. En particular, este TFG aborda la validación de los aspectos formales de la entrega de un ejercicio, en el sentido en que dicha entrega cumple una serie de requisitos especificados por el profesor. Por ejemplo, un requisito puede imponer que el nombre del fichero entregado tenga una determinada extensión, o que sea un directorio que no contenga en su interior ficheros con extensión `.class`. También pueden ponerse restricciones sobre el propio contenido del fichero, como por ejemplo, que tenga una línea con el nombre de los estudiantes. El conjunto de requisitos que pueden especificarse será extensible mediante añadidos (*plugins*).

2. Objetivos

A continuación, enumeraremos y detallaremos los objetivos del proyecto `PoVALE`.

- **Implementación del comprobador**

Desarrollar una herramienta para comprobar que el ejercicio entregado por el estudiante respeta unas especificaciones de entrega.

- **Implementación de la interfaz del comprobador**

Desarrollar la interfaz gráfica de la herramienta del comprobador. Dicha herramienta debe permitir a los alumnos validar su entrega, ver los resultados de evaluar la entrega y finalmente exportarla para uso del profesor.

- **Implementación de la herramienta para generar especificaciones**

Implementar una herramienta visual para generar documentos de especificación. La herramienta debe permitir a un profesor definir los requisitos que debe cumplir una entrega y exportarlos de forma que el documento generado pueda ser empleado en la herramienta del comprobador.

3. Plan de trabajo

En este apartado explicaremos las fases de desarrollo del proyecto, además de indicar en qué capítulo se explicará cada fase.

3.1. Estudio de herramientas y tecnologías

En esta primera fase evaluamos las tecnologías existentes que podíamos emplear para desarrollar el proyecto. La primera decisión que tomamos fue emplear JavaFX con la herramienta Scene Builder en lugar de Swing para el diseño de las interfaces gráficas. Teníamos experiencia usando Swing por su uso en las asignaturas de Ingeniería del Software, Tecnología de la Programación y Ampliación de Bases de Datos. Sin embargo, nos parecía interesante aprender a usar JavaFX ya que se prevé que es la tecnología que sucederá a Swing y además aporta diversas ventajas. Asimismo, decidimos usar la herramienta Scene Builder con la que es fácil crear interfaces y que requiere un esfuerzo de aprendizaje mínimo.

Nunca habíamos utilizado la reflexión de Java en las asignaturas que cursamos. Vimos que era necesario estudiarla y emplearla para realizar distintas funcionalidades del proyecto.

No teníamos un conocimiento previo de cómo trabajar en Java con ficheros XML. Nos documentamos para ver qué alternativas teníamos. Primero realizamos pruebas básicas usando JAXB. Entonces vimos que esta tecnología era muy compleja, por lo que tras ver cómo funcionaba la API DOM de W3C decidimos que sería esta la tecnología que usaríamos para trabajar con ficheros XML.

Estudiamos los posibles entornos de desarrollo con los que podríamos llevar a cabo el proyecto, y nos decantamos por NetBeans.

En el capítulo 3 se explicarán en detalle las tecnologías utilizadas.

3.2. Implementación del comprobador

Primero procedimos a desarrollar las clases principales para representar los asertos. Tras ver que los creábamos y se evaluaban correctamente, comenzamos la segunda fase en la que definimos un lenguaje para representar los asertos en un fichero XML. Después de esto, implementamos un lector de ficheros XML que cargaría su contenido para poder ser evaluado. Una vez logrados correctamente la lectura y el cargado de especificaciones, realizamos pruebas con el plugin desarrollado por Manuel Montenegro para determinar que toda la funcionalidad que tiene lugar en el proceso de validación era correcta.
Explicaremos esta fase en el capítulo 5.

3.3. Implementación de la interfaz del comprobador

Tras implementar el comprobador, desarrollamos la interfaz que utilizará el alumno para comprobar que su entrega cumple los requisitos especificados por el profesor. Esta herramienta también sirve para exportar a un fichero comprimido los resultados de la entrega si la validación es favorable. Por el contrario, si la validación es desfavorable, debería mostrarse al estudiante un mensaje indicando qué requisitos no han sido satisfechos.
Tras realizar pruebas con la herramienta, vimos que la información que se le daba al alumno antes y después de validar su entrega era insuficiente y confusa por lo que pasamos a la siguiente fase. Esta fase se explica en el capítulo 6.

3.4. Mejora de la mensajería en el comprobador

En la fase 2 habíamos introducido un sistema de generación de mensajes legibles por el usuario para los asertos, que se usaría en la interfaz del proyecto. Tras ver que no informaban al usuario correctamente, realizamos una nueva implementación en el comprobador para que la información mostrada en la interfaz sobre la validación fuera más clara e informase correctamente al alumno.
En los capítulos 5 y 6 explicamos el proceso de generación de mensajes de la herramienta y los cambios que sufrió en esta fase de desarrollo.

3.5. Implementación del generador de especificaciones

Una vez terminados los módulos de validación y de interfaz gráfica, pasamos a desarrollar la aplicación para generar especificaciones. Primero, definimos las funcionalidades que queríamos que tuviese la aplicación y diseñamos la interfaz. A continuación, implementamos las clases necesarias para especificar variables y asertos. Finalmente, introducimos las mecánicas para importar plugins y poder abrir archivos XML. Explicamos el generador de especificaciones en el capítulo 7.

2. Introduction

In this chapter, we will explain the reasons that make it necessary to develop a system to evaluate exercises, as well as the objectives that the system must meet. Finally, we will specify the work plan that we have followed, detailing which objectives have been addressed in each of the development phases.

1. Motivation

As a result of the Bologna Process, in which twenty-nine EU countries agree to reform Higher Education systems to create a unified educational framework, the European Higher Education Area (EHEA) [14] emerges. Its establishment has resulted in the adjustment of teaching methodologies to put more emphasis on evaluating the student's effort through continuous assessment systems.

The continuous assessment system consists of carrying out exercises over a period of time, the evaluation of these exercises, and the subsequent feedback provided to students. This system favors a learning process in which students develop and acquire competences in a progressive way, boosting their ability to succeed in their subjects. However, the continuous assessment system increases the teacher's workload, in terms of time and effort, significantly and proportionately to the number of students. As a consequence, this methodology is not feasible in subjects studied by a large number of students, where the workload makes it impossible to give detailed feedback about the exercises to the students with enough time in advance that they have time to correct and learn from the mistakes made.

The PoVALE exercise evaluation system aims to develop a set of tools that facilitate the correction of exercises submitted digitally. In particular, this final degree project addresses the validation of the formal aspects of the submission of an exercise, in the sense that such submission fulfills a series of requirements specified by the teacher. For example, a requirement may impose that the name of the delivered file has a certain extension, or that it is a directory that does not contain files with a .class extension. You can also specify restrictions on the content of the file itself, such as that it contains a line with the students' names. The set of requirements that can be specified will be extensible by means of plugins.

2. Objectives

The objectives of the PoVALE project are listed and detailed below.

- **Implementation of the validator**

Develop a tool to verify that the exercise submitted by the student respects a series of requirements.

- **Implementation of the validator's interface**

Develop the graphical interface of the validator tool. This tool must allow students to validate their exercise, see the results of the evaluation and finally export the results for the teacher's use.

- **Implementation of the application that generates specifications**

Implement a visual tool to generate documents with the requirements that the exercises must meet. The tool must allow a teacher to define the requirements and export them so that the generated document can be used in the validator tool.

3. Work Plan

In this section we will explain the development phases of the project, in addition to indicating in which chapter each phase will be further explained.

3.1. Research of tools and technologies

In this first phase we assessed the existing technologies that we could use to develop the project. The first decision we made was to use JavaFX with the Scene Builder tool instead of Swing to design the graphical interfaces. We had experience with Swing from using it in the subjects of Software Engineering, Programming Technology and Advanced Databases. However, we thought it would be interesting to learn how to use JavaFX as it is thought to be the technology that will replace Swing, and also provides several advantages. We also decided to use the Scene Builder tool, which allows you to easily create interfaces and requires minimum effort to learn how to use it.

We had never used Java reflection in our subjects. We saw that it was necessary to study and use it to implement various functionalities of the project.

We didn't have prior knowledge of how to work in Java with XML files. We did some research to figure out what alternatives we had. First we performed basic tests using JAXB. Then we saw that this technology was very complex, so after researching how the W3C DOM API worked we decided that this would be the technology we would use to work with XML files.

We studied the possible development environments with which we could carry out the project, and we chose NetBeans.

In chapter 3 we will explain in detail the technologies used.

3.2. Implementation of the validator

First we developed the main classes to represent the assertions. After making sure that we were creating and evaluating them correctly, we started the second phase in which we defined a language to represent the assertions in an XML file. After that, we implemented an XML file reader that would load its contents to make it possible to evaluate the assertions. Once we could successfully read and load specifications, we did some tests with the plugin developed by Manuel Montenegro to determine that all the functionality that takes place in the validation process was working correctly.

We will explain this phase in chapter 5.

3.3. Implementation of the validator's interface

After implementing the validator, we developed the interface that would be used by the student to verify that their submission meets the requirements specified by the teacher. If the validation if is favorable, this tool can be used to export to a compressed file the results. On the contrary, if the validation is unfavorable, the student is shown a message indicating which requirements have not been met.
After carrying out tests with the tool, we observed that the information that was given to the student before and after validating their exercise was insufficient and confusing so we moved on to the next phase. This phase is explained in chapter 6.

3.4. Improvement of the message generation system of the validator

In phase 2 we introduced a user readable message generation system for assertions, which would be used in the validator interface. After seeing that they did not inform the user correctly, we made adjustments in the verifier so that the information displayed in the interface after validation was clearer and informed the student correctly.
In chapters 5 and 6 we explain the process of generating messages and the changes the validator tool underwent in this development phase.

3.5. Implementation of the application that generates specifications

Once the validation modules and graphical interface modules were finished, we moved on to develop the application that generates specifications. First, we defined the features that we wanted the application to have and we designed its interface. Next, we developed the classes that were necessary to specify variables and assertions. Finally, we developed the mechanics to import plugins and to be able to open and edit existing XML files. This phase is explained in chapter 7.

3. Selección de herramientas y tecnologías

En este apartado, detallaremos las herramientas y tecnologías empleadas en el desarrollo de las aplicaciones.

API de reflexión de Java [4]
La API de reflexión de Java permite acceder, en tiempo de ejecución, a la metainformación de cada clase, atributo y método disponible. Partimos de que todas clases de Java heredan de la clase `java.lang.Object` y que, por tanto, heredan un método `getClass` que retorna un objeto de la clase `Class` que contiene información sobre la clase de la cual un objeto es instancia en tiempo de ejecución. También es posible instanciar un objeto `Class` sabiendo únicamente el nombre calificativo de la clase mediante el método `Class.forName("...")`.

Gracias a la reflexión, aunque desconozcamos la clase instanciada por un objeto, podemos averiguar si esa clase tiene un determinado método e invocarlo, si lo tiene, de la siguiente manera:

```
Method method = foo.getClass().getMethod("hazAlgo”);
method.invoke(foo);
```

El primer parámetro pasado a `method.invoke` es el objeto sobre el que se realizará la llamada al método `hazAlgo` (en nuestro caso, `foo`). Además, a `method.invoke` se le pueden pasar los parámetros que se desean pasar al método `foo`.

Nosotros hemos usado la reflexión en conjunto con las anotaciones. Mediante reflexión se buscarán, dentro de una clase, aquellos métodos que tengan una determinada anotación y será posible invocarlos.

También hacemos uso de la reflexión en la herramienta del comprobador, concretamente en la pestaña de requisitos y en la de validación, para mostrar mensajes más precisos y descriptivos de las funciones y del resultado de ejecutarlas.

JavaFX [2]
JavaFX es la plataforma para la creación de aplicaciones de escritorio y Rich Internet Applications (RIAs) que hemos empleado para desarrollar el aspecto visual de nuestras aplicaciones.

A pesar de contar con experiencia en el uso de Swing como librería para desarrollar interfaces gráficas de usuario, decidimos optar por usar JavaFX primordialmente porque se espera que Swing se deje de mantener y que JavaFX sea su sucesor. Es especialmente interesante usar tecnologías más nuevas para permitir la extensibilidad y continuidad del proyecto.

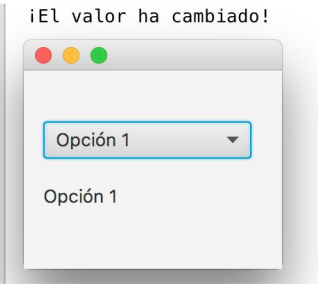
Asimismo, JavaFX aporta diversas ventajas frente a Swing como su uso de propiedades y vinculación en la gestión de eventos, el soporte para CSS y la herramienta de diseño Scene Builder.

JavaFX hace un uso extenso de las propiedades, que son variables cuyo valor se puede observar. A través de propiedades se pueden gestionar casi todas las características y elementos de una interfaz gráfica. Además, a estas propiedades se les puede registrar *listeners* para ejecutar código cuando la propiedad cambie, lo que hace que la gestión de eventos en JavaFX sea muy potente. Otra funcionalidad relacionada con las propiedades que ofrece JavaFX es la posibilidad de poder vincular una propiedad a otra propiedad, de forma que si una cambia la otra cambia automáticamente. Esta funcionalidad es conocida como Binding. En nuestra herramienta para la creación de documentos de requisitos usamos Binding para que al añadir nuevos plugins, se actualice en todos los elementos que sea necesario las entidades, funciones de aplicación y predicados disponibles.

En el siguiente ejemplo, vemos cómo se pueden vincular propiedades, así como registrar un *listener* a una propiedad. El texto del componente `Combobox` y del componente `Label` están vinculados, de modo que cuando el primero cambia, el segundo también lo hace. Además, mediante `'addListener'` capturamos este evento para mostrar un mensaje por la consola.

```
Bindings.bindBidirectional(combo.valueProperty(),label.textProperty());

combo.valueProperty().addListener(new ChangeListener<String>() {
    @Override
    public void changed(ObservableValue<? extends String> observable,
            String oldValue, String newValue) {
        System.out.println("¡El valor ha cambiado!");
    }
});
```



En JavaFX se pueden añadir hojas de estilo CSS permitiendo formatear cada aspecto visual de la interfaz. Al poder añadir varias hojas de estilo, se puede permitir que el usuario elija el estilo que más se adecue a sus necesidades fácilmente. Esto resulta especialmente interesante para poder adaptar la interfaz a personas con discapacidades visuales y hacer la aplicación más accesible.

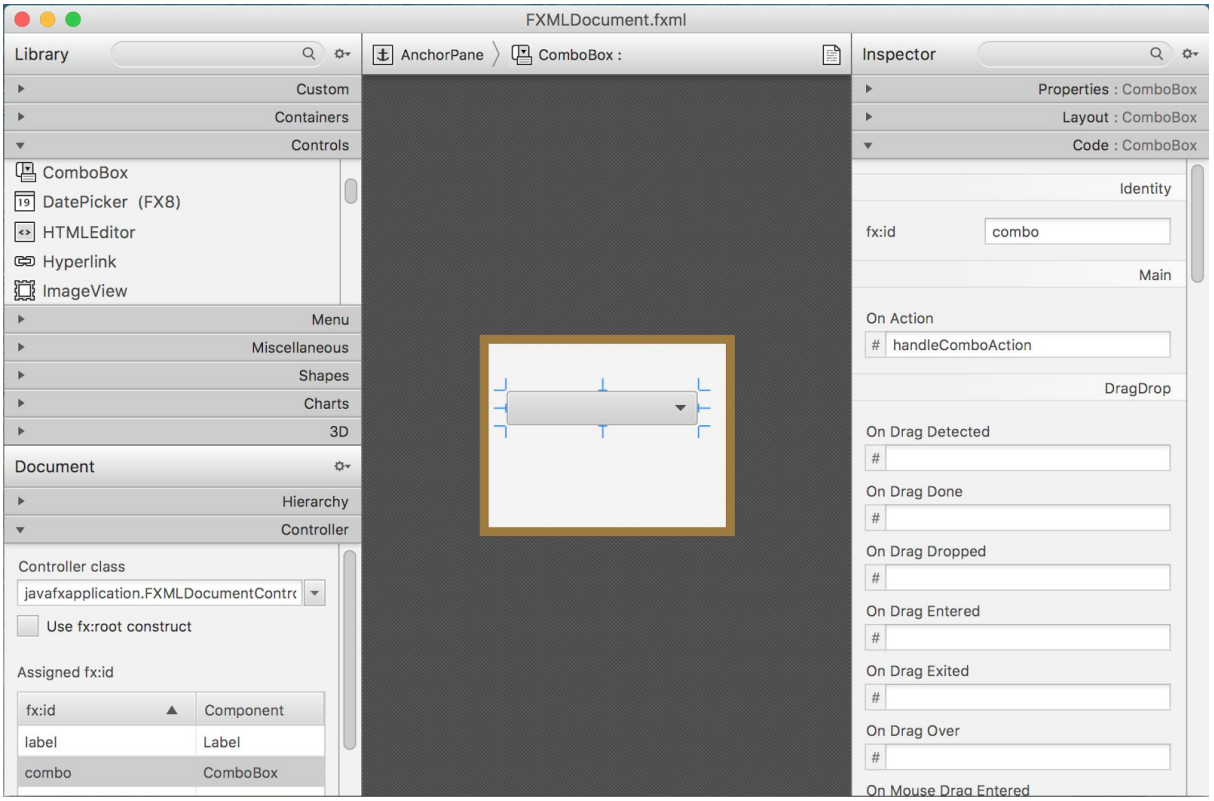
Scene Builder [6]

Scene Builder es una herramienta de JavaFX para el diseño de interfaces de usuario que permite un prototipado rápido de forma sencilla y que asegura la separación de la vista y la lógica de una aplicación.

Arrastrando y soltando componentes visuales en la ventana es posible crear interfaces sin la necesidad de escribir código. Además es posible modificar las propiedades de dichos componentes y especificar los métodos, definidos en el controlador, que gestionarán los eventos que sean lanzados. También es posible añadir hojas de estilo usando la herramienta. El resultado final es un fichero FXML que se genera automáticamente y que contiene toda la especificación de la interfaz.

En la siguiente imagen podemos ver la interfaz de la herramienta Scene Builder. En el ejemplo, nuestra GUI tiene dos componentes: un `Label` y un `ComboBox`. Se puede observar que le hemos asociado un `id` al elemento `ComboBox` y hemos definido cómo se llamará el

método que gestionará su evento `On Action`. Este método estará definido en el controlador de la GUI, que está especificado en la zona inferior izquierda.



XML [7]

XML es un lenguaje que permite almacenar datos de forma comprensible mediante el lenguaje de marcas, definiendo una serie de reglas. Lo hemos utilizado para definir una gramática de lenguaje para la especificación de requisitos que empleamos para almacenar y leer documentos de especificaciones.

XML favorece la extensibilidad, lo que supone que si nuestro lenguaje de especificación de requisitos sufriera cambios se podrían añadir nuevas etiquetas a nuestro lenguaje sin mucho esfuerzo y sin afectar a las etiquetas ya existentes.

Durante el grado hemos trabajado puntualmente con XML pero nunca habíamos utilizado el estándar DOM de W3C [9] ni las utilidades que posee su API en Java para trabajar con ficheros XML. El estándar DOM consiste en modelizar los ficheros XML como estructuras arbóreas, pudiendo recorrer los árboles fácilmente y realizar búsquedas de nodos u obtener colecciones de nodos concretos por su etiqueta.

Por ejemplo, hemos especificado que los plugins empleados para la validación de requisitos se definen de la siguiente manera:

```
<import>plugin</import>
```

En nuestro lector de especificaciones realizamos una búsqueda por la etiqueta `import`.

NetBeans [10]
Netbeans es un entorno de desarrollo libre con soporte para Java 8 que está construido de manera modular. Para el desarrollo de aplicaciones Java ya están todos los módulos necesarios instalados por defecto, al contrario a lo que ocurre con el entorno de desarrollo Eclipse donde a menudo es el usuario quien tiene que instalar los plugins deseados, lo que hace que la puesta en marcha inicial sea lenta. Hemos utilizado Netbeans y sus plugins de integración con GitHub como sistema de control de versiones y con Maven para la gestión y construcción de proyectos.

Git y Github [11]
Git es una herramienta para el control de versiones. Github es una aplicación web que permite almacenar un repositorio Git en la nube. En conjunto permiten a desarrolladores descargarse el repositorio en una máquina local y descargar y subir cambios a ese repositorio, acciones conocidas como *pull* y *push*. Además permite realizar bifurcaciones (*branches*) que luego se pueden incorporar al proyecto principal, mediante la acción *merge*. Nos ha resultado fundamental su uso para poder desarrollar en paralelo y gestionar los diferentes proyectos de PoVALE.

Por otra parte, Github es una herramienta muy útil para exhibir nuestro código y fomentar las contribuciones de otros desarrolladores a él en un futuro.

Es posible acceder a nuestros repositorios en: https://github.com/PoVALE/.

Maven [12]
Maven es una herramienta para realizar la construcción de proyectos de forma sencilla y automática al definir cómo están construidos y qué dependencias tienen. Usa el archivo `pom.xml`, en el que se describe el proyecto software que se construirá, cómo se empaquetará y de qué librerías o proyectos depende. Para nosotros es especialmente útil está capacidad, al estar los proyectos de `PoVALE` separados por funcionalidad, y de esta forma se pueden añadir como dependencias aquellos que sean necesarios, de forma modular.

Cada proyecto Maven queda identificado unívocamente por tres valores: `groupId`, `artifactId` y `version`. Todos los proyectos de `PoVALE` forman parte del groupId: "`es.ucm.povale`".

A continuación vemos el archivo pom.xml del proyecto `PoVALE-reader`, donde primero aparecen datos sobre el proyecto: a qué grupo pertenece (`groupId`), el nombre del proyecto (`artifactId`), su versión y cómo se empaqueta (en nuestro caso, como un fichero `.jar`). `PoVALE-reader` analiza el contenido de un fichero XML haciendo uso de las clases de `PoVALE-core` y, por tanto, `PoVALE-core` aparece como dependencia.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3
    <modelVersion>4.0.0</modelVersion>
    <groupId>es.ucm.povale</groupId>
    <artifactId>PoVALE-reader</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>es.ucm.povale</groupId>
            <artifactId>PoVALE-core</artifactId>
            <version>1.0-SNAPSHOT</version>
            <type>jar</type>
        </dependency>
    </dependencies>
</project>
```

4. Lenguaje de especificación de requisitos

En esta sección explicaremos el lenguaje que usamos para especificar los requisitos que una entrega debe cumplir.

1. Lenguaje de especificación

El lenguaje que utilizaremos para expresar la especificación de requisitos de las entregas se asemeja a la lógica de primer orden y está diseñado con el objetivo de que sea extensible mediante plugins. A través de plugins se pueden extender el conjunto de tipos de entidad soportados, así como el conjunto de funciones y predicados disponibles.

Entidades
Una entidad es algo sobre lo que se pueden poner restricciones. Los tipos de entidad básicos son `IntegerEntity` para la representación de enteros, `StringEntity` para la representación de cadenas, y `ListEntity` para representar una lista de entidades.
Mediante el plugin *Plugin-Files* desarrollado por Manuel Montenegro y completado por nosotros, se añaden los tipos de entidad, `FileEntity` y `DirectoryEntity`, que representan ficheros y directorios, respectivamente. Sobre un fichero se pueden poner restricciones, como por ejemplo, exigir que tenga un nombre determinado o una extensión. Asimismo, sobre un directorio, además de poder exigir que tenga un nombre determinado, se pueden especificar restricciones sobre los ficheros y directorios comprendidos dentro de él.

El plugin *Plugin-Lines* desarrollado por Manuel Montenegro utiliza los tipos de entidad `FileEntity`, `DirectoryEntity` y define un tipo de entidad nuevo: `MatchResult`. Permite inspeccionar y comparar líneas dentro de ficheros. De este modo se permite, por ejemplo, especificar que un fichero tenga una línea que contenga determinada cadena, o que se ajuste a una determinada expresión regular.

Un plugin puede añadir nuevos tipos de entidad, implementando la interfaz `Entity` y sus métodos abstractos. El método `getType` permite hacer comprobaciones de tipo cuando se pasan entidades como parámetros a funciones.

```
public interface Entity {

    default Class<? extends Entity> getType(){
        return this.getClass();
    }

    public abstract void toXML(Element contents, Document doc);

    public abstract void writeToZip
        (ZipOutputStream z, String outputFile) throws IOException;
}
```

Términos
Los términos sirven para denotar una entidad. Pueden ser:

- Entidades directamente (literales). Por ejemplo, una cadena representada con `StringEntity`.
- Variables.
- La aplicación de una función a uno o varios términos: $f(t_1, ..., t_n)$.
- Una colección de términos: $(t_1, ..., t_n)$.

Funciones
Una función recibe uno o varios términos como parámetros y produce una o varias entidades. Por ejemplo, la función `children`, dentro del plugin de ficheros, recibe una entidad de tipo `DirectoryEntity` que denota un directorio y devuelve una `ListEntity` compuesta por `FileEntity` y `DirectoryEntity`. Es decir, devuelve una lista con los ficheros y directorios contenidos dentro del directorio indicado.

Para ampliar el conjunto de funciones mediante plugins, es necesario que las funciones extiendan la clase abstracta `Function`.

Variables
Una variable es un tipo de término que, en un determinado momento de la comprobación de una restricción, hará referencia a una entidad.

En el anterior ejemplo, nuestra función `children` podría haber recibido por parámetro una entidad de tipo `variable`, representado el directorio que escogerá el alumno al iniciar la aplicación y seleccionar las variables de entorno. Una vez elegido el directorio, la variable denota una entidad de tipo `DirectoryEntity`.

Asertos
Los requisitos de una entrega se representan mediante asertos. Un aserto representa una restricción sobre una o varias entidades. Tenemos los siguientes tipos de asertos, donde t hace referencia a un término y φ hace referencia a un aserto:

- Verdad lógica: $true$
- Falsedad lógica: $false$
- Negación: $\neg\varphi$
- Igualdad: $t_1 = t_2$
- Conjunción: $\varphi_1 \wedge ... \wedge \varphi_n$
- Disyunción: $\varphi_1 \vee ... \vee \varphi_n$
- Implicación: $\varphi_1 \rightarrow \varphi_2$
- Cuantificador universal: $\forall X \in t.\varphi$
- Cuantificador existencial: $\exists X \in t.\varphi$

- Cuantificador existencial de unicidad: $\exists !X \in t.\varphi$
- Aplicación de predicado: $P(t_1, ..., t_n)$

Por ejemplo, suponemos una práctica de programación donde la variable ROOT hace referencia al directorio que entrega el estudiante. Se podría imponer la siguiente restricción:

El directorio se llama "src" y contiene un único fichero llamado "Main"

Que se representa mediante el siguiente aserto:

$$baseName(ROOT) = "src" \ \wedge \ \exists !X \in children(ROOT).baseName(X) = "Main"$$

Predicados

Una aplicación de predicado es un tipo de aserto compuesto por un símbolo de relación y por uno o varios términos como parámetros. Una relación puede satisfacerse o no según el valor de los argumentos que recibe.

Por ejemplo, la relación `is-directory?` recibe una entidad de tipo fichero y determina si es un directorio o no.

Un plugin puede añadir nuevos predicados extendiendo la clase abstracta `Predicate` e implementado sus métodos abstractos `getName` y `getMessage` que sirven para poder dar realimentación precisa al usuario.

```
public abstract class Predicate extends DynamicallyCallable<Entity,Boolean>{
    @Override
    public Boolean call(Entity... params) {
        return super.call(params);
    }

    public abstract String getName();

    public abstract String getMessage();
}
```

Al aserto usado en el anterior ejemplo se le podría añadir la restricción de que la entrega fuese un directorio. Este requisito se representaría de la forma: $isDirectory?(ROOT)$.

5. Implementación del comprobador

Está herramienta tiene como objetivo comprobar que el ejercicio entregado por el estudiante respeta unas especificaciones de entrega. El profesor define los requisitos mediante la herramienta de creación de documentos de especificación que detallaremos más adelante.

Los usuarios de esta herramienta son los estudiantes y dicha herramienta se ha diseñado con la intención de que les proporcione una retroalimentación detallada para que puedan corregir sus entregas si fuese necesario.

En primer lugar, mediante el proyecto `PoVALE-reader` se lee y analiza el documento XML con los requisitos y se cargan las variables, los plugins y los asertos a validar. En segundo lugar mediante el proyecto `PoVALE-core` se comprueba si se cumplen o no los las especificaciones de entrega.

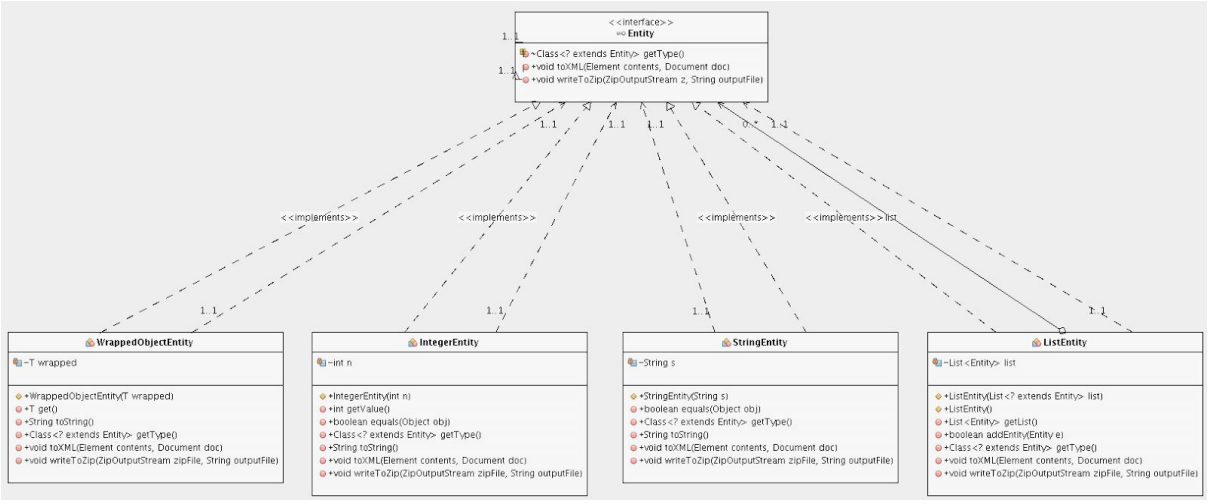
1. PoVALE-core

A continuación vamos a explicar la funcionalidad que aporta el proyecto `PoVALE-core`. Este proyecto es el encargado de realizar la validación de las restricciones previamente cargadas.

1.1. Estructura de clases

Este proyecto está dividido en paquetes que agrupan clases según su funcionalidad.

Entidades
Como hemos mencionado anteriormente una entidad es algo que puede ser sometido a restricciones. Este proyecto contiene tres tipos de entidades básicas: la de enteros (`IntegerEntity`), la de cadenas (`StringEntity`), y la lista de entidades (`ListEntity`). Además tenemos la clase `WrappedObjectEntity` que permite hacer uso, mediante reflexión, de nuevos tipos de entidades. Cualquier entidad que se quiera extender a la aplicación debe implementar a la interfaz `Entity`.



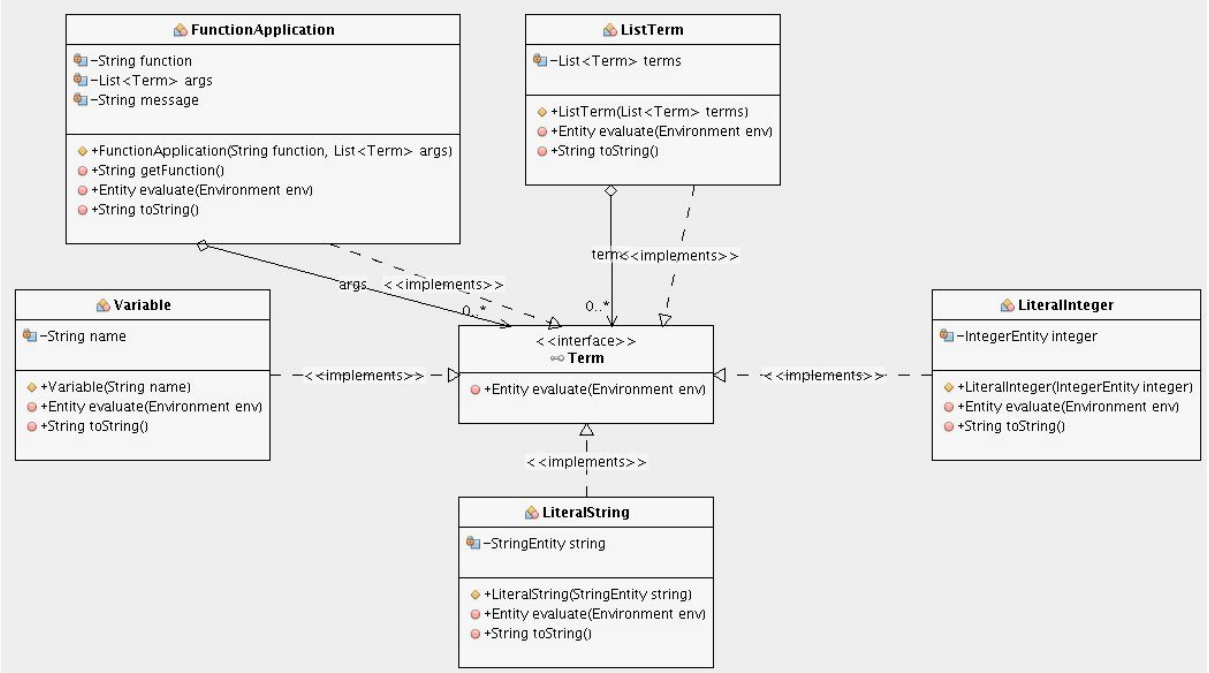
Entorno
En la clase `Environment` guardamos las variables que están en ámbito durante el proceso de validación. Además de cargar en ella los plugins que sean importados y así poder hacer uso de sus funciones y predicados, también almacena las entidades nuevas que proporcionen los plugins importados.

Variables
Representadas en la clase `Var`, son las variables cuyos valores serán determinados por el estudiante a la hora de comprobar que su entrega es correcta, para después realizar la validación. Dependiendo del tipo de entidad al que pertenezcan, las variables tendrán una representación visual u otra en la vista del validador (proyecto `PoVALE-view`) ya que cada tipo de dato requerirá un método de entrada diferente para ser tratado correctamente. Por ejemplo, para que el alumno precise cuál es su número de grupo, que sería una variable de tipo `IntegerEntity`, se le mostraría un cuadro de texto. En cambio, con una variable para especificar el fichero a entregar, de tipo `FileEntity`, al alumno se le mostraría un selector de ficheros.

Términos
Todos los términos implementan la interfaz `Term` . En este paquete se encuentran las clases de todos los términos que ya hemos mencionado anteriormente: `LiteralInteger`, `LiteralString`, `Variable`, `FunctionApplication`, y `ListTerm`. Todas tienen un método para evaluar el término y otro para mostrar su contenido evaluado, que se añadirá dentro del mensaje del aserto que lo contenga y proporcionará la retroalimentación necesaria a los alumnos en la validación.

La evaluación de los términos devuelve una entidad o una lista de entidades.
En los términos que representan a cadenas y enteros, devuelven las entidades que representan a estos dos tipos. La evaluación de una variable involucra la búsqueda de la misma en el entorno (clase `Environment`), y la devolución del valor al que va asociada.
Para evaluar una lista de términos, se evalúa cada término que contenga dicha lista y se agrupan los resultados en una lista de entidades.

Las funciones de evaluación evalúan la función de un plugin que se haya especificado en ella mediante reflexión. Devolviendo una entidad concreta o una lista de entidades, dependiendo de la función.
Por último, para evaluar una aplicación de un símbolo de función, se busca en el entorno la implementación de dicho símbolo (el plugin habrá introducido previamente dicha implementación en el entorno). Se evalúan, por otro lado, cada uno de los argumentos a los que la función está aplicada, obteniendo una serie de entidades, y se realiza una llamada a la implementación de la función con dichas entidades. La entidad devuelta por esta función será el resultado del término.
Por ejemplo, de las funciones `Name` que devuelve el nombre de un fichero o directorio y `Children` que devuelve una lista de los hijos que contiene un directorio



Asertos

Cada aserto especificado en nuestro lenguaje de especificaciones tiene asociado una clase que implementa la interfaz `Assertion`. Para implementar dicha clase es necesario desarrollar el método `check(Environment env)` que devuelve un objeto de la clase `AssertInformation`. Este objeto contiene toda la información relacionada con la evaluación del aserto y detallaremos su funcionalidad en el siguiente apartado. De forma análoga a los elementos que contienen los asertos en lenguaje XML, las clases de los asertos contienen ciertos atributos. Por ejemplo, el aserto `ForAll`, que representa el cuantificador universal, tiene como atributos un atributo de tipo `String` para el nombre de la variable ligada, un atributo de tipo `Term` para el término que denota el conjunto en el que tomará valores la variable ligada, y un atributo de tipo `Assertion` que representa el aserto que especifica los requisitos que deben cumplir todas entidades.

Todos los asertos tienen un atributo de tipo `String` para el mensaje que proporcionará al usuario cuando se valide el aserto. Si en la especificación de requisitos se ha incluido un mensaje específico se mostrará ese. Si no, se mostrará un mensaje por defecto para ese tipo de aserto.

El diagrama de clases del paquete *assertion* se encuentra en el Capítulo 14.

Assert Information

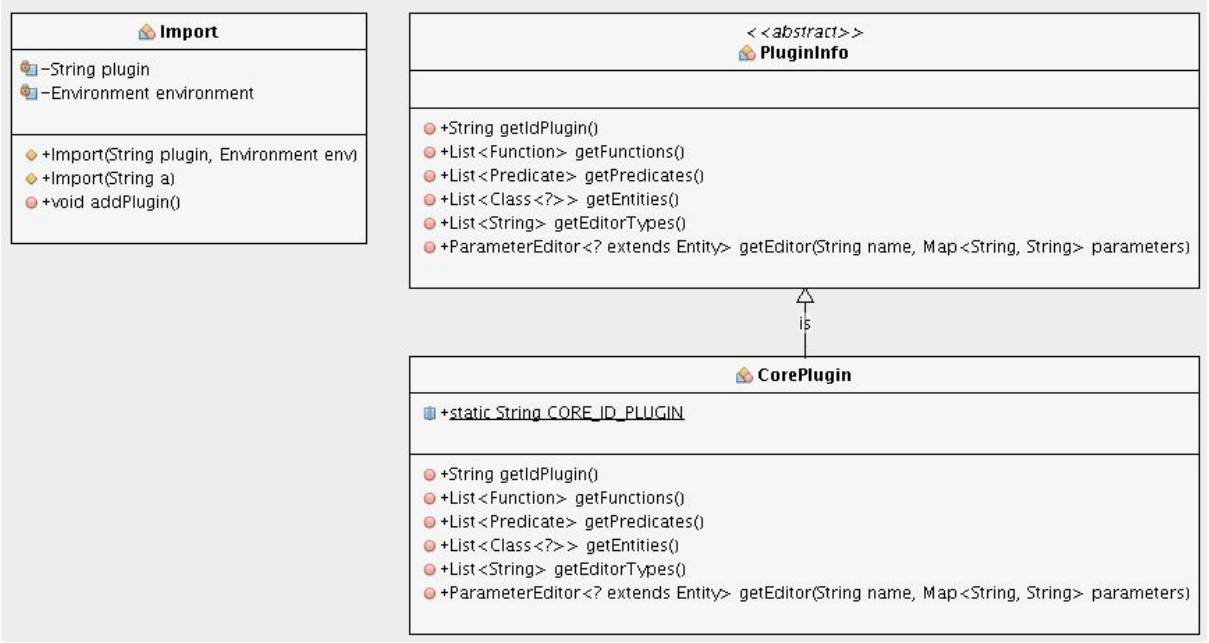
Como mencionamos anteriormente, implementamos `AssertNode` en el paquete `PoVALE-reader` para ofrecer información específica a los alumnos sobre los requisitos de entrega. La aplicación necesitaba dar también retroalimentación sobre los resultados de validar la entrega, por lo que creamos `AssertInformation`. Esta clase funciona como un nodo arbóreo y representa el resultado de comprobar la validez de un aserto. Contiene el mensaje que se mostrará para ese aserto, un booleano con el resultado de validación del aserto y una lista de `AssertInformation` que representa los resultados de evaluar los

asertos hijo que contiene el aserto (en el caso del que este contenga asertos). Por ejemplo, el aserto de tipo conjunción posee un conjunto de asertos, que son aquellos sobre los que se realiza la conjunción.

Plugins

Nuestra aplicación está diseñada para que se puedan agregar plugins a ella que amplíen su funcionalidad. Para crear un plugin es necesario crear una clase que extienda a la clase abstracta `PluginInfo`. En el apéndice *Apéndice: ¿Cómo crear un plugin en PoVALE?* se detallan los pasos necesarios para crear un plugin.

Cuando se realiza la acción de importar, mediante el método `import` de la clase del mismo nombre, la única información de la que se dispone es el entorno donde se deben cargar los nuevos datos y la cadena que identifica inequívocamente a un plugin. Este método llama a las funciones `getFunctions`, `getPredicates`, `getEntities` y `getEditorTypes`.

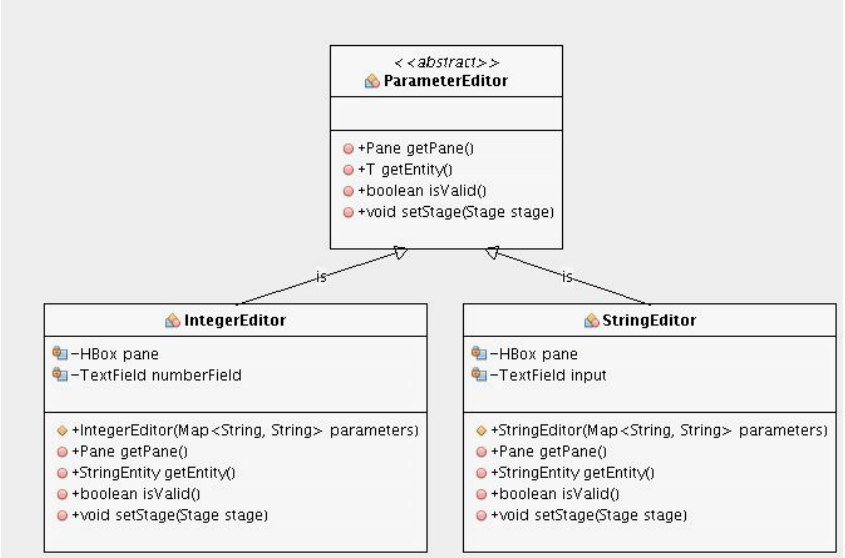


Parameter Editor

La clase `ParameterEditor` es una clase abstracta parametrizada con un tipo de entidad. Aquellas entidades que vayan a ser utilizadas como variables para que el usuario introduzca sus valores, deberán tener su correspondiente clase que extienda a `ParameterEditor`. En `PoVALE-core` están las clases que extienden a `ParameterEditor` para representar variables que notan enteros (`IntegerEditor`) y cadenas (`StringEditor`). Cada una de las clases que extiendan a ParameterEditor debe implementar el método `getPane`, que devolverá el panel JavaFX asociado al editor y que se colocará en la ventana de `PoVALE-view`, permitiendo al usuario introducir un valor que se asociará a la variable.

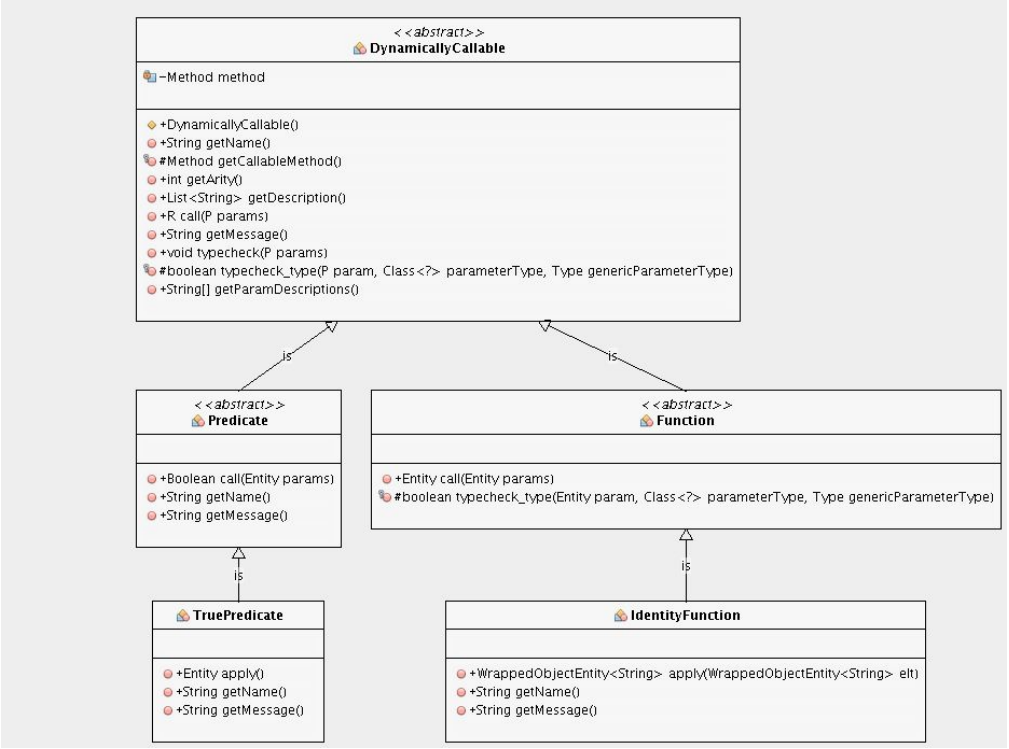
Para el plugin `PoVALE-plugin-files` hemos desarrollado las clases `FileEditor` y `DirectoryEditor`, que extienden a `ParameterEditor` y que permiten al usuario

seleccionar un fichero y un directorio respectivamente, para asignar una entidad de tipo `FileEntity` o `DirectoryEntity` a una variable.



Function, Predicate, Internal, Annotation

En las clases `Function`, `Predicate`, `Internal` y `Annotation` se hace uso de la reflexión de Java para poder invocar en tiempo de ejecución a funciones de aplicación y predicados desconociendo la clase y método al que habrá que llamar en tiempo de compilación.



A continuación, usando un ejemplo, vamos a explicar cómo utilizamos la reflexión de Java para poder llamar en tiempo de ejecución a métodos de los plugins y así poder comprobar los asertos que hagan uso de las funciones de aplicación o predicados de los plugins.

Para nuestro ejemplo, suponemos que la variable ROOT hace referencia al directorio principal del proyecto que entrega el estudiante. La restricción de entrega es que ROOT sea un directorio, es decir: $directory(ROOT) = true$.

Para comprobar este requisito haríamos uso del plugin `PoVALE-plugin-files` y su función de predicado `is-directory?`.

En el proceso de validación, el primer paso es llamar al método `check` del aserto, en este caso el de `PredicateApplication`. Dentro de este método se realiza lo siguiente:

```
if (!p.call(list.toArray(new Entity[list.size()]))) {
    result = false;
}
```

donde `p` es en nuestro caso de tipo `IsDirectory`, clase que extiende a la clase abstracta `Predicate`.

El método `call` tiene como parámetro una lista de entidades, en este caso contendrá lo que pasa el alumno para comprobar si es un directorio.

El método `call`, que se encuentra en la clase abstracta `Predicate`, hace lo siguiente:

```
@Override
public Boolean call(Entity... params) {
    return super.call(params);
}
```

La clase `Predicate` extiende a `DynamicallyCallable`. Esta clase, creada por Santiago Saavedra, tiene un atributo privado de tipo `Method` que se inicializa en su constructor llamando al método `getCallableMethod` que hace lo siguiente:

```
Method current = null;
for (Method m : getClass().getMethods()) {
    if (m.getAnnotation(CallableMethod.class) != null) {
        if (current != null) {
            // More than one method with the annotation!
            throw new RuntimeException(
                    String.format(
                            "More than one CallableMethod found in " +
                                    "class %s",
                            getClass()));
        }
        current = m;
    }
}
```

Donde con `m.getAnnotation(CallableMethod.class)` se buscará a un método con la anotación `Callable` y este será el que se asigne al atributo de tipo `Method` de la clase `DynamicallyCallable` si existe y no hay más de uno.

Con `method` inicializado es posible realizar la llamada a `call` de `DynamicallyCallable`, a la que llama `Predicate` en su propio método `call`:

```
public R call(P... params) {
    try {
        typecheck(params);
        return (R) method.invoke(this, (Object[]) params);
    } catch (IllegalAccessException | InvocationTargetException e) {
        throw new RuntimeException(e);
    }
}
```

Donde vemos que mediante `method.invoke(this, (Object[])params)` llamará en nuestro caso al `CallableMethod` de la clase `IsDirectory`:

```
@CallableMethod
public boolean isDirectory(@ParamDescription("es un directorio ") FileEntity f) {
    return f instanceof DirectoryEntity;
}
```

A través de estos pasos y gracias a la reflexión, aunque en tiempo de compilación se desconozca la clase exacta, que extiende a `Predicate`, y el método al que tendremos que llamar tras nuestra llamada a `call` para resolver el aserto, podemos realizar la validación.

2. PoVALE-reader

A continuación vamos a explicar la funcionalidad que aporta el proyecto `PoVALE-reader`. Primero describiremos cómo se expresan los elementos que componen nuestro lenguaje de especificación, detallado en la sección anterior, mediante XML.

2.1. Lenguaje de especificación mediante XML

Toda la información a leer se encuentra entre las etiquetas `<spec></spec>`. Los elementos que leemos son plugins, variables, asertos y términos.

Plugins

Cada plugin a leer debe estar entre las etiquetas `<import></import>` y en el interior de esas etiquetas debe encontrarse el nombre (incluyendo paquete) de la clase principal del plugin, de la que crearemos una instancia mediante reflexión. Más adelante en este capítulo se describirá la estructura de la clase principal de un plugin.
Por ejemplo:
`<import>es.ucm.povaleFiles.FilesPlugin</import>`
Donde `FilesPlugin.java` es la clase principal del plugin `PoVALE-plugin-files`.

Variables

Estas son las variables cuyo valor especifica el alumno al iniciar la aplicación `PoVALE-view`. Por ejemplo, el fichero o directorio a entregar, nombre de los alumnos, o número de grupo. Son representadas mediante la clase `Var` del proyecto `PoVALE-core`.

Las variables se encuentran entre las etiquetas `<var></var>`. A continuación mostraremos y explicaremos un ejemplo de una variable:

```
<var>
      <label>Identificador</label>
      <name>nombre_grupo</name>
      <desc>Formato: LXXGXX </desc>
      <type>StringEntity</type>
</var>
```

Donde:

- `<label>` se corresponde con la etiqueta que aparecerá al lado del campo de texto que rellenará el usuario (en la herramienta de `PoVALE-view`).

- `<name>` es el nombre con el que se introducirá la variable en el entorno.

- `<desc>` se corresponde con la descripción opcional que podrá ver el usuario.

- `<type>` Determina de qué tipo es el dato a rellenar por el usuario. Se cargará un `ParameterEditor` acorde al tipo de entidad señalada. Explicaremos esto más adelante.

Términos

Aparecen siempre en el interior de algunos asertos. Según el tipo de término, su representación en XML es:

- **Cadenas:**
  `<literalString>string concreto</literalString>`

- **Enteros:**
  `<literalInteger>integer concreto</literalInteger>`

- **Variables:**
  `<variable>nombre de la variable</variable>`

- **Lista de términos:**
  Se compone por la etiqueta `<listTerm>` y en su interior se introducen los términos que vayan a pertenecer a la lista. Abajo podemos ver cómo se expresaría una lista de términos formada por dos `LiteralString`.

  ```
  <listTerm>
        <literalString>PoVALE</literalString>
        <literalString>example</literalString>
  </listTerm>
  ```

- **Aplicaciones de función**:
  Se representan mediante la etiqueta `<functionApplication>`, y en su interior se encuentra primero la etiqueta `<function>` en la que se encuentra el nombre de una

función de un plugin. En segundo lugar se encuentran un término o una lista de términos, que son los argumentos pasados a la función.
A continuación vemos la representación de la función de aplicación `base-name` del plugin `PoVALE-plugin-files`, que devuelve el nombre de un fichero denotado por la variable “x”.

```
<functionApplication>
      <function>base-name</function>
      <variable>x</variable>
</functionApplication>
```

Asertos

Todos los asertos se encuentran bajo una única etiqueta: `<assertion>`. Cada aserto principal se encuentra bajo la etiqueta: `<assert>`. La herramienta verificará que se cumplen todos estos asertos principales. Los términos usados en los asertos se representan como hemos mostrado en la sección anterior.

Cada aserto tiene un mensaje por defecto que se muestra durante la validación en el proyecto `PoVALE-view`. Si se desea utilizar un mensaje distinto en algún aserto es posible añadiendo dentro de la etiqueta del aserto que queramos el atributo `msg =”mensaje personalizado”`.
Por ejemplo si queremos modificar el mensaje de una verdad lógica:

```
<assertTrue msg="Siempre se cumple este aserto"></assertTrue>
```

A continuación mostraremos cómo se representa cada aserto:

- **Verdad y falsedad lógica**:
  Son elementos vacíos y nos han servido para realizar pruebas.
  `<assertTrue/>` y `<assertFalse/>`

- **Negación:**
  Su etiqueta es `<not>` y en su interior se encuentra el aserto que se niega, por ejemplo negar una falsedad lógica en el lenguaje XML se representa:

  ```
  <not>
        <assertFalse/>
  </not>
  ```

- **Igualdad:**
  Su etiqueta es `<equals>` y en su interior se encuentran dos etiquetas, primero la etiqueta
  `<lhs>` que representa el operando izquierdo de la igualdad y que contiene un término. La segunda etiqueta que se encuentra en la igualdad es `<rhs>` que representa al operando derecho y que contiene otro término. Por ejemplo, la representación de la igualdad `x = 1` es:

  ```
  <equals>
          <lhs>
              <variable>x</variable>
          </lhs>
  ```

```
        <rhs>
            <literalInteger>1</literalInteger>
        </rhs>
</equals>
```

- **Conjunción (AND) y disyunción (OR):**
  La conjunción se representa con la etiqueta `<and>` y la disyunción con la etiqueta `<or>`. En su interior se encuentran los asertos que pertenecen al aserto. La conjunción de una verdad y una falsedad lógica se presenta de la forma:

```
<and>
      <assertTrue/>
      <assertFalse/>
</and>
```

- **Implicación:**
  Se representa con la etiqueta `<entail>` y en su interior alberga dos etiquetas: `<lhs>`, que contiene en su interior la representación del aserto que actúa como operando izquierdo en la implicación y la etiqueta `<rhs>`, que contiene al aserto operando derecho.
  Ejemplo que representa la implicación si `x = 1` entonces `true` (verdad lógica):

```
<entail>
      <lhs>
            <equals>
                  <lhs>
                        <variable>x</variable>
                  </lhs>
                  <rhs>
                        <literalInteger>1</literalInteger>
                  </rhs>
            </equals>
       </lhs>
       <rhs>
            <assertTrue/>
       </rhs>
</entail>
```

- **Cuantificadores universal, existencial y existencial único:**
  El cuantificador existencial se representa mediante la etiqueta `<exist>`, el existencial de unicidad con la etiqueta `<existOne>` y el universal con `<forAll>`. Los tres tipos de aserto contienen en su interior las mismas etiquetas: En primer lugar la variable ligada por el cuantificador, que está situada dentro de la etiqueta `<variable>`. En segundo lugar un término que corresponde al conjunto al que pertenece la variable. Y en tercer lugar la condición que se tiene que cumplir para que el aserto sea correcto.
  A continuación mostramos la representación de “Existe X perteneciente a {1, 2 ,5} tal que X es igual a 6 o X es igual a 5”, en el que la variable ligada es X, el conjunto {1, 2, 5} sería una lista de términos donde cada término es un entero y la condición sería una disyunción de la igualdad X igual a 6 o X igual a 5.

```
<exist>
	<variable>X</variable>
	<listTerm>
		<literalInteger>1</literalInteger>
		<literalInteger>2</literalInteger>
		<literalInteger>5</literalInteger>
	</listTerm>
	<or>
		<equals>
			<lhs>
				<variable>x</variable>
			</lhs>
			<rhs>
				<literalInteger>6</literalInteger>
			</rhs>
		</equals>
		<equals>
			<lhs>
				<variable>x</variable>
			</lhs>
			<rhs>
				<literalInteger>5</literalInteger>
			</rhs>
		</equals>
	</or>
</exist>
```

- **Aplicaciones de predicado:**
  Se representan con la etiqueta `<predicateApplication>` y tienen un contenido similar a las del término función de aplicación. En primer lugar `<predicate>` contiene la palabra clave de un método de un plugin y en segundo lugar un término que puede ser una lista de términos en el caso de predicados con más de un parámetro.

```
<predicateApplication>
	<predicate>is-directory?</predicate>
	<variable>x</variable>
</predicateApplication>
```

2. 2. Estructura de clases

El proyecto `PoVALE-reader` está formado por las clases `AssertNode`, `AssertParser`, `TermParser` y `XMLParser`.

AssertNode: Inicialmente esta clase no existía porque creábamos directamente los asertos utilizando la clase `Assertion` de `PoVALE-core`. Para mejorar el sistema de retroalimentación con el usuario utilizado en `PoVALE-view`, donde se muestran en forma de árbol mensajes descriptivos de los requisitos a cumplir en cada aserto en la validación y en los resultados tras la validación, tuvimos que crear esta clase.

Cada `AssertNode` representa un aserto y contiene tres atributos: el aserto concreto (de tipo `Assertion`) al que representa, un String que contendrá el mensaje del aserto que se mostrará en los mensajes de requisitos de `PoVALE-view`. Y una lista de `AssertNode` que representa los asertos hijo que tiene el aserto (si tiene).

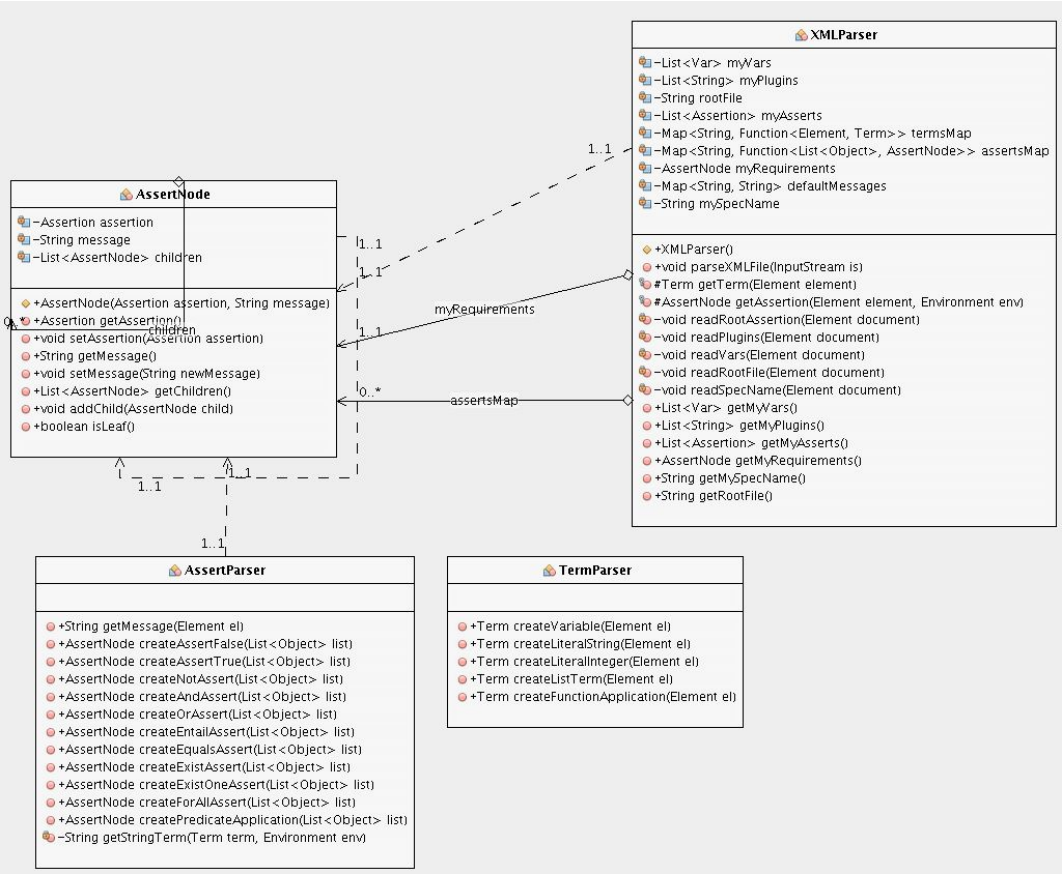
AssertParser: Contiene los métodos necesarios para leer y crear cada tipo de aserto. Inicialmente creaba asertos de tipo `Assertion` pero, como hemos explicado antes, al mejorar la calidad de la información mostrada en los mensajes de requisitos ahora devuelven un `AssertNode` con toda la información sobre el aserto y sobre sus hijos, en el caso de que este tuviera.

TermParser: Posee los métodos para analizar cada tipo de término. Estos métodos son utilizados dentro de la clase `AssertParser` cuando un aserto contiene términos. Los métodos devuelven un término concreto que extiende a la clase `Term` de `PoVALE-core`.

XMLParser: Es el núcleo del proyecto `PoVALE-reader`. Aquí cargamos los datos que luego utilizará `PoVALE-core` para validar los requisitos: asertos, plugins, variables. También cargamos una estructura de `AssertNode` utilizada en los mensajes de retroalimentación para el usuario del proyecto `PoVALE-view`.

Para hacer más rápida la lectura y creación de asertos, usamos dos hashmap (uno para términos y otro para asertos) que tienen como clave las etiquetas de los términos y asertos especificados en el lenguaje XML y como valor, los métodos de las clases `AssertParser` y `TermParser` que crean ese elemento concreto.

En este proyecto hemos hecho uso de la reflexión de Java para crear mensajes descriptivos con sentido en los asertos que utilizan aplicaciones de función y de predicados de los plugins. Estos mensajes son utilizados en la vista de la aplicación.

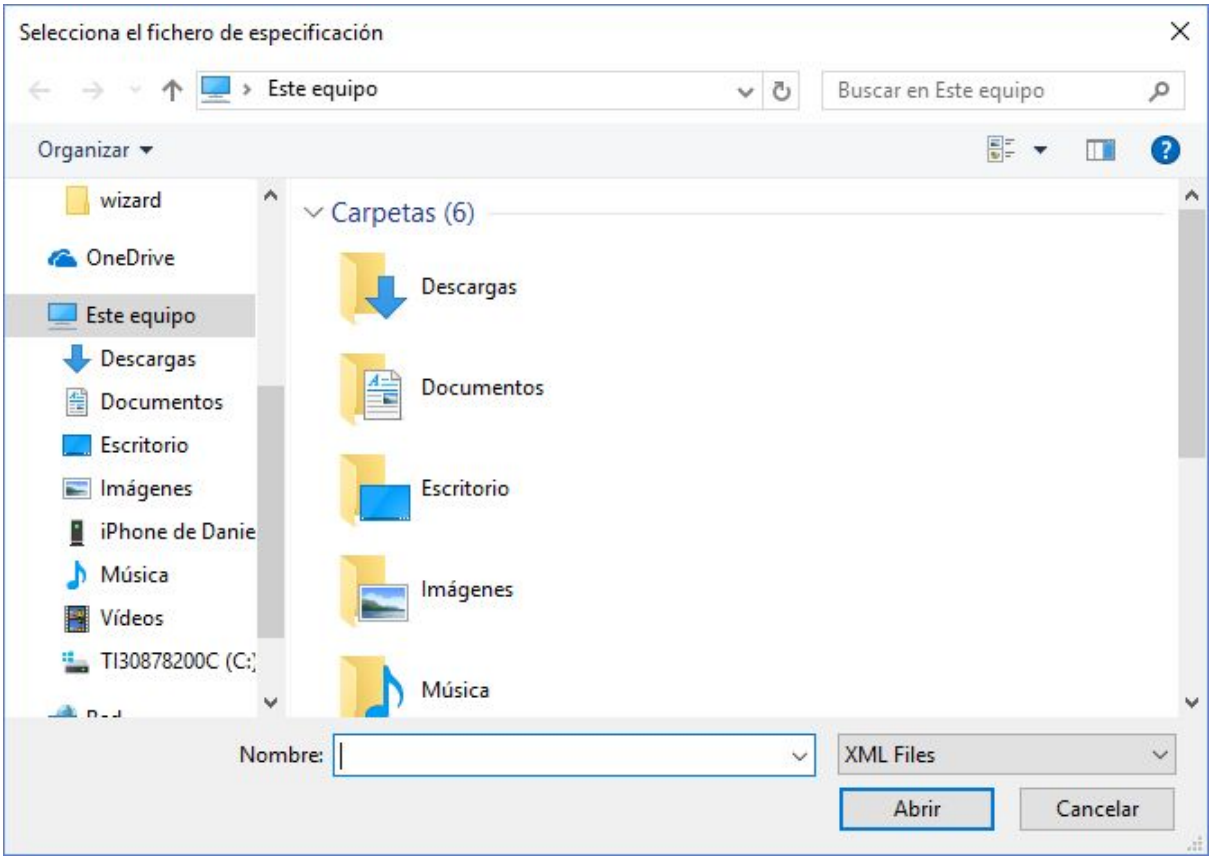


6. Implementación de la interfaz gráfica

El proyecto `PoVALE-view` desarrolla la interfaz gráfica de la herramienta del comprobador. Permite a los alumnos cumplimentar los valores de las variables, ver qué requisitos debe cumplir la entrega, validar si su entrega es correcta y finalmente poder guardar en un zip la entrega y los resultados de la validación.

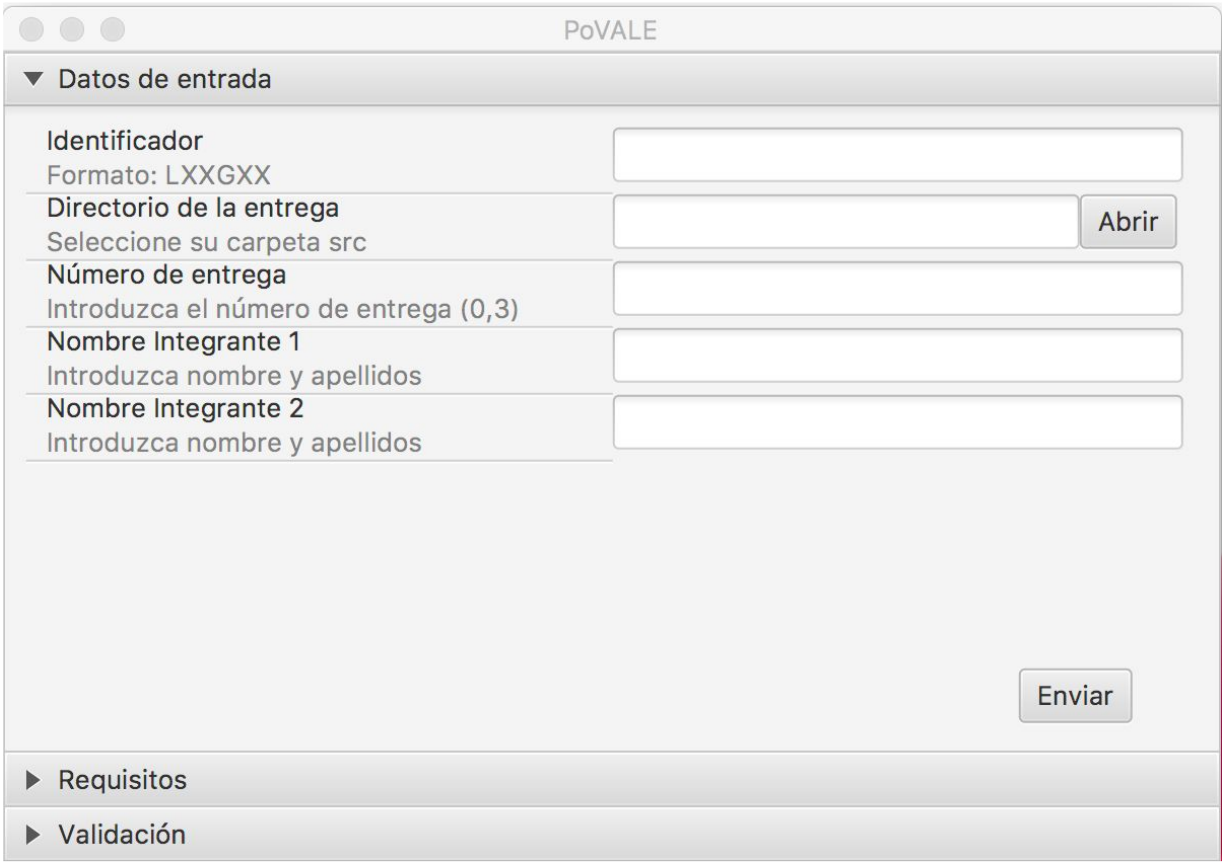
1. Manual de usuario

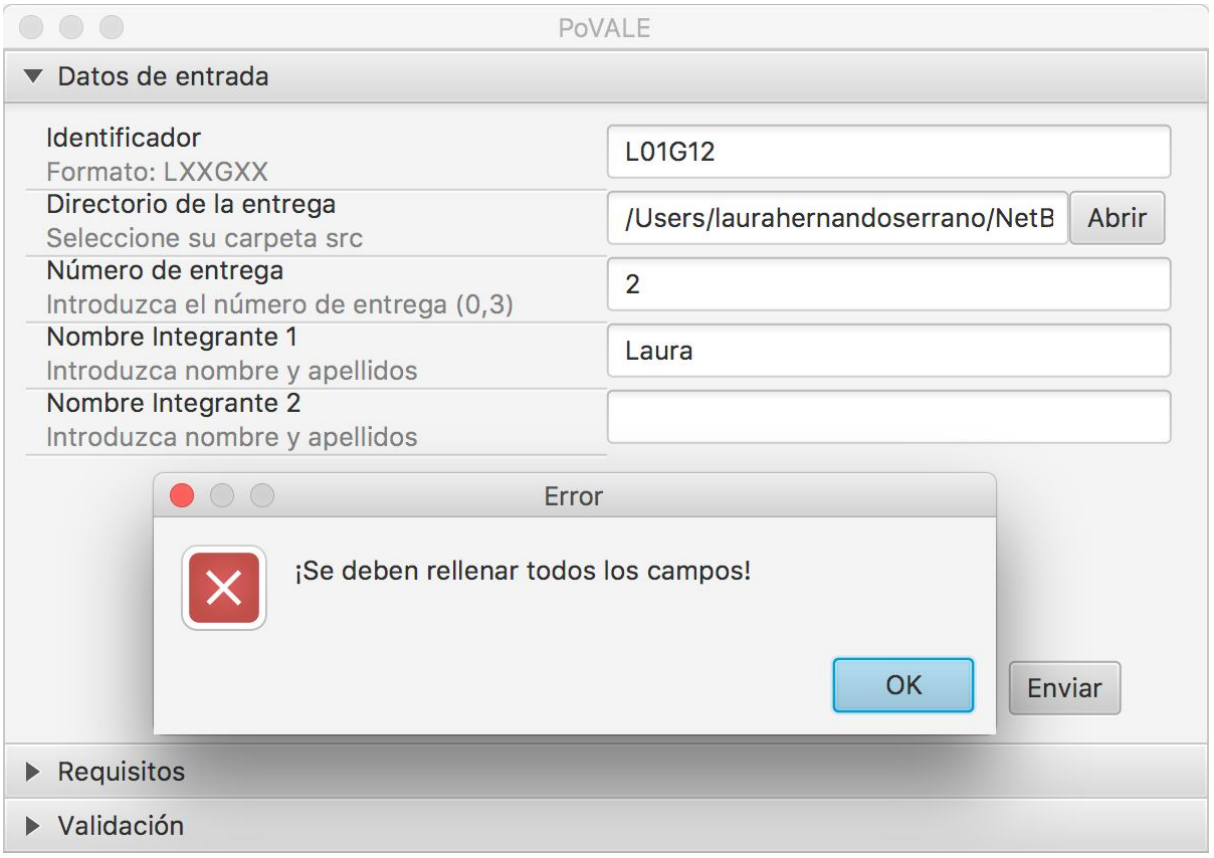
Al iniciar la aplicación el usuario debe seleccionar el fichero de especificación proporcionado por el profesor. Una vez seleccionado se mostrará la pantalla principal que dispone de tres pestañas: Datos de entrada, Requisitos y Validación.



1.1. Datos de entrada

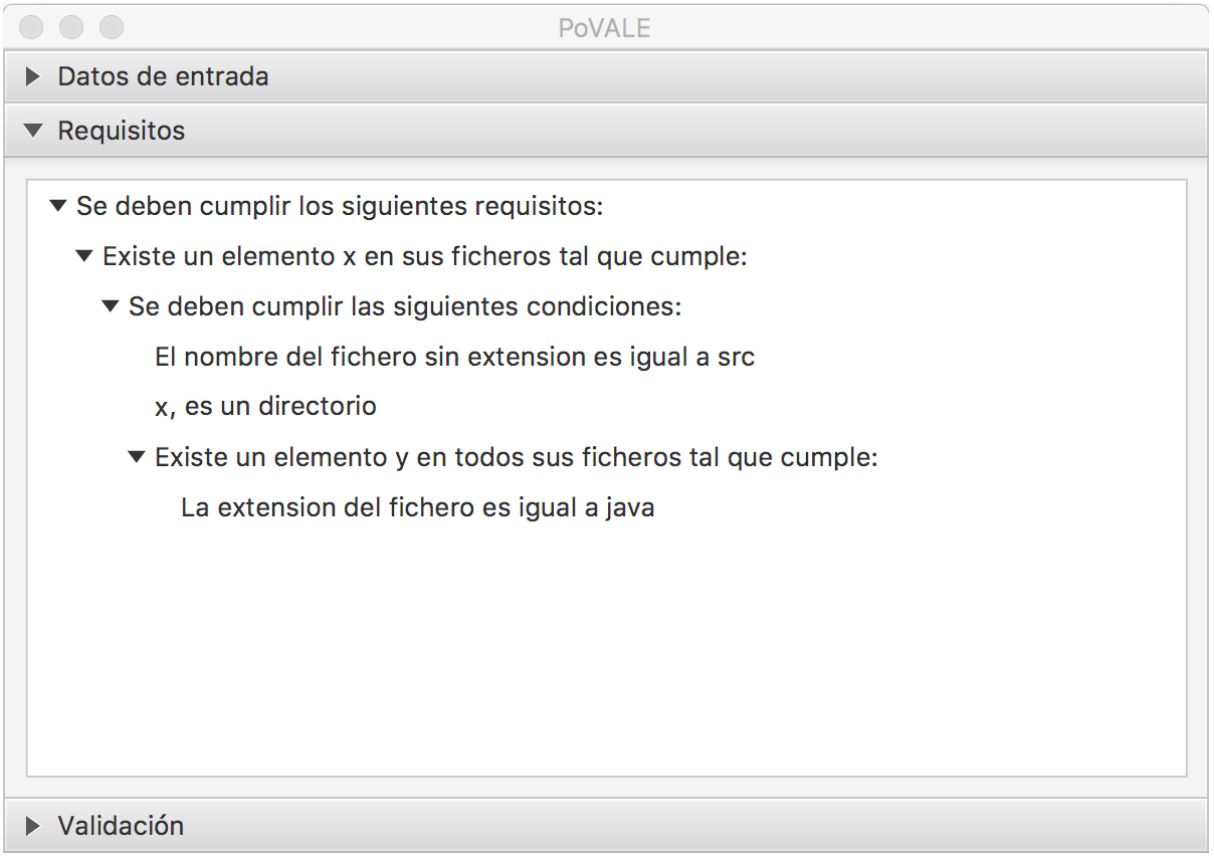
El objetivo de la primera pestaña es que el alumno rellene el valor de las variables. A la izquierda se puede ver el nombre de la variable y debajo, en gris, su descripción. A la derecha de cada variable, el usuario introduce el valor de la misma. Cuando ha rellenado todos los datos de entrada, pulsa *Enviar*. Se le mostrará un mensaje indicando si los datos introducidos son correctos o no.




PoVALE
Datos de entrada
Identificador
Formato: LXXGXX
L01G12
Directorio de la entrega
Seleccione su carpeta src
/Users/laurahernandoserrano/NetB
Abrir
Número de entrega
Introduzca el número de entrega (0,3)
2
Nombre Integrante 1
Introduzca nombre y apellidos
Laura
Nombre Integrante 2
Introduzca nombre y apellidos
Error
¡Se deben rellenar todos los campos!
OK
Enviar
Requisitos
Validación

1.2. Requisitos

En la pestaña de *Requisitos* el alumno puede ver qué restricciones se comprobarán para que pueda saber de antemano si necesita realizar alguna modificación a su entrega. Más adelante detallaremos cómo generamos está información.



1.3. Validación

En la pestaña de *Validación* el alumno puede comprobar los resultados de validación de su entrega. Si esta cumple los requisitos especificados por el profesor, podrá exportar los resultados de evaluación, que generará un fichero zip con los ficheros de su entrega y con ficheros adicionales que contienen los datos completados por el alumno en la pestaña *Datos de entrada*.

2. Detalles de implementación

Cambios en los plugins

Al desarrollar la aplicación de `PoVALE-view` nos dimos cuenta de que para que el alumno pudiera introducir el valor de las variables, era necesario un método de entrada acorde al tipo de entidad de la variable. Por ello, era necesario modificar los plugins. Cada tipo de entidad que pudiera ser indicado explícitamente por el usuario (como, por ejemplo, nombres de ficheros, cadenas de texto, etc.), necesitaba tener asociado un componente gráfico. De esta decisión surgieron los `ParameterEditor`. La clase principal de un plugin debe desarrollar el método `getEditor` que, dado el nombre de una entidad, devuelve la clase que extiende la clase abstracta `ParameterEditor` y contiene el componente gráfico para dicha entidad. De esta forma, dada, por ejemplo, una variable de tipo `DirectoryEntity`, la clase principal `FilesPlugin` del plugin `PoVALE-plugin-files`, le asociará la clase

`DirectoryEditor`, que proporcionará el panel con el componente que permite al usuario seleccionar un directorio.

Cambios en Reader, Core y en métodos de los plugins

Cuando hacíamos pruebas en `PoVALE-view`, vimos que tanto la información proporcionada en la pestaña *Requisitos* como la retroalimentación mostrada en la pestaña *Validación*, no eran del todo claras cuando se quería mostrar una representación textual de las aplicaciones de símbolos de función y predicado. En particular, no se mostraba un mensaje relacionado con el método concreto utilizado del plugin y, por tanto, la retroalimentación que recibía el alumno era incompleta y no le servía para poder corregir su entrega.

Para asociar un mensaje informativo preciso a un símbolo de función o predicado de un plugin, añadimos a cada uno de sus argumentos la etiqueta `@ParamDescription`, que contiene la descripción precisa del método. Este mensaje se usa para mostrar la información de la pestaña de *Requisitos*. Por ejemplo, para la función `base-name` de `PoVALE-plugin-files,` que devuelve el nombre de un fichero sin su extensión, la declaración del método quedaría de la siguiente manera:

```
public StringEntity baseName(@ParamDescription("El nombre del fichero sin extension")FileEntity f)
```

Cada función también dispone de un método `getMessage` que devuelve el mensaje a mostrar una vez comprobado el requisito que imponga dicha función. Este mensaje, que se muestra en la pestaña *Validación,* puede ser una cadena literal como ocurre para `base-name` cuyo método `getMessage` devuelve `"el nombre del fichero",` o puede devolver el resultado de la ejecución de la función. Por ejemplo, para el símbolo de función `children`, que devuelve la lista de ficheros que se encuentran dentro de un directorio, el método `getMessage` devuelve una cadena con los nombres de los ficheros contenidos en este.

Para que los asertos y los términos puedan obtener los mensajes generados en las funciones de los plugins, añadimos en la clase `DynamicallyCallable` del proyecto `PoVALE-core`, los métodos que permiten extraer estos mensajes mediante la reflexión de Java. Esto lo utilizamos en `PoVALE-core` y en `PoVALE-reader`. En `PoVALE-core` usamos estos métodos en la evaluación (método `evaluate`) de los asertos de funciones de aplicación y predicados. Con ello, arreglamos el problema de proporcionar al alumno una retroalimentación detallada y útil. En `PoVALE-reader`, los utilizamos al generar los mensajes informativos para la pestaña de *Requisitos*.

Estructura de clases

El proyecto `PoVALE-view` consta de cuatro clases: `MainApp`, `FXMLController`, `XMLExport` y `ZipExport`. El diseño de la interfaz gráfica está definido, mediante la herramienta Scene Builder, en `FXMLDocument`.

MainApp

La clase `MainApp` se encarga de cargar el fichero de especificación seleccionado por el alumno. Así mismo, también es la responsable de cargar recursos como el controlador y el fichero FXML.

Haciendo uso del proyecto `PoVALE-reader` se cargan las variables a rellenar por el usuario, los asertos que tendrá que superar la entrega, y la pestaña de "Requisitos" que muestra al alumno, de forma detallada, los requisitos que debe cumplir su entrega.

FXMLController
`FXMLController` es una clase generada mediante la herramienta Scene Builder. Es el controlador de la interfaz gráfica. En él gestionamos la interacción con el usuario y controlamos que los datos que introduce el alumno son correctos.

En esta clase se encuentran los métodos que sirven para mostrar información antes y después de validar una entrega. Uno de estos métodos recibe una lista de `AssertNode` construida en `PoVALE-reader` y genera un árbol con la información de los requisitos de validación que se introducirá en la pestaña de *Requisitos* de la vista. Otro método recibe una lista de `AssertInformation`, generada por el proyecto `PoVALE-core` cuando el alumno procede a validar una entrega. Este método se encarga de crear un árbol con los resultados, que se insertará en la pestaña de *Validación* de la vista.

En esta clase validamos que el alumno haya completado los datos a rellenar para cada variable, llamando al método `isValid` de las clases `ParameterEditor` asociadas a las variables. Si no fuera así, no podría realizar la validación. Una vez rellenadas correctamente las variables, al pulsar el botón *Comprobar* se valida la entrega realizada por el alumno y se carga la retroalimentación sobre el proceso de validación en la pestaña *Validación* mencionada anteriormente.

Por último, si la entrega satisface los requisitos especificados por el profesor, al pulsar *Enviar*, su fichero o directorio entregado así como el resto de variables introducidas se comprimen en una fichero zip. La carpeta se aloja donde especifique el alumno.

XMLExport y ZipExport
Inicialmente no estábamos muy seguros de cómo almacenaríamos la entrega y el resto de variables introducidas por el alumno.

La primera opción que propusimos fue exportar los datos a un fichero XML. Para ello creamos la clase `XMLExport` que contiene el método `export` que crea un fichero XML y se encarga de introducir en él los elementos necesarios, en el orden adecuado. En la clase `Entity` añadimos el método `toXML` que debían implementar todos los tipos de entidad para poder ser exportadas. Traducir las variables de tipo cadena o enteros (`StringEntity` y `IntegerEntity`) a XML era muy sencillo y traducir entidades de tipo fichero (`FileEntity`) también resultó ser viable y relativamente fácil. Sin embargo, tuvimos problemas con la exportación de directorios. Debido al gran tamaño que podía alcanzar un fichero XML y que volcar un conjunto de ficheros en un documento XML no era la forma más sencilla para que un profesor pudiese revisar una entrega, descartamos exportar las variables a un

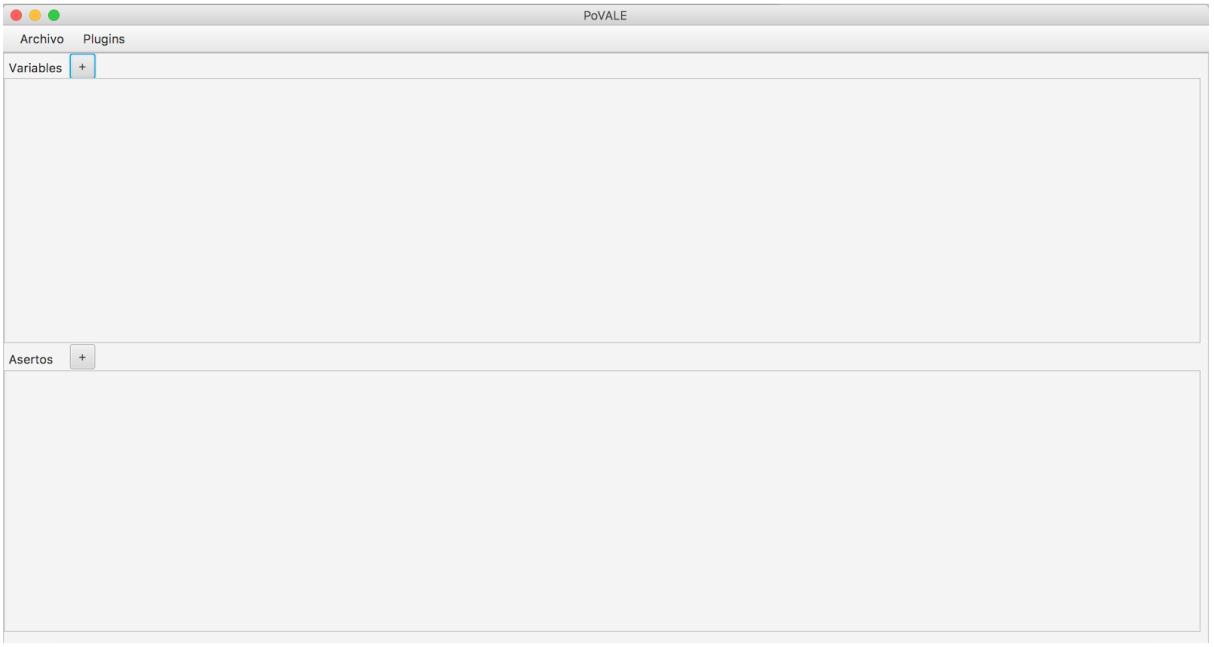
documento XML. En su lugar, decidimos optar por almacenar la entrega y el resto de variables en una carpeta zip. Para ello almacenamos la entrega del alumno directamente en el fichero comprimido. Las variables se guardan en ficheros de texto cuyo nombre es el identificador especificado por el profesor en el documento de requisitos. `ZipExport` es la clase en la que implementamos la exportación de las variables a la carpeta comprimida. A la interfaz `Entity` le añadimos el método `toZip` que implementan todas las entidades.

7. Implementación del generador de especificaciones de entrega

El generador de especificaciones es una herramienta visual para la creación de documentos de especificación. Con dicha herramienta, un profesor puede definir los requisitos que debe cumplir una entrega y exportarlos de forma que el documento generado pueda ser empleado en la herramienta del comprobador. Además, permite abrir y editar documentos de especificación ya creados.

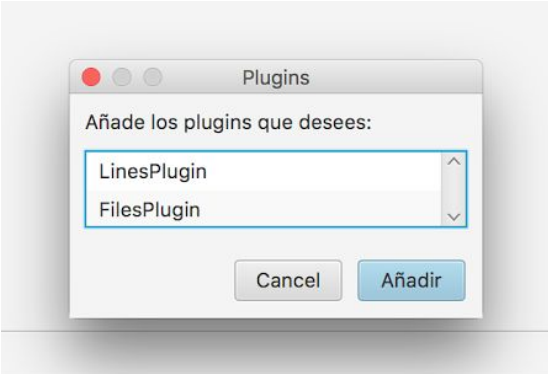
1. Manual de usuario

Al iniciar la aplicación el profesor puede comenzar a crear un documento de requisitos nuevo o abrir y editar uno existente. La interfaz está dividida en dos paneles: *Variables* y *Asertos*.

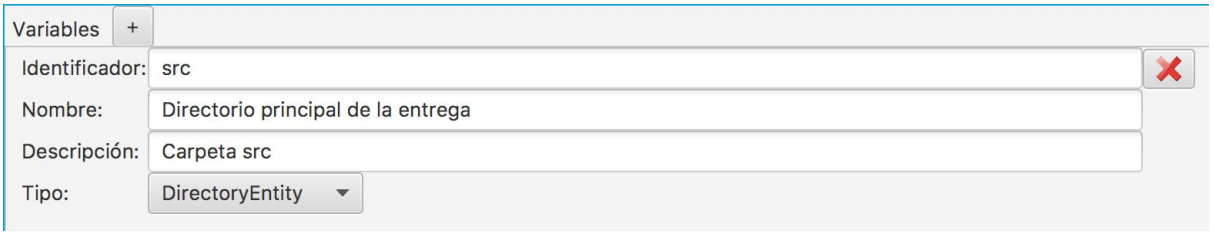


El menú *Archivo* contiene las opciones *Abrir*, *Guardar* y *Guardar como*, con las que es posible abrir un archivo o guardar el actual.

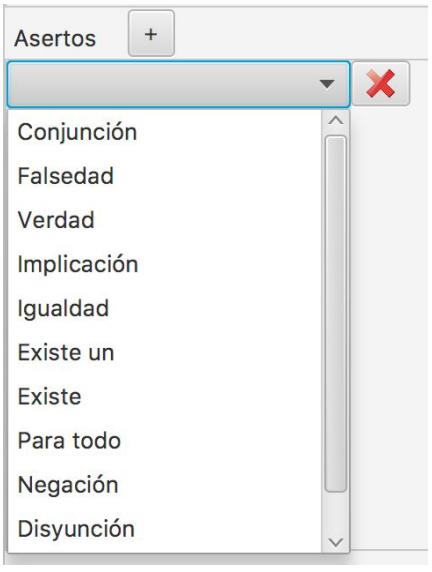
El menú Plugins contiene las opciones *Ver Plugins*, *Añadir Plugins* y *Eliminar Plugins*. Con la primera opción podemos ver la lista de plugins ya añadidos. Por defecto está añadido `CorePlugin`, que contiene las definiciones de los tipos de entidad básicos (`IntegerEntity`, `StringEntity`, etc.). La segunda y tercera opción permiten añadir nuevos plugins al fichero actual y eliminar plugins existentes.



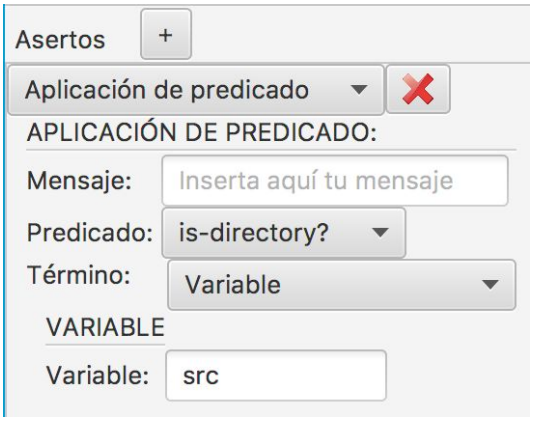
Pulsando el botón de añadir situado a la derecha de *Variables*, podemos crear una nueva variable de entrada. Este tipo de variables son las que posteriormente se muestran al usuario para que cumplimente su valor. Para cada una de estas variables es necesario especificar su identificador, nombre, descripción y tipo. Como se observa en la imagen, también es posible eliminar variables.



Pulsando el botón de añadir situado a la derecha de *Asertos*, aparece un componente *ComboBox* para seleccionar el tipo de aserto que queremos crear.



Por ejemplo, podemos crear el siguiente aserto con el que comprobamos que la variable declarada arriba es un directorio:



2. Detalles de implementación

Para llevar a cabo esta herramienta tuvimos que determinar de qué manera el profesor especificaría las variables y los requisitos que tiene que cumplir su entrega, por lo que implementamos cómo se mostraría cada tipo de aserto y de término.
Para que el profesor pudiera utilizar la funcionalidad que otorgan los plugins desarrollados en PoVALE, implementamos cómo importar los plugins. Por último, si tiene ya un fichero de especificaciones creado y desea modificarlo tuvimos que añadir la opción de importar un fichero ya existente a la aplicación.

2.1. Estructura de clases

Este proyecto está dividido en los paquetes `specification`, `variables`, `assertionRepresentation`, `termRepresentation`, `imports`, y `plugins` que agrupan clases según su funcionalidad.

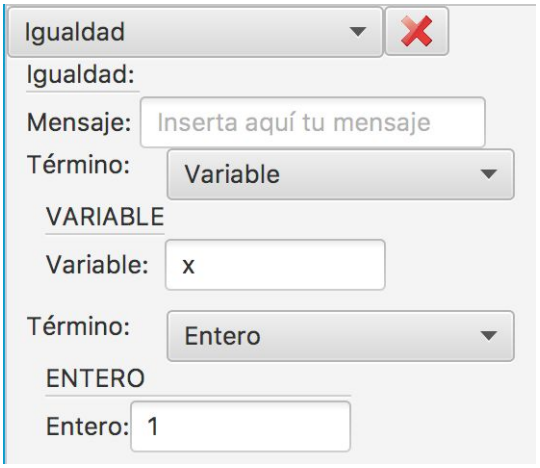
Specification
En la clase `Specification` se almacenan los datos del modelo: la lista de asertos y de variables definidas, los plugins añadidos, las funciones de aplicación, predicados y entidades disponibles para la especificación de requisitos.

Variables
Como hemos mencionado anteriormente, hay ciertas variables cuyo valor debe especificar el alumno. La clase `VarRep` contiene la representación visual para definir las variables. Permite al profesor especificar el identificador, nombre, descripción y tipo de la variable. El identificador es importante, ya que si el profesor desea añadir alguna restricción sobre la variable, deberá introducir éste identificador en el término perteneciente al aserto que utilice. El nombre y la descripción de la variable se mostrará al alumno. Finalmente, el tipo de la variable es necesario para saber qué tipo de entidad denotará dicha variable cuando tenga un valor. Además, es importante para poder mostrarle al usuario, mediante la clase asociada al tipo de entidad (que extienda a `ParameterEditor`), el panel donde puede introducir el valor de esa variable.

El profesor tiene la posibilidad de añadir y borrar variables. Al guardar el documento, mediante el método `exportVariable` se guardarán los datos de la variable de forma que puedan ser leídos por el comprobador de requisitos y la aplicación `PoVALE-view`.

AssertionRepresentation

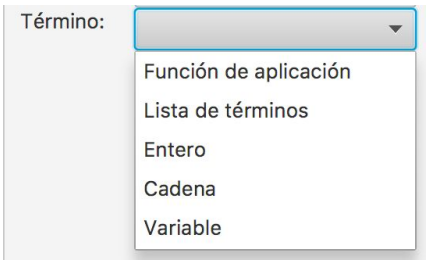
Cada aserto de nuestro lenguaje de especificaciones tiene asociado una clase para su representación visual, que extiende la clase abstracta `AssertionRep`. En esta clase se define el aspecto visual común a todos los distintos tipos de aserto. Las clases que la extienden deben añadir los componentes visuales adicionales que sean necesarios para representar su aserto. Por ejemplo, el aserto de igualdad añade la representación de los dos términos sobre los que se evaluará la igualdad.



TermRep

Cada tipo de aserto tiene una clase para representarlo y para que el usuario pueda definir su valor. Todas estas clases extienden la clase `TermRep`, que crea el panel con los componentes comunes a todos los tipos de términos. Además, esta clase contiene una *ObservableList* con la lista de símbolos de función disponibles. Esta lista está vinculada, mediante *Binding*, a la lista de símbolos de función de la clase principal `Specification`. Mediante este vínculo, cada vez que se añade un plugin se actualiza la lista en `Specification` y, a su vez, se actualiza la lista en los `TermRep`. De esta forma, el usuario puede observar el cambio automáticamente.

El diagrama de clases de `TermRep` se encuentra el en capítulo 16.



Imports

Como hemos mencionado, la herramienta permite abrir documentos de especificación de requisitos para poder editarlos. Para realizar esto es necesario traducir el documento seleccionado a su representación en el generador. Esto se realiza mediante las clases `AssertParser`, `TermParser`, `Import`, y `XMLParser`.

- `Import`: Contiene el método `importFile` que recibe el fichero seleccionado mediante un parámetro de tipo `InputStream`. Devuelve al controlador la información traducida de las variables, asertos y plugins para que está información sea añadida en la clase `Specification`.
- `XMLParser`: En esta clase se lee y traduce el fichero xml a una estructura formada por las clases de `PoVALE-specification`. Los asertos son traducidos a objetos de tipo `AssertionRep` y las variables a `VarRep`. Además se leen los plugins para importarlos en nuestra aplicación.
- `AssertParser`: Contiene los métodos necesarios para crear cada tipo de aserto dado un elemento XML. Cada método devuelve la representación de un aserto concreto mediante su clase correspondiente, que extiende a la clase `AssertionRep`.
- `TermParser`: Posee los métodos para crear cada tipo de término. Los métodos devuelven la representación de un término concreto, que extiende a la clase `TermRep`.

Plugins

La aplicación permite que el profesor añada los plugins que va a necesitar para especificar los requisitos que deben cumplir las entregas.

La clase `PluginActions` contiene métodos para poder importar un plugin dado el identificador del plugin. Al importar un plugin se añaden las funciones de aplicación, los predicados y entidades nuevas que aporta el plugin. Igualmente es posible eliminar plugins ya importados.

En el paquete `plugins` también se encuentran la clase `BasePluginDialog` que extiende la clase `Dialog` de JavaFX y que es a su vez una clase abstracta de la cual extienden las clases `AddPluginDialog`, `RemovePluginDialog`, `ViewPluginDialog`. La clase `BasePluginDialog` crea un diálogo genérico con una lista de plugins. Las clases `AddPluginDialog`, `RemovePluginDialog`, `ViewPluginDialog` modifican y añaden los componentes necesarios a la ventana de diálogo para que el usuario pueda añadir, eliminar y visualizar plugins, respectivamente. Por ejemplo, la clase `RemovePluginDialog` modifica el mensaje de información y añade el botón de *Eliminar*.

Los plugins importados también se exportan al documento de especificación de requisitos, para que puedan ser cargados posteriormente por la herramienta validadora.
El diagrama de clases de `PluginActions` está en el capítulo 15.

8. Conclusiones

1. Valoración de los resultados

En este último capítulo haremos una retrospección de los objetivos marcados para valorar el grado de éxito que hemos tenido en este proyecto.

El primer objetivo era desarrollar una herramienta que comprobase que el ejercicio entregado por un estudiante respetaba unas especificaciones de entrega. Este objetivo lo hemos cumplido con creces mediante el desarrollo del proyecto `PoVALE-core` puesto que no nos hemos limitado a proporcionar los mecanismos necesarios para poder comprobar que un ejercicio es correcto. `PoVALE-core` está diseñado de forma que es relativamente fácil ampliar su funcionalidad mediante plugins, y además proporciona información detallada de la evaluación de los asertos. Resultó más difícil de lo que esperábamos determinar qué devolver como resultado de la evaluación de un aserto. Al principio devolvíamos simplemente un booleano con el resultado de la evaluación. Más tarde quisimos realizar un manejo de los errores, devolviendo un `Optional<AssertionError>` cuyo valor sería nulo en el caso de que se cumpliese el aserto, y un objeto de tipo `AssertionError` en el caso contrario, que contendría los métodos necesarios para representar el error. Finalmente, nos decantamos por devolver siempre un objeto de tipo `AssertInformation` que dispone de un booleano que representa si se ha cumplido el aserto, un mensaje usado para proporcionar retroalimentación al alumno y una lista de `AssertInformation` para los subasertos contenidos dentro del aserto.

El segundo objetivo era desarrollar la interfaz de la herramienta del comprobador para que pudiese ser usada por un alumno para evaluar su entrega, ver los resultados y poder exportar la entrega. Este objetivo lo hemos abordado mediante el desarrollo del proyecto `PoVALE-view`. La interfaz implementada es simple e intuitiva y permite a un alumno evaluar un ejercicio, ver los resultados y crear un archivo comprimido con los detalles de su entrega. Además, mostramos los requisitos que debe cumplir el ejercicio para que el usuario conozca de antemano respecto a qué restricciones va a ser evaluada su entrega antes de comprobarla y pueda hacer las modificaciones necesarias. Encontramos diversos problemas a la hora de elegir el método e implementar el mecanismo para exportar la entrega de un alumno, pasando de generar un único archivo XML a un fichero comprimido. También tuvimos que hacer diversas adaptaciones para que los mensajes generados sobre los asertos fueran específicos y detallados, especialmente para los asertos de tipo aplicación de función y predicado. En cualquier caso, consideramos que el objetivo de desarrollar una interfaz intrínsecamente supone el hacer una valoración de su usabilidad con usuarios finales y debido a esto podemos considerar que el objetivo se ha cumplido parcialmente.

Finalmente, el tercer objetivo era implementar una herramienta visual que permitiese a un profesor definir y generar documentos de especificación que pudiesen ser exportados para ser empleados en la herramienta del comprobador. Este objetivo lo hemos cumplido

mediante el desarrollo del proyecto `PoVALE-specification`. La herramienta permite importar plugins para poder hacer uso de sus funciones de aplicación y predicados. En la fase de diseño y definición del comportamiento de la aplicación, identificamos que queríamos que el usuario pudiese añadir plugins en cualquier momento y que automáticamente se actualizaran las funciones de aplicación, predicados y entidades disponibles. Tuvimos problemas en conseguir dicho comportamiento, pero finalmente lo solventamos con el uso de la vinculación de JavaFX. Nos llevó mucho tiempo y esfuerzo el desarrollo de la funcionalidad de abrir documentos de especificación existentes y que estos se representasen adecuadamente en la interfaz, por las complicaciones que implica anidar asertos. Aunque la visualización de los asertos es correcta, se podría mejorar la indentación de los diversos asertos para mostrar y separar claramente el contenido de cada uno.

Globalmente, consideramos que se han cumplido los objetivos planteados inicialmente y que estos satisfacen la motivación de desarrollar un conjunto de herramientas que faciliten y agilicen la corrección de ejercicios. En primer lugar, el profesor podrá definir los requisitos que debe cumplir una entrega de forma visual y generar el documento de especificación que podrá distribuir a los alumnos. Cada alumno podrá utilizar la herramienta visual del comprobador para validar que su entrega se ajusta los requisitos marcados. Aunque posteriormente podría ser necesario que el profesor examine más en detalle ciertos aspectos de los ejercicios, este ya se habría ahorrado comprobar que los ejercicios cumplen unos requisitos base. De esta forma, se agiliza el proceso de corrección de ejercicios para favorecer el uso de la evaluación continua en grupos de alumnos numerosos.

Este proyecto nos ha sido útil para ampliar nuestros conocimientos sobre Java, al utilizar la API de reflexión y la API de W3C para el manejo de ficheros XML. Por otro lado, hemos aprendido a utilizar JavaFX y Scene Builder, con lo que se agiliza el desarrollo de interfaces. Realizar el proyecto, nos ha servido también para aprender a trabajar en paralelo mediante el uso de la tecnología Git. Además, a lo largo de la carrera nunca habíamos tenido que desarrollar un trabajo tan extenso donde tuviéramos que crear diversos proyectos separados por funcionalidad, por lo que hemos aprendido a saber delimitar proyectos.

2. Trabajo futuro

Esta herramienta puede extenderse de distintas formas. En esta sección se proponen las siguientes:

- Teniendo en cuenta que JavaFX permite utilizar hojas de estilo, podríamos mejorar la interfaz gráfica del generador de especificaciones creando hojas CSS.

- Añadir una pestaña de ayuda tanto en la interfaz del comprobador, como en la del generador de especificaciones con información útil para los usuarios que usen por primera vez estas herramientas.

- Para agilizar la labor del profesor cuando revise y corrija entregas, proponemos que el comprobador suba directamente a un servidor el fichero comprimido con los

resultados de validación de cada alumno. Así el profesor podría bajarse todos de manera simultánea, agilizando el tiempo que dedicaría descargando uno a uno los resultados de validación. En este sentido sería útil el uso de la API de *Moodle* para subir ejercicios directamente desde nuestra herramienta hasta el Campus Virtual.

- Implementar un generador de informes con los resultados de validación de una entrega para que el profesor no tenga que inspeccionar el contenido del fichero comprimido que contiene los resultados de la entrega. Esto haría más sencilla la revisión de las entregas ya que el profesor solo tendría que leer un documento con un resumen de los resultados.

- Realizar una aplicación para el procesamiento por lotes de ejercicios. Con ella, el profesor podría trabajar con las entregas subidas al servidor mencionado en el punto anterior. La herramienta permitiría realizar acciones simultáneamente sobre todas las entregas. Por ejemplo, compilar y ejecutar las entregas, aplicar casos de prueba sobre ellas, extraer información de estas y unificarlas en un archivo.
  Estas operaciones permitirán al profesor reducir el tiempo que dedica a corregir las entregas.

- Los plugins además de poder añadir funciones de aplicación, predicados y entidades, podrían añadir acciones para la herramienta de procesamiento por lotes. Estas acciones se aplicarían sobre las entidades con lo que habría que adaptar la interfaz `Entity` para permitir este comportamiento y la clase abstract `PluginInfo` para poder importar dichas acciones.

9. Conclusions

1. Evaluation of the results

In this last chapter we will look back at the objectives set to assess the degree of success we have had in this project.

The first objective was to develop a tool that validates that the student's exercise meets the specified requirements. This objective has been fully accomplished by developing the `PoVALE-core` project and we have not limited ourselves to only providing the necessary mechanisms to verify that an exercise is correct. `PoVALE-core` is designed so that it is relatively easy to extend its functionality through plugins, and also provides detailed information of the result of evaluating assertions. It turned out be more difficult than we expected to determine what to return as a result of the assessment of an assertion. At first we simply returned a boolean with the result of the evaluation. Later on we decided we wanted to handle errors more precisely, returning an `Optional<AssertionError>` whose value would be null if the assert was fulfilled, and an object of type `AssertionError` otherwise, that would contain the necessary methods to represent the error. Finally, we decided to always return an `AssertInformation`, which has a boolean that represents if the assertion has been fulfilled, a message used to provide feedback to the student and a list of `AssertInformation` for the subasserts contained within the assert.

The second objective was to implement the interface for the validator tool so that it could be used by a student to evaluate their exercise, see the results and be able to export it. This objective has been addressed through the development of the `PoVALE-view` project. The implemented interface is simple and intuitive and allows a student to evaluate an exercise, see the results and create a compressed file with the details of their submission. In addition, we show the requirements that the exercise must meet in order for the user to know in advance what restrictions are going to be evaluated. In this way the user is able to check these requirements beforehand and can make the necessary modifications. We faced several problems when choosing the method and implementing the mechanism to export the submission of a student, going from generating a single XML file to a compressed file. We also had to make various adaptations so that the messages generated for the assertions were specific and detailed, especially for function and predicate type assertions. In any case, we consider that the objective of developing an interface intrinsically involves making an assessment of its usability with end users and because of this we can consider that the objective has been partially fulfilled.

Finally, the third objective was to develop a visual tool that would allow a teacher to define and generate a specification document that could be exported for its use in the validator application. This objective has been fulfilled through the development of the `PoVALE-specification` project. The application allows the user to import plugins to be able to make use of its functions and predicates. In the initial phase of designing and defining the behavior of the application, we identified that we wanted the user to be able to add plugins at

any time and that the available functions, predicates and entities would be automatically updated in the application. We had problems achieving such behavior, but we finally solved it with the use of JavaFX binding. It took a lot of time and effort to develop the functionality of being able to open an existing specification document and representing it correctly in the interface, due to the complications involved in nesting assertions. Although the visualization of the assertions is correct, the indentation of the various assertions could be improved to clearly show and separate the content of each one.

Overall, we consider that the objectives initially set have been met and that they satisfy the motivation to develop a set of tools that facilitate and speed up the correction of exercises. The teacher can visually define the requirements that an exercise must fulfill and generate the specification document that can be distributed to students. Each student will be able to use the validator application to verify that their submission meets the specified requirements. Although it may be necessary afterwards for the teacher to examine in more detail certain aspects of the exercises, time would have been saved by automatically checking that the exercises meet some basic requirements. In this way, the process of correction of exercises takes less time to favor the use of continuous assessment in large groups of students.

This project has been useful to us to deepen our knowledge of Java, using the reflection API and the W3C API for handling XML files. Additionally, we have learned to use JavaFX and Scene Builder, which eases the development of interfaces. Carrying out the project has also helped us to learn to work in parallel using the Git technology. In addition, throughout our degree we had never had such an extensive assignment where we had to create different projects separated by functionality, so we have learned to delimit projects.

2. Future work

This project can be extended in different ways. In this section we propose the following:

- Given that JavaFX allows you to use style sheets, we could improve the graphical interface of the specification generator by creating various CSS sheets.

- Add a help tab in the interface of the validator as well as in the interface of the specification generator with useful information for users who use these tools for the first time.

- To further ease and speed up the work of the teacher when reviewing and correcting exercises, we propose that the validator directly uploads to a server the compressed file with the validation results of each student. Thus, the teacher could download all of them simultaneously, speeding up the time that would be spent downloading one by one the validation results. Likewise, it would be useful to use the Moodle API to upload exercises directly from our tool to the Virtual Campus.

- Implement a report generator with the results of validating an exercise so that the teacher doesn’t have to inspect the contents of the compressed file. This would make

it easier to review submissions since the teacher would only have to read a document with a summary of the results.

- Make an application for batch processing of exercises. With it, the teacher could work with deliveries uploaded to the server mentioned in the previous point. The tool would allow simultaneous actions on all deliveries. For example, it would compile and execute the submissions, apply test cases on them, extract information from them and merge them into a file. These operations would allow the teacher to reduce the time devoted to correcting exercises.

- In addition to being able to add functions, predicates, and entities, plugins could add actions for the batch processing tool. These actions would be applied on entities. Therefore, we would have to adapt the `Entity` interface to allow this behavior and the abstract `PluginInfo` class to be able to import these actions.

10. Apéndice: Contribución de los integrantes

En esta sección explicaremos qué labores ha desempeñado cada alumno durante el desarrollo del proyecto.

1. Contribución de Laura Hernando Serrano

Estudio de herramientas y tecnologías:

- Para la realización de interfaces, sabíamos que pese a que era lo que conocíamos, la librería de Swing iba a dejar de mantenerse. Decidimos buscar una alternativa y dimos con JavaFX por lo que tuve que documentarme al respecto. También vi las ventajas que nos daba usar la herramienta Scene Buider.

- Por otro lado, nuestro tutor Manuel Montenegro nos introdujo que durante el proyecto tendríamos que utilizar la API de reflexión de Java. Por lo que tuve que estudiarla y realizar pruebas para poder manejarla correctamente.

PoVALE-core:

- Creé las clases principales del proyecto, las que representaban a asertos y términos. Tras completar el proyecto, lo subí al repositorio de Github.

- Posteriormente, cuando quisimos implementar cómo recoger información de los resultados de validación que luego mostraríamos al usuario. Cambié el método `check` de cada aserto para que éste, en lugar de devolver un booleano devolviera un `Optional<AssertionError>`, tuve que implementar la clase `AssertionError` y otras clases, necesarias para llevar a cabo este sistema. Aunque finalmente nos decantamos por usar otro sistema.

- Con el fin de poder utilizar plugins en el proyecto, implementé `PluginInfo` y otras clases necesarias que permitieran poder importar y usar plugins en un futuro. Para ello tuve también que modificar la clase `Entity`.

- Completé la documentación de varias clases de `PoVALE-core`.

- Implementé el paquete `parameter`, para que las entidades tuvieran una representación determinada en la interfaz de `PoVALE-view` dependiendo del tipo de entidad.

- Como primero decidimos que exportaríamos los resultados de validación a un fichero XML, implementé los métodos necesarios para poder realizar esta exportación.

PoVALE-plugin-files:

- Modifiqué la clase `FilesPlugin` para adaptar nuevos editores (`ParameterEditor)`.

- Implementé las clases `FileEditor` y `DirectoryEditor` para dotar de representación visual en la interfaz del proyecto `PoVALE-view` a las entidades desarrolladas en el plugin.

PoVALE-view:

- Diseñé y desarrollé la interfaz del proyecto.

- Implementé la gestión para manejar correctamente y validar las variables de la pestaña *Datos de entrada* de la interfaz del proyecto. También manejé la funcionalidad encargada de interactuar con el usuario.

- Creé y completé la clase `XMLExport` para poder realizar la exportación de los resultados de validación a un fichero XML.

PoVALE-specification:

- Diseñé e implementé la interfaz usada en el proyecto.

- Desarrollé las clases para crear las representaciones visuales de las variables, los términos y los asertos.

- Implementé la importación de plugins, para que el profesor pudiera utilizar funciones y predicados de los plugins que desee.

- Ajusté la indentación de los asertos y términos, dejando a la izquierda de la interfaz a los asertos principales, mientras que sus hijos avanzan gradualmente a la derecha según su profundidad.

- Implementé un método en cada tipo de aserto y término, además de las variables, para exportar los requisitos especificados por el profesor a un fichero XML.

2. Contribución de Daniel Rossetto Bermejo

Estudio de herramientas y tecnologías:

- Cuando decidimos que los ficheros que portasen los requisitos de validación, fueran documentos XML. Primero realicé pruebas con JAXB [13], pero la documentación no era clara ni era fácil de usar, por lo que procedí a realizar pruebas usando la API DOM de W3C. Finalmente fue esta API la que utilizamos.

- Al igual que mi compañera desconocía la API de reflexión de Java por lo que tuve familiarizarme con ella y realizar pruebas para usarla correctamente en el proyecto.

PoVALE-core:

- Trabajé en paralelo con mi compañera para asentar las bases de este proyecto implementando el entorno, las entidades y la clase `Var` para gestionar las variables de entorno.

- Al igual que mi compañera documenté varias clases del proyecto.

- Para mejorar la calidad de los mensajes que la aplicación proporciona al usuario antes y después de la validación. Para ello, desarrollé la clase `AssertInformation`, modifiqué el método `check` de cada aserto y el método `call` de cada término. Además tuve que cambiar otras clases como las clases `Predicate` y `Function` entre otras.

- Cambié la gestión de variables, para poder borrar una variable del entorno, una vez que el aserto que la estuviera usando se haya validado.

- Desarrollé la funcionalidad necesaria para importar los resultados de validación a un fichero zip.

PoVALE-reader:

- Definí el lenguaje de especificación usado en los ficheros de requisitos XML.

- Desarrollé un lector para poder tratar la información que portan los ficheros de validación XML. Pudiendo cargar, los plugins necesarios para la validación, las variables que completarán los alumnos y los asertos y términos existentes en el fichero.

- Creé un sistema para generar mensajes que usaríamos en la pestaña de *Requisitos* de `PoVALE-view`, que después modifiqué para que estos proporcionaran una información útil al usuario.

PoVALE-plugin-files:

- Añadí los métodos necesarios para validar y exportar a un fichero zip las entidades `FileEntity` y `DirectoryEntity`.

- Introduje los mensajes descriptivos a cada función y predicado implementado en el plugin, para que luego se mostrasen correctamente en la información proporcionada al usuario en el proyecto `PoVALE-view`.

PoVALE-view:

- Fui el encargado de gestionar la información que se le proporciona al usuario en las pestañas de *Validación* y *Requisitos*.

  Para la pestaña de *Requisitos*, implementé un método que presentara la información en forma de árbol. Permitiendo así al usuario poder leer la información de una manera más clara y sencilla.

  En la pestaña de *Validación*, utilicé un método similar al que implementé para mostrar información en la pestaña de *Requisitos*. En este caso, la información contenida se generaba al validar los requisitos de la entrega del alumno, dependiendo del cumplimiento de cada aserto, se presentaba la información de ese aserto junto a una imagen que representa si este se cumple o no.

- Implementé la clase `ZIPExport` para poder realizar la exportación de los resultados de validación a un fichero zip.

PoVALE-specification:

- Habilité la importación de ficheros de requisitos XML ya creados, por si el profesor quisiera editarlo.

11. Apéndice: PoVALE-plugin-files

El plugin `PoVALE-plugin-files` desarrollado principalmente por Manuel Montenegro aporta nuevas entidades, funciones de aplicación y predicados para la gestión de archivos.

Se puede ver el repositorio en el siguiente enlace:
https://github.com/PoVALE/PoVALE-plugin-files

Entidades

`File` - Representación de archivos.
`Directory` - Representación de directorios, extiende a `File`.

Predicados

`id-directory?(File)` - Se cumple el predicado cuando el fichero indicado es un directorio.

Funciones

`name(File) : StringEntity` - Nombre de un fichero (con extensión).
`base-name(File) : StringEntity` - Nombre de un fichero (sin extensión).
`extension(File) : StringEntity` - Extensión de un fichero.
`files(File) : ListEntity` - Ficheros regulares contenidos en un directorio.
`files-rec(File) : ListEntity` - Ficheros regulares contenidos en un directorio y sus subdirectorios.
`children(File) : ListEntity` - Ficheros y directorios contenidos en un directorio.
`children-rec(File) : ListEntity` - Ficheros y directorios contenidos en un directorio y sus subdirectorios.

Editores

`FileEditor` - Extiende a `ParameterEditor`, y es un componente visual para seleccionar variables de tipo `File`.
`DirectoryEditor` - Extiende a `ParameterEditor`, y es un componente visual para seleccionar variables de tipo `Directory`.

12. Apéndice: PoVALE-plugin-lines

`PoVALE-plugin-lines` es un plugin desarrollado por Manuel Montenegro. Aporta nuevas entidades, funciones de aplicación y predicados, que sirven para establecer restricciones sobre el texto contenido en los ficheros de la entrega. Este plugin requiere la presencia de `PoVALE-plugin-files`, ya que permite hacer referencia al contenido de un fichero.

Se puede ver el repositorio en el siguiente enlace:
https://github.com/PoVALE/PoVALE-plugin-lines

Entidades

`MatchResult` - Representación de coincidencias presentes en un archivo.

Predicados

`line-contains(StringEntity cadena, StringEntity subcadena)` - Se cumple el predicado cuando la subcadena se encuentra en la cadena recibida.
`line-ends-with(StringEntity cadena, StringEntity subcadena)` - Se cumple el predicado cuando la subcadena coincide con el final de la cadena recibida.
`line-starts-with(StringEntity cadena, StringEntity subcadena)` - Se cumple el predicado cuando la subcadena coincide con el principio de la cadena recibida.
`matches(MatchResult)` - Se cumple el predicado cuando se encuentran coincidencias en el `MatchResult`.

Funciones

`line-match-against(StringEntity cadena, StringEntity expresión regular, StringEntity flags) : MatchResult` - Devuelve un `MatchResult` con los resultados coincidentes de una expresión regular con la cadena. Dependiendo del flag se activa el modo MULTILINE (m) o el modo CASE_INTENSITIVE (i) de la clase `Pattern`. Más información en `https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html`
`line-trim(StringEntity cadena) : StringEntity` - Devuelve la cadena sin espacios.
`lines(File archivo) : ListEntity` - Devuelve una lista con el contenido del archivo recibido.
`match-group(MatchResult matcher, IntegerEntity i) : StringEntity` - Devuelve la coincidencia alojada en la posición i del matcher.

13. Apéndice: ¿Cómo crear un plugin en PoVALE?

Cómo desarrollar la clase principal de un plugin

La clase principal del plugin debe extender a la clase `PluginInfo` e implementar todos sus métodos abstractos, que explicaremos a continuación:

`getIdPlugin`: en él se devuelve un String con el identificador del plugin desarrollado. El identificador debe tener esta estructura: "es.ucm.Nombre_del_paquete_principal.Nombre_de_la_clase_principal".
Por ejemplo, en el proyecto `PoVALE-plugin-files` su id es: "es.ucm.povaleFiles.FilesPlugin", donde `povaleFiles` es el paquete principal del plugin y `FilesPlugin` la clase principal.

Cada símbolo de función y predicado que se implemente en el plugin debe estar en una clase. Tras explicar los métodos restantes a implementar de `PluginInfo`, detallaremos cómo crear predicados y funciones en el plugin.

`getFunctions`: aquí se devuelve una `List<Function>` con los símbolos de función desarrollados en el plugin. Por ejemplo, este es el método `getFunctions` de `PoVALE-plugin-files`:

```
@Override
public List<Function> getFunctions() {
    return Arrays.asList(
            new BaseName(),
            new Children(),
            new ChildrenRec(),
            new Extension(),
            new Files(),
            new FilesRec(),
            new Name()
    );
}
```

`getPredicates`: es similar al método anterior, pero devuelve una `List<Predicate>` con los predicados desarrollados en el plugin.

`getEntities`: en este método se debe devolver `List<Class<?>>`, una lista con las clases de las entidades que se implementen (`NombreEntidad.class`). Tras explicar los métodos restantes a implementar de `PluginInfo`, detallaremos cómo crear nuevas entidades.

`getEditorTypes`: devuelve una lista con los nombres de las entidades desarrolladas en el plugin que disponen de un editor.

`getParamEditors`: en este método se recibirá un String con el nombre del editor y un Map<String, String> con los parámetros necesarios para crear un editor y devolverá el editor creado. Explicaremos cómo crear `ParameterEditor` concretos.

Cómo desarrollar funciones y predicados

Se debe crear una clase que extiende a la clase `Function` en el caso de implementar funciones y a la clase `Predicate` en el caso de implementar predicados. En ambas opciones se deben implementar 3 métodos:

- `getName`: que devuelve el identificador de la función o del predicado.
- Método bajo la etiqueta `@CallableMethod`, que será el método en el que se ejecutará la función o el predicado desarrollado. Este es el método del predicado `IsDirectory` de `PoVALE-plugin-files`:

```
@CallableMethod
public boolean isDirectory
    (@ParamDescription("es un directorio ") FileEntity f) {
    return f instanceof DirectoryEntity;
}
```

  Por último, en el argumento del método se debe añadir la etiqueta `@ParamDescription` con el mensaje que se mostrará en `PoVALE-view` en las pestañas de Requisitos y de Validación en el caso de los predicados y en la pestaña de Requisitos en el caso de las funciones.

- `getMessage`: Debe implementarse en las funciones que se implementen. Devuelve el mensaje que aparecerá en `PoVALE-view`, en la pestaña de Validación.

Cómo crear nuevas entidades

Cada entidad nueva creada debe ser un interfaz que extienda a la clase `Entity` de `PoVALE-core` y contendrá los métodos necesarios para su funcionamiento y los que ejecuten las funciones y predicados que se desarrollen en el plugin.
Tras crear la interfaz, se debe desarrollar una clase que implemente a la interfaz y completar los métodos pertenecientes a la misma.
Si la entidad quiere ser guardada en el archivo zip de los resultados de validación de `PoVALE-view`, se deberá hacer @Override del método `writeToZip`.

Cómo crear nuevas nuevos editores (ParamEditor)

Si se crea una nueva entidad y esta puede ser usada como `Variable` a rellenar por el alumno, se debe implementar un `ParameterEditor` para crear la representación visual de la entidad en la aplicación `PoVALE-view`. Para ello, se tiene que crear una clase que extienda a `ParameterEditor<Nombre_de_la_entidad_creada>`. En esta clase se implementan los métodos:

- `getPane`: que devuelve el objeto de tipo `Pane` que se cree para representar a la entidad.

- `getEntity`: que devuelve un objeto del tipo de la clase que implementa a la interfaz de la entidad creada.

- `isValid`: método que devuelve un booleano. En él se comprueba si el alumno ha rellenado correctamente la variable.

- `setStage`: que recibe un objeto `Stage` para actualizar el `Stage` propio del editor implementado.

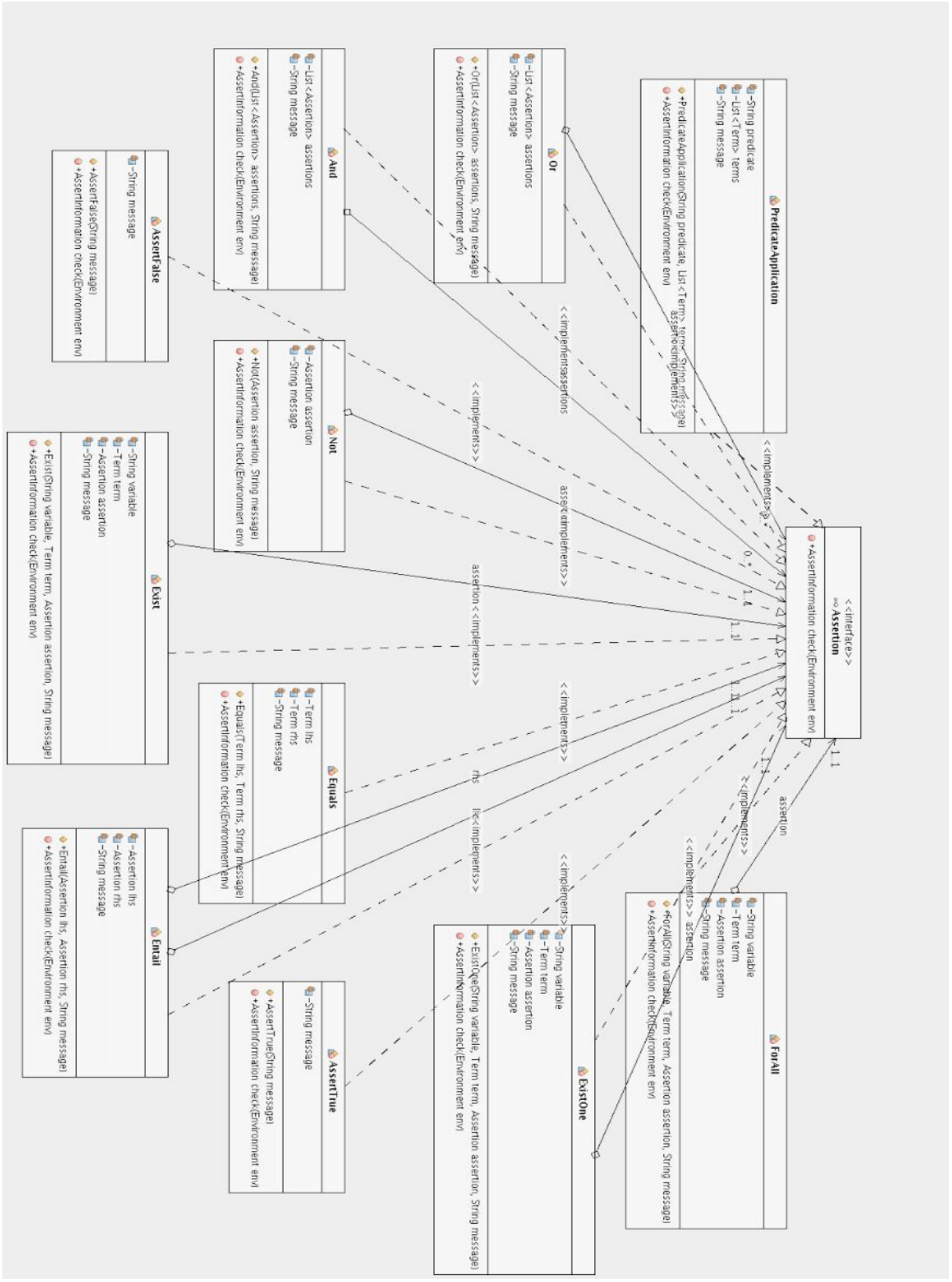
Requisitos a cumplir por el pom.xml

El proyecto debe tener de groupId: `es.ucm.povale` y el artifactId por convenio debe ser: `PoVALE-nombre-del-proyecto`. Por último, se debe añadir en sus dependencias el proyecto `PoVALE-core`, ya que tanto los predicados como las funciones de aplicación que se implementen en el plugin se ejecutarán mediante los términos y asertos de `PoVALE-core`. A continuación mostraremos dos fragmentos del fichero pom.xml del plugin `PoVALE-plugin-lines`:

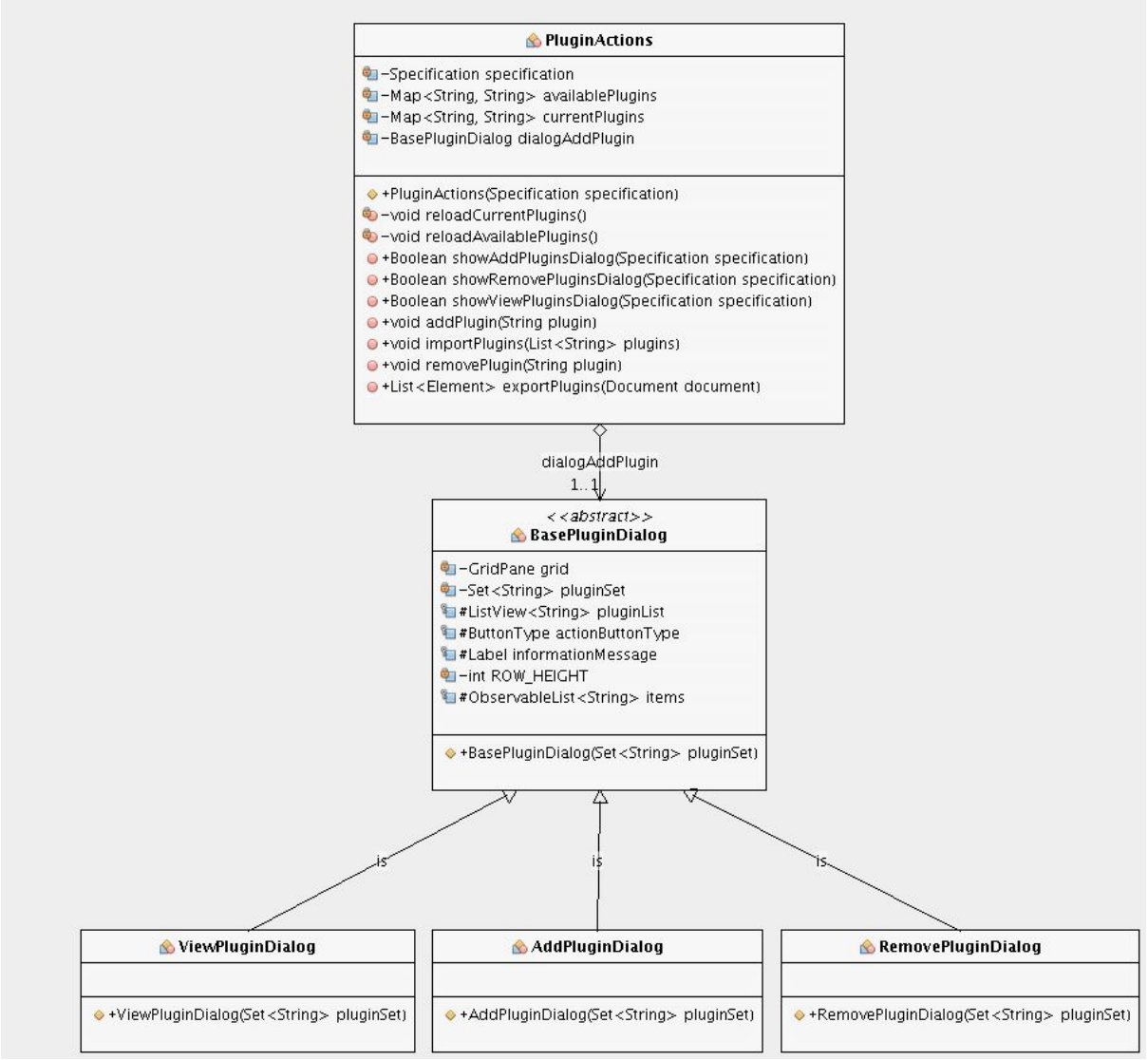
```
<groupId>es.ucm.povale</groupId>
<artifactId>PoVALE-plugin-files</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>

<dependency>
    <groupId>es.ucm.povale</groupId>
    <artifactId>PoVALE-core</artifactId>
    <version>1.0-SNAPSHOT</version>
    <type>jar</type>
</dependency>
```

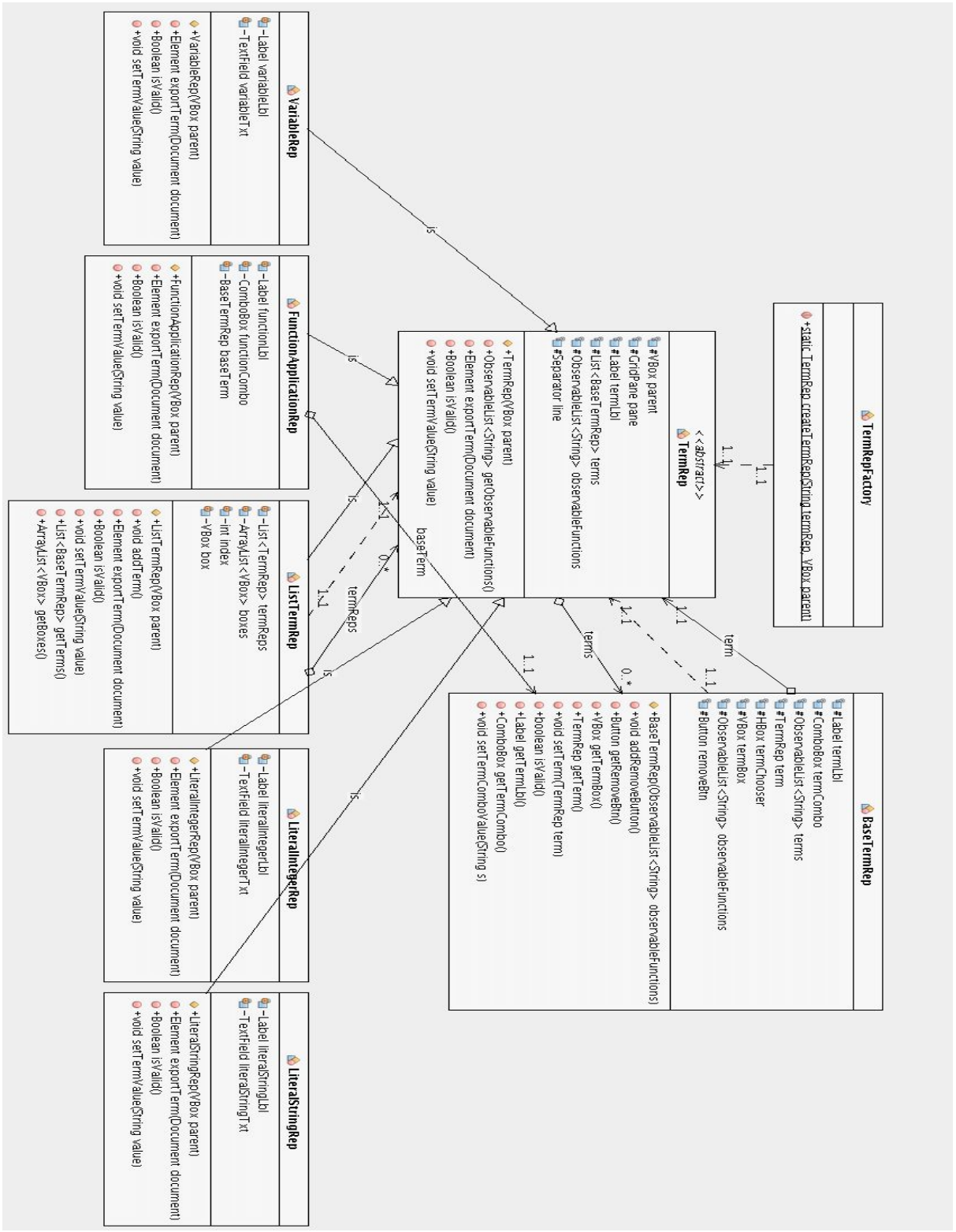
14. Apéndice: PoVALE-core: Assertion



15. Apéndice: PoVALE-specification: PluginSpec



16. Apéndice: PoVALE-specification: TermRep



Bibliografía

[1] Heckler, Grunwald, Pereda, Phillips, Dea, **JavaFX 8: Introduction by Example**, Apress, ISBN: 978-1430264606, 2014.

[2] Sharan, **Learn JavaFX 8: Building User Experience and Interfaces with Java 8**, Apress, ISBN: 978-1484211434, 2014.

[3] Weaver, Gao, Chin, Iverson, Vos, **Pro JavaFX 8: A Definitive Guide to Building Desktop, Mobile, and Embedded Java Clients**, Apress. ISBN: 978-1430265740, 2014.

[4] Eckel, **Thinking in Java**, Prentice Hall. ISBN: 978-0131872486, 2006.

[5] Forman, R. Forman, **Java Reflection in Action**, Manning. ISBN: 978-1932394184, 2004.

[6] Oracle, **JavaFX Scene Builder**, https://docs.oracle.com/javase/8/scene-builder-2/JSBID.pdf.

[7] T. Ray, **Learning XML**, O'Reilly. ISBN: 978-0596004200, 2001.

[8] St. Laurent, Fitzgerald, **XML Pocket Reference**, O'Reilly. ISBN: 978-0596100506, 2005.

[9] W3C, **XML DOM**, https://www.w3schools.com/xml/xml_dom.asp.

[10] Salter, **NetBeans IDE 8 CookBook**, Packt Publishing. ISBN: 978-1782167761, 2014.

[11] Chacon, Straub, **Pro Git**, Apress. ISBN: 978-1484200773, 2014.

[12] Sonatype Company, **Maven: The Definitive Guide**, O'Reilly Media. ISBN: 978-0-596-51733-5, 2008.

[13] Ort, Metha (Oracle), **Java Architecture for XML Binding (JAXB)**, www.oracle.com/technetwork/articles/javase/index-140168.html.

[14] EEES, www.eees.es.