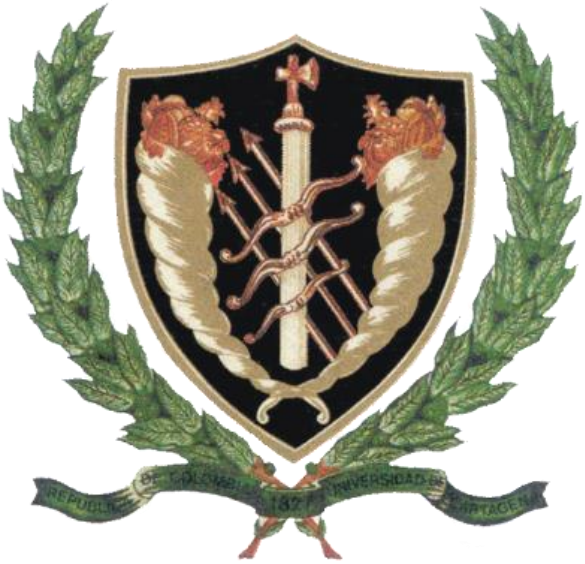
TRAZABILIDAD DEL PASO DE MENSAJES A PARTIR DE CÓDIGO FUENTE ORIENTADO A OBJETOS

INVESTIGADORES

Víctor Fernando Del Rio Prens



UNIVERSIDAD DE CARTAGENA

FACULTAD DE INGENIERÍA

PROGRAMA DE INGENIERÍA DE SISTEMAS

CARTAGENA DE INDIAS, 2017

TRAZABILIDAD DEL PASO DE MENSAJES A PARTIR DE CÓDIGO FUENTE ORIENTADO A OBJETOS

TÉSIS DE GRADO

Ingeniería De Software

INVESTIGADORES

Víctor Fernando Del Rio Prens

Director: Martín Monroy Ríos, MSc, PhD. (Universidad de Cartagena)

UNIVERSIDAD DE CARTAGENA

FACULTAD DE INGENIERÍA

PROGRAMA DE INGENIERÍA DE SISTEMAS

CARTAGENA DE INDIAS, 2017

UNIVERSIDAD DE CARTAGENA

Tesis de Grado:	TRAZABILIDAD DEL PASO DE MENSAJES A PARTIR DE CÓDIGO FUENTE ORIENTADO A OBJETOS
Autores:	VICTOR FERNANDO DEL RIO PRENS
Director:	Msc. MARTÍN MONROY RÍOS

Nota de Aceptación

_____

_____

_____

_____

_____

Presidente del Jurado

_____

Jurado

_____

Jurado

Cartagena de Indias, ____ de _____ de 2017

TABLA DE CONTENIDO

INDICE DE ILUSTRACIONES

ABSTRACT

A software tool that allows to record the traceability of message passing from object oriented source code in Java language was developed, thus creating object flow graphs. Later, the architectural analysis of both the design and the real world systems was carried out. In addition, the bases on which the criteria for the creation of the software are established, and the antecedents that are had in the field to which it aims were approached. The extreme programming model adapted for individual development was used as the basis for the creation of this system. Also, the progress made with the software was shown and included each of the results obtained from it, among which are the creation of the system architecture and the implementation of the system, creating a component capable of generating an OFG From object-oriented java source code. Finally, the conclusions from the development and the final product were drawn, among which the care with the handling of the regular expressions stands out, since they have the peculiarity that if the repetitive characters are used of wrong form as it is the case of the Asterisk (*) and of the plus (+), when analyzing them they realize a quantity of iterations too big that ends in an operations overload for the PC that executes the analysis.

RESUMEN

Se desarrolló una herramienta software que permite registrar la trazabilidad del paso de mensajes a partir de código fuente orientado a objetos en lenguaje Java, creando así grafos de flujo de objetos. Luego se realizó el análisis arquitectónico del sistema tanto de diseño como del mundo real. Además, se abordaron las bases sobre las cuales se establecen los criterios para la creación del software, y los antecedentes que se tienen en el campo al cual apunta. Se usó como base para la creación de este sistema el modelo de programación extrema adaptado para el desarrollo individual. También, se mostró el avance realizado con el software y se incluyeron cada uno de los resultados obtenidos a partir del mismo, entre los que se encuentran el diseño de la arquitectura del sistema y la implementación de ésta, creando un componente capaz de generar un OFG a partir de código fuente java orientado a objetos. Finalmente, se plasmaron las conclusiones obtenidas del desarrollo y del producto final, entre las que destaca el cuidado con el manejo de las expresiones regulares, ya que poseen la peculiaridad de que si se utilizan de forma errada los caracteres de repetición como es el caso del asterisco (*) y del mas (+), al analizarlas realizan una cantidad de iteraciones demasiado grande que termina en una sobrecarga de operaciones para el PC que ejecuta el análisis.

INTRODUCCION

A partir de la literatura encontrada se analizó la arquitectura necesaria para el diseño y la implementación del software de trazabilidad del paso de mensajes a partir de código fuente orientado a objetos. Se tomó especialmente como base la literatura de Tonella y Potrich (2005), la cual posee todo el planteamiento para crear un generador de OFG. El aplicativo se desarrolló como tesis de grado para la universidad de Cartagena, y hace parte de la línea de investigación "ingeniería de software", la cual es una de las líneas investigativas del grupo E-Soluciones. Los resultados obtenidos en la investigación, sirven como instrumento de verificación de la gramatica propuesta por Tonella y Potrich (2005).

1.1. ANTECEDENTES

Actualmente los sistemas o también llamados productos software, son completamente indispensables para casi todas las labores que se realizan, ya sea en el ámbito laboral, en el hogar, o incluso en la calle. Todo esto es debido a que esas líneas de código en realidad representan una a una la forma de solucionar problemas, problemas que existen en el día a día de las personas y que van desde los más sencillos, hasta los más complejos.

El asunto con estos productos software es su vasta cantidad de código, la cual los hace difíciles de analizar, y en la actualidad ese problema se ha vuelto aun mayor, ya que antes gracias a la poca capacidad que tenían los ordenadores, los programadores debían ser más cuidadosos a la hora de programar, porque siempre se hacía necesario tener en cuenta el tamaño de los programas. Pero en estos momentos la situación es muy diferente, con el gran avance realizado en estos últimos 20 o 30 años los artefactos tecnológicos ya no se ven atados al espacio de memoria existente, y gracias a eso, los programadores han obtenido libertades que son contraproducentes si no se manejan con cuidado.

Ahora, si bien es verdad que los programas actuales generalmente poseen una gran cantidad de líneas de código, también es verdad que existen los diagramas UML, los cuales permiten

obtener una vista determinada dependiendo de lo que se desee observar en el código (estructura o comportamiento).

Estos diagramas tienden a ser generados antes de crear el código, pero en ciertas circunstancias se crean aplicativos los cuales no poseen ningún diagrama estructural creado previamente, lo único que tienen es su código fuente. Para estos casos se utilizan herramientas que permiten la recuperación de diagramas estructurales a partir de código fuente, este proceso se llama ingeniería inversa. La mayoría de las herramientas que permiten este tipo de recuperación de diagramas estructurales, son conocidas como herramientas CASE (computer aided software engineering, ingeniería de software asistida por computadora), estas herramientas como su nombre lo indica, brindan apoyo informático para los procesos de ingeniería de software, como es el caso de: StarUML, Enterpryse Architect, EasyCASE, entre otras. No todas las herramientas CASE permiten procesos de recuperación de diagramas, solo aquellas que poseen un entorno de ingeniería inversa.

Es precisamente el fin de esta investigación, ayudar al desarrollo de un entorno de ingeniería inversa, el cual se encuentra embebido dentro de la tesis doctoral desarrollada en el programa perteneciente al docente Martin Emilio Monroy Ríos, llamada "Marco de referencia para la recuperación y análisis de vistas arquitectónicas de comportamiento". La forma en la que se brinda ese apoyo es a partir de un módulo, cuya finalidad es la recuperación del grafo de flujo de objetos, el cual permite el análisis detallado del comportamiento del sistema, por lo tanto, se convierte en un componente esencial en el desarrollo de la tesis doctoral. Ahora bien, este módulo se diferencia de otros módulos encontrados en entornos de ingeniería inversa por tres motivos, el primero es que podrá realizar la recuperación del grafo a partir de una entrada especifica (sentencia), el segundo es su diseño, el cual permitirá una implementación en cualquier lenguaje de programación orientado a objetos (diseño reutilizable), y el tercero es su salida la cual se encontrara tanto de forma gráfica, como en un formato XML, que puede ser usado para la recuperación del grafo por cualquier programa que interprete la estructura del diagrama que representa. Por lo anterior, se puede afirmar que esta investigación constituye un aporte al estado del arte, porque complementa la literatura actual

1.2. FORMULACION DEL PROBLEMA

¿Cómo implementar una herramienta software que permita registrar la trazabilidad del paso de mensajes a partir de código fuente orientado a objetos mediante la representación del grafo de flujo de objetos?

1.3. JUSTIFICACION

El desarrollo de la herramienta apoyó los procesos de investigación del programa. Esto debido a que es un componente que se encuentra embebido dentro de la tesis doctoral titulada "Marco de referencia para la recuperación y análisis de vistas arquitectónicas de comportamiento", realizada por el profesor Martin Emilio Monroy Ríos, que como ya se había mencionado con anterioridad tiene como resultado crear un entorno de ingeniería inversa, el cual es similar a una herramienta CASE, que permita la recuperación de vistas arquitectónicas de comportamiento de producto software. Pero además de eso, el proyecto estaría acompañando a otro conjunto de herramientas que son investigaciones realizadas en el programa, y que también conforman el entorno de ingeniería inversa, como es el caso de: "Entorno de ingeniería inversa para la integración de funcionalidades basado en una arquitectura orientada a componentes" por Eugenio Pájaro Almagro y Mario Barrios Pacheco; "Mecanismo para la conversión de archivos XML al lenguaje de intercambio de modelos software (XMI)" por Juan Camilo Beleño Vega; al igual que otras.

El producto final generado a partir de este proyecto, es el grafo de flujo de objetos, el cual permite obtener una perspectiva del comportamiento del sistema a partir de las relaciones que presentan sus objetos. He allí la importancia y el aporte significativo que brinda al sector de la ingeniería de software.

Es necesario recalcar, que esta herramienta no pretendió tomar todas las estructuras dentro de un código, sino que simplemente tomó aquellas relacionadas con invocaciones de métodos a partir de objetos, ya que tratar de abarcar todas y cada una de las diferentes partes pertenecientes a un programa a partir de su código fuente, era simplemente algo demasiado amplio para el ámbito de esta investigación. Sólo se quiso brindar una herramienta que facilite el análisis del comportamiento de una aplicación orientada a objetos.

1.4. OBJETIVO GENERAL

Desarrollar un componente software para el registro de la trazabilidad del paso de mensajes a partir de código fuente orientado a objetos, mediante grafos de flujo de objetos y usando una estructura de invocación especifica.

1.5. OBJETIVOS ESPECIFICOS

- Elaborar el modelo del negocio correspondiente a la trazabilidad del paso de mensajes a partir de código fuente orientado a objetos (Modelo de dominio y diagrama de actividades).
- Diseñar las estructuras necesarias para la creación del grafo de flujo de objetos.
- Diseñar el componente a partir del modelo del negocio obtenido.
- Implementar el componente en código java para registrar la trazabilidad del paso de mensajes a partir de código fuente orientado a objetos, garantizando la visualización del diagrama de flujo de objetos.
- Evaluar el componente para determinar posibles fallos del sistema y para saber si cumple con los requerimientos establecidos.

1.6. ALCANCE

El resultado del proyecto será un componente software que generará el grafo de flujo de objetos para código fuente java usando como entrada una estructura de invocación específica. El grafo de flujo de objetos obtenido se guardará en un repositorio con formato XML para que pueda ser visualizado y utilizado por cualquier herramienta que interprete la estructura del diagrama que representa.

El componente que se obtendrá como resultado hace parte integral del Entorno de Ingeniería Inversa desarrollado en la Tesis doctoral del docente Martín Monroy Ríos, y su funcionalidad se limita a la establecida en el objetivo general previamente expuesto. Las distintas representaciones que se pueden obtener a partir del diagrama de flujo de objetos, se encuentran fuera del alcance del presente proyecto.

ESTADO DEL ARTE Y MARCO TEORICO

Actualmente en el campo de la ingeniería inversa enfocada en el software, se han realizado bastantes avances significativos, lo que ha dado como resultado un conjunto de herramientas bastante robustas que facilitan la vida de los programadores. Esto ha permitido un mejoramiento significativo de las prácticas y modelos empresariales en el ámbito del software en todo el mundo.

2.1. NIVEL INTERNACIONAL

2.1.1. EUROPA

El Reino Unido fue la tierra para el nacimiento de una herramienta bastante interesante en aspectos de ingeniería inversa llamada Visual Paradigm [15], la cual puede crear una gran cantidad de diagramas a partir de código fuente en distintos lenguajes de programación.

En Alemania las propuestas se basaron meramente en la creación de diagramas de clase a partir de código fuente como es el caso de [9], quienes lo hacen a partir de C++, mientras que en Portugal [5], definen los elementos básicos del diagrama de comunicación, como actores, objetos y mensajes. Otros como [11] obtienen los diagramas de secuencias de UML 2.0 a partir de JAVA.

2.1.2. ASIA

En Taiwán se encuentran investigaciones relacionadas con la creación de diagramas de clase a partir de código C++, como es el caso de [17].

2.1.3. AMERICA DEL NORTE

Las investigaciones realizadas en Estados Unidos son verdaderamente variadas y notables. Casos específicos como el de las herramientas o también llamadas entornos de programación, Netbeans [8], y Visual Studio [6], exceptuando visual estudio, que solo permite la generación de ciertas estructuras de datos desde código, las demás de forma muy exitosa, generan diagramas de clases desde distintos lenguajes de programación. Aunque todas ellas presentan inconvenientes a la hora de generar el diagrama de secuencias, puesto que no obtienen la gran mayoría de los elementos pertenecientes al estándar de UML 2.0. Generalmente presentan problemas en la identificación del fragmento combinado LOOP, a partir de sentencias DO Y DO WHILE. Otra investigación importante es la de [12], que obtienen diagramas de flujo y expresiones en lógica de predicados a partir de código C.

Se encuentran otros investigadores que solo se enfocan en el diagrama de clases como por ejemplo [13] desde C++.

También se puede encontrar un grupo de propuestas encargadas meramente de la parte de diagramas de interacción, como el de comunicación (colaboración en el estándar de UML 1.4) y el de secuencias. Tal es el caso de [14], que realizan una revisión de las técnicas disponibles para la ingeniería inversa de código orientado a objetos y dedican un capítulo especial a los diagramas de interacción, pero sin obtener los elementos nuevos definidos por el estándar de UML 2.0 (como los fragmentos combinados). Se puede mencionar de igual manera el caso de [10], que obtienen los diagramas de secuencias de UML 2.0 a partir de JAVA, incluyendo algunos fragmentos combinados, pero presentan problemas con el fragmento LOOP a partir de las sentencias DO y DO WHILE.

En Canadá por otro lado también existen aportes notables al campo de la ingeniería inversa como por ejemplo en el caso de [2], quienes a partir de las descripciones textuales de los casos de uso, aplican ingeniería inversa a la obtención del diagrama de casos de uso, que es uno de los diagramas de comportamiento de UML, el grupo de diagramas del cual descienden los de interacción. Otros aportes que se pueden mencionar ya específicamente para el caso de los diagramas de clases son los de [4] y [16], quienes los generan a partir de

código JAVA; [3] desde lenguaje C; [1], que obtienen el diagrama de secuencias de UML a partir de código C++.

2.1.4. AMERICA CENTRAL

En México se ven investigaciones relacionadas con la creación de diagramas de interacción y el de secuencias, como es el caso de [7], que obtienen el diagrama de secuencias de UML a partir de código C++.

2.2. NIVEL NACIONAL

En Colombia se puede encontrar el caso de [18], quienes proponen un método para la realización de ingeniería inversa de código java hacia diagramas de secuencias UML 2.0.

También se menciona el caso de la tesis doctoral del profesor Martin Monroy Ríos, quien plantea un marco de referencia para la recuperación de vistas arquitectónicas y de comportamiento, con el propósito de mejorar los procesos de recuperación de conocimiento a partir de sistemas de software implementados. [19]

Dentro de las investigaciones mencionadas anteriormente en cada uno de los continentes existe un patrón relacionado con la creación de diagramas de diseño, haciendo énfasis específicamente en la creación de diagramas de secuencia. El proyecto en curso, no se centra en la creación de este diagrama, pero si en la creación de un OFG, el cual permite ver el flujo de objetos a partir de un código fuente.

2.3. MARCO TEORICO

2.3.1. ANTEDECENDES HISTORICOS

Las primeras investigaciones relacionadas con los traductores e intérpretes, fueron el estudio del problema de lectura de fórmulas algebraicas complicadas y su interpretación. Así la notación polaca inversa (Reverse Polish Notation), o notación libre de paréntesis fue introducida por J. Lukasiewicz a finales de los años 20. Las formulas con paréntesis, en su caso más general fueron estudiadas por Kleene en 1934, y por Church en 1941. Continuando con los estudios previamente realizados, K. zuse especifico, en 1945, un algoritmo que determinaba cuando una expresión con paréntesis estaba bien formada. En 1951, H. Rutishauser describió el nivel de paréntesis de una formula algebraica, como un perfil entre distintos niveles, que parte del nivel cero, y que va ascendiendo nivel a nivel, para descender por último, otra vez a nivel cero. Todos los estudios mencionados, estaban situados en el campo de la lógica. [22]

En un principio se llamó compilador a un programa que "reunía" subrutinas (compilaba). Así, cuando en 1954 empezó a utilizarse el término "compilador algebraico", su significado es el actual, empleándose en lugar de traductor. [22]

Los lenguajes de alto nivel empezaron a desarrollarse en los años 50, y algunos de ellos todavía juegan un papel importante hoy en día, aunque han sido remodelados en revisiones periódicas. Este es el caso del FORTRAN, LISP Y COBOL que fueron diseñador originalmente en esa época. El primer "compilador algebraico" se desarrolló para el lenguaje FORTRAN (1954). [22]

El proceso de compilación se facilitó restringiendo la notación primeramente se suprimió la precedencia de los operadores, y por tanto se necesitaban muchos paréntesis. Otra forma era requiriendo el completo uso de paréntesis, pues se conocía una manera mecánica y siempre de insertar los paréntesis, obteniéndose expresiones con el "máximo número de paréntesis", desarrollador por los estudios ligados al dominio de la lógica. En 1952 C.

Böhm centro su interés en este último método, y lo aplico a las formulas secuencialmente. Böhm estuvo al frente del equipo que realizo el primer compilador de FORTRAN para IBM, en 1954. [22]

En 1957 Kantorovic mostro como se puede representar una formula con estructura de árbol lo cual supuso el primer paso para la conexión con el análisis sintáctico de los lingüistas. La jerarquía de los lenguajes había sido introducida por Chomsky (1956), como parte de un estudio sobre lenguajes naturales. Su utilización para especificar la sintaxis de los lenguajes de programación la realizo Backus, con el borrador del que sería más tarde ALGOL 60. [22]

Las técnicas fundamentales del análisis sintáctico se desarrollaron y formalizaron en los años sesenta. En un principio las investigaciones se centraron en el análisis sintáctico ascendente, aunque posteriormente se completaron con el estudio del análisis descendente. [22]

El análisis sintáctico ascendente se basa en reconocer las sentencias desde las hojas hasta la raíz del árbol. Floyd, en 1963, fue el primero en introducir las gramáticas de precedencia, que dieron lugar a los compiladores basados en precedencia. En 1966 McKeeman obtuvo la precedencia débil como una generalización de la precedencia simple. Otro desarrollo en el análisis sintáctico ascendente se debe a Knuth, que generalizo los autómatas de pila de Bauer y Samuelson, para reconocer los lenguajes generados por gramáticas LR (k). [22]

Los métodos descendentes de análisis sintáctico, se basan en construir el árbol sintáctico desde la raíz a las hojas, es decir en forma descendente. Los compiladores dirigidos por sintaxis, en la forma de análisis recursivo descendente fueron propuestos por primera vez de modo explícito por Lucas en 1961, para describir un compilador simplificado de ALGOL 60 mediante un conjunto de subrutinas recursivas, que correspondían a la sintaxis BNF. El problema fundamental del desarrollo de este método fue el retroceso, lo que hizo que su uso práctico fuera restringido. La elegancia y comodidad de la escritura de compiladores dirigidos por sintaxis fue pagada en tiempo de compilación por el usuario. Así mientras que con esta técnica tenían que ejecutarse 1000 instrucciones para introducir

una instrucción máquina, con las técnicas de los métodos ascendentes se tenían que ejecutar 50. La situación cambio cuando comenzó a realizarse análisis sintáctico descendente dirigido por sintaxis sin retroceso, por medio del uso de las gramáticas LL (l), obtenidas independientemente por Foster (1965) y Knuth (1967). Generalizadas posteriormente por Lewis, Rosenkrantz y Stearns en 1969, dando lugar a las gramáticas LL (l), que pueden analizar sintácticamente sin retroceso, en forma descendente, examinando en cada paso, todos los símbolos procesados anteriormente y los k símbolos más a la derecha. Evidentemente, tanto el análisis de las gramáticas LL (l) como el de las LR (k), es eficiente, y la discrepancia entre métodos de análisis sintáctico ascendente no vas más allá de su importancia histórica. [22]

A finales de los años sesenta surgen las primeras definiciones semánticas elaboradas de lenguajes de programación, como la dada para PL/I por el laboratorio de IBM de Viena, en 1969. [22]

La década de los setenta se caracteriza por la aparición de nuevos lenguajes de programación, resultando en parte de la formalización de las técnicas de construcción de compiladores de la década anterior, y por otra por el gran desarrollo del hardware. Entre los nuevos lenguajes se destaca el PASCAL, diseñado para la enseñanza de técnicas de programación estructurada, por N. Wirth en 1971 en el campo de la programación de sistemas destacan el lenguaje C, diseñado por Kernighan y Ritchie en 1973, y lenguaje Modula-2, diseñado por Wirth a finales de los setenta. Hay también que señalar la aparición de los pequeños sistemas (micro-ordenadores) y el abaratamiento de estos. [22]

La década de los ochenta comienza marcada por la aparición de lenguaje ADA, que pretende ser multipropósito, y con unas prestaciones muy elevadas. El diseño de este lenguaje se hizo según las directrices marcadas por el ministerio de defensa de los Estados Unidos. Es un lenguaje basado en pascal, que incluye características tales como abstracción de datos, multitarea, manejo de excepciones, encapsulamiento y módulos genéricos. Para la construcción de sus compiladores se han utilizado las técnicas más avanzadas, y también se ha visto la necesidad de considerar nuevas técnicas de diseño de compiladores. Se avanza

en el concepto de tipo abstracto de datos. Se desarrollan otras arquitecturas de computadoras, que faciliten el procesamiento paralelo. La tecnología VLSI, abre expectativas para el desarrollo de nuevas arquitecturas, y en el campo de compiladores de silicio. [22]

A través del tiempo, se han desarrollado técnicas sobre la estructura del compilador y se han modularizado, con un objetivo claro para cada módulo. Esto ha originado las técnicas sistemáticas para mayor ayuda al programador, que simplifica y mejora el diseño. Un refinamiento de estas técnicas es que el programador no escriba el módulo, sino que el ordenador los programe, es decir, de disponer de herramientas de ayuda en la construcción del compilador. [22]

2.3.2. ANTEDECENTES INVESTIGATIVOS

Título: Improving Behavioral Design Pattern Detection through Model Checking
Autor: Lucia, A.; Deufemia, V.; Gravino, C.; Risi, M.
En: Software Maintenance and Reengineering (CSMR), 2010 14th European Conference
No. Páginas: 9
Año de publicación: 2010

Descripción: El artículo propone un enfoque para la recuperación de instancias de patrones de diseño en sistemas de software. Se analizan las instancias desde un punto de vista estructural y luego se procede a analizarlas desde un punto de vista dinámico.

Diferenciador: Se habla específicamente de un modelo a seguir, pero no se realiza ningún tipo de implementación.

Título: Reverse engineering of object oriented code
Autor: Tonella, P.
En: Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference

No. Páginas: 208
Año de publicación: 2005

Descripción: En este libro se encuentran diversos modelos para el desarrollo de técnicas de ingeniería inversa, las cuales pueden ser bastante útiles en la creación de un entorno de ingeniería inversa.

Diferenciador: No está orientado a la creación de una implementación específica, Aunque puede ser tomado como referente para el contexto del problema presente.

Título: Design recovery for software testing of object-oriented programs
Autor: Kung, C.H.; Gao, J.; Hsia, P.; Lin, J.; Yoyoshima, Y.
En: Reverse Engineering, 1993. Proceedings of Working Conference
No. Páginas: 9
Año de publicación: 1993

Descripción: Se presenta un enfoque de ingeniería inversa para realizar testeo de software a programas creados con base a lenguajes orientados a objetos usando un modelo gráfico, llamado Object-Oriented Test Model (OOTM). El modelo consiste en 3 tipos de diagramas: 1) Un diagrama de relación de objetos; 2) Un conjunto de diagramas que describen las estructuras de control de las funciones y las relaciones de control entre funciones; 3) Un conjunto de diagramas de estado de objetos que describen comportamientos dependientes de los objetos.
Diferenciador: Se limita exclusivamente a la descripción de un modelo, pero no realiza ningún tipo de implementación.

Título: Summarizing the Content of Large Traces to Facilitate the Understanding of the Behaviour of a Software System
Autor: Hamou-Lhadj, A.; Lethbridge, T.

En: Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference
No. Páginas: 9
Año de publicación: 2006

Descripción: Se presenta un enfoque semiautomático para resumir largas trazas de ejecución. Se toman trazas de ejecución como entrada y se retorna un resumen del contenido. El resumen del contenido puede luego ser convertido en un diagrama de secuencia UML, que puede ser usado por ingenieros de software para entender aspectos de comportamiento fundamentales respecto a un sistema.

Diferenciador: Al igual que en casos anteriores se habla de un modelo, pero no se realiza una implementación del mismo.

Título: Un método de ingeniería inversa de código java hacia diagramas de secuencias de uml 2.0
Autor: Carlos Mario Zapata; Óscar Andrés Ochoa; Camilo Vélez
En: Rev.EIA.Esc.Ing.Antioq no.9 Envigado Jan./June 2008
No. Páginas: 12
Año de publicación: 2008

Descripción: En este artículo se propone un método que automatiza la conversión de código JAVA en diagramas de secuencias de UML 2.0, por medio de la aplicación de reglas de transformación que convierten los elementos del código en elementos del diagrama. Se presenta también un ejemplo de aplicación del método con un prototipo que lo emplea, el UNC-Inversor.

Diferenciador: Se enfoca en diagramas de secuencia y además su entrada no es una invocación específica.

2.3.3. MARCO CONCEPTUAL

Cuando se habla de trazabilidad específicamente en el caso de código, se debe tener en cuenta que se necesitan hallar estructuras para representarla. Estas estructuras para este caso son invocaciones, las cuales se realizan respecto a métodos. En general si se visualiza el proyecto desde una perspectiva amplia de lo que se debe hacer, resalta el hecho de que es necesario un análisis léxico de cada una de las palabras que conforman la invocación. Pero sucede algo interesante y es que ese analizador léxico no puede ser general, porque en tal caso el conjunto de palabras a evaluar sería demasiado grande y probablemente el analizador haría su función de una forma poco optima, además de que se desaprovecharía la capacidad computacional orientando todo a un solo proceso. [20]

Dicho esto, se hace necesario realizar varios analizadores léxicos con conjuntos de palabras específicas a validar. Luego de que se tienen los analizadores léxicos, se hace evidente que simplemente evalúan si una palabra pertenece o no al conjunto de términos aceptados por el lenguaje de programación en el cual se encuentra el código fuente. Es decir, no se tiene una herramienta capaz de validar la estructura de estas palabras aceptadas dentro de una misma sentencia, Esto hace referencia a que no necesariamente por el hecho de que sean palabras aceptadas dentro del lenguaje, se encuentren en un orden aceptado por el mismo lenguaje. Es allí donde se implementa el analizador sintáctico, el cual simplemente validaría que la estructura se encontrara de forma adecuada según los parámetros del lenguaje. Si bien se observa, estas dos herramientas hacen parte de los actualmente conocidos compiladores. [20] En forma más específica son la base de los mismos según el siguiente orden:

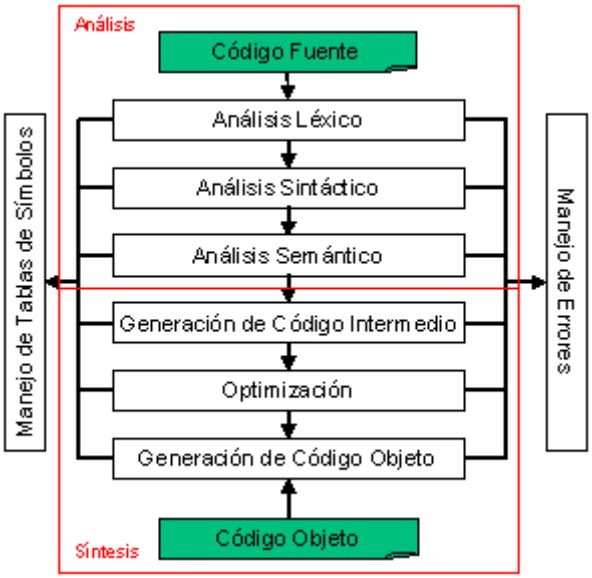


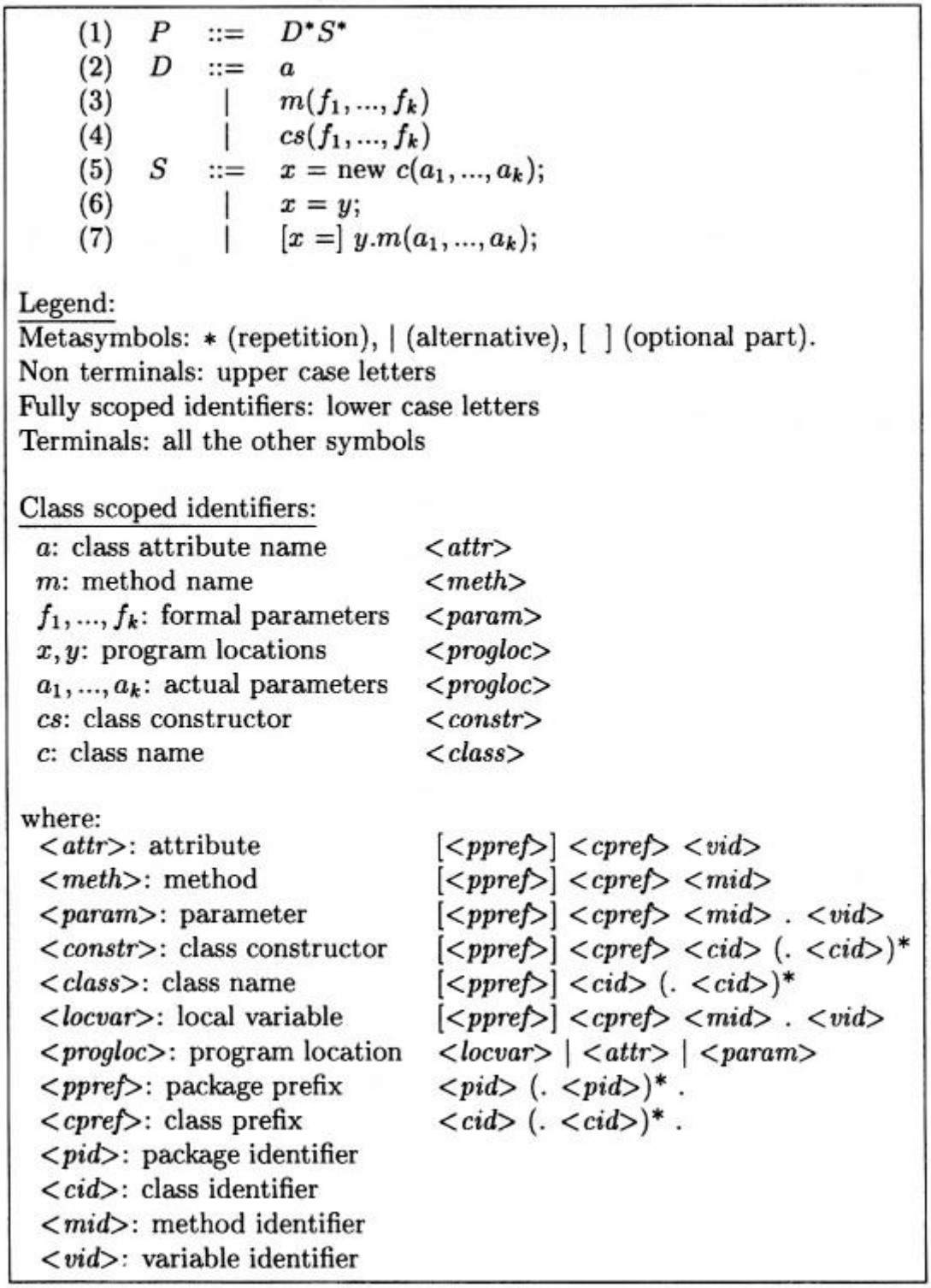
Fig. 1.1. Estructura de un compilador. [20]

Sabiendo que el proyecto tomara una parte tan importante de la maquina conocida como compilador, se hace indispensable un cierto conocimiento de la teoría de compiladores, al igual que la teoría de autómatas y lenguajes formales.

Ahora bien, se conoce la necesidad de un conjunto de estructuras capaces de realizar análisis sintácticos y léxicos ¿pero con base en que se realizaría lo anterior?, hablando de forma más concreta, ¿Cómo se llevaría a cabo el manejo del flujo de objetos a través del código fuente (paquetes, clases, métodos y sentencias)? Pues bien, para eso existe el Object Flow Graph (OFG) que es la representación básica del programa para el análisis estático del código fuente. El OFG permite el rastreo del flujo de información acerca de los objetos, desde la creación del objeto mediante la asignación de declaraciones, a través de la asignación de objetos a variables, hasta el almacenamiento de objetos en campos de clase o su uso en llamadas a métodos. La información que se propaga a través del OFG varía dependiendo sobre los efectos del análisis en el que se emplea. Por ejemplo, el tipo al que los objetos se convierten por medio de expresiones de conversión puede ser la información que se propaga cuando un análisis está definido a estáticamente determinar un tipo de objeto más preciso que el objeto perteneciente a la declaración. [14]

Si bien ya se conoce la representación mediante la cual se pueden obtener los flujos de objetos, se desconoce la estructura interna de este paradigma, el cual esta expresado anteriormente de forma demasiado general como para ser usado de forma práctica en el contexto del problema presente. Las partes que conforman esta compleja estructura son las siguientes:

- **El lenguaje abstracto:** Es una sintaxis simplificada del lenguaje de programación respecto al que se desea obtener la representación. Dentro de esta categoría entran dos tipos:
    - Declaraciones: Las cuales pueden ser de 3 tipos. Las **declaraciones de atributos** (2) consisten meramente en el nombre del atributo, es decir, una lista de puntos separados de paquetes, seguidos de una lista separada por puntos de las clases, seguidas por el identificador del atributo. Una **declaración de método** (3) consiste en el nombre del método seguido de la lista de parámetros formales. Las **declaraciones de constructores** (4) poseen una sintaxis abstracta parecida a la de las declaraciones de métodos, resaltando meramente el hecho de que en vez de poseer el nombre del método, poseen el nombre de la clase a la cual pertenecen. [14]
    - Sentencias: También son de 3 tipos (**Sentencias de locación** (5)**, sentencias de asignación** (6) e **invocaciones de métodos** (7)). Del lado izquierdo ***x*** de todas las sentencias existe una locación de programa (una dirección, la cual es opcional para las invocaciones de métodos), y del lado derecho ***y*** de las sentencias de asignación, al igual que el blanco ***y*** de las invocaciones de métodos, también existe una locación del programa. [14]



(1) P ::= D^*S^*
(2) D ::= a
(3) | $m(f_1, ..., f_k)$
(4) | $cs(f_1, ..., f_k)$
(5) S ::= x = new $c(a_1, ..., a_k)$;
(6) | $x = y$;
(7) | $[x =]\ y.m(a_1, ..., a_k)$;

Legend:
Metasymbols: $*$ (repetition), | (alternative), [ ] (optional part).
Non terminals: upper case letters
Fully scoped identifiers: lower case letters
Terminals: all the other symbols

Class scoped identifiers:

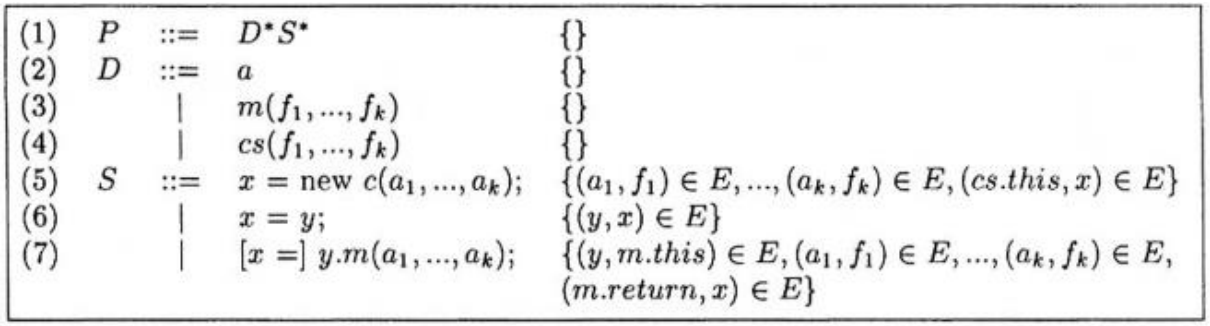
a: class attribute name	<*attr*>
m: method name	<*meth*>
$f_1, ..., f_k$: formal parameters	<*param*>
x, y: program locations	<*progloc*>
$a_1, ..., a_k$: actual parameters	<*progloc*>
cs: class constructor	<*constr*>
c: class name	<*class*>

where:

<*attr*>: attribute	[<*ppref*>] <*cpref*> <*vid*>
<*meth*>: method	[<*ppref*>] <*cpref*> <*mid*>
<*param*>: parameter	[<*ppref*>] <*cpref*> <*mid*> . <*vid*>
<*constr*>: class constructor	[<*ppref*>] <*cpref*> <*cid*> (. <*cid*>)*
<*class*>: class name	[<*ppref*>] <*cid*> (. <*cid*>)*
<*locvar*>: local variable	[<*ppref*>] <*cpref*> <*mid*> . <*vid*>
<*progloc*>: program location	<*locvar*> | <*attr*> | <*param*>
<*ppref*>: package prefix	<*pid*> (. <*pid*>)* .
<*cpref*>: class prefix	<*cid*> (. <*cid*>)* .
<*pid*>: package identifier	
<*cid*>: class identifier	
<*mid*>: method identifier	
<*vid*>: variable identifier	

Fig. 1.2. Sintaxis abstracta del lenguaje java simplificado. [14]

- **El OFG:** Ya se habló de la representación general que abarca el OFG, pero tocando el tema de forma más específica, el OFG es un par (*N, A*), que representa un conjunto de nodos N y un conjunto de aristas A. Dentro del OFG se adhiere un nodo por cada locación de programa (Variable local, atributo o parámetro formal). [14]



(1)	P	::=	D^*S^*	$\{\}$
(2)	D	::=	a	$\{\}$
(3)		|	$m(f_1,...,f_k)$	$\{\}$
(4)		|	$cs(f_1,...,f_k)$	$\{\}$
(5)	S	::=	x = new $c(a_1,...,a_k)$;	$\{(a_1,f_1)\in E,...,(a_k,f_k)\in E,(cs.this,x)\in E\}$
(6)		|	$x = y$;	$\{(y,x)\in E\}$
(7)		|	$[x =]\ y.m(a_1,...,a_k)$;	$\{(y,m.this)\in E,(a_1,f_1)\in E,...,(a_k,f_k)\in E,$ $(m.return,x)\in E\}$

Fig. 1.3. Aristas del OFG inducidas por cada sentencia abstracta de java. [14]

Las aristas son añadidas al OFG respecto a las reglas especificadas en la Fig. 1.3. Ellas representan el flujo de datos que ocurre en el programa analizado. El conjunto de aristas A pertenecientes al OFG contiene todos y solamente los pares que resultan para al menos una regla en la Fig. 1.3. [14]

- **Contenedores:** El OFG tiene todo el flujo de datos que ocurre en el programa. Sin embargo, algunos de estos flujos están relacionados con instrucciones específicas de java, como la asignación o la llamada a un método, mientras que otros lo están con el uso de una librería de clases. Sucede que cada vez que una librería introduce un flujo de datos desde una variable ***x*** a una variable ***y***, una arista (***x, y***) debe ser incluida dentro del OFG. [14]

  Una categoría de librerías de clases que introduce flujos de datos externos adicionales, está representada por **contenedores**. En java, un ejemplo de esto es

cualquier clase que implemente la interface *collection* (*Vector, LinkedList, Hashset,* y *TreeSet*) o la interface *Map* (*Hashtable, HashMap,* y *TreeMap*). [14]

- **Algoritmo de flujo de propagación:** Se sabe que el OFG es una fuente enorme de información que permite conocer muchas características respecto a los objetos pertenecientes a un programa, dicho esto, es posible obtener esa información para analizar el comportamiento del mismo a través del algoritmo de flujo de propagación. Se pueden obtener características tan importantes como el tipo al cual un objeto es casteado o la locación de un objeto en una parte especifica de un programa. [14]
- **Sensibilidad de objetos:** Respecto a la sintaxis abstracta presente en la figura 1.2, se puede decir que tanto atributos de clase, como nombres de métodos, como locaciones de programa, etc., se encuentran en el ámbito del nivel de clase. Esto significa que es posible distinguir entre dos locaciones cuando ellas pertenecen a distintas clases (ej. atributos de clase), pero no es posible entonces distinguirlas cuando pertenecen a la misma clase y están relacionadas a instancias distintas (objetos). Por tal motivo, se puede decir que si se realiza un OFG basándose en las reglas presentes en la figura 1.3, se estaría creando un OFG sin sensibilidad de objetos. Este aspecto es muy importante, ya que la capacidad de distinguir locaciones que pertenecen a distintas instancias, puede ayudar mucho al mejoramiento de los resultados del análisis. [14]

  Un OFG con sensibilidad de objetos, puede ser construido dando ámbitos de objetos a nombres de programa no estáticos, en vez de dar ámbitos de clase (atributos estáticos y locaciones de programa que pertenecen a métodos estáticos, mantienen un ámbito de clase). Los objetos pueden ser identificados de forma estática por sus puntos de locación con un OFG con sensibilidad de objetos. [14]

Luego de conocer la estructura interna del OFG, se hace necesario representarla de forma gráfica. Esto se hace por medio del diagrama de objetos, el cual muestra las relaciones entre las distintas instancias que pertenecen al programa (objetos) y sus relaciones. [14]

También se puede decir que el diagrama de clases es la vista básica para entender un programa creado en base a código orientado a objetos, pero este no es muy informativo respecto al comportamiento que el programa mostrara durante el tiempo de ejecución enfocándose en las relaciones estáticas entre clases. Mientras que por el contrario, el diagrama de objetos representa las instancias de las clases y la relación entre objetos. Esta representación del programa provee de información adicional respecto al diagrama de clases en la forma en que las clases están siendo utilizadas. [14]

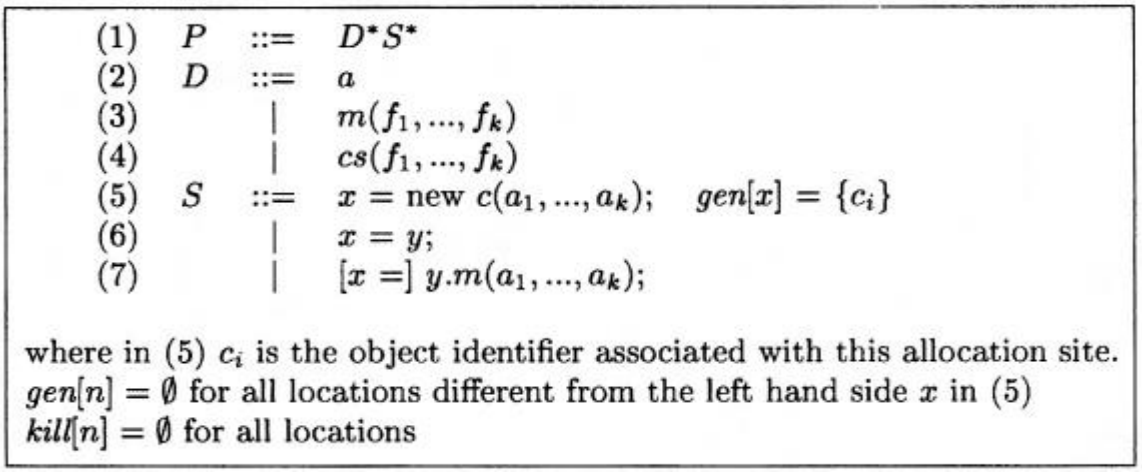
Dicho esto, y al igual que en el OFG, también se hace necesario resaltar ciertos aspectos que son esenciales para crear un diagrama de objetos de la forma adecuada:

- **El diagrama de objetos:** representa un conjunto de objetos creados dentro de un programa y las relaciones que se presentan entre cada uno de estos objetos. Este conjunto de elementos (objetos y relaciones) son instancias de los elementos (clases y asociaciones) en el diagrama de clases. La diferencia entre un diagrama de objetos y un diagrama de clases es que el primero instancia al último. Como consecuencia de ello, los objetos dentro del diagrama de objetos representan casos específicos de clases relacionadas. Por cada clase en el diagrama de clases, debe haber varios objetos instanciándola dentro del diagrama de objetos. Por cada relación entre clases en el diagrama de clases, deben existir pares de objetos instanciándolas y pares de objetos no relacionados por ellas. [14]
- **Recuperación del diagrama de objetos:** La computación estática del diagrama de objetos explota el flujo de propagación en el OFG para transmitir información respecto a objetos que están creados en el programa hasta los atributos que hacen referencia a ellos. Los objetos están identificados por su locación (la línea de código que contiene la declaración de locación), sin tener en cuenta el número real de veces que se ejecuta (que es en general, imposible de descifrar para un análisis estático).

  La figura 1.4 muestra el flujo de información que es propagado en el OFG para recuperar el diagrama de objetos. Cada uno de los sitios de locación (declaración de tipo (5)) está asociado con un único objeto identificador, construido con el nombre de clase *c*, subíndice por un entero incrementado i (que da el identificador de

objeto). Tal flujo de información se propaga en el OFG respecto al algoritmo dado en el capítulo 2.

La construcción del diagrama de objeto es un post-procesamiento sencillo de la computación descrita anteriormente. Cada identificador de objeto c_i genera un nodo correspondiente en el diagrama de objetos. Cada nodo en el OFG asociado a otro atributo de objeto, es tomado en consideración cuando son generadas asociaciones entre objetos. [14]



(1)	P	::=	D^*S^*	
(2)	D	::=	a	
(3)		|	$m(f_1, ..., f_k)$	
(4)		|	$cs(f_1, ..., f_k)$	
(5)	S	::=	x = new $c(a_1, ..., a_k)$;	$gen[x] = \{c_i\}$
(6)		|	$x = y$;	
(7)		|	$[x =] \; y.m(a_1, ..., a_k)$;	

where in (5) c_i is the object identifier associated with this allocation site.
$gen[n] = \emptyset$ for all locations different from the left hand side x in (5)
$kill[n] = \emptyset$ for all locations

Fig. 1.4. Flujo especializado de propagación para determinar el conjunto de objetos asignados en el programa que se son referenciados por cada ubicación del programa. [14]

- **Sensibilidad de objetos:** Un estimado más acertado respecto a las relaciones entre objetos presentes dentro del programa, puede ser obtenido de un análisis con sensibilidad de objetos. Las locaciones de programa son distinguidas por el objeto al que pertenecen en lugar de su clase. Teniendo en cuenta los sitios de asignación en el programa bajo análisis, un identificador de objeto c_i está asociado a cada uno de ellos. Una ubicación del programa de ámbito originalmente por clase *n* da lugar a un conjunto de nodos n' de OFG, en el ámbito de identificadores de objetos c_i cuando un OFG sensible a objetos se construye. [14]

- **Análisis dinámico:** La construcción dinámica del diagrama de objetos es obtenida trazando la ejecución de un programa en un conjunto de casos de testeo. Las facilidades de rastreo requeridas son básicamente la posibilidad de inspeccionar el objeto respectivo y sus atributos cada vez que un método es invocado en un objeto y sus sentencias son ejecutadas. El rastreo de información debería incluir un identificador de objeto para el objeto respectivo y también por cada objeto referenciado por cualquiera de los atributos de ese objeto. [14]

  Es posible obtener esa información dinámica explotando herramientas de rastreo disponibles o instrumentando el programa dado. En caso de instrumentar el programa, son necesarias las siguientes adiciones:

  - Las clases son aumentadas con un objeto identificador, el cual es computado y rastreado durante la ejecución de los constructores de clase.
  - Durante un cambio de atributo, los identificadores de los objetos referenciados por dicho atributo son añadidos al trazo de ejecución.
  - Las marcas de tiempo son producidas y trazadas cuando cualquiera de los dos eventos arriba mencionados ocurre. [14]

2.3.4. MARCO TECNOLOGICO

Para este caso en específico, la implementación del modelo se realizó en Java, el cual es un lenguaje de programación libre, que presenta varias facilidades para la creación de analizadores sintácticos, ya que tiene múltiples librerías reusables que poseen analizadores léxicos. Además, el tipo de lenguaje a analizar será también en código java.

METODOLOGIA

El objetivo general propuesto para este Proyecto fue el de "Desarrollar un componente software para el registro de la trazabilidad del paso de mensajes a partir de código fuente orientado a objetos, mediante diagramas de flujo de objetos y usando una estructura de invocación específica". Para cumplir ese objetivo, se realizó una investigación aplicada, gracias a la cual se tomó como base metodológica el modelo de programación extrema adaptado para el desarrollo individual. Actualmente todas las metodologías ágiles se basan en el desarrollo en equipo, por lo cual se han olvidado completamente del desarrollo de software orientado para una sola persona, dado esto y por las condiciones del actual proyecto, se hizo necesario orientar el modelo XP al desarrollo individual. Este desarrollo fue realizado en la ciudad de Cartagena de indias durante un lapso de tiempo aproximado de año y medio. [21]

Este nuevo modelo tuvo como base un conjunto de fases que son las siguientes: análisis, diseño, implementación y pruebas. Siempre teniendo en cuenta el mejoramiento del prototipo inicial.

3.1. FASE DE ANÁLISIS

Objetivo específico: Elaborar el modelo del negocio correspondiente a la trazabilidad del paso de mensajes a partir de código fuente orientado a objetos.

En esta primera fase, se abordó el análisis del problema de manera general, utilizando como técnica de recolección de información una revisión documental sobre la trazabilidad del paso de mensajes. Dentro de esta revisión fue necesario un conjunto de actividades que son: Definir cadenas de búsqueda, definir motores de búsqueda, aplicar las búsquedas, filtrar documentos y analizar documentos. Luego de haber recolectado, organizado y analizado la información, se tuvo en cuenta la técnica de análisis basada en la síntesis de todos los

documentos pertinentes al ámbito de estudio. Todo esto para crear una base sólida de conocimientos sobre la cual se sustentó la solución. Además, se elaboró un modelo de negocio el cual está conformado por un modelo de dominio y un diagrama de casos de uso general. Luego de realizar estos modelos, se llevó a cabo una corrección secuencial y creación de nuevas versiones de cada uno de ellos en el lapso de tiempo estimado para tal motivo.

3.2. FASE DE DISEÑO

Objetivo específico: Diseñar el componente a partir del modelo del negocio obtenido.

Luego de obtener un modelo de negocio pertinente al ámbito del problema, se realizó un diseño del sistema orientado a aspectos más específicos de este. Para tal motivo se crearon los siguientes modelos: diagrama de clases, diagrama de componentes, diagrama de flujo, diagrama de actividades, diagrama de despliegue, analizadores léxicos y analizador sintáctico. Donde en cada uno de ellos se llevó a cabo una corrección secuencial y creación de nuevas versiones.

3.3. FASE DE IMPLEMENTACIÓN

Objetivo específico: Implementar el componente en código java para registrar la trazabilidad del paso de mensajes a partir de código fuente orientado a objetos, garantizando la visualización del diagrama de flujo de objetos.

Para esta fase, se crearon las implementaciones en código de cada uno de los diagramas realizados en la fase de diseño, y así mismo, también se llevaron a cabo la corrección secuencial y la creación de nuevas versiones propuestas a través de todo el modelo anteriormente.

3.4. FASE DE PRUEBAS

Objetivo específico: Evaluar el componente para determinar posibles fallos del sistema y para saber si cumple con los requerimientos establecidos.

Dentro de esta última fase, se realizaron una serie de pruebas al componente software en las cuales se determinó que cumple con los requerimientos del sistema establecidos. Al finalizar todo este procedimiento, se procedió a crear la documentación.

RESULTADOS

A partir de cada uno de los objetivos desarrollados en el proyecto, se obtuvieron un conjunto de resultados que serán presentados en este capítulo. Estos se enfocan en modelos arquitectónicos, que brindan un soporte respecto a la estructura que posee el proyecto y también se enfocan en el aplicativo implementado, con sus funcionalidades específicas.

4.1. MODELO DE NEGOCIO

Dentro de la investigación se usó toda la información recolectada en la documentación presentada en la bibliografía, principalmente del libro de Tonella y Potrich, con el fin de conocer las necesidades que presentaba el proyecto. A partir de estas necesidades, se creó un esquema del funcionamiento del software, el cual se encuentra representado en los diagramas a continuación.

4.1.1. CASOS DE USO DEL MUNDO REAL

A partir del análisis de la problemática, se estableció un conjunto de acciones a realizar por el sistema para lograr cumplir con su objetivo, además se definieron 2 actores principales, que realizan los casos de uso representados en la **figura 2.1 Diagrama de casos de uso del mundo real**, Inicialmente se tiene al "experto de software", quien se encarga de definir el código a analizar, y también la estructura sintáctica a partir de la cual se generará el OFG. Luego, se tiene al "Autómata", quien recibe el código fuente, y lo analiza a partir de un String, que no es más que la estructura sintáctica definida anteriormente. Durante el análisis sintáctico, se ejecuta la gramática definida para la generación de nodos relacionados, que al final terminan conformando el OFG.

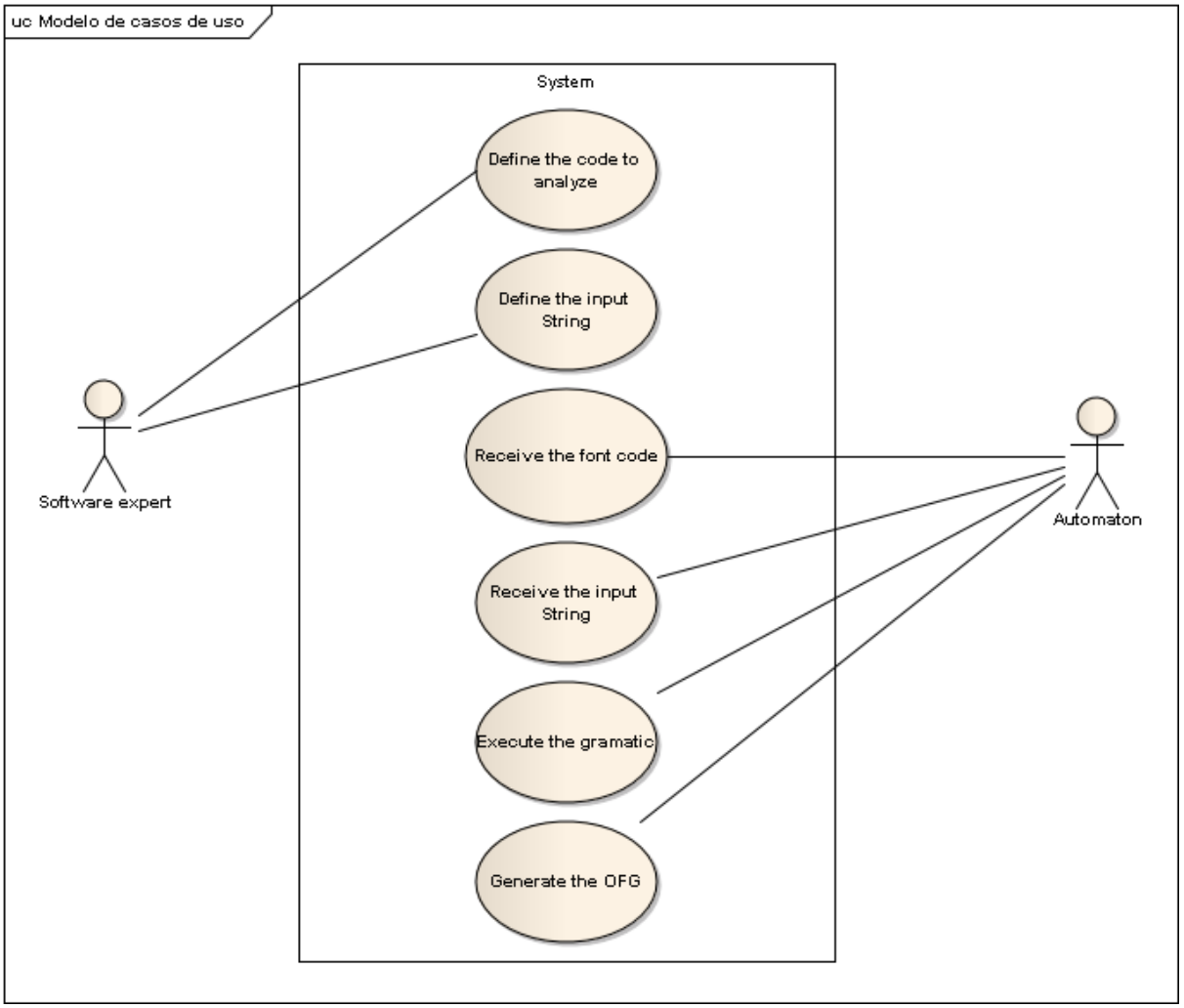


Fig. 2.1. Diagrama de casos de uso del mundo real.

4.1.2. MODELO DE DOMINIO

Para la creación del modelo de dominio representado en la **figura 2.2 Modelo de dominio**, se tuvieron en cuenta el conjunto de estructuras que hacen parte del sistema en concreto, además de las relaciones entre cada una de ellas.

Por medio del análisis de los actores, se determina que los involucrados finalmente son: “el experto de software” (La persona que usa el aplicativo), “El código fuente” (De donde se obtienen las relaciones entre objetos), “La cadena de entrada” (Estructura inicial respecto a

la cual se genera el OFG), “El autómata” (Genera el OFG), “El OFG” (Grafo que muestra las relaciones entre los objetos) y la “Gramática del OFG” (Reglas sintácticas usadas por el autómata para el análisis del código).

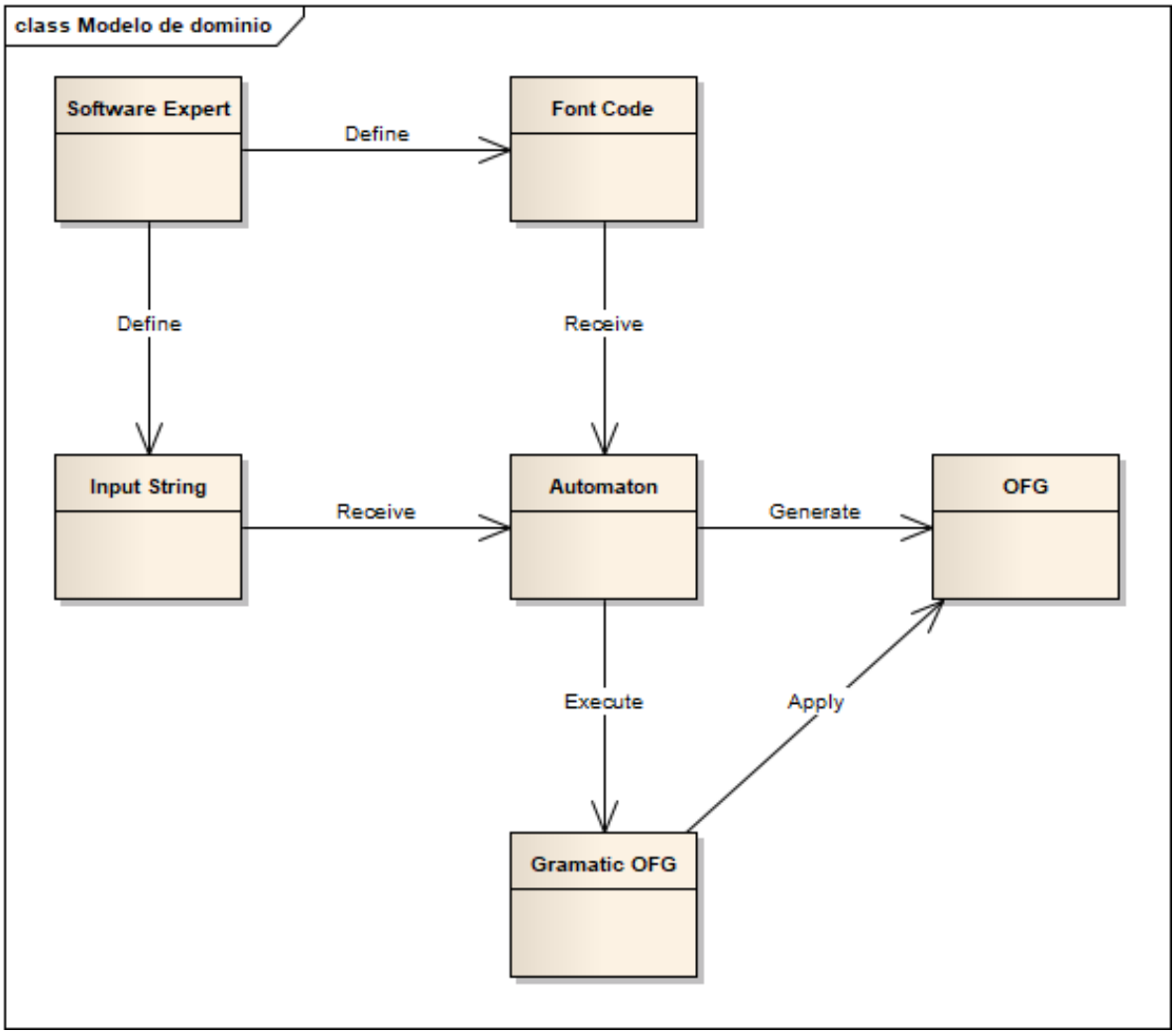


Fig. 2.2. Modelo de dominio.

4.1.3. DIAGRAMA DE ACTIVIDAD

Para la consecución de este diagrama representado en la **figura 2.3. Diagrama de actividades**, se tuvieron en cuenta los casos de uso representados en el comportamiento

interno de las actividades dadas por el usuario en el sistema, por ello en el gráfico se muestra el flujo de inicio a fin de las entradas generadas por el "Experto de software" y el uso de estas en procesos orientados a la generación del OFG.

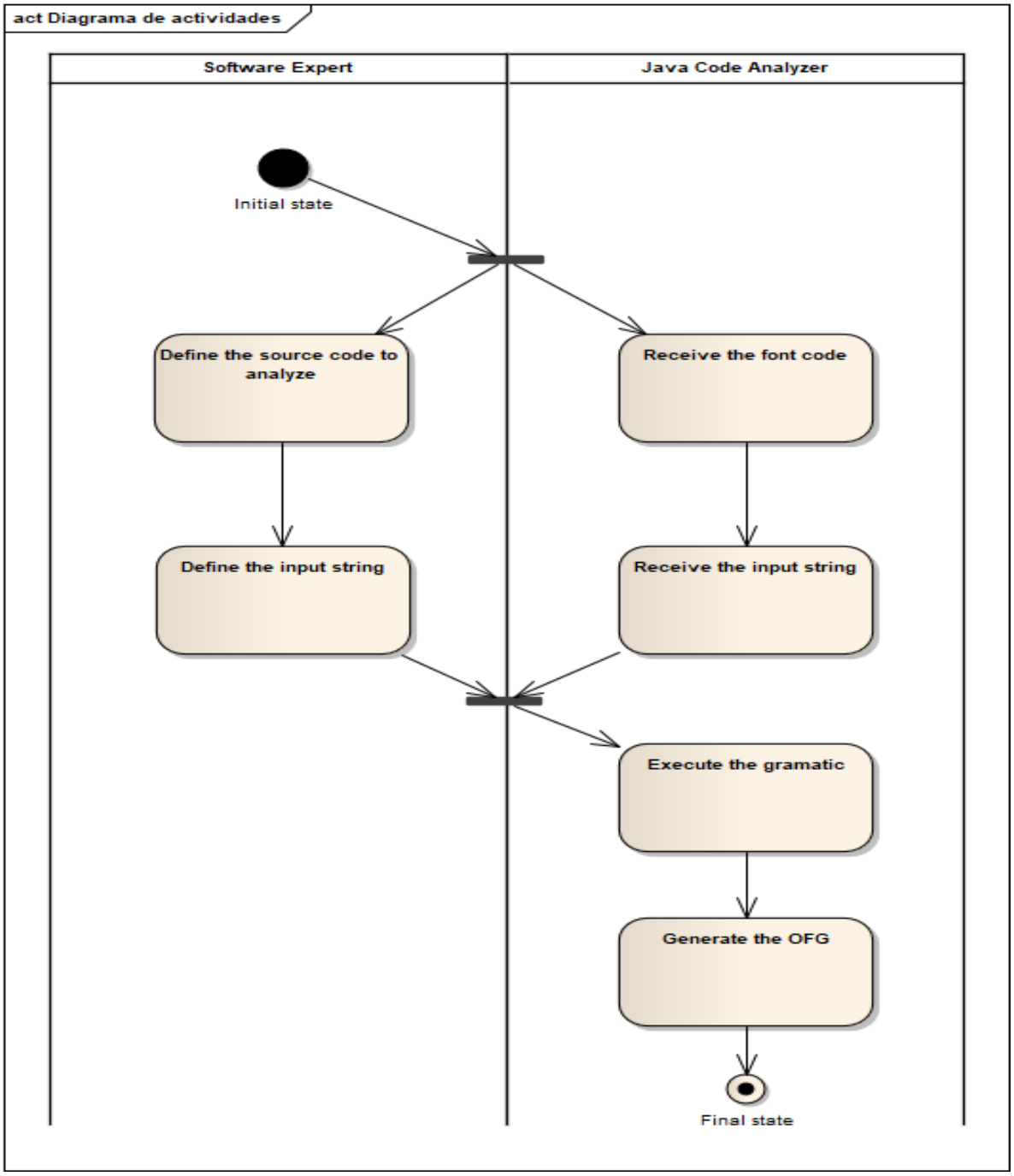


Fig. 2.3. Diagrama de actividades.

El proceso comienza por un estado inicial, del cual se derivan dos actividades, la primera ejecutada por el experto en software la cual consiste en elegir el código fuente que se va a utilizar para el análisis. El analizador de código recibe la ruta del proyecto, y toma el código fuente que se encuentra dentro. Luego, el experto elige la línea de código que actuara como raíz del OFG. El analizador de código recibe este parámetro, determina si cumple con la estructura de un nodo raíz, y genera un OFG que se visualiza en la pantalla.

4.2. DISEÑO DE LAS ESTRUCTURAS PARA LA CREACIÓN DEL GRAFO DE FLUJO DE OBJETOS

A partir de la gramática propuesta por Tonella y Potrich (2005), se definieron diferentes estructuras que son la base para la creación del grafo de flujo de objetos. Primero se creó una clase que contiene las expresiones léxicas y sintácticas que representan parte del lenguaje Java (ConstantsManager), como lo son nombres de los paquetes, los identificadores de métodos y clases, las declaraciones de métodos, las declaraciones de clases, etc. Luego de tener las expresiones, se creó una nueva estructura cíclica dentro un archivo de formato drl (Analizer) interpretado por drools que es un motor de reglas de negocio (lo cual se explica con detalle más adelante), para analizar y catalogar cada una de las expresiones regulares y definir si eran o no, potenciales elementos pertenecientes al OFG o que ayudan a la creación de éste, el cual está representado en la **figura 2.11. Diagrama de paquetes**. También se crearon las clases para el análisis de la estructura de las carpetas y de las distintas clases pertenecientes a ellas, así como para la creación del OFG a partir de las expresiones obtenidas y sus relaciones (ProjectReader, OFGconverter). Para esto se necesitó el uso de funciones recursivas que realizan el análisis de cada uno de los nodos las veces que fueran necesarias para encontrar nuevos nodos generados a partir de los padres. Por último, fueron creadas las clases para la interpretación del OFG y la creación tanto del xml como de la vista de usuario (XmlConversor, OFGView). Todas las clases mencionadas se pueden ver en la figura **figura 2.4. Diagrama de clases paquete model** y en la **figura 2.6. Diagrama de clases paquete view**.

4.3. MODELO DE DISEÑO

A partir de la definición de un esquema del mundo real, se procedió a crear las vistas arquitectónicas que representan el sistema de forma interna, y que permiten tener una perspectiva más clara del funcionamiento del proyecto a través de cada uno de sus elementos.

4.3.1. DIAGRAMA DE CLASES

Este modelo se basa en 3 paquetes (Modelo, Vista y Controlador), en los cuales se pueden notar relaciones de uso. El modelo es el encargado de contener los objetos que se manejan dentro del análisis léxico-sintáctico, además de otras estructuras que permiten manejar una lógica del negocio para realizar acciones que van desde el guardado de archivos, hasta la recolección de expresiones usadas para la creación del OFG. El Controlador mientras tanto, permite la comunicación entre los paquetes Vista y Modelo. Por último, la Vista toma todas las acciones realizadas por el experto de software y las procesa a través del controlador, además recibe los datos del OFG y los transforma para que sean visibles.

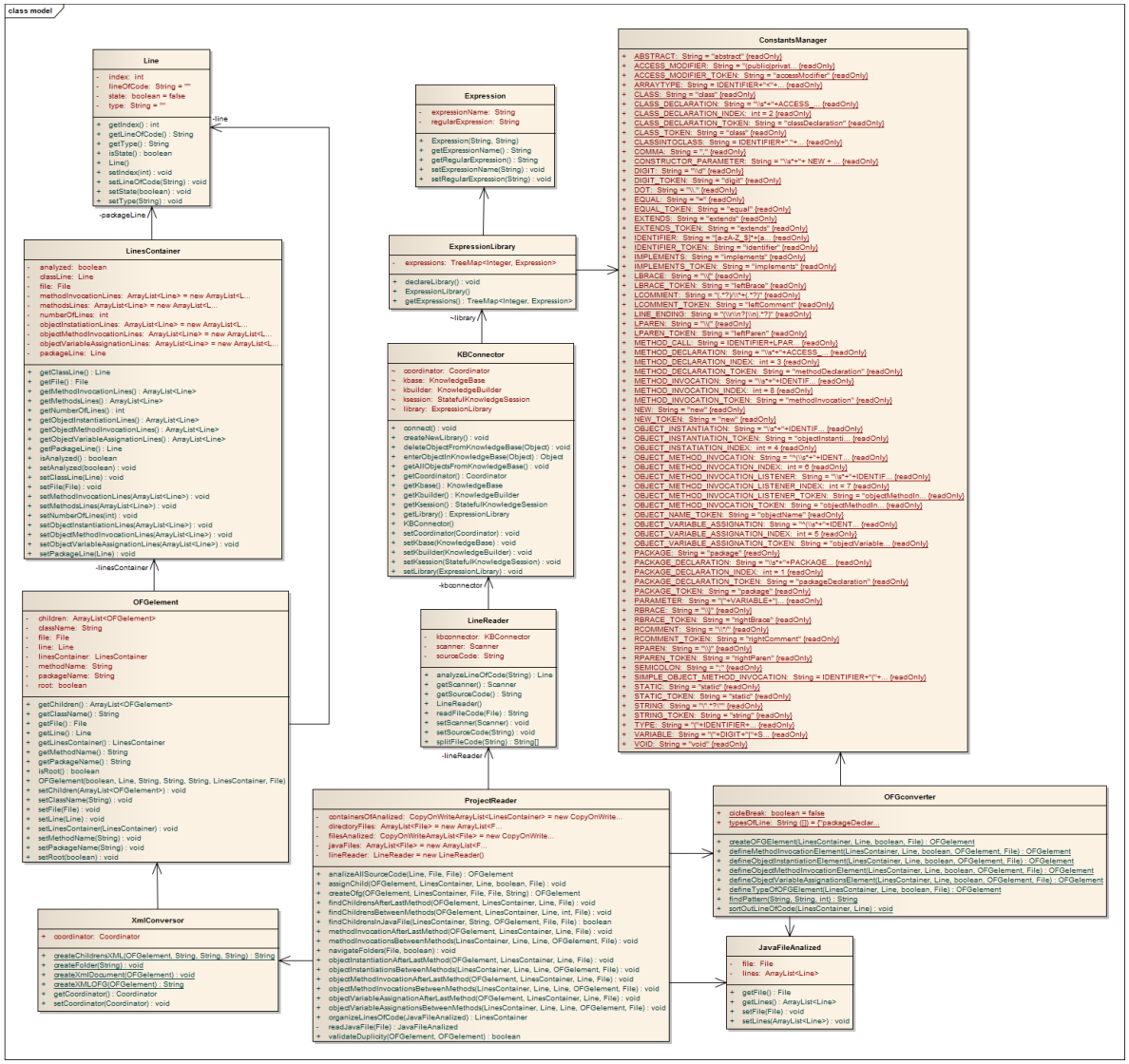


Fig. 2.4. Diagrama de clases paquete model.

Este paquete representado por la **figura 2.4. Diagrama de clases paquete model**, se encuentra conformado por varias clases que son las que permiten el análisis léxico-sintáctico de las estructuras provenientes de la vista (ConstantsManager). Así como el análisis de los archivos donde se encuentra guardado el código (ProjectReader y JavaFileAnalized) y la creación del OFG (OFGconverter y OFGelement, etc.)

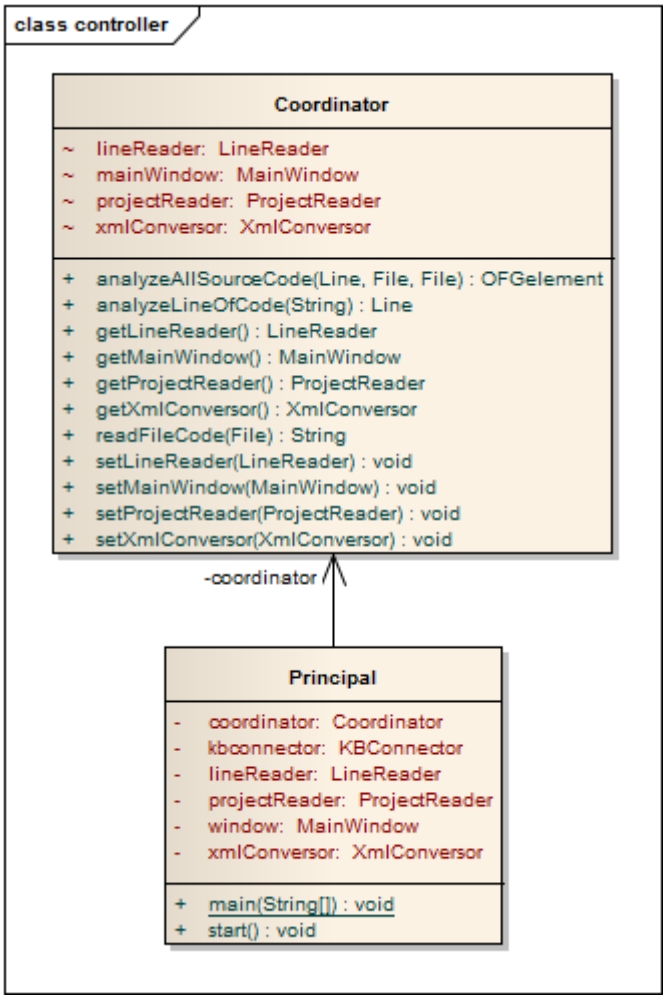


Fig. 2.5. Diagrama de clases paquete controller.

El paquete controller representado por la **figura 2.5. Diagrama de clases paquete controller**, se encarga de la comunicación entre los componentes vista y modelo, manejando los llamados a través de la clase coordinador. La clase principal como su nombre lo indica, es quien da inicio al procesamiento de datos dentro de todo el proyecto.

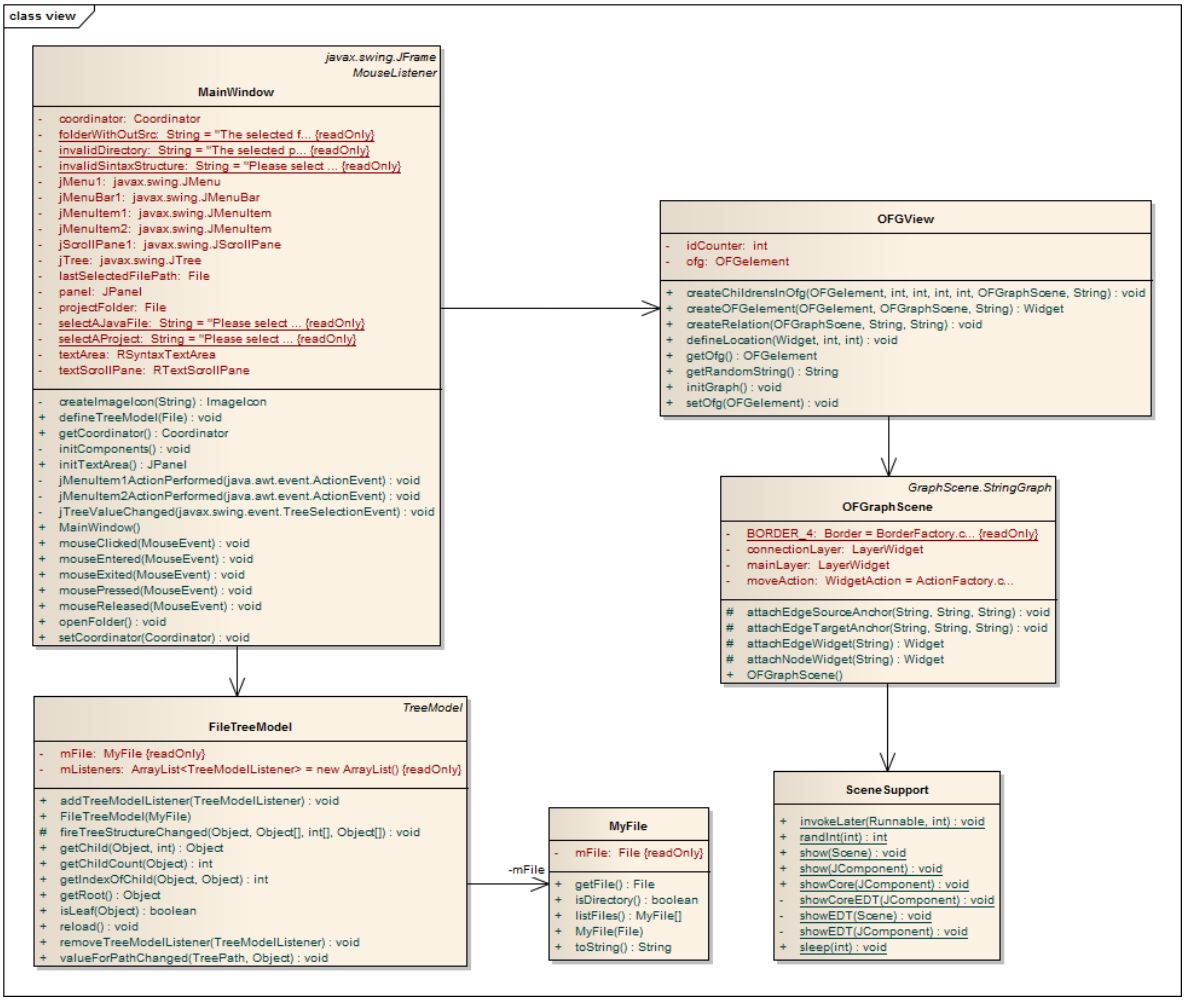


Fig. 2.6. Diagrama de clases paquete view.

El paquete view representado por la **figura 2.6. Diagrama de clases paquete view**, consta de 6 clases que se comunican entre sí para brindar la experiencia grafica del usuario. MainWindow es la clase principal de la vista, desde la cual se puede realizar la búsqueda de los proyectos que se van a analizar, las estructuras que van a tomarse como raíces del OFG y también realizar llamados a OFGView para mostrar el OFG obtenido, la cual se apoya en las clases OFGraphScene y SceneSupport para crear las estructuras graficas que le permiten mostrar el grafo.

4.3.2. DIAGRAMA DE COMPONENTES

Este diagrama representado por la **figura 2.7. Diagrama de componentes**, muestra 4 componentes (Autómata de Drools, Lógica, Controlador, Vista), cada uno de ellos posee servicios que son satisfechos, ofrecidos o ambos. Para el caso del Autómata de Drools por ejemplo, ofrece un servicio llamado "identificar estructuras gramáticas", este servicio implica el análisis de todas y cada una de las expresiones brindadas por la lógica. La lógica por su lado, ofrece un servicio llamado "Generar el OFG", este servicio toma todas las expresiones identificadas por el "Autómata de Drools" y las convierte ya sea en una imagen o un archivo XML, los cuales son una forma de representar el OFG. El Controlador con su servicio "Mostrar el código y el OFG", recibe la información obtenida de la lógica, y la transmite a la vista, que para este caso puede ser código, o simplemente el OFG. La vista por ultimo muestra el código respecto al cual se va a realizar el análisis, y después de realizado visualiza el OFG.

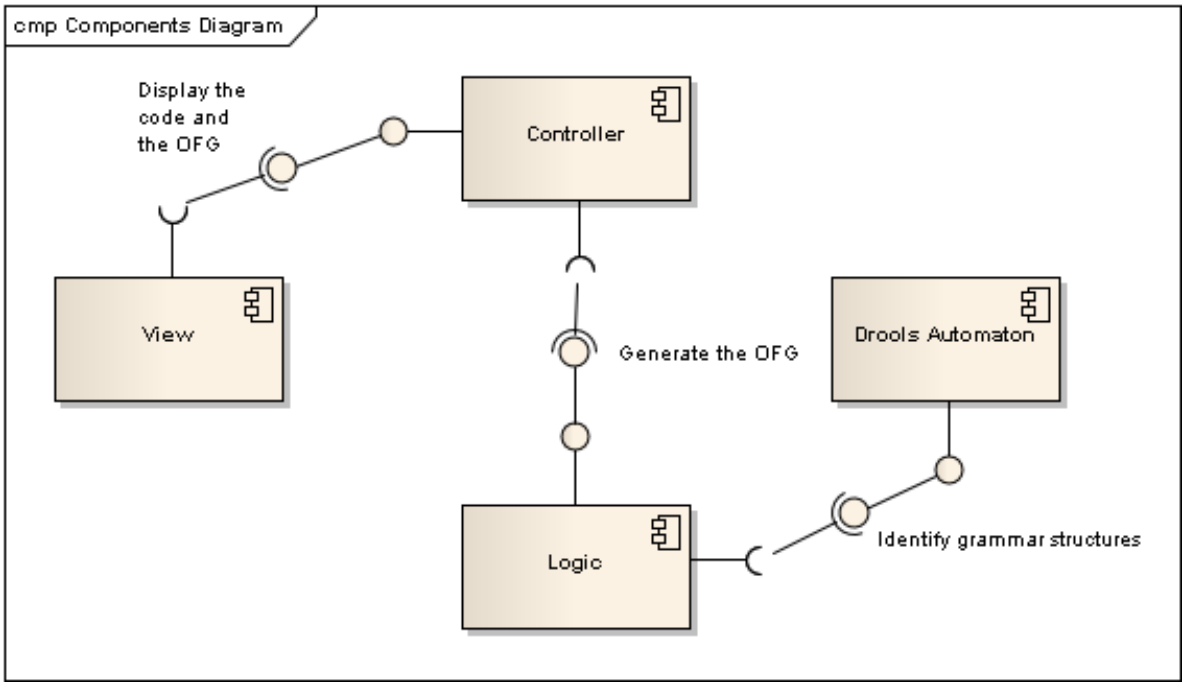


Fig. 2.7. Diagrama de componentes.

4.3.3. DIAGRAMAS DE CASOS DE USO

Además de mostrar que es lo que debe realizar el sistema en el mundo real, también se debe enfatizar sobre las funciones que debe realizar el aplicativo software a nivel de diseño, en otras palabras, las necesidades que debe satisfacer para cumplir con los objetivos específicos del funcionamiento del software.

4.3.3.1. VISUALIZAR EL CODIGO Y EL OFG

Este caso de uso representado por la **figura 2.8. Caso de uso "Visualizar el código y el OFG"**, está centrado en la vista del proyecto, donde un experto de software quiere visualizar el código y el OFG, pero para esto necesita realizar dos acciones, la primera es elegir el proyecto respecto al cual se va a crear el OFG, luego de haber hecho esto puede elegir la estructura que será el nodo inicial del OFG.

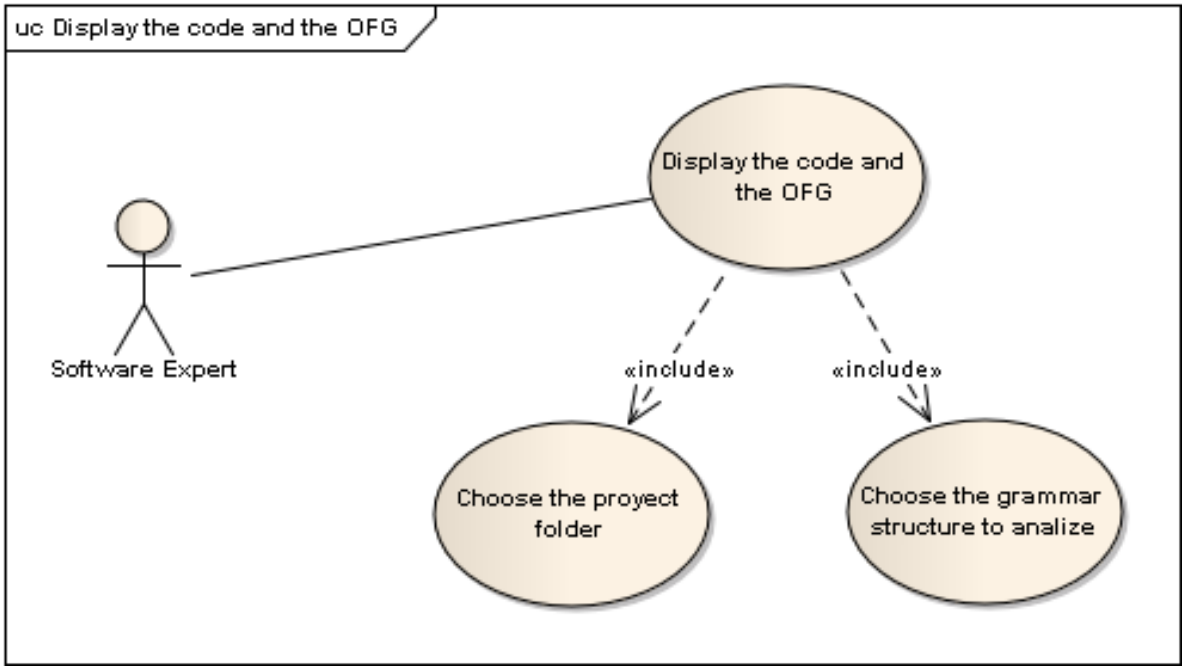


Fig. 2.8. Caso de uso "Visualizar el código y el OFG".

4.3.3.2. GENERAR EL OFG

Este caso de uso representado por la **figura 2.9. Caso de uso "Generar el OFG"**, está basado en las acciones del sistema luego de que el usuario haya elegido la el código y la estructura inicial respecto a la cual va a crear el OFG. Como se ve, el actor principal es el sistema, y el caso de uso global es "Generar el OFG", pero este caso de uso incluye varios más como son:

- Analizar la estructura del código: navega a través del proyecto convirtiendo cada una de las clases en Strings para luego ser analizadas.
- Definir los nodos del OFG: como su nombre lo indica, es quien toma cada una de las expresiones definidas en la sintaxis del sistema, y las convierte en nodos del OFG. Para esto, analiza la sintaxis del código y a su vez el léxico.
- Interconectar los nodos entre ellos: Toma cada uno de los nodos generados y los relaciona para generar el OFG.
- Crear el formato del OFG (XML o JPG): A partir del OFG generado, crea una imagen o un archivo XML dependiendo de lo que el usuario desee.

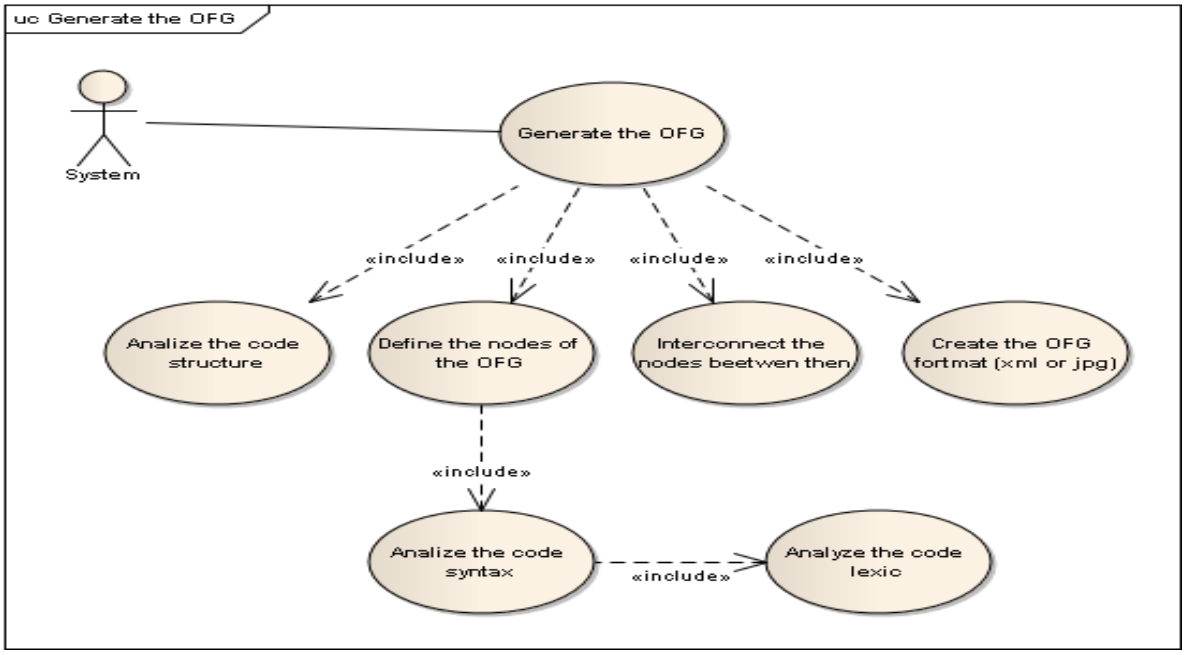


Fig. 2.9. Caso de uso "Generar el OFG".

4.3.4. DIAGRAMA DE SECUENCIA

Muestra de forma detallada el proceso para el desarrollo de las acciones solicitadas por el usuario, desde la primera capa que es la capa de la vista, hasta la capa del paquete rules. Como se puede observar en la **figura 2.10. Diagrama de secuencia**, el proceso inicia con la búsqueda del proyecto por parte del usuario, luego de esto, se verifica que el archivo tomado sea un folder y también que posea una estructura en la cual dentro debe haber un folder src. Luego de que el archivo pasa la verificación, el sistema procede a mostrar el proyecto seleccionado, permitiendo navegar a través de sus clases y de esa misma forma, permitiendo al usuario iniciar con el siguiente proceso, que es el de la selección de una estructura de entrada para la creación del OFG. Cuando el usuario selecciona la estructura, se envían mensajes a través de las capas controller y model, hasta llegar a la capa de rules, donde se analiza y se determina si es válida para ser el nodo raíz del OFG. Cuando la vista obtiene la respuesta de que la estructura seleccionada es válida, procede a enviar la información del proyecto a través de la capa controller hacia la capa model, donde se realiza el análisis de del proyecto buscando posibles candidatos para ser nodos hijos del nodo raíz, todo esto siempre validando las estructuras con la capa rules.

Si el proceso finaliza exitosamente, se devuelve un objeto de clase OFGelement, el cual contiene dentro de si otros elementos OFGelement, que a su vez, también pueden contener nuevos elementos, que es la base del árbol que conforma el OFG. Quien toma ese elemento es la vista y lo transforma en un diagrama visible al usuario. Luego de ser visualizado, la vista envía el objeto de nuevo a la capa model pasando a través del controller, este proceso se realiza para crear el archivo xml, lo cual es el fin para la cadena de procesos que se deben llevar a cabo.

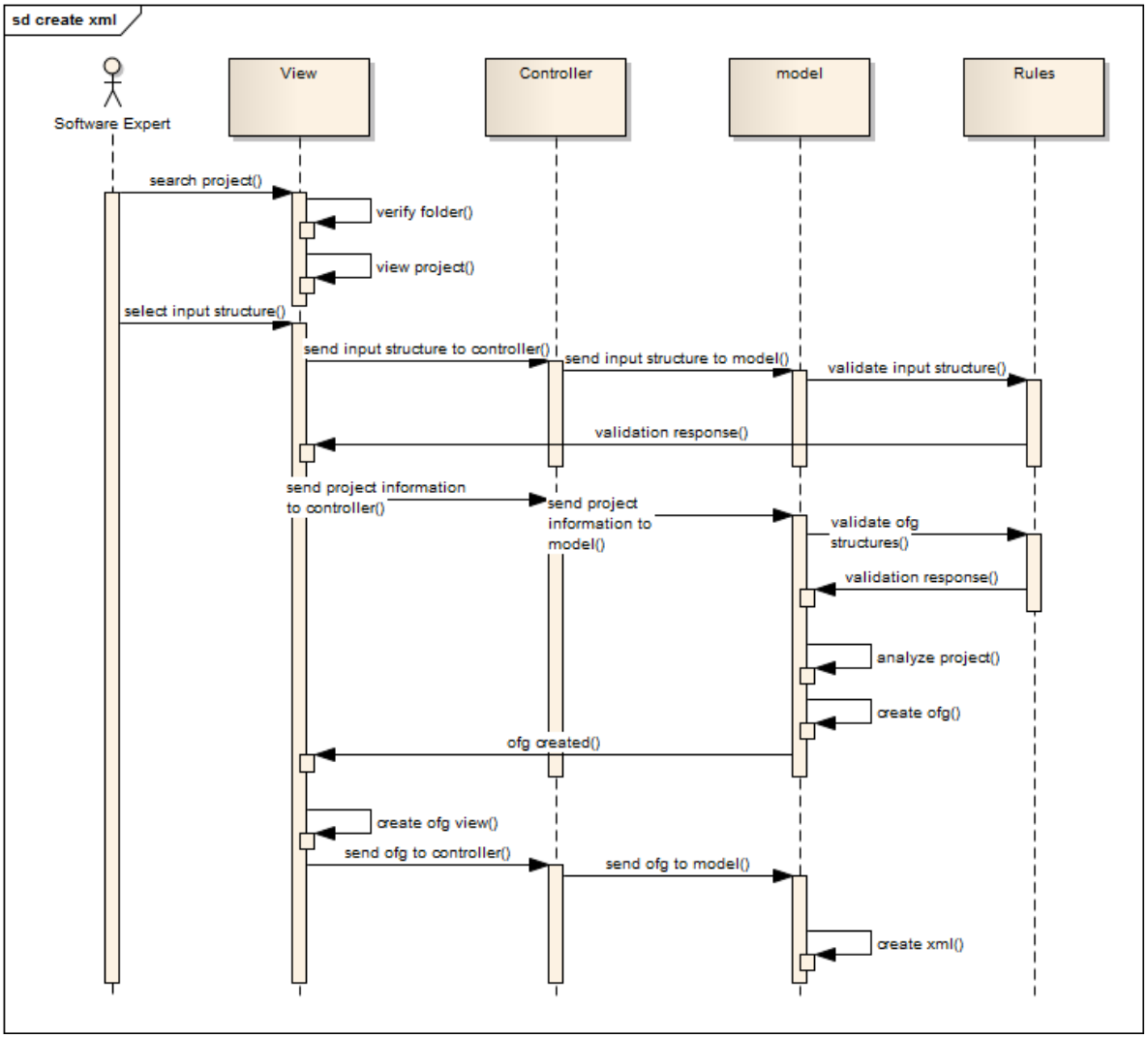


Fig. 2.10. Diagrama de secuencia.

4.4. IMPLEMENTACION DEL COMPONENTE

Como lenguaje de programación base, se utilizó Java para la implementación del componente, complementando este lenguaje, también existen varias tecnologías que ayudan a su funcionamiento y a la creación del OFG. Entre estas se mencionan:

- Gradle: es una API basada en el concepto de un ecosistema maduro de plugins e integraciones que permiten la automatización de tareas, especialmente de builds (construcciones). Se apoya en Groovy y en un DSL (Domain Specific Languague) para trabajar con un lenguaje sencillo y claro al momento de crear el build. Otro aspecto a resaltar es su gestión de dependencias, la cual permite que al momento de crear el build, se puedan añadir diferentes dependencias de forma fácil y rápida, que permitan el correcto funcionamiento del aplicativo. [24]
- Drools: es una solución BRMS (Business Rules Management System) que provee un BRE (Business Rules Engine). En otras palabras un sistema de gestión de reglas de negocio que usa un motor de reglas el cual se basa en inferencia de tipo fordward chaining y backward chaining, la cual hace una implementación avanzada del algoritmo Rete. [25]
- Javax.xml: es una librería de java que permite el procesamiento y la comunicación de documentos xml a través de sus diferentes paquetes. [26]
- RsyntaxTextArea: es una librería para Java Swing que extiende de JTextComponent y que permite el resaltado de sintaxis de manera rápida y eficiente en cualquier aplicación que necesite editar o ver código fuente. [27]

4.4.1. DIAGRAMA DE PAQUETES

Este diagrama representado por la **figura 2.11. Diagrama de paquetes**, muestra las relaciones que se presentan entre los distintos paquetes respecto a los cuales está organizado el proyecto. Existen 6 paquetes fundamentales que son los que permiten el funcionamiento del aplicativo, y cada uno de ellos tiene un fin específico. El paquete view es el encargado de todo lo relacionado con la interfaz de usuario. El paquete controller

regula el paso de mensajes entre la vista y las capas inferiores del aplicativo. El paquete model maneja toda la lógica de negocio, así como los procesamientos más importantes del sistema en relación a la creación del OFG. El paquete rules, contiene el archivo necesario para el análisis de las estructuras sintácticas con las cuales se conforma el OFG. Por último, el paquete javax.xml contiene clases usadas para la creación del xml y el paquete RSyntaxTextArea contiene las clases usadas para la visualización y resaltado del código fuente.

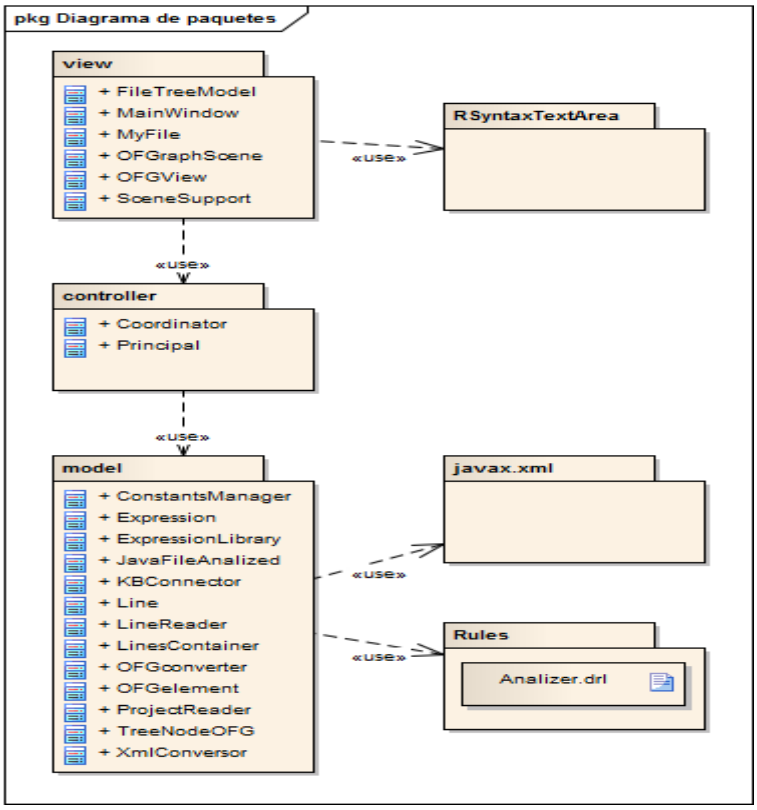


Fig. 2.11. Diagrama de paquetes.

4.4.2. DIAGRAMA DE DESPLIEGUE

Este diagrama representado por la **figura 2.12. Diagrama de despliegue**, muestra la interacción entre la máquina, el aplicativo y los distintos componentes que hacen parte del proceso de ejecución. Como se puede ver el dispositivo principal en todo esto es el PC, ya

que es el medio que realiza el procesamiento de la información. Luego de él, existe algo llamado JVM o máquina virtual de java, esta es la que ejecuta e interpreta el lenguaje de programación java, y permite la comunicación entre el sistema operativo del PC y el aplicativo. Una capa más arriba de la JVM, se encuentran el aplicativo y los artefactos relacionados con la ejecución del mismo. Por un lado está el código fuente, el cual es necesario para la compilación y generación del aplicativo, por otra parte está el archivo XML, que es el resultado de la ejecución del aplicativo.

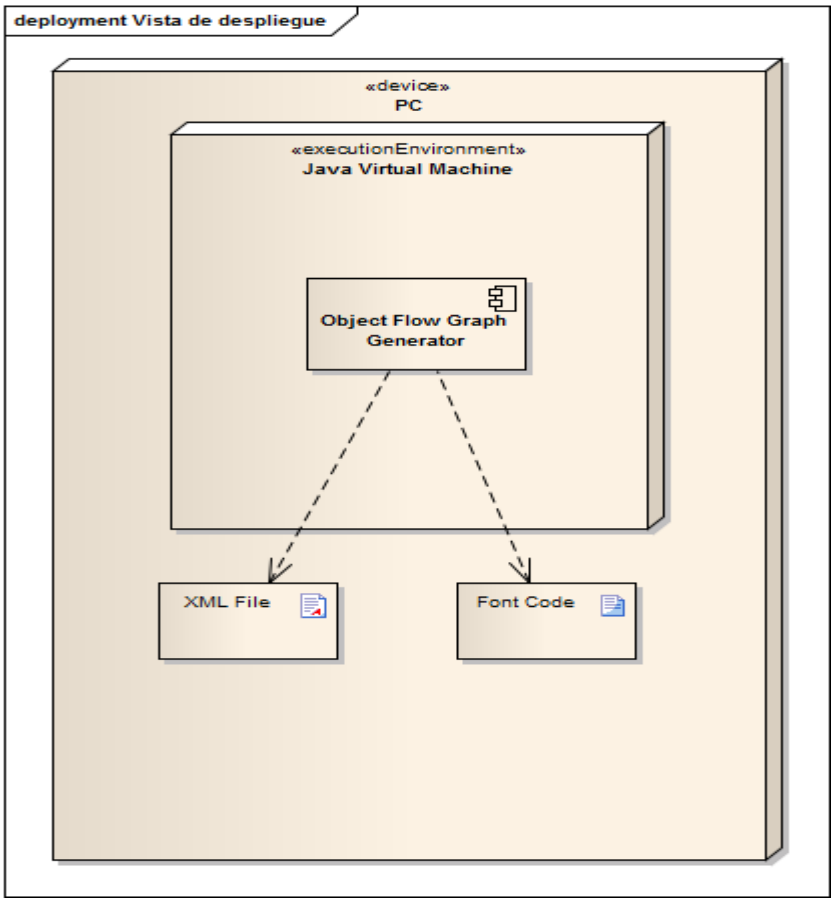


Fig. 2.12. Diagrama de despliegue.

4.5. EVALUACION DEL COMPONENTE

Los objetivos fundamentales de esta tesis son dos. El primero consiste en la visualización de un diagrama el cual muestra el OFG, y el segundo consiste en la creación de un repositorio XML que permite la persistencia del diagrama. Para validar estos objetivos, se realizaron diferentes pruebas, las primeras fueron con estructuras creadas desde 0, pero que permitían observar el comportamiento del aplicativo en sus etapas iniciales, las últimas pruebas se realizaron con el software JHotDraw, el cual es un framework de gráficos bidimensionales, creado por Erich Gamma's y otros.

4.5.1. FUNCIONAMIENTO

Lo primero que se encuentra al ejecutar el aplicativo como se puede evidenciar en la **figura 3.1. Vista principal**, es una pantalla en donde se tiene un pequeño menú, el cual permite seleccionar el proyecto respecto al cual se desea generar el OFG.

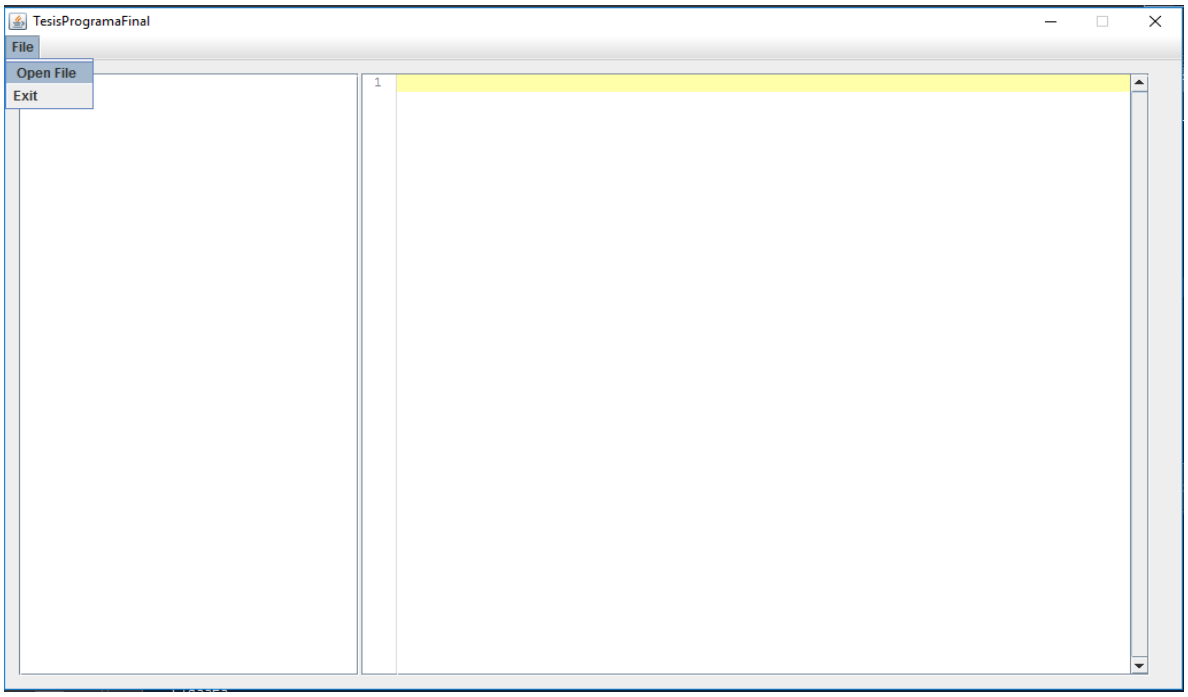


Fig. 3.1. Vista principal.

Al seleccionar la opción Open File, se abre una ventana en la cual se puede buscar el proyecto deseado, como se ve en la **figura 3.2. Seleccionar proyecto**.

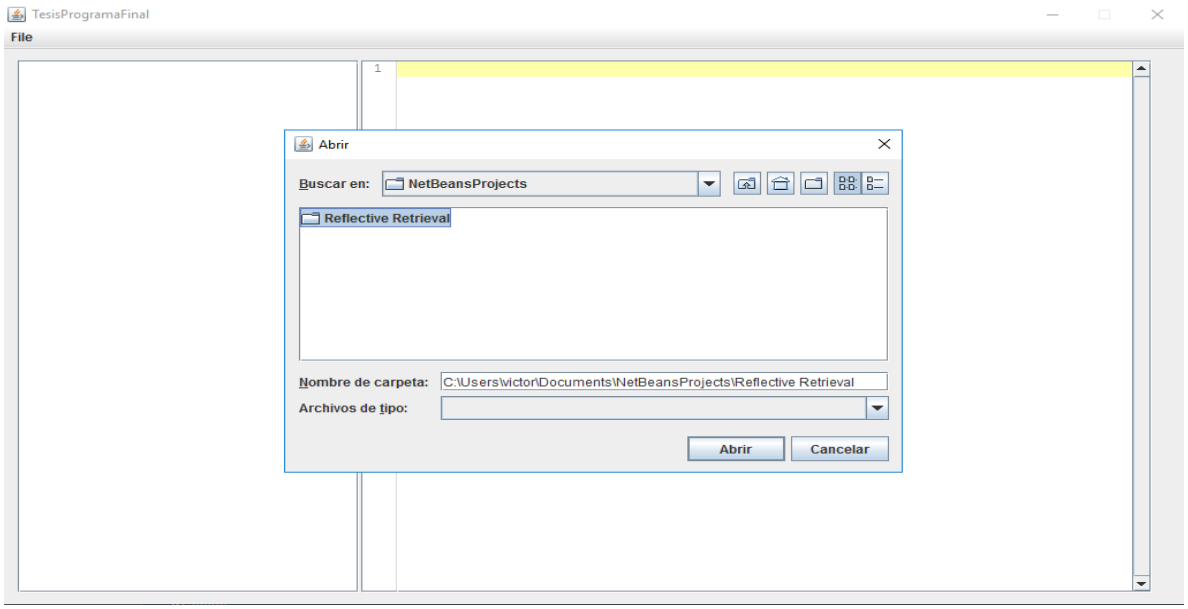


Fig. 3.2. Seleccionar proyecto.

Al abrir el proyecto como se evidencia en la **figura 3.3. Seleccionar Archivo y estructura raíz**, se carga el árbol de archivos en la parte de la izquierda y en la parte derecha, se carga el código fuente al seleccionar un archivo con extensión .java.

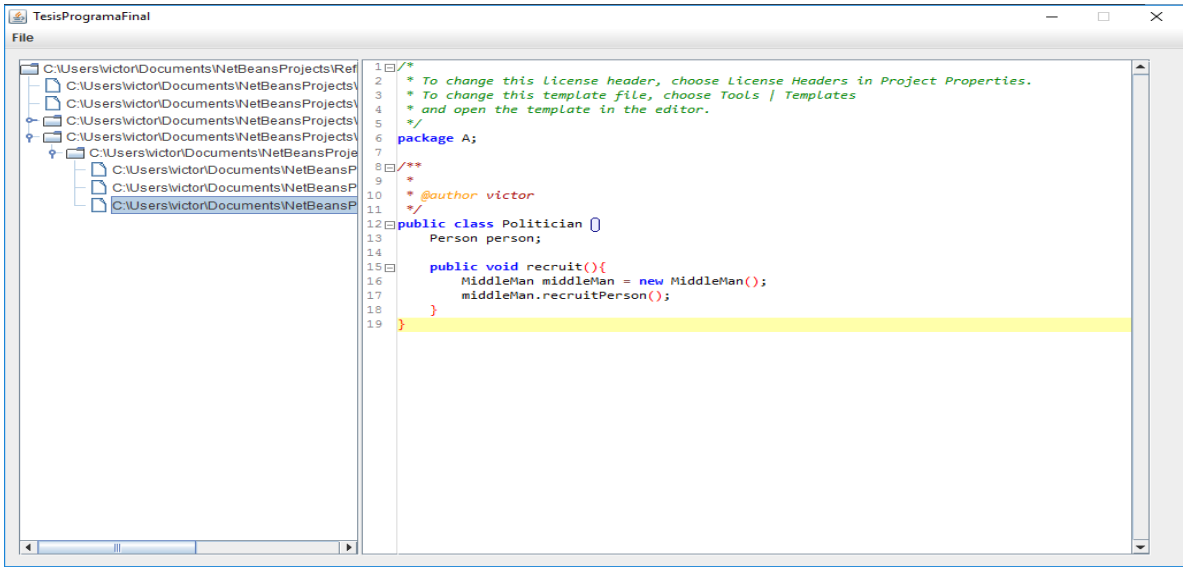


Fig. 3.3. Seleccionar Archivo y estructura raíz.

Luego de dar clic sobre una estructura valida como se ve en la **figura 3.4. Mostrar OFG**, se genera el OFG dependiendo de cómo sean los llamados del proyecto, Cada uno de los nodos es completamente movible a través del panel, y si se coloca el cursor encima de cualquiera de ellos, indicaran a que paquete, clase y método pertenece.

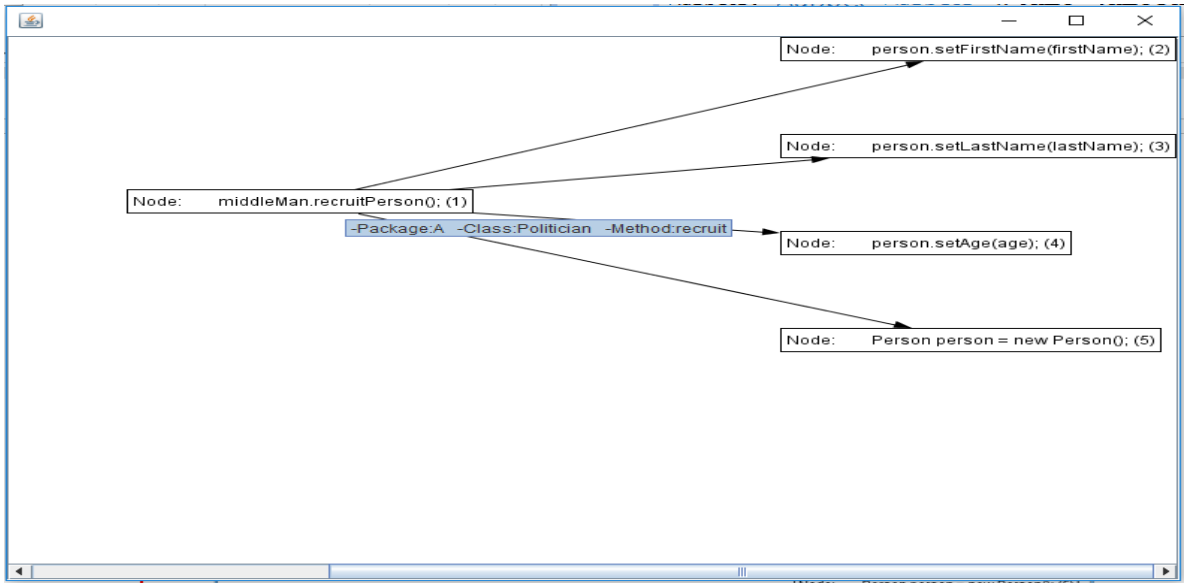


Fig. 3.4. Mostrar OFG.

Para ver el archivo XML hay que dirigirse a la carpeta tmp generada por el programa, como se evidencia en la **figura 3.5. Carpeta tmp**.

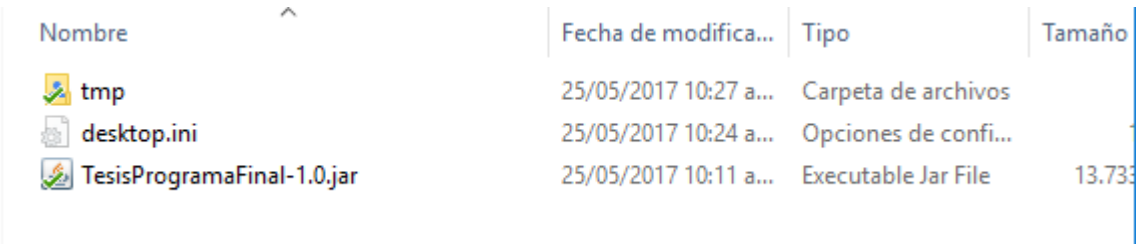


Fig. 3.5. Carpeta tmp.

Dentro de esta, se encuentra un archivo XML llamado output generado por el aplicativo. En este archivo, se puede observar el OFG en el formato XML. Esto se evidencia en la **figura 3.6. Archivo XML.**

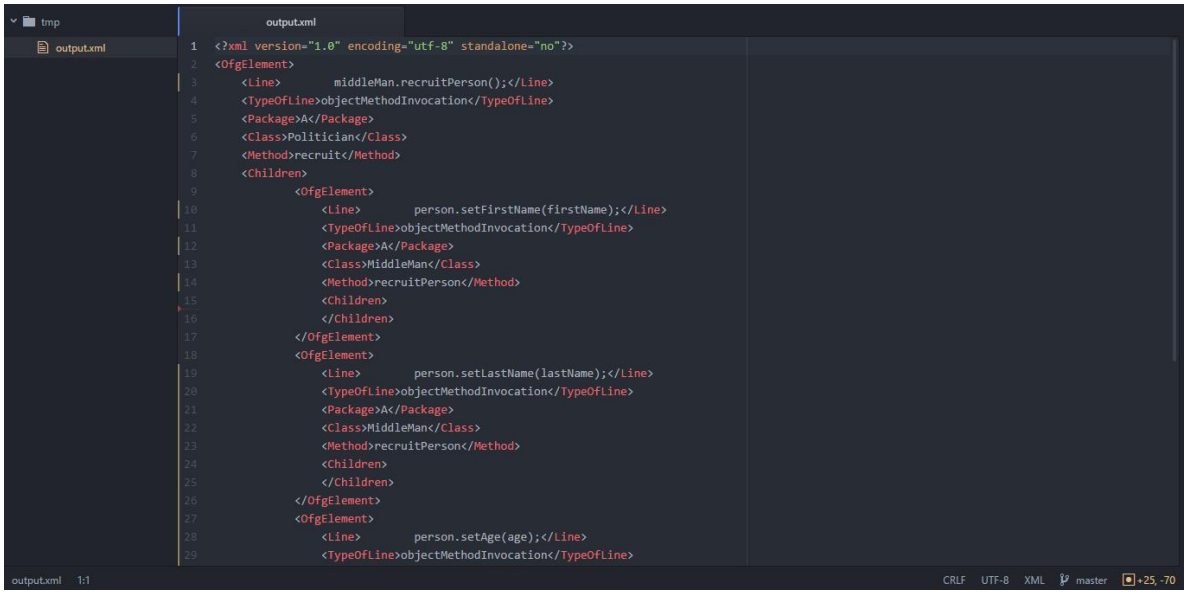


Fig. 3.6. Archivo XML.

4.5.2. PRUEBAS

Al inicio se realizaron enfocadas a dos puntos específicos, el primero era el tamaño del árbol, y el segundo era si podía armar el OFG a partir de las estructuras que se encuentran en diferentes partes del aplicativo (no solo en la misma carpeta). Para el primer caso, se probó con árboles de 2, 3 y 4 niveles, generando con éxito hasta el árbol de 4 niveles. Esto se puede evidenciar en la **figura 3.7. Árbol de 4 niveles**.

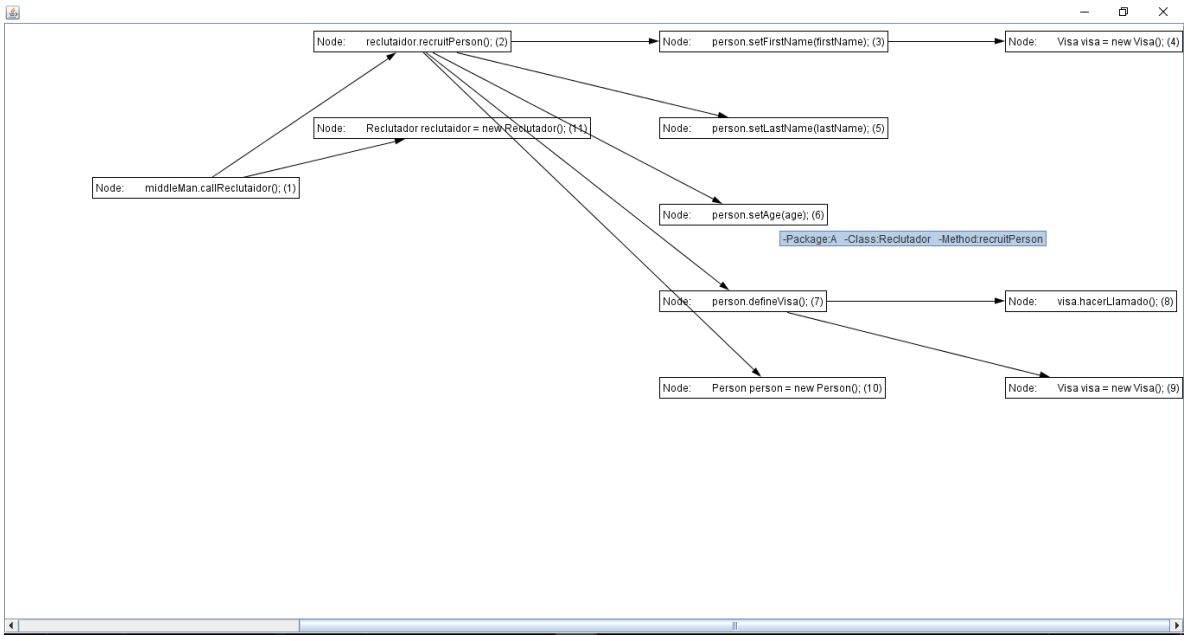


Fig. 3.7. Árbol de 4 niveles.

Para el segundo caso representado por la **figura 3.8. Árbol de JHotDraw**, se usó el software JHotDraw antes mencionado. Con el cual se tomó como parámetro inicial la línea 384 (mb.add(m);) del archivo DefaultMDIApplication.java que se encuentra en la carpeta app.

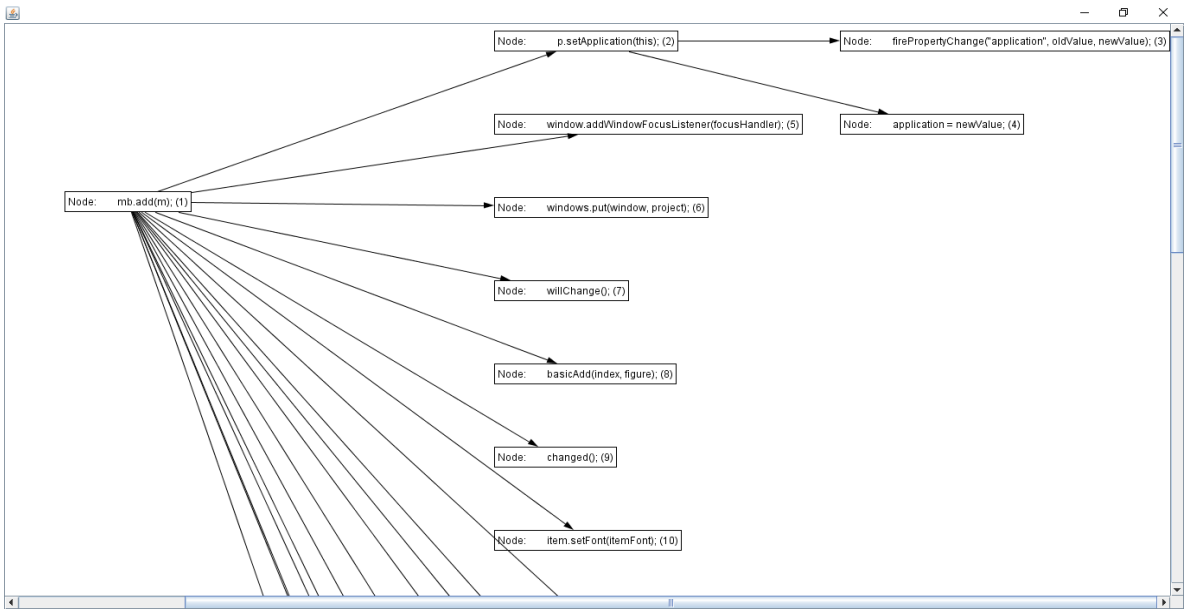


Fig. 3.8. Árbol de JHotDraw.

También fue generado con éxito al igual que su archivo XML. Todos estos archivos de prueba se pueden encontrar con el dentro del cd de entrega.

El proyecto actualmente se encuentra desarrollado en gradle, se hizo usando el IDE NetBeans, por lo cual es necesario descargar el plugin “Gradle support” para poder ejecutarlo desde el IDE.

ANALISIS DE LOS RESULTADOS

Se presentaron dos dificultades principales las cuales hicieron que el proyecto generara inconvenientes. La primera de ellas fue el uso de los métodos recursivos para la generación y lectura del OFG. Cuando un método es llamado recursivamente, cada vez que se llama se genera una copia del método original, con nuevos parámetros locales listos para ser usados, pero, los parámetros globales permanecen intactos, el problema de esto es que si modificas los parámetros globales, mientras estás haciendo cualquier especie de ciclo con ellos, se genera un concurrentmodificationexception. Para solucionar este inconveniente, se hizo uso de la clase CopyOnWriteArrayList, la cual genera una copia del array cada vez que es modificado, manteniendo la original intacta mientras se recorre el ciclo.

La segunda dificultad presentada fue la cantidad de pasos necesaria para la validación de las expresiones regulares, debido a que se dieron casos en los que se realizaban más de 135000 pasos, en estas ocasiones el aplicativo simplemente no avanzaba, debido al enorme procesamiento se quedaba estancado. La solución para esto fue el uso de lazy quantifiers y possesive quantifiers los cuales delimitan un poco más el retroceso de las expresiones y permiten un menor número de pasos. Aunque el uso de esta solución permitió el avance en el desarrollo, el conjunto de expresiones identificadas como nodos del OFG que el software permite analizar, quedó acotado por cuestiones de rendimiento en el número de pasos a las siguientes:

- **Expresiones simples:** identificadorObjeto.identificadorMetodo();
- **Expresiones simples con parámetros simples:**
    - identificadorObjeto.identificadorMetodo(parametroSimple);
    - identificadorObjeto.identificadorMetodo(parametroSimple1, parametroSimple2);
    - De la misma forma que los anteriores, aumentando la cantidad de parametros, hasta un numero n que varía dependiendo de la capacidad de procesamiento del equipo.

- **Expresiones con asignación a variable y parámetros simples:**
    - ((tipoDeObjeto)? identificadorObjeto | (tipoDeVariable)? identificadorVariable)) = identificadorObjeto.identificadorMetodo(parametroSimple);
    - ((tipoDeObjeto)? identificadorObjeto | (tipoDeVariable)? identificadorVariable)) = identificadorObjeto.identificadorMetodo(parametroSimple1, parametroSimple2);
    - De la misma forma que los anteriores, aumentando la cantidad de parametros, hasta un numero n que varía dependiendo de la capacidad de procesamiento del equipo.
- **Expresiones con declaraciones de nuevos objetos:**
    - ((tipoDeObjeto)? identificadorObjeto | (tipoDeVariable)? identificadorVariable)) = identificadorObjeto.identificadorMetodo(new identificadorMetodo());
    - ((tipoDeObjeto)? identificadorObjeto | (tipoDeVariable)? identificadorVariable)) = identificadorObjeto.identificadorMetodo(new identificadorMetodo1(), new identificadorMetodo2());
    - De la misma forma que los anteriores, aumentando la cantidad de parámetros, hasta un numero n que varía dependiendo de la capacidad de procesamiento del equipo.
- **Expresiones con declaraciones de nuevos objetos que reciben parámetros simples a través del constructor:**
    - ((tipoDeObjeto)? identificadorObjeto | (tipoDeVariable)? identificadorVariable)) = identificadorObjeto.identificadorMetodo(new identificadorMetodo(parámetroSimple));
    - ((tipoDeObjeto)? identificadorObjeto | (tipoDeVariable)? identificadorVariable)) = identificadorObjeto.identificadorMetodo(new identificadorMetodo(parámetroSimple1, parámetroSimple2));

    - ((tipoDeObjeto)? identificadorObjeto | (tipoDeVariable)? identificadorVariable)) = identificadorObjeto.identificadorMetodo(new identificadorMetodo1(parámetroSimple1), new identificadorMetodo2(parámetroSimple2));
    - ((tipoDeObjeto)? identificadorObjeto | (tipoDeVariable)? identificadorVariable)) = identificadorObjeto.identificadorMetodo(new identificadorMetodo1(parámetroSimple1, parámetroSimple2), new identificadorMetodo2(parámetroSimple1, parámetroSimple2));
    - De la misma forma que los anteriores, aumentando la cantidad de parámetros y declaraciones de nuevos objetos, hasta un numero n que varía dependiendo de la capacidad de procesamiento del equipo.
- **Expresiones con diferentes tipos de parámetros:**
    - ((tipoDeObjeto)? identificadorObjeto | (tipoDeVariable)? identificadorVariable)) = identificadorObjeto.identificadorMetodo(new identificadorMetodo(parámetroSimple), "cadena de caracteres", identificadorObjeto.identificadorMetodo(), identificadorMetodo());
    - De la misma forma que la anterior, aumentando la cantidad de parámetros y declaraciones de nuevos objetos, hasta un numero n que varía dependiendo de la capacidad de procesamiento del equipo.
- **Expresiones de instanciación de objetos:**
    - (tipoDeObjeto)? identificadorObjeto = new identificadorMetodo();

Cabe destacar, que todas estas expresiones son válidas como nodos del OFG, pero para el caso del **nodo raíz** se exceptúan las **expresiones de instanciación de objetos**. También es importarte indicar, que estas expresiones no se leen cuando se encuentran inmersas dentro de condicionales de otras expresiones, por ejemplo: **if(objeto.metodo()){** , solo pueden ser leídas cuando se encuentran en el cuerpo, por ejemplo: **if(condicional){objeto.metodo();}**

Para un mayor entendimiento, se definen los siguientes términos:

- **Parámetro Simple:** Cadena de caracteres, identificador de objeto, identificador de variable y valor numérico como se encuentran aceptados dentro del lenguaje Java.
- **Tipo de variable:** Nombre aceptado del tipo al que pertenece la variable dentro del lenguaje Java.
- **Tipo de objeto:** Nombre aceptado de la clase a la que pertenece el objeto dentro del lenguaje Java.
- **Identificador objeto:** identificador aceptado como nombre de un objeto dentro del lenguaje Java.
- **Identificador variable:** identificador aceptado como nombre de una variable dentro del lenguaje Java.
- **Identificador método:** identificador aceptado como nombre de un método dentro del lenguaje Java.
- **()?:** Para aquellas expresiones que se encuentren dentro de este término, indica que pueden existir una o ninguna vez (igual como funciona en el caso de las expresiones regulares.
- **|:** Al igual que en el caso de las expresiones regulares, este término representa un "o", lo que quiere decir que se puede dar el caso de lo que se encuentra a la derecha o a la izquierda del término.

Al final del desarrollo y luego de superar las dificultades presentadas anteriormente, se obtuvo un producto software funcional, el cual comprueba los planteamientos realizados por Tonella y Potrich (2005), ya que toma gran parte de su estructura base y la implementa para crear el OFG. Además, representa un aporte a los estudios realizados en el campo de la ingeniería inversa, porque la herramienta desarrollada sirve para hacer análisis de código fuente implementado, apoyando procesos de comprensión de sistemas heredados.

CONCLUSIONES

Como resultado del desarrollo de este proyecto, se dio respuesta a la pregunta de investigación planteada ¿Cómo implementar una herramienta software que permita registrar la trazabilidad del paso de mensajes a partir de código fuente orientado a objetos mediante l**a** representación del grafo de flujo de objetos?, obteniendo como resultado un conjunto de pasos, para los cuales se tomaron en cuenta las bases establecidas por Tonella y Potrich (2005), además se elaboró un modelo de negocio correspondiente a la trazabilidad del paso de mensajes a partir de código fuente orientado a objetos, se diseñaron las estructuras necesarias para la creación del OFG, se diseñó también el componente a partir del modelo de negocio, se implementó el componente en código Java garantizando la visualización del diagrama de flujo de objetos y por último, se evaluó el componente en busca de posibles fallos del sistema y en busca del cumplimiento con los requerimientos establecidos en un inicio, encontrando algunos detalles en cuanto a las estructuras analizadas y ampliando el conjunto de expresiones soportadas como parte del OFG.

Por otra parte, este proyecto generó para el investigador un nuevo conocimiento respecto a la forma en cómo se debe abordar un desarrollo de software, también se obtuvieron nuevas experiencias relacionadas con el manejo de estructuras de código complejas, como lo son las estructuras ciclomáticas conocidas con el nombre de métodos recursivos, otras estructuras como las expresiones regulares y nuevas tecnologías como por ejemplo Drools. La importancia de este estudio radica en el conocimiento obtenido a partir del mismo y a su vez, en el aporte que le brinda al campo investigativo de la ingeniería inversa, mediante la creación de un mecanismo que facilita la generación de los OFG a partir de código fuente. Algo que no se esperaba dentro de este proceso es la dificultad para la implementación de los diseños obtenidos a partir del modelo de negocio, ya que dentro de estos radica un conjunto de procesos de análisis bastante amplio y complejo. Al final, lo que dio luz al desarrollo fue el entendimiento de la recursividad, asi como el uso de los “lazy” y los “possesive quantifiers”.

RECOMENDACIONES

Se recomienda a cualquier persona que quiera tomar un proyecto de índole similar, crear una arquitectura solida basada en un modelo de negocio bien estructurado, en donde cada una de sus clases tenga una razón bien definida, y cada una de sus funciones posea una sola labor. De lo contrario puede verse afectado por la complejidad del análisis de este tipo de sintaxis y también en el entendimiento de su propio código. Como punto a parte, es necesario recalcar que se debe tener mucho cuidado con las expresiones regulares, las cuales pueden tornarse en un verdadero problema si no se las maneja bien, expresado de otra manera, esto puede presentar una limitante respecto al manejo de memoria, es decir, para realizar el análisis del código fuente, se necesita pasar la línea de código a través de un analizador, el cual realiza un gran numero regresiones para determinar si la línea pertenece o no a la expresión regular, como el tamaño de la memoria RAM para hacer estas operaciones no es infinito, se deben definir cuidadosamente las expresiones regulares para evitar el mayor número de regresiones posibles. Dentro de las limitaciones de esta investigación, se encuentran tanto el manejo de estructuras pertenecientes al OFG dentro de condicionales, como el manejo de estructuras de anidación de parámetros más complejas de las que se especifican en el capítulo de análisis de resultados, esto es debido a que agregar estas validaciones, representa un conjunto mucho más amplio de posibilidades y variaciones del que se pretendía en un inicio con el alcance de esta investigación. De forma general fue una gran experiencia en la que se aprendió mucho respecto al mundo de la programación y así mismo respecto al funcionamiento de los arboles especialmente los OFG.

Como trabajo futuro, para complementar el desarrollo propuesto en esta investigación, se recomienda agregar una nueva funcionalidad a la herramienta desarrollada que le permita generar diagramas de secuencia, bajo el estándar UML, y si es posible también se recomienda agregar la validación de expresiones pertenecientes al OFG dentro de condicionales y expresiones con anidación de parámetros más complejas.

BIBLIOGRAFIA

[1]. Briand, L. C.; Labiche, Y. and Miao, Y. (2003). Towards the reverse engineering of UML sequence diagrams. Proceedings of the 10th Working Conference on Reverse Engineering, Victoria (Canada), pp. 57-66.

[2]. El-Attar, M. and Miller, J. (2008). Producing robust use case diagrams via reverse engineering of use case descriptions. Software and System Modeling 7:67-73.

[3]. Guéhéneuc, Y. (2004). A reverse engineering tool for precise class diagrams. Proceedings of the 2004 Conference on the Centre for Advanced Studies on Collaborative Research, Markham (Canadá), pp. 28-41.

[4]. Keschenau, M. (2004). Reverse engineering of UML specifications from Java programs. Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Vancouver (Canadá), pp. 326-327.

[5]. Kollman, R. and Gogolla, M. (2001). Capturing dynamic program behaviour with UML collaboration diagrams. Proceedings of the 5th Conference on Software Maintenance and Reengineering, Lisboa (Portugal).

[6]. Levinson, J. and Nelson, D. 2006. Pro Visual Studio 2005 team system. Apress, Berkeley.

[7]. López, M.; Alfonzo, A.; Pérez, J.; González, J. and Montes, A. (2006). A metamodel to carry out reverse engineering of C++ Code into UML Sequence Diagrams. Proceedings of the Electronics, Robotics and Automotive Mechanics Conference, Cuernavaca (México).

[8]. Myatt, A. 2007. Pro NetBeans IDE 5.5 Enterprise Edition. Apress, New York.

[9]. Philippow, I.; Streitferdt, D.; Riebisch, M. and Naumann, S. (2005). An approach for reverse engineering of design patterns. Software and System Modeling 4(1): 55-70.

[10]. Rountev, A. and Connell, B. (2005). Object naming analysis for reverse-engineered sequence diagrams. Proceedings of the 27th International Conference on Software Engineering, St. Louis (Estados Unidos), pp. 254-263.

[11]. Rountev, A.; Volgin, O. and Reddoch, M. (2005). Static control-flow analysis for reverse engineering of UML sequence diagrams. Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Lisboa (Portugal), pp. 96-102.

[12]. Snavely, N.; Debray, S. and Andrews, G. (2005). Unpredication, unscheduling, unspeculation: reverse engineering Itanium executables. IEEE Transactions on Software Engineering, 31(2): 99-115.

[13]. Sutton, A. and Maletic, J. I. (2005). Mappings for accurately reverse engineering UML class models from C++. Proceedings of the 12th Working Conference on Reverse Engineering, Pittsburgh (Estados Unidos), pp. 175-184.

[14]. Tonella, P. and Potrich, A. 2005. Reverse engineering of object oriented code. Springer, New York.

[15]. Tsang, C. 2005. Object-oriented technology from diagram to code with Visual Paradigm for UML. McGraw-Hill, Maidenhead.
[16]. Wang, X. and Yuan, X. (2006). Towards and AST-based approach to reverse engineering. Proceedings of the Canadian Conference on Electrical and Computer Engineering, Ottawa (Canadá), pp. 422-425.
[17]. Yeh, D.; Sun, P.; Chu, W.; Lin, C. and Yang, H. (2007). An empirical study of a reverse engineering method for the aggregation relationship based on operation propagation. Empirical Software Engineering, 12:575-592.
[18]. Carlos Mario Zapata; Óscar Andrés Ochoa; Camilo Vélez (2008). Un método de ingeniería inversa de código java hacia diagramas de secuencias de uml 2.0, ev.EIA.Esc.Ing.Antioq no.9 Envigado Jan. /June 2008.
[19]. Martin Monroy R.; José L. Archiniegas H. y Julio Rodríguez R. (2012). Framework for recovery and analysis of behavioral architectural views. Cartagena, Bolívar, Colombia.
[20]. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman (1998). Compiladores: principios, técnicas y herramientas, Naucalpan de Juárez, Edo. de México.
[21]. MSc(c). Javier Mogollón Afanador, MSc. Luis Alberto Esteban Villamizar (2011). el desarrollo individual de proyectos de software: una realidad sin método. Pamplona, Norte de Santander, Colombia.
[22]. Gonzalo Sánchez Dueñas, Juan Antonio Valverde Andreu (1984). compiladores e intérpretes un enfoque pragmático. Los llanos, Madrid.
[23]. Kenneth E. Kendall, Julie E. Kendall (2005). Análisis y diseño de sistemas. Sexta edición. México.
[24]. Gradle Inc. 2017. Gradle Build Tool. [ONLINE] Available at: https://gradle.org/. [Accessed 10 May 2017].
[25]. Red Hat, Inc. or third-party contributors. 2006. Drools. [ONLINE] Available at: https://www.drools.org/. [Accessed 10 May 2017].
[26]. Oracle and/or its affiliates. 1993. javax.xml (Java Platform SE 7). [ONLINE] Available at: https://docs.oracle.com/javase/7/docs/api/javax/xml/package-summary.html. [Accessed 10 May 2017].
[27]. Fifesoft. 2015. RSyntaxTextArea. [ONLINE] Available at: http://bobbylight.github.io/RSyntaxTextArea/. [Accessed 10 May 2017].