Máster en Cloud Apps
Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2019/2020

Trabajo de Fin de Máster

UML Diagrams Lib

Autor: Pablo José Calvo Sastre
Tutor: Luis Fernández

Resumen

Este Trabajo de Fin de Máster ha tenido como objetivo crear un prototipo de herramienta que nos permita generar documentación UML de manera efectiva, eficaz, eficiente, sin errores y de forma automatizada. Para ello se han seguido varias vías de trabajo.

La primera vía de trabajo ha consistido en facilitar la generación de documentación UML mediante la creación de un modelo de dominio. Para ello se ha implementado un patrón builder para ir añadiendo entidades al dominio, haciendo así la creación del modelo del dominio más amigable para una persona con conocimientos de programación.

En relación con la anterior se ha realizado otra vía de trabajo consistente en analizar un código ya creado obteniendo de éste la documentación UML de su arquitectura. Para ello, mediante ingeniería inversa se analizan todas las clases existentes y las relaciones que hay entre ellas, cargando dinámicamente estas sobre un modelo de dominio.

De esta forma podemos obtener documentación UML que nos permita comparar el diseño inicial de la arquitectura y su evolución a lo largo del tiempo con distintos tipos de plantillas para distintos diagramas.

Por último, se ha trabajado en obtener la documentación en formato gráfico del documento UML creado anteriormente. Para llegar a este punto, primero se ha tenido que convertir el modelo del dominio a lenguaje PlantUML, y posteriormente se ha realizado una investigación, para poder enviar esta información en el formato adecuado para los servidores de PlantUML y que estos nos devuelvan la imagen.

Introducción y objetivos

Este trabajo surge con la idea de poder agilizar la creación de documentación UML para el diseño arquitectónico de aplicaciones. Así como el análisis de su evolución.

Actualmente una de las maneras de generar documentación UML es mediante la herramienta de código abierto PlantUML, que consiste en un lenguaje de texto sin formato que interpretado por el servidor de PlantUML, nos genera el diagrama UML que hayamos creado con este lenguaje.

El crear documentación UML mediante este sistema es costoso en tiempo, ya que modelar la arquitectura de una aplicación de cierto tamaño requiere de mucho tiempo y esfuerzo, pues es sencillo cometer errores de sintaxis en este lenguaje.

El objetivo de este TFM ha sido crear un prototipo de herramienta que nos permita generar documentación UML basándonos en el lenguaje de PlantUML.

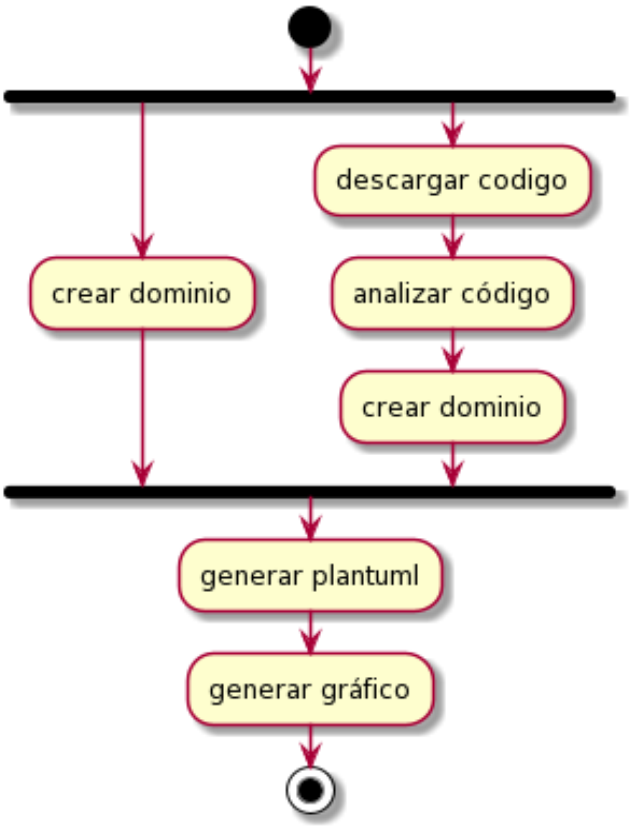
Para ello se ha optado por facilitar la generación de documentación sobre proyectos ya creados, mediante un analizador de código, que con ingeniería inversa hace un análisis de las clases existentes y sus relaciones. Este código puede estar alojado en nuestra máquina, o podemos descargarlo de un repositorio de Git para su análisis.

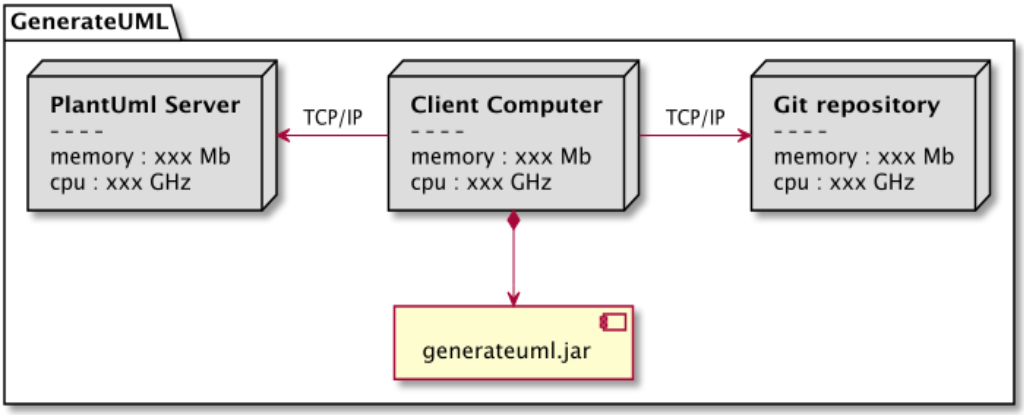
También se ha desarrollado una forma para la creación de nuevos modelos de dominio de manera programática, facilitando así su creación y reduciendo los posibles errores.

Tanto si el modelo del dominio lo hemos obtenido analizando un código ya existente, como si lo hemos creado nosotros, para poder obtener la documentación UML, primero convertiremos el modelo del dominio a lenguaje PlantUML.

Una vez tenemos nuestro modelo de dominio convertido a lenguaje PlantUML, haremos una petición a los servidores de PlantUML que convertirá la información enviada en lenguaje PlantUML a una imagen, que guardaremos en nuestra máquina.

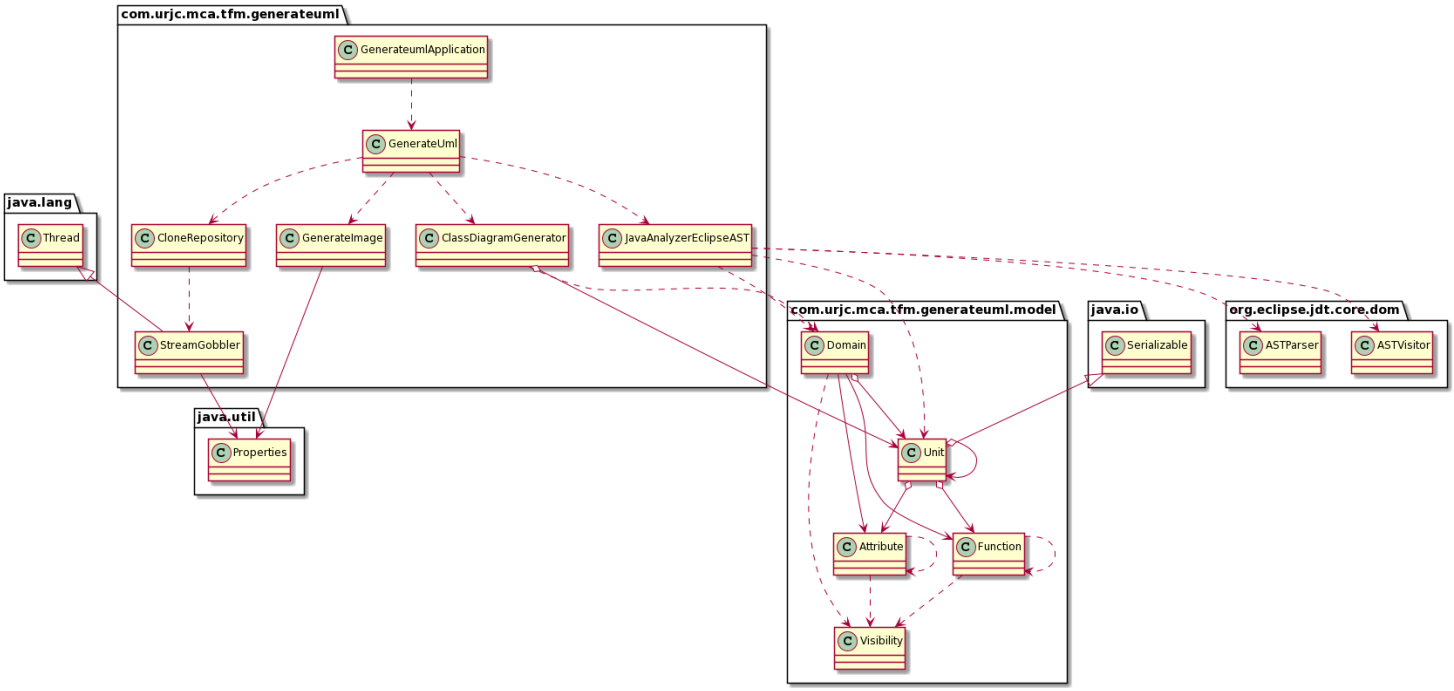
Podemos resumir la funcionalidad de la herramienta de esta manera.





Aquí vemos las conexiones que puede realizar la herramienta con los servidores de PlantUML, para la generación de los gráficos, y un repositorio de Git para la descarga del proyecto.

Diagrama de la arquitectura de la aplicación

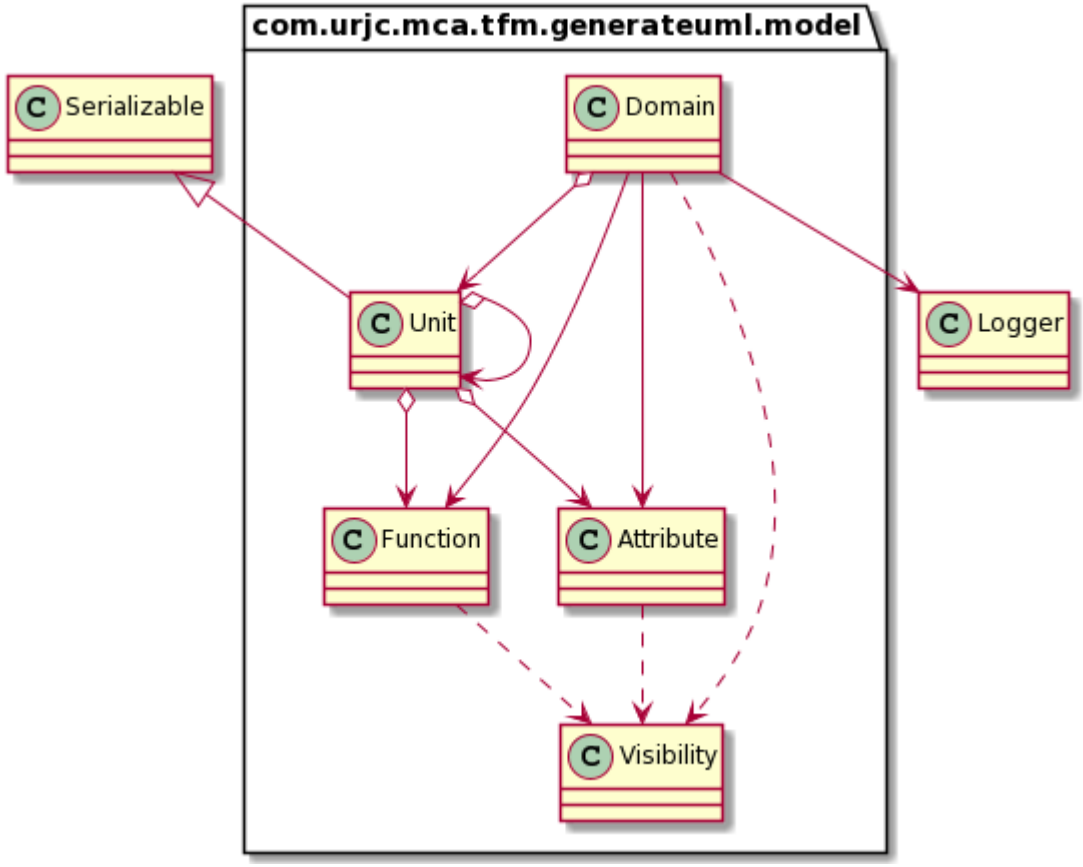


El repositorio de la herramienta se encuentra aquí.

1. Creación del modelo

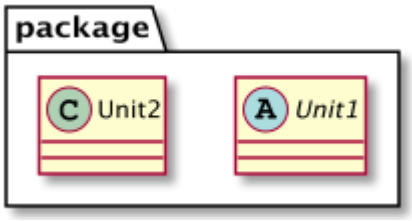
Para crear el modelo de dominio se ha decidido implementar un patrón bulider especial con pila de tipo unidades, de esta manera añadimos información sobre la última unidad apilada. Esto nos permite tener un grafo con las relaciones de las unidades del modelo del dominio.

La clase sobre la que iremos creando el modelo del dominio es la clase Dominio y usando los diferentes métodos de los que dispone, podremos ir añadiendo unidades al modelo, o funciones y/o atributos a la unidad activa.



Para poder añadir correctamente la información al dominio tendremos un elemento activo (el último apilado) que será sobre el que iremos añadiendo la información. Este puede ser de tipo unidad, atributo o función.

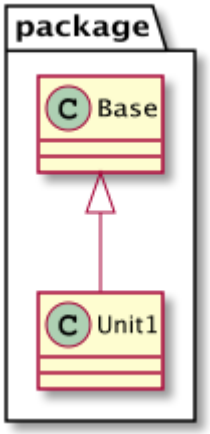
```
domain.addPackage("package")
      .addUnit("Unit1")
      .setAbstractUnit()
      .addUnit("Unit2");
```



A la última unidad apilada le podemos añadir mediante estos métodos, unidades, funciones o atributos.

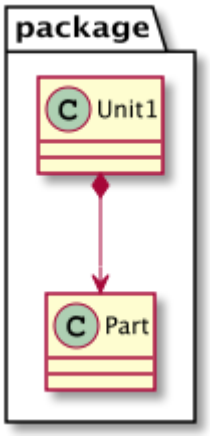
1.1. addBase

```
domain.addPackage("package")
      .addUnit("Unit1")
      .addBase("Base");
```



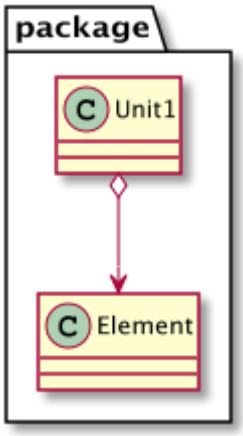
1.2. addPart

```
domain.addPackage("package")
      .addUnit("Unit1")
      .addPart("Part");
```



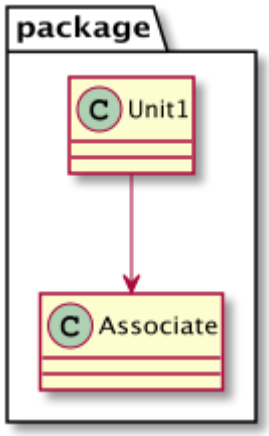
1.3. addElement

```
domain.addPackage("package")
      .addUnit("Unit1")
      .addElement("Element");
```



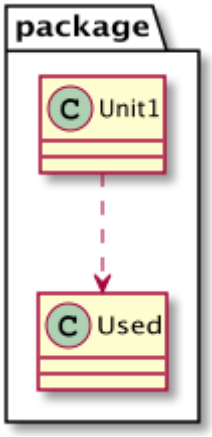
1.4. addAssociate

```
domain.addPackage("package")
      .addUnit("Unit1")
      .addAssociate("Associate");
```



1.5. addUsed

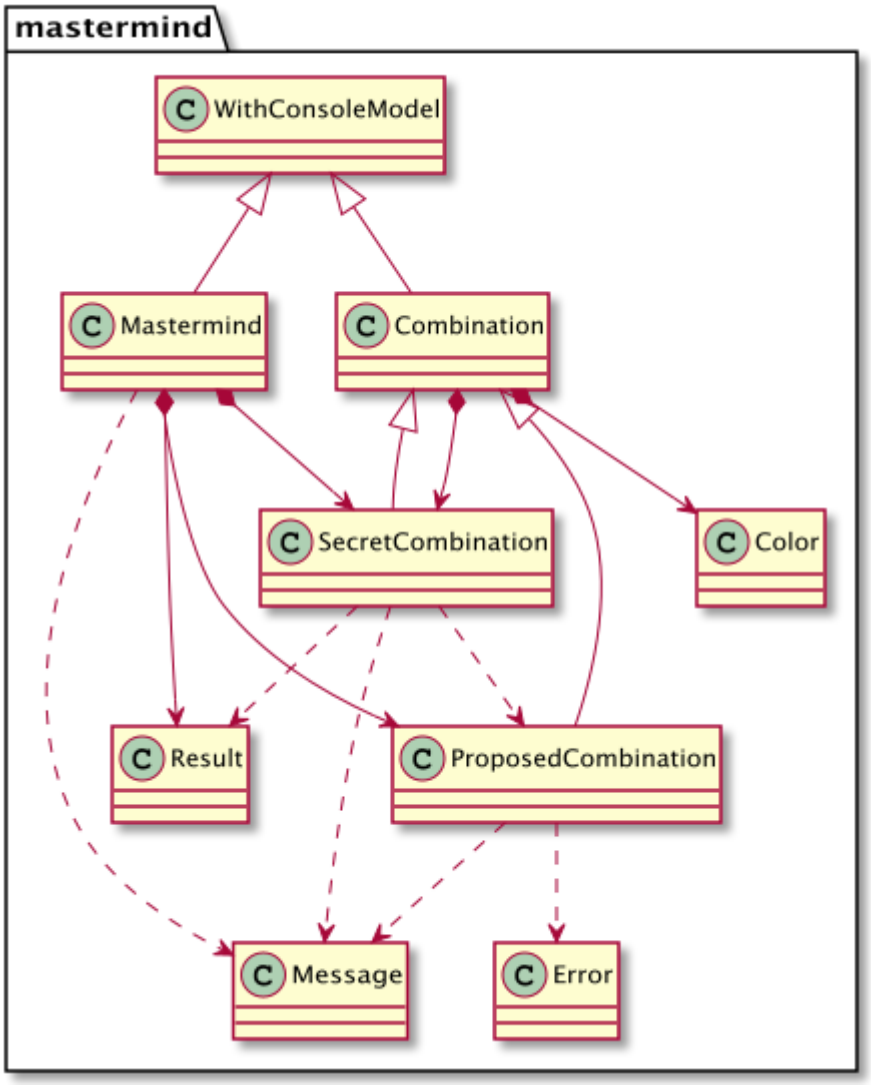
```
domain.addPackage("package")
      .addUnit("Unit1")
      .addUsed("Used");
```



Ahora vamos a ver un ejemplo usando todos a la vez usando como referencia la práctica 1 de la asignatura de diseño y calidad del software del máster cloud apps.

```
domain.addPackage("mastermind")
      .addUnit("Mastermind")
            .addBase("WithConsoleModel")
            .addPart("SecretCombination")
            .addPart("ProposedCombination")
            .addPart("Result")
```

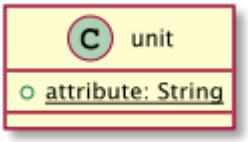
```
        .addUsed("Message")
    .addUnit("Combination")
        .addBase("WithConsoleModel")
        .addPart("Color")
        .addPart("SecretCombination")
    .addUnit("SecretCombination")
        .addBase("Combination")
        .addUsed("ProposedCombination")
        .addUsed("Message")
        .addUsed("Result")
    .addUnit("ProposedCombination")
        .addBase("Combination")
        .addUsed("Error")
        .addUsed("Message");
```



Además de los métodos que hemos visto, también podemos añadir a una unidad atributos o funciones.

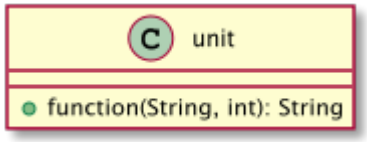
1.6. Atributos

```
domain.addUnit("unit")
      .addAttribute("attribute")
      .setType("String")
      .addVisibility(Visibility.PUBLIC)
      .setStatic(true);
```

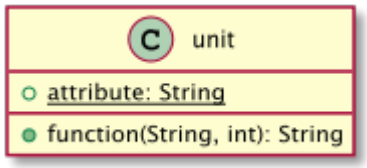


1.7. Funciones

```
String[] parameters = {"String", "int"};
domain.addUnit("unit")
     .addFunction("function")
     .addVisibility(Visibility.PUBLIC)
     .addReturnType("String")
     .addParameters(parameters);
```



Y la combinación de ambos



2. Generar documentación

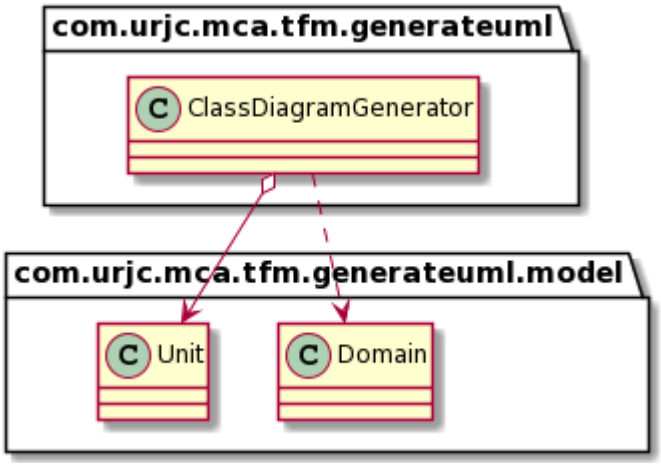
Para poder generar la documentación UML vamos a hacer uso de la herramienta de código abierto PlantUML que permite a los usuarios crear diagramas UML a partir de un lenguaje de texto sin formato.

La obtención de este texto se realiza a partir de la información del modelo del dominio recorriendo cada una de las unidades, delegando en estas la generación de su información en lenguaje PlantUML.

Con el texto generado enviaremos esa información al servidor de PlantUML y nos descargará la imagen generada del diagrama UML. Este punto lo ampliaremos más adelante.

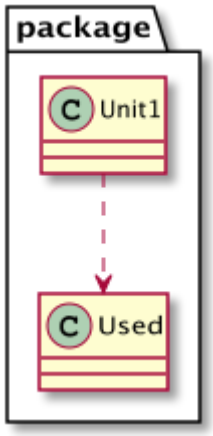
2.1. Generar PlantUML

Para poder generar PlantUML a partir del modelo del dominio, hacemos uso de la clase **ClassDiagramGenerator** que recorrerá toda la colección de unidades que previamente le habremos facilitado, delegando en cada una la responsabilidad de generar su información en formato texto para PlantUML.



Un ejemplo de texto PlantUML obtenido de la clase ClassDiagramGenerator

```
class package.Unit1
class package.Used
package.Unit1 ..> package.Used
```



Además de obtener la información de todas las unidades añadidas, la clase **ClassDiagramGenerate** nos provee de una serie de posibilidades a la hora de generar el texto para el PlantUML dentro de la información del dominio.

2.1.1. Obtener PlantUML del dominio

Partiendo de este código de ejemplo visto anteriormente de la práctica 1 de diseño y calidad del software.

Código

```
domain.addPackage("mastermind")
      .addUnit("Mastermind")
            .addBase("WithConsoleModel")
            .addPart("SecretCombination")
            .addPart("ProposedCombination")
            .addPart("Result")
            .addUsed("Message")
      .addUnit("Combination")
            .addBase("WithConsoleModel")
            .addPart("Color")
            .addPart("SecretCombination")
      .addUnit("SecretCombination")
            .addBase("Combination")
            .addUsed("ProposedCombination")
            .addUsed("Message")
            .addUsed("Result")
      .addUnit("ProposedCombination")
            .addBase("Combination")
            .addUsed("Error")
            .addUsed("Message");
```

Para obtener el diagrama de clases visto antes debemos añadir el dominio y posteriormente llamar al método **print** de la clase **ClassDiagramGenerate**

```
classDiagram.addDomain(domain);
classDiagram.print();
```

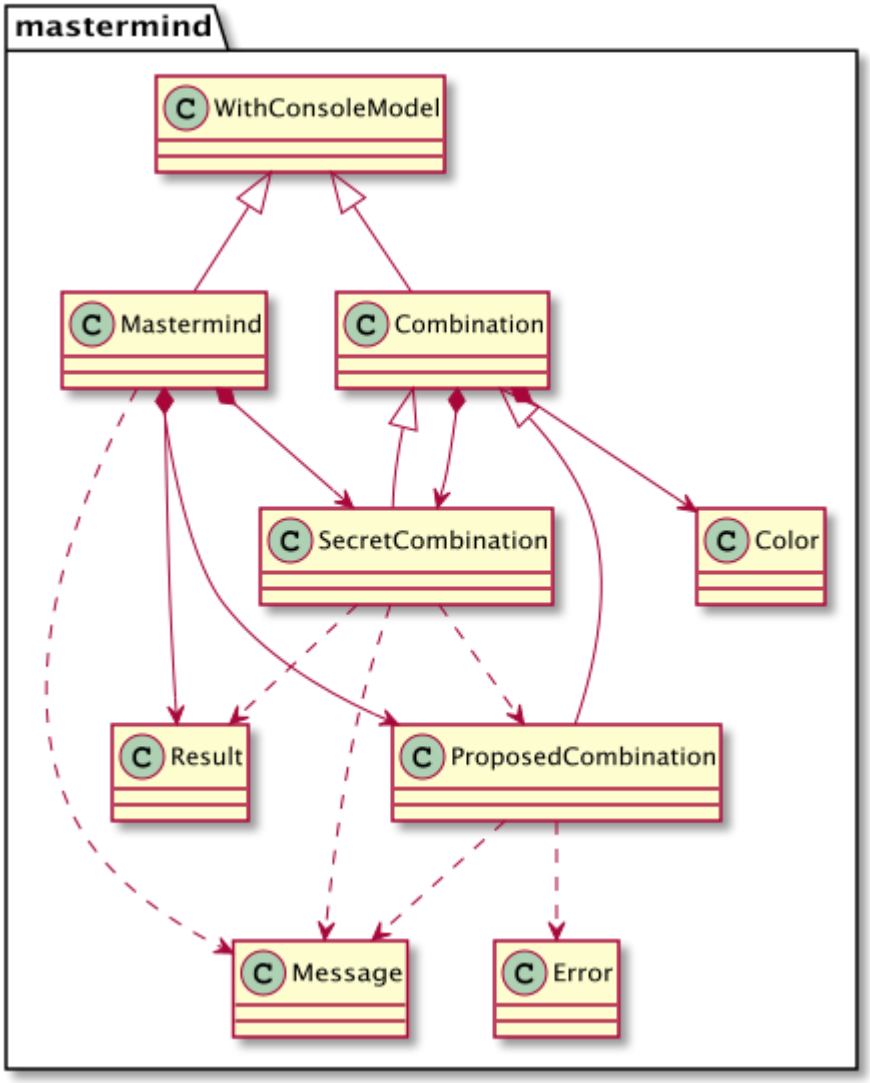
Print() nos devolverá esta cadena de texto

Texto

```
class mastermind.Mastermind
class mastermind.WithConsoleModel
class mastermind.SecretCombination
class mastermind.ProposedCombination
class mastermind.Result
class mastermind.Message
class mastermind.Combination
class mastermind.Color
class mastermind.Error
mastermind.WithConsoleModel <|-- mastermind.Mastermind
mastermind.Mastermind *--> mastermind.SecretCombination
mastermind.Mastermind *--> mastermind.ProposedCombination
mastermind.Mastermind *--> mastermind.Result
mastermind.Mastermind ..> mastermind.Message
mastermind.Combination <|-- mastermind.SecretCombination
mastermind.SecretCombination ..> mastermind.Message
mastermind.SecretCombination ..>
mastermind.ProposedCombination
mastermind.SecretCombination ..> mastermind.Result
mastermind.Combination <|-- mastermind.ProposedCombination
mastermind.ProposedCombination ..> mastermind.Message
mastermind.ProposedCombination ..> mastermind.Error
mastermind.WithConsoleModel <|-- mastermind.Combination
mastermind.Combination *--> mastermind.Color
mastermind.Combination *--> mastermind.SecretCombination
```

Que genera la imagen vista antes

Imagen



2.1.2. Acoplamiento Eferente

Siguiendo con el ejemplo anterior, vamos a obtener sólo la unidad SecretCombination y sus relaciones.

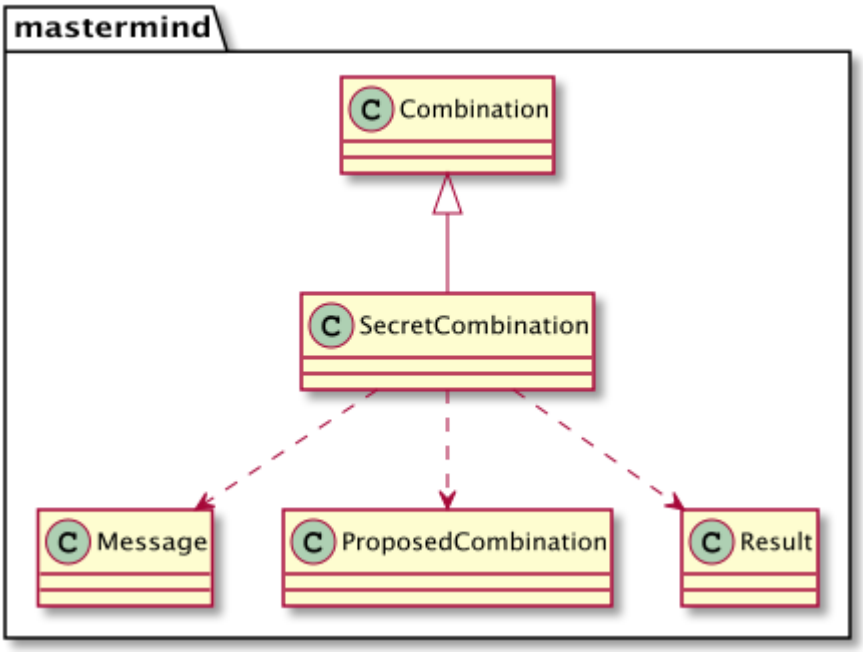
Código

```
classDiagram.addUnits(domain.getEfferent("SecretCombination"));
classDiagram.print()
```

Texto

```
class mastermind.SecretCombination
mastermind.Combination <|-- mastermind.SecretCombination
mastermind.SecretCombination ..> mastermind.Message
mastermind.SecretCombination ..> mastermind.ProposedCombination
mastermind.SecretCombination ..> mastermind.Result
```

Imagen



2.1.3. Acoplamiento Aferente y Eferente

Además de obtener las relaciones de una de las unidades del dominio como acabamos de ver, también es posible obtener el grafo de acoplamiento eferente y aferente tanto de una unidad en concreto como de todo el dominio.

Para estos 4 ejemplos vamos a usar el mismo dominio.

```
domain.addUnit("X")
            .addBase("Base_de_X")
            .addPart("Parte_de_X")
            .addAssociate("Asociada_de_X")
            .addUsed("Usada_por_X")
      .addUnit("Todo_de_X")
            .addPart("X")
      .addUnit("Usa_X")
            .addUsed("X")
      .addUnit("Asociado_a_X")
            .addAssociate("X")
      .addUnit("Descendiente_de_X")
            .addBase("X");
```

2.1.3.1. Aferentes de todas las unidades del dominio

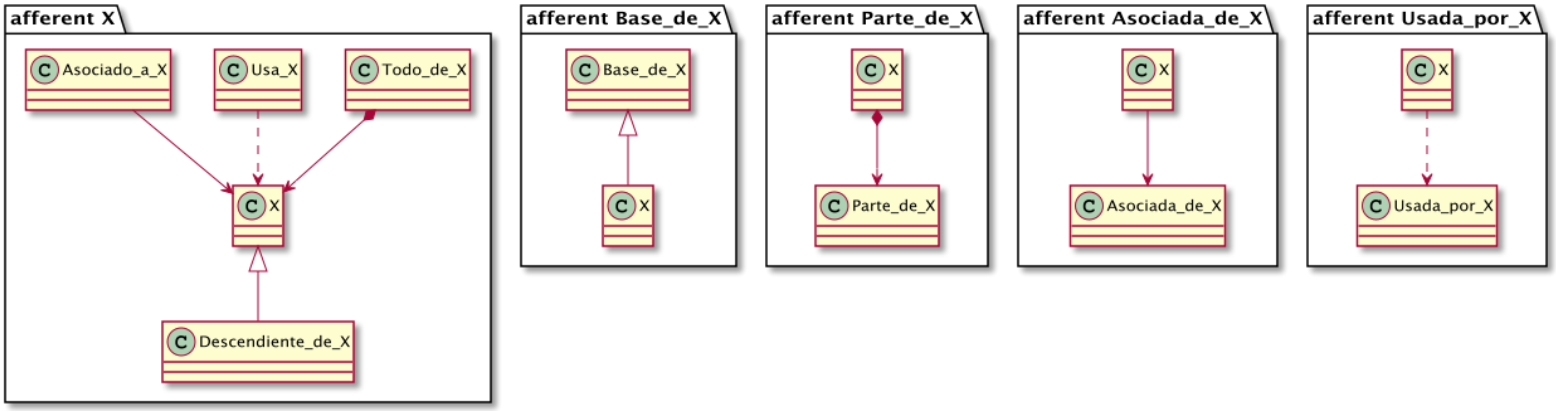
Código

```
classDiagram.addUnits(domain.getAllAfferent());
classDiagram.print();
```

Texto

```
class "afferent X.Todo_de_X"
class "afferent X.Usa_X"
class "afferent X.Asociado_a_X"
class "afferent X.Descendiente_de_X"
class "afferent Base_de_X.X"
class "afferent Parte_de_X.X"
class "afferent Asociada_de_X.X"
class "afferent Usada_por_X.X"
"afferent X.Todo_de_X" *--> "afferent X.X"
"afferent X.Usa_X" ..> "afferent X.X"
"afferent X.Asociado_a_X" --> "afferent X.X"
"afferent X.X" <|-- "afferent X.Descendiente_de_X"
"afferent Base_de_X.Base_de_X" <|-- "afferent Base_de_X.X"
"afferent Parte_de_X.X" *--> "afferent Parte_de_X.Parte_de_X"
"afferent Asociada_de_X.X" --> "afferent
Asociada_de_X.Asociada_de_X"
"afferent Usada_por_X.X" ..> "afferent Usada_por_X.Usada_por_X"
```

Imagen



2.1.3.2. Aferentes de una Clase

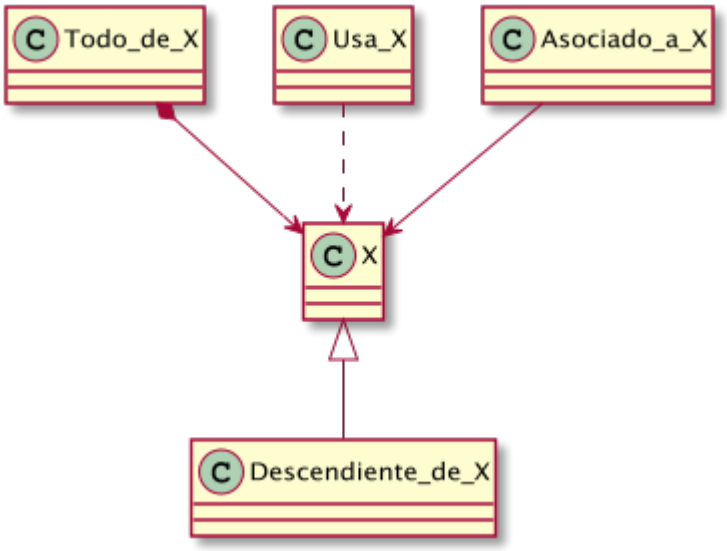
Código

```
classDiagram.addUnits(domain.getAfferent("X"));
classDiagram.print();
```

Texto

```
class Todo_de_X
class Usa_X
class Asociado_a_X
class Descendiente_de_X
Todo_de_X *--> X
Usa_X ..> X
Asociado_a_X --> X
X <|-- Descendiente_de_X
```

Imagen



2.1.3.3. Eferentes de todas las unidades del dominio

Código

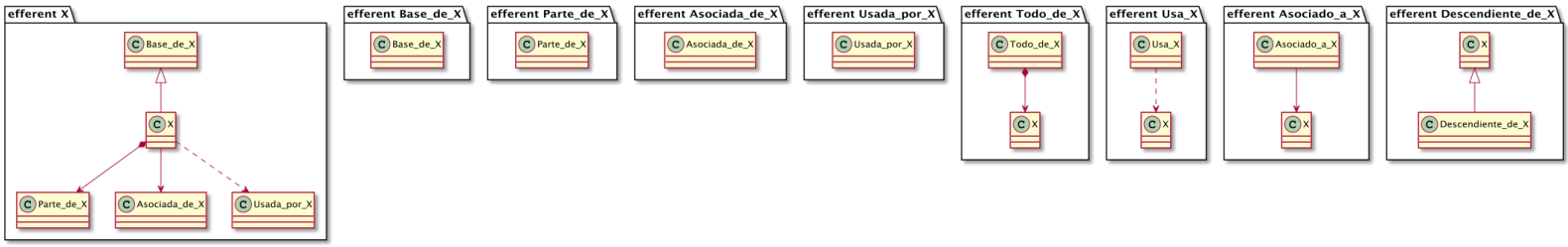
```
classDiagram.addUnits(domain.getAllEfferents());
classDiagram.print();
```

Texto

```
class "efferent X.X"
class "efferent Base_de_X.Base_de_X"
class "efferent Parte_de_X.Parte_de_X"
```

```
class "efferent Asociada_de_X.Asociada_de_X"
class "efferent Usada_por_X.Usada_por_X"
class "efferent Todo_de_X.Todo_de_X"
class "efferent Usa_X.Usa_X"
class "efferent Asociado_a_X.Asociado_a_X"
class "efferent Descendiente_de_X.Descendiente_de_X"
"efferent X.Base_de_X" <|-- "efferent X.X"
"efferent X.X" *--> "efferent X.Parte_de_X"
"efferent X.X" --> "efferent X.Asociada_de_X"
"efferent X.X" ..> "efferent X.Usada_por_X"
"efferent Todo_de_X.Todo_de_X" *--> "efferent Todo_de_X.X"
"efferent Usa_X.Usa_X" ..> "efferent Usa_X.X"
"efferent Asociado_a_X.Asociado_a_X" --> "efferent Asociado_a_X.X"
"efferent Descendiente_de_X.X" <|-- "efferent
Descendiente_de_X.Descendiente_de_X"
```

Imagen



2.1.3.4. Eferentes de una Clase

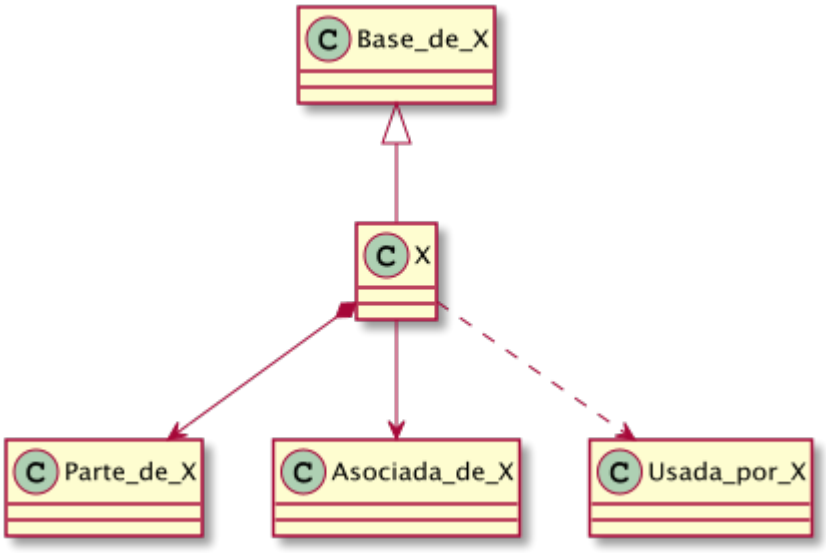
Código

```
classDiagram.addUnits(domain.getEfferent("X"));
classDiagram.print();
```

Texto

```
class X
Base_de_X <|-- X
X *--> Parte_de_X
X --> Asociada_de_X
X ..> Usada_por_X
```

Imagen



2.1.4. Mostrar un paquete concreto

En el caso de que tengamos un dominio con varios paquetes, podemos mostrar únicamente un paquete y sus relaciones con los otros paquetes de primer nivel.

Código

```
      domain.addPackage("mastermind")
                  .addUnit("MastermindStandalone")
                      .addBase("Mastermind")
                  .addPackage("mastermind.controllers")
                      .addUnit("AcceptorController")
                           .addUsed("ControllersVisitor")
                           .addUnit("Logic")
                  .addPackage("masterind")
                      .addUnit("Mastermind")
                      .addUsed("AcceptorController")
                      .addPart("Logic");

classDiagram.addDomain(domain);
classDiagram.print("mastermind");
```

Texto

```
class mastermind.MastermindStandalone
class mastermind.Mastermind
class mastermind.controllers.AcceptorController
class mastermind.controllers.Logic
mastermind.Mastermind <|-- mastermind.MastermindStandalone
mastermind.Mastermind *--> mastermind.controllers.Logic
mastermind.Mastermind ..> mastermind.controllers.AcceptorController
```

Imagen de todo el dominio

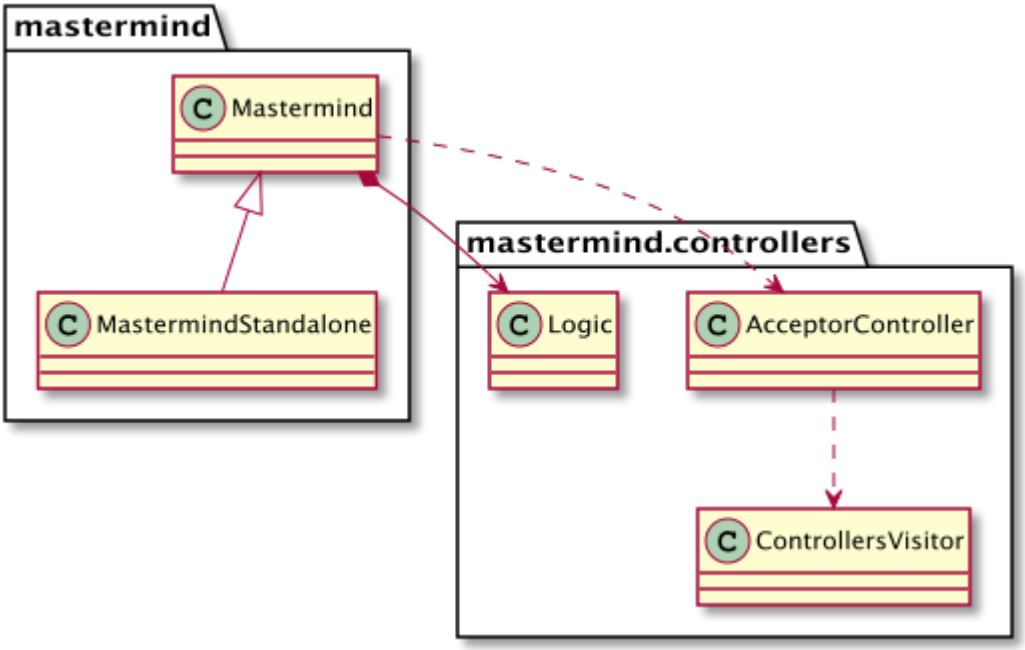
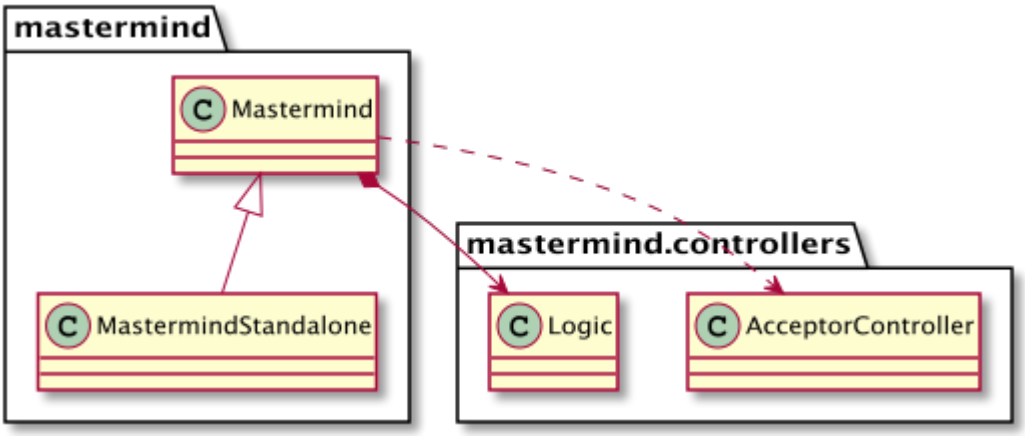


Imagen del dominio mastermind



Podemos ver como únicamente aparecen las relaciones a primer nivel con los demás paquetes.

2.1.5. Mostrar sólo paquetes

Y por último también podemos ver únicamente los paquetes que componen el dominio.

Código

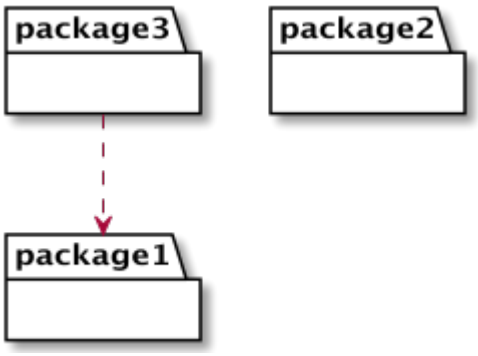
```
domain.addPackage("package1")
           .addUnit("unit1")
           .addUnit("unit2")
                .addUsed("used")
      .addPackage("package2")
           .addUnit("unit3")
             .addPackage("package3")
           .addUnit("unit4")
           .addUsed("unit1");

classDiagram.addDomain(domain);
classDiagram.printPackage();
```

Texto

```
package package1 {}
package package2 {}
package package3 {}
package3 ..> package1
```

Imagen



2.2. Generar gráfico

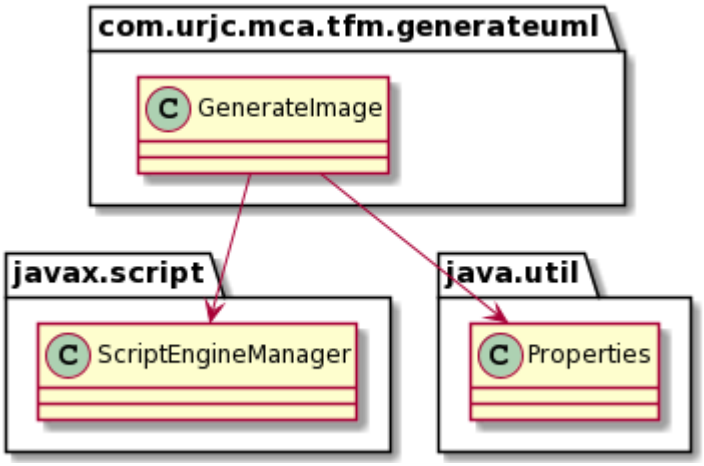
En este punto se hizo una investigación para poder invocar al servidor PlantUML desde la propia herramienta, facilitando así la creación de la documentación UML en formato gráfico y no sólo en lenguaje PlantUML.

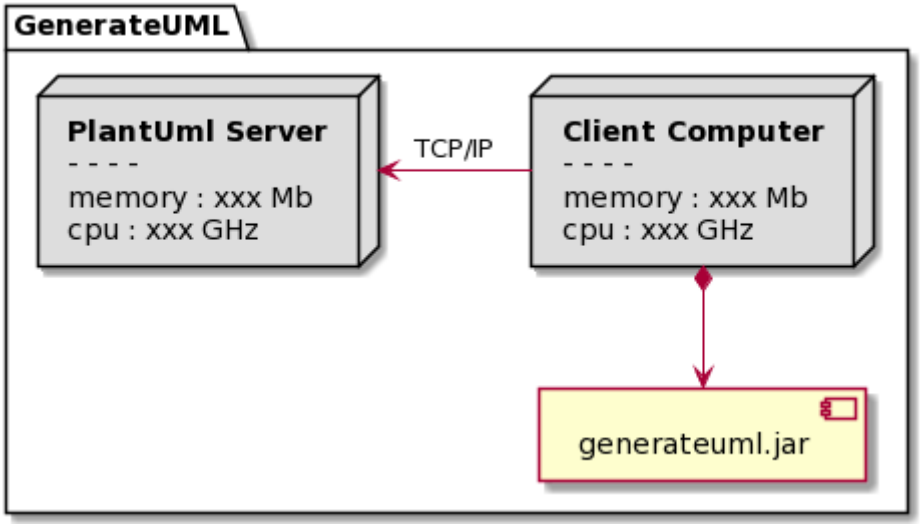
Para ello se hizo un trabajo de ingeniería inversa de la web planttext.com, para saber como codificar de forma adecuada el lenguaje PlantUML de manera que el servidor lo entienda correctamente y de esta forma nos genere la imagen.

En el análisis se descubrió que son necesarias dos ficheros JavaScript, en las cuales se encuentran una serie de métodos que son los encargados de codificar la información, por lo que se ha optado por descargar estos ficheros cada vez que vayamos a generar una imagen por si hubiera actualizaciones, para poder usarlos localmente. Estos ficheros se encuentran en la ruta ***resources/js***

La dirección del servidor, así como los nombres de los ficheros JavaScript, junto con otras propiedades, están definidas en una archivo de configuración (**application.properties**).

Este sería el diagrama de clases de la clase **GenerateImage** encarga de contactar con el servidor PlantUML para descargar la imagen en nuestra máquina.



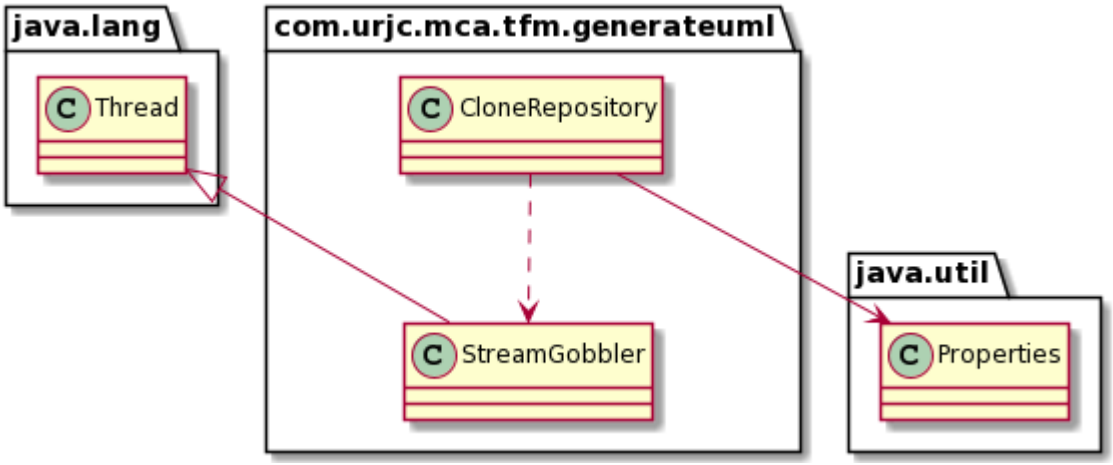


En este diagrama se muestra la conexión que establece nuestra máquina con el servidor de PlantUML.

3. Ingeniería Inversa

3.1. Descargar código

También disponemos de la opción de descargarnos el código de un repositorio git para su posterior análisis. La clase encargada de esto es **CloneRepository**, que clonará el repositorio en la ruta ***resources/repositories/{nombreProyecto}***



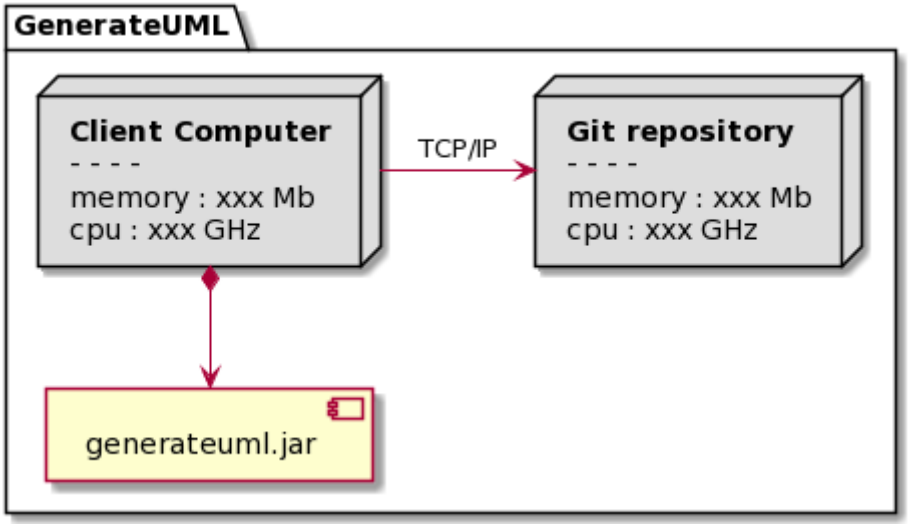


Diagrama que muestra la conexión de nuestra máquina con el servidor que aloja el repositorio de de Git.

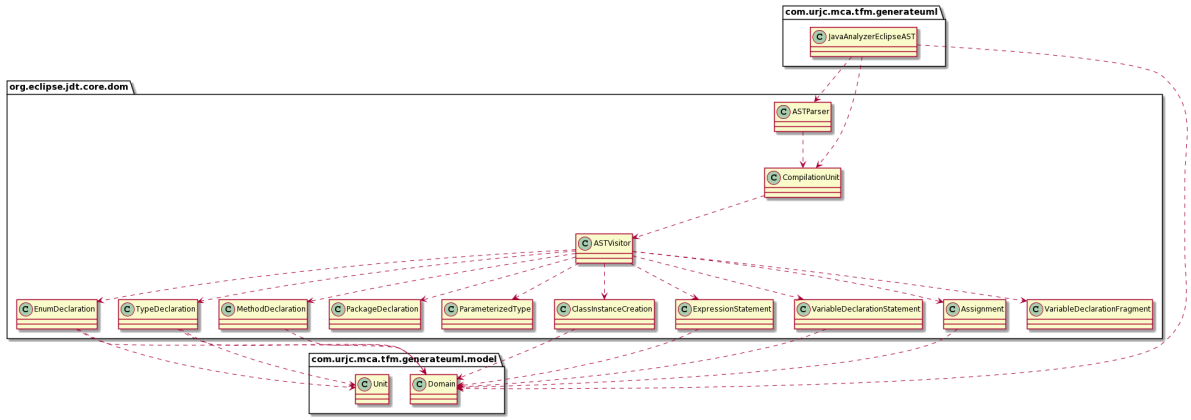
3.2. Analizar código

Como se ha comentado anteriormente, también es posible realizar un análisis mediante ingeniería inversa del código de una carpeta, usando el patrón Interpreter y el patrón Visitor, para crear un grafo del modelo del dominio. Este análisis también es posible realizarlo sobre una única clase.

El modelo del dominio generado será únicamente de las relaciones entre las diferentes unidades sin entrar en los atributos o funciones de estas.

Para poder realizar el análisis es necesario invocar el método **run** de la clase **JavaAnalyzerEclipseAST** pasándole por parámetro la ruta donde se encuentra el código a analizar.

Este es el diagrama de clases de la clase **JavaAnalyzerEclipseAST**



A la clase **ASTParser** se le suministra la ruta con el fichero que va a interpretar y el procesado de ese fichero, se lo pasamos por parámetro a la clase CompilationUnit.

Esta clase **CompilationUnit** usa la clase **ASTVisitor**, y mediante el patrón visitor, pasa por las diferentes clases que se observan en el diagrama de clases para poder obtener la información e ir modelando el modelo del domino.

El resultado de la ejecución nos devolverá el modelo del dominio creado.

4. Conclusiones

Como se ha podido comprobar a lo largo del documento, en el desarrollo de este prototipo de herramienta se han abordado patrones de diseño, diseño, investigación e ingeniería inversa.

Se ha logrado un prototipo funcional que mejora de manera sustancial los tiempos en la creación de documentación UML y su posterior análisis de su evolución.

Para ello se ha seguido una línea de trabajo diferenciada en los tres bloques que hemos visto a lo largo del documento.

En la primera parte, para la creación del modelo se ha buscado un diseño que nos permita que sea fácilmente extensible, pudiendo de esta manera ampliar el dominio incluyendo información para generar diagramas de secuencia, actividad, etc.

En la segunda, en la parte de la generación de la documentación para generar PlantUML, habría que crear vistas que se adapten a los nuevos diagramas que queramos obtener y que previamente han sido añadidos en el modelo del dominio. Al delegar la responsabilidad en cada uno de los elementos del dominio, el obtener su información en lenguaje PlantUML, nos permite que sea muy extensible.

Para la obtención del gráfico, al ser lenguaje PlantUML, no habría que realizar ninguna modificación en esta parte.

En la parte de ingeniería inversa, en un primer momento se optó por usar la librería archUnit, esta librería crea un árbol sintáctico abstracto del código, el cual interpretamos para obtener el modelo del dominio. Por contra, tiene la limitación de que solo analiza código compilado, por ese motivo se cambió la línea de trabajo con la librería ASTParse de Eclipse. Pese a ello no se ha eliminado del código ya que posee métodos que te analizan la arquitectura, como por ejemplo decirte los ciclos que tiene y esto podría ser una línea de trabajo en el futuro a convivir con la librería de Eclipse.

Respecto a la librería ASTParser, nos permite analizar el código, aunque no esté compilado, pero tiene el problema de que es más complejo analizar las relaciones entre clases ya que en sentencias no te interpreta los tipos de los objetos como sí hace la librería de archUnit, y que, si la clase no se encuentra en la carpeta analizada, no tendrá un paquete en el grafo que genera del modelo del dominio.

Este apartado se puede mejorar depurando las dependencias entre unidades, añadiendo métodos y atributos a las unidades, etc.

También como posible mejora sería añadirle opciones de personalización de plantillas, colores, tamaños y fuentes a la hora de generar el diagrama en PlantUML.

Como trabajo futuro se podrá trasladar la manera de crear el modelo de dominio mediante builders a una web exponiendo un api que interprete esa información y nos devuelva la imagen del diagrama creado.

Una vez el punto anterior estuviera resuelto, también se podría crear usuarios para poder tener una estructura de dominios los cuales tuvieras definidos en tu perfil y los puedas consultar y modificar en la web.

Bibliografía

Material de la asignatura de Diseño y Calidad del Software del Master CloudApps.

Material de la asignatura de Patrones y Arquitectura Software del Master CloudApps.

Documentación de la página de PlantUML https://plantuml.com/

Documentación de la página de archUnit https://www.archunit.org/

Documentación con ejemplos sobre la librería de AST de Eclipse
https://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html
https://www.vogella.com/tutorials/EclipseJDT/article.html
https://www.programcreek.com/2011/01/best-java-development-tooling-jdt-and-astparser-tutorials/