UNIVERSIDAD
REY JUAN CARLOS

INGENIERÍA INFORMÁTICA
Curso Académico 2003 / 2004

Proyecto de fin de carrera

javaMod: una API para la gestión de programas Java

Tutor: Francisco Gortázar Bellas
Autor: Micael Gallego Carrillo

A Ana, por su cariño y compañía.
A Patxi, por su colaboración extrema.
A Soto, por su ayuda.
A Víctor, por su otro punto de vista.
A mis compañeros de clase, por sufrirme día a día.
A mis compañeros de café, por su tertulia.
A Luis, por su forma de ser.

Y a todos aquellos, que de una forma
u otra han ayudado y colaborado en este proyecto.

Índice general

Índice de tablas.....4

Índice de ilustraciones.....5

1 Introducción.....6

2 Estado del arte.....9

2.1 Herramientas de gestión de programas Java.....9

2.1.1 Taxonomía.....9

2.1.2 Herramientas de visualización del software, programación visual y gestión del software.....11

2.1.3 Herramientas Educativas.....13

2.2 APIs de gestión de programas Java.....14

2.2.1 Taxonomía.....15

2.2.2 APIs disponibles.....16

Información estructural.....16

Árbol de sintaxis abstracta.....16

Árbol de sintaxis abstracta con información semántica y elementos del lenguaje.....17

Intérprete.....17

Depurador.....18

Traza.....18

2.3 Conclusiones.....19

2.4 Referencias.....19

3 Metodologías y Tecnologías.....21

3.1 Metodología.....21

3.1.1 Proceso Unificado de Desarrollo.....21

3.1.2 Programación extrema.....23

3.1.3 Elección de la metodología.....24

3.2 Tecnologías software.....24

3.2.1 Orientación a objetos.....25

3.2.2 Java.....27

3.2.3 Procesadores de lenguajes.....27

3.2.4 JavaCC.....29

3.2.5 Depuradores....31
3.2.6 JPDA....31

4 Especificación de requisitos....33
4.1 Descripción general....34
4.1.1 Perspectiva del producto....34
Interfaces del sistema....34
Interfaces de usuario....34
Interfaces hardware....34
Interfaces software....34
4.1.2 Funciones del producto....34
4.1.3 Características del usuario....35
4.2 Requisitos específicos....35
4.2.1 Interfaces Externos....35
4.2.2 Requisitos funcionales....35
Requisitos para gestionar la definición de un programa Java....35
Requisitos para gestionar la ejecución de un programa Java y su traza....38
4.2.3 Requisitos no funcionales....42
4.3 Conclusiones....42

5 Diseño de la API javaMod....44
5.1 Definición de un programa Java....44
5.1.1 Elementos principales de un programa Java....44
5.1.2 Ficheros fuente....45
Diseño orientado a objetos basado en la gramática....46
Árbol de sintaxis abstracta....51
Representación textual de los elementos del código fuente....53
5.1.3 Elementos de Java que no aparecen en ficheros fuente....55
5.1.4 Referencias y Definiciones....56
5.1.5 Información semántica del programa....58
5.1.6 Modelado de las librerías....58
5.1.7 Asociación de objetos a los elementos....59
5.1.8 Repositorios de tipos....59
5.2 Ejecución de un programa Java....61
5.2.1 Inicio, finalización y control de la ejecución....61
5.2.2 El estado de la ejecución....64

5.2.3 La información de un programa Java....65
5.2.4 Interacción con los objetos....67
5.2.5 Asociación de objetos a los elementos....67
5.2.6 Ejecución paso a paso....67
5.2.7 Entrada y salidas del programa en ejecución....68
5.2.8 Recibir notificaciones ante eventos de interés....68
5.3 Distribución en paquetes....70

6 Implementación y pruebas de la API....72
6.1 Definición de un programa Java....72
6.1.1 Identificación de los paquetes de las librerías....74
6.1.2 Inicialización de los tipos primitivos y operadores....74
6.1.3 Tratamiento especial de algunas clases de las librerías....75
6.1.4 Construcción del árbol de sintaxis abstracta....75
6.1.5 Enlaces de las referencias a las definiciones....77
6.2 Ejecución de un programa Java....78
6.2.1 Construcción de los ficheros fuente adaptados a JPDA....79
6.2.2 Compilación de los ficheros fuente....79
6.2.3 Gestión de la ejecución con JDI....80
Adaptación del modelo de eventos de JDI al modelo de javaMod....80
Control paso a paso de la ejecución basado en elementos del código....80
Gestión eficiente de los valores de la ejecución....81
6.3 Pruebas....82
6.4 Instalación y uso de la implementación....82

7 Conclusiones y trabajos futuros....83
7.1 Principales aportaciones de javaMod....83
7.1.1 javaMod: An Integrated Java Model for Java Software Visualization....83
7.1.2 Depuración Estructural: Acercando la Práctica a la Teoría de la Programación....85
7.2 Trabajos futuros....86
7.2.1 Ampliación de la API para cubrir todos los requisitos....86
7.2.2 Implementación de la API como plugin para los entornos de desarrollo....87
7.2.3 Adaptación de la API javaMod a la versión del lenguaje incluido en J2SE 1.5....87
7.2.4 Incorporación los comentarios convencionales y de JavaDoc a javaMod. .88

7.2.5 Automatizar la construcción de APIs con los conceptos de javaMod para otros lenguajes.....88
7.2.6 Detección de patrones de diseño en programas Java.....89

Bibliografía.....90

Índice de tablas

Tabla 1. Comparativa de las características relevantes de diversas herramientas de visualización y gestión del software.....12
Tabla 2. Comparativa de las características relevantes de las diversas herramientas educativas.....14
Tabla 3. Direcciones de Internet y referencias bibliográficas con información sobre las herramientas y APIs.....20

Índice de ilustraciones

Figura 1. El programa java y sus componentes principales.....45
Figura 2. Interfaz de la API en UML para representar la sentencia while.....47
Figura 3. Interfaz de la API para representar un parámetro.....48
Figura 4. Jerarquía de herencia de las sentencias.....48
Figura 5. Jerarquia de herencia de CodeElementRepresenter.....54
Figura 6. Jerarquía de herencia de los tipos.....55
Figura 7. Jerarquía de herencia de las operaciones.....56
Figura 8. Diagrama de clases de los interfaces Definition, Reference e Identifier...57
Figura 9. Jerarquía de herencia de los repositorios de tipos.....60
Figura 10. Relaciones de composición de los repositorios de tipos.....61
Figura 11. Diagrama de clases que muestra los interfaces y métodos que controlan la ejecución.....63
Figura 12. Diagrama de clases que muestra los interfaces que representan el estado de la ejecución.....65
Figura 13. Diagrama de clases que muestra los items de informacion.....66
Figura 14. Diagrama de clases que muestra la jerarquía de valores.....66
Figura 15. Transformación de todos los ficheros que forman el programa Java en una estructura orientada a objetos.....73
Figura 16. Arquitectura de la aplicación JavaDebugger usando javaMod.....84
Figura 17. Captura de pantalla de JavaDebugger en la que se muestran las diferentes visualizaciones sincronizadas de los distintos aspectos del programa.....85

1 Introducción

En la actualidad, y debido a la creciente complejidad del software, existen cada vez un mayor número de herramientas cuyo fin es la *gestión de programas,* entendiendo por *gestión,* tratamiento, manipulación, análisis, traducción, etc., y por *programa,* el código, las librerías, la documentación, la ejecución, etc. El objetivo de esta gestión de programas es mejorar, tanto los programas en sí, como el proceso de desarrollo de los mismos.

Algunas herramientas que gestionan programas son los entornos de desarrollo integrados, herramientas de cálculo de métricas, compiladores, etc. Al construir este tipo de herramientas es necesario gestionar un programa en su definición y ejecución, y para ello, hay que usar tecnologías de bajo nivel, muy diferentes y poco integradas entre sí. En este contexto, sería interesante disponer de una librería que permitiese la gestión de un programa en todos sus aspectos de una forma integrada, ortogonal, completa, adaptable y fácil de usar.

Uno de los principales campos de investigación del grupo ViDo[1] es el desarrollo de herramientas de *visualización del software, programación visual* y *de la enseñanza de la programación.* En este contexto, surge la necesidad de construir una librería de programación o API (Application Programming Interface) que represente y permita gestionar todos los aspectos de un programa escrito en un determinado lenguaje de programación.

La *visualización del software* [Stasko98] es una disciplina que estudia cómo representar, normalmente de forma gráfica, los elementos que aparecen en el desarrollo software, tanto en su definición como en su ejecución. El software es intangible, no tiene forma ni tamaño, y normalmente se encuentra definido en diversos ficheros fuente y librerías, con distintos formatos. Al ejecutarlo se representa en estructuras en la memoria del sistema. Las herramientas de visualización del software usan técnicas para hacerlo visible, para mostrar los elementos que lo definen y su comportamiento en ejecución.

1 Grupo de Visualización y Documentación electrónica de la ESCET en la Universidad Rey Juan Carlos

La *programación visual* [Myers86] es una disciplina que estudia el uso de las representaciones gráficas para la definición de programas, estableciendo técnicas que permiten programar mediante diagramas, iconos, etc. El objetivo es conseguir que el proceso de desarrollo sea más sencillo, rápido y cómodo.

Las *herramientas para la enseñanza de la programación* facilitan el proceso de aprendizaje de la misma y para ello, se suelen utilizar diagramas, se simplifica la interfaz de usuario, los mensajes de error están adaptados a las necesidades específicas de los alumnos, etc. Estas herramientas están muy relacionadas con las dos disciplinas anteriores, ya que representaciones visuales adecuadas ayudan a la comprensión y asimilación de los conceptos del desarrollo software.

En este Proyecto de Fin de Carrera se define e implementa una API, llamada *javaMod*, que representa y permite gestionar todos aspectos de un programa escrito en el lenguaje de programación Java. El lenguaje elegido es Java porque es uno de los lenguajes más utilizados en la actualidad, orientado a objetos, multiplataforma e impartido por muchas instituciones.

Esta API representa un programa Java desde dos puntos de vista: su definición, formada por código y librerías; y su ejecución, la cual se puede ejecutar paso a paso, consultando el estado del sistema, o bien permitiendo registrar una traza de la misma. Está definida e implementada también en lenguaje Java y sus objetivos principales son: facilidad de comprensión y manejo, ortogonalidad al representar los conceptos y tratamiento integrado de la definición y la ejecución.

En la definición de javaMod se han usando los principios de la programación orientada a objetos. Por otro lado, es independiente de la implementación, es decir, se ha definido usando interfaces. Esto permite que se puedan desarrollar implementaciones diferentes que obedezcan a distintos criterios.

El resto de esta memoria se estructura en diferentes capítulos. En el capítulo 2 se presenta un conjunto de los trabajos más representativos relacionados con este proyecto. Se describen algunas herramientas educativas, de visualización del software y programación visual desarrolladas para Java. Además, se muestran algunas de las APIs que permiten gestionar los programas en Java. En el capítulo 3, se presenta la metodología seguida y se introducen conceptos básicos acerca de las tecnologías utilizadas en el desarrollo. El capítulo 4 describe los requisitos usados para construir la API. En el capítulo 5 se muestran los pasos seguidos en el diseño de la misma. El capítulo 6 describe brevemente la implementación desarrollada. Por último, en el capítulo 7, se muestran las conclusiones de este proyecto, indicando los logros alcanzados y los trabajos futuros.

2 Estado del arte

JavaMod tiene como objetivo principal proporcionar un conjunto de servicios que permitan la gestión integrada de los distintos aspectos de un programa Java. Es interesante estudiar las APIs existentes para gestionar programas Java. Cada una de estas APIs no proporciona una forma integrada de gestionar los distintos aspectos, sino que se centra en un aspecto determinado. Para el desarrollo de javaMod se ha puesto especial interés en que resulte cómodo construir herramientas educativas, tanto de visualización del software como de programación visual.

En la primera parte de este capítulo se muestran varias herramientas que gestionan código Java. Se comienza con una pequeña taxonomía de los distintos aspectos que se pueden visualizar en ellas y posteriormente se describen algunas de las más representativas. En la segunda parte, se muestran varias APIs que permiten gestionar código Java. También se comienza con una pequeña taxonomía basándose en el aspecto del programa que permiten gestionar y posteriormente se describen algunas de las más representativas. Por último se ofrecen las conclusiones y en la Tabla 3 se indican la direcciones de Internet y/o referencias bibliográficas donde encontrar información sobre las herramientas y APIs mencionadas en este capítulo.

2.1 Herramientas de gestión de programas Java

Las herramientas educativas, de visualización del software y programación visual, pueden manejar diversos aspectos del software y a diferentes niveles de detalle. A continuación se muestra una pequeña taxonomía de las aplicaciones dependiendo de de la información que se visualiza y su grado de detalle. Posteriormente se describen algunas de las herramientas más representativas agrupadas en herramientas de visualización del software, programación visual y gestión del software y en herramientas educativas, todas ellas para el lenguaje Java.

2.1.1 Taxonomía

Atendiendo al aspecto del programa Java que visualizan y al diferente nivel de detalle las visualizaciones que pueden aparecer en las herramientas se pueden clasificar en:

- **Definición del software:** Está formada por ficheros fuente y librerías. La información que se visualiza es la que se obtiene del proceso de compilación.
  - **Estructura del software (alto nivel):** En este apartado se encuentran los diagramas que muestran el software desde un punto de vista arquitectónico. Se suelen usar diagramas que muestran los módulos o paquetes de un programa, desde que módulos se usan otros módulos, etc...
  - **Código fuente (bajo nivel):** La visualización habitual es el código fuente con resaltado de sintaxis y con la identación adecuada. Actualmente se está comenzando a realizar un coloreado semántico, es decir, usando la información semántica de cada elemento del texto y no sólo los aspectos léxico-sintácticos. Las visualizaciones también pueden incluir iconos en el código fuente para mostrar alguna característica, subrayando partes para mostrar errores, insertando listas desplegables, etc... Igualmente se han construido visualizaciones interactivas que permiten ocultar ciertas partes del código, no relevantes en un momento determinado, mejorando la legibilidad del mismo.
- **Ejecución del software:** La ejecución del software puede visualizarse desde dos puntos de vista: por un lado, se pueden mostrar los elementos que aparecen en un instante determinado durante la ejecución paso a paso de un programa; y por otro lado se puede visualizar una traza de la ejecución.
  - **Ejecución paso a paso:** Se representan los valores de las variables en un instante determinado y los métodos que se están ejecutando en ese mismo momento. Más concretamente, se muestran los hilos de ejecución, la pila, la memoria dinámica, etc...
    - **Alto nivel:** Se visualizan las estructuras de datos sin mostrar los detalles. Se pueden representan mostrando su topología lógica en vez de mostrar los detalles de implementación de la misma.
    - **Bajo nivel:** De forma detallada se muestran los valores de cada una de las variables del programa en ejecución, la próxima sentencia que se va a ejecutar, etc...
  - **Traza de la ejecución:** En este tipo de visualización se muestra la evolución temporal de todas las acciones llevadas a cabo durante la ejecución de un programa. Se pueden visualizar los valores que han tomado las distintas variables, las llamadas que se han realizado a los diversos métodos, los objetos que han existido, etc...

- **Alto nivel:** Se muestran aquellas acciones más representativas de la ejecución. Suelen visualizarse los métodos que han sido ejecutados a los largo de la ejecución, la evolución de la memoria, etc...
- **Bajo nivel:** Están más enfocadas a mostrar cada una de las acciones realizadas durante la ejecución, visualizando los valores de cada una de las variables a lo largo del tiempo, las sentencias ejecutadas, los valores temporales de las variables, etc...

El principal objetivo de javaMod es construir herramientas que visualicen los aspectos aquí mencionados. Para ello, deberá reflejar cada uno de éstos con alto grado de detalle y además, deberá proporcionar mecanismos para extraer información de alto nivel de una forma sencilla.

2.1.2 Herramientas de visualización del software, programación visual y gestión del software

En este apartado, se muestra una relación de algunas de las herramientas de visualización del software y programación visual más representativas. También se incluyen diversas herramientas que permiten gestionar programas java de una u otra forma, por ejemplo para cálculo de métricas, generación de HTML con resaltado de sintaxis, etc... Cada una de estas herramientas está concebida con distintos objetivos. Lo que se pretende es tener una idea general de los servicios que debe proporcionar una API que permita gestionar programas para realizar con él cualquier tipo de proceso. A continuación se muestra una breve descripción de las herramientas más representativas:

- **Java Roudtrip Engineering:** Obtiene una modelo UML[2] en formato XMI[3] partiendo de un código Java.
- **Jacot**: Visualiza la concurrencia en Java mediante diagramas UML.
- **SeeSoft[4]:** Ayuda en la visualización de gran cantidad de código fuente mostrando información estadística sobre cada línea de una forma compacta.

2 Lenguaje unificado de modelado (Unified Modeling Language). Define el formato de una serie de diagramas para el modelado de aplicaciones.
3 Formato estándar basado en XML para representar diagramas UML.
4 Pese a no estar específicamente diseñada para java, es muy representativa de sistemas de visualización de grandes cantidades de código fuente

- **Jinsight**: Desarrollada por IBM. Pensada para orientación a objetos y multihilo, especialmente para análisis de rendimiento y depuración. Utiliza un fichero de traza generado por la máquina virtual de Java.
- **Omniscient Debugging:** Depura un programa desde una traza de su ejecución, lo que permite ir hacia delante y hacia atrás en la ejecución.
- **Jrat**: Obtiene información de un programa en ejecución con diversas técnicas como JPDA[5], instrumentación de código, etc...
- **Evolve:** Es una herramienta para crear visualizaciones de la ejecución de un programa java basado en su traza.
- **Fujaba:** Basa la programación en Java en el uso de gráficos UML. Permite construir objetos e invocar métodos sobre ellos. Muestra las relaciones entre ellos como un diagrama de objetos UML. La implementación de los métodos se define usando diagramas de actividad, muy cercano a la programación visual. Tratan de permitir depurar directamente con esos diagramas.

En la Tabla 1 se muestra una comparativa de las características mas relevantes de cada una de estas herramientas.

Herramienta	Definición del software	Ejecución paso a paso	Traza	Tecnología
Java Roundtrip Engineering	Sí / Alto nivel	No	No	¿?
Jacot	No	Sí / Alto nivel	No	JPDA
SeeSoft	Sí / Bajo nivel	No	No	¿?
JInsight	No	No	Sí / Bajo nivel	JVM modificada
Omniscient Debugging	No	No	Sí / Bajo nivel	Instrumentación
JRat	No	Sí / Bajo nivel?	No	Varias
Evolve	No	No	Sí / Alto y bajo nivel	Protocolo de traza
Fujaba	Sí / Alto nivel	Sí /Alto nivel	No	¿?

Tabla 1. *Comparativa de las características relevantes de diversas herramientas de visualización y gestión del software.*

5 Arquitectura de Depuración para la Plataforma Java (Java Platform Debugger Architecture)

2.1.3 Herramientas Educativas

La visualización del software y la programación visual son técnicas que mejoran el aprendizaje de la programación. Por otro lado, la programación orientada a objetos se impone como un paradigma adecuado para la construcción de la gran mayoría de las aplicaciones actuales. Por tanto, el uso de técnicas de visualización en herramientas educativas será beneficioso. Están en fase de desarrollo técnicas de visualización específicas para la comprensión de los conceptos presentes en la programación orientada a objetos [Mehner00] [Systä00]. El uso de diferentes visualizaciones a medida que se van aprendiendo los conceptos resultarían muy útiles.

A continuación se muestran una serie de herramientas diseñadas para la enseñanza de la programación con el lenguaje Java. Describiremos de cada una de ellas las características principales:

- **JavaVis:** Herramienta que muestra la ejecución de programas java mediante diagramas de objetos y de secuencia.
- **BlueJ:** Herramienta de IBM en la que los programas se construyen de forma visual usando una notación similar a los diagramas de clases de UML. Se pueden instanciar objetos interactivamente y cambiar los valores de los atributos o invocar métodos sobre los objetos.
- **DrJava**: Herramienta con el compilador de Java integrado. Dispone de un panel de interacción que permite invocar métodos, declarar variables, etc... al estilo de un intérprete de comandos. También dispone de JavaDoc integrado, de forma que puede generarse documentación rápida y cómodamente. El programa puede ejecutarse paso a paso.
- **Fujaba life**: Es la versión educativa de FUJABA (ver sección anterior), en la que han tratado de poner especial cuidado en la especificación de las distintas fases del desarrollo de programas.
- **JGRASP**: Entorno de desarrollo que utiliza tanto visualizaciones de código fuente (decoradas de forma algo más avanzada que el simple resaltado de sintaxis) como diagramas UML de clases. El entorno ofrece las características usuales en cuanto a depuración.

- **ProfessorJ:** Herramienta evolutiva que introduce la programación orientada a objetos en tres niveles: principiante, intermedio y avanzado. Los errores se presentan de una forma más explicativa, y evolucionan según se avanza por los distintos niveles. ProfessorJ es un plugin[6] para un entorno de desarrollo genérico denominado DrScheme.

En la Tabla 2 se muestra una comparativa de las características mas relevantes de cada una de estas herramientas.

Herramienta	Definición del software	Ejecución paso a paso	Traza	Tecnología
JavaVis	No	Sí / Alto nivel	No	JPDA
BlueJ	Sí / Bajo y alto nivel	Sí / Alto nivel	No	JPDA
DrJava	Sí / Bajo nivel	Sí / Bajo nivel	No	Intérprete
Fujaba life	Sí / Alto nivel	Sí / Alto nivel	No	?¿
JGRASP	Sí / Bajo y alto nivel	Sí / Bajo y alto nivel	No	DynamicJava y JPDA
ProfessorJ	Sí / Evolutivo	Sí / Bajo nivel	No	Plugin para DrScheme

Tabla 2. *Comparativa de las características relevantes de las diversas herramientas educativas.*

2.2 APIs de gestión de programas Java

En la actualidad existen diversas APIs que permiten gestionar programas Java en alguno de sus aspectos. El principal inconveniente de usar estas APIs para construir herramientas es que sólo permiten gestionar uno de los muchos aspectos del programa. No proporcionan una visión integrada de todos ellos como se propone con javaMod.

En el siguiente apartado se muestra una taxonomía de las APIs dependiendo del aspecto del programa Java que permiten gestionar. Por último se describen algunas de las implementaciones de los distintos tipos más representativas.

6 Módulo que se incorpora a un programa para añadirle más funcionalidad

2.2.1 Taxonomía

Atendiendo al aspecto del programa Java que permiten gestionar, las APIs se pueden clasificar en:

- **Información estructural:** Representa parcialmente un programa en tiempo de compilación. Tan sólo se pueden gestionar las entidades de alto nivel de la definición de un programa Java. Esta información incluye paquetes, clases, interfaces, métodos, constructores, atributos y tipos. La información está representada mediante una estructura de objetos.
- **Árbol de sintaxis abstracta:** Representa parcialmente un programa en tiempo de compilación. Mantiene el árbol de sintaxis abstracta como una estructura de datos orientado a objetos. Este árbol es el resultado de un proceso de análisis léxico-sintáctico del código fuente. Este tipo de APIs representan la información de un fichero de código fuente, lo que incluye la declaración de clases, interfaces, métodos, atributos, sentencias, locales, expresiones, literales, etc...
- **Árbol de sintaxis abstracta con información semántica y elementos del lenguaje:** Representa totalmente un programa en tiempo de compilación. Es el resultado de añadir al árbol de sintaxis abstracta toda la información semántica. Es decir, asociar todas las referencias con sus definiciones, determinar el tipo de las expresiones, etc... Es un superconjunto que engloba al árbol de sintaxis abstracta y a la información estructural.
- **Intérprete**: Representa un programa Java en tiempo de ejecución. La propia API es la encargada de interpretar el programa que gestiona y generar el resultado esperado de la ejecución de dicho programa. Representa todos los elementos del programa en cada instante de la ejecución: valores, hilos, locales activas, etc...
- **Depurador**: Representa un programa Java en tiempo de ejecución. La API no interpreta el código del programa, lo que hace es controlar la ejecución del mismo usando la máquina virtual en modo depuración. Ésta comunica a la API algunos aspectos de interés relativos a la ejecución del programa. Normalmente no se puede obtener toda la información de la ejecución, sólo la más relevante. La comunicación entre la máquina virtual y la API de depuración se realiza usando técnicas nativas de memoria compartida o mediante conexiones de red entre procesos.

- **Traza**: Representa un programa Java en tiempo de ejecución. Es una API que guarda la información a lo largo de la ejecución de un programa. La diferencia con el intérprete y el depurador, es que en la traza se registra la ejecución completa (o una parte relevante de ella) y en el depurador e intérprete sólo se dispone de información instantánea evoluciona a lo largo del tiempo.

2.2.2 APIs disponibles

Algunas de las implementaciones más representativas de estas APIs son:

Información estructural

- **API Reflections en Java:** Esta API permite obtener la información estructural de las clases y objetos del propio programa que la usa. Además de obtener la información estructural se permite la metaprogramación, que consiste en la invocación de métodos y acceso al valor de los atributos que se eligen en tiempo de ejecución y no en tiempo de compilación.
- **BCEL, The Byte Code Engineering Library:** Esta API permite obtener la información estructural de las clases e interfaces representados por ficheros .class. Las clases e interfaces que se representan en estos ficheros no se cargan en el programa que usa esta API, por tanto no se pueden instanciar objetos de estas clases.
- **Javassist:** Similiar a BCEL, con la diferencia de que se puede construir un fichero . class dinámicamente con código Java en texto.

Árbol de sintaxis abstracta

- **PMD:** Modela el código fuente de un programa Java, para la búsqueda de elementos que puedan ser origen de errores, o para restringir la semántica del lenguaje. Existe en formato de plugin para muchos IDEs[7].

7 Entorno de desarrollo integrado (Integrated Development Environment)

Árbol de sintaxis abstracta con información semántica y elementos del lenguaje

- **RECODER:** Representa un programa java en tiempo de compilación completamente. Mantiene el árbol de sintaxis abstracta obtenido de los ficheros fuente, la información estructural obtenida de los ficheros .class y todo ello está relacionado semánticamente para obtener en la estructura de datos las referencias asociadas a las definiciones. Permite modificar el código y regenerar los ficheros fuente del programa Java.
- **BARAT:** Representa un programa Java en tiempo de compilación completamente. Mantiene el árbol de sintaxis abstracta de los ficheros fuente y mantiene información de los ficheros .class. No permite la modificación de la estructura de datos que representa el programa.
- **OpenJava:** Representa el árbol de sintaxis abstracta de los códigos fuente de un programa Java. Su principal objetivo es la incorporación de metainformación al código fuente que será usada para generar código fuente 100% puro Java.
- **Eclipse JDT core:** Está integrada en el entorno de desarrollo de Eclipse[8]. Representa un programa Java en tiempo de compilación completamente. Se permite modificar la estructura de datos que representa al programa. Al estar integrado con un entorno de desarrollo, se dispone de componentes textuales en los que se puede modificar el código fuente y los cambios son reflejados en la estructura de datos.

Intérprete

- **DynamicJava:** Representa un programa Java completo en tiempo de compilación, es decir, mantiene el árbol de sintaxis abstracta de los ficheros fuente. Se accede a la información de las librerías mediante las capacidades de metaprogramación del lenguaje Java. La representación de los ficheros de código fuente es similar al modelo presentado en BARAT y RECODER. El módulo que interpreta el programa está altamente desacoplado de la estructura de datos.
- **BeanShell:** Representa un programa Java completo en tiempo de compilación. No permite acceder fácilmente a dicho programa. Es equivalente en la funcionalidad proporcionada a DynamicJava.

8 Entorno de desarrollo de código abierto (http://www.eclipse.org)

Depurador

- **JPDA:** Representa la ejecución de un programa Java en una máquina virtual distinta de la que usa dicha API. Permite registrar gestores de eventos que serán notificados cuando se ejecute un método, se acceda a un atributo, etc... Permite consultar los objetos y el valor de las locales que se encuentra en la pila de ejecución de cada uno de los hilos que forman la ejecución del programa. No permite acceder a los resultados de la evaluación de subexpresiones. No permite controlar la ejecución paso a paso basado en sentencias, si no en líneas de código fuente. No representa el árbol de sintaxis abstracta de los ficheros fuentes del programa Java.
- **Eclipse JDT debug:** Es básicamente un recubrimiento de JPDA incorporado en el entorno de desarrollo Eclipse. Aunque JDT core y JDT debug están incluidos dentro del mismo entorno de desarrollo, no están integrados entre sí y sus interfaces son totalmente diferentes.

Traza

- **JVMPI – Java Virtual Machine Profiler Interface:** Permite obtener información de la traza de un programa Java en ejecución en una máquina virtual. El formato de la información es muy compacto y de muy bajo nivel, necesario para no afectar al funcionamiento del programa cuando hay que realizar medidas de tiempos sobre el mismo. Si esta información se guarda, se obtiene la traza.
- **STEP, Extensible Program Trace Encoding :** Define una serie de formatos para grabar la traza de la ejecución de un programa en Java. Este formato reduce el tamaño en disco que sería necesario si se grabase directamente la información proporcionada por JVMPI.

2.3 Conclusiones

En función del objetivo de cada una de las herramientas, éstas proporcionan unas u otras visualizaciones del software. Es necesario que una API ofrezca la posibilidad de desarrollar cualquier tipo de visualización, ya que no se sabe con certeza cuáles serán las necesidades en el futuro. Una característica presente en algunos de estas herramientas es la de proporcionar formas de navegar desde las vistas de ejecución hacia las de definición, siendo preciso establecer relaciones entre la información de los distintos aspectos. La presencia de diferentes niveles de detalle obliga a que la información tenga amplio contenido semántico.

2.4 Referencias

La Tabla 3 contiene las direcciones de Internet y las referencias bibliográficas donde puede encontrarse información sobre las herramientas y APIs.

Herramienta/API	Dirección
Java Roundtrip Engineering	http://javare.sourceforge.net/
JACOT	http://www.st.informatik.tu-darmstadt.de/pages/workshops/ASARTI03/papers.html
SeeSoft	http://dx.doi.org/10.1109/32.177365
Jinsight	http://www.alphaworks.ibm.com/aw.nsf/download/jinsight
	Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, Jeaha Yang, *Visualizing the Execution of Java Programs*, Software Visualization, May 2001. Wim De Pauw, Doug Kimelman, John Vlissides, *Modeling Object-Oriented Program Execution*, Proceedings of the 8th European Conference, ECCOP, pages 163--182, July 1994.
Omniscient Debugging	http://www.lambdacs.com/debugger/debugger.html
Jrat	http://jrat.sourceforge.net/
Evolve	http://www.sable.mcgill.ca/evolve/
	Qin Wang, Wei Wang, Rhodes Brown, Karel Driesen, Bruno Dufour, Laurie Hendren and Clark Verbrugge, *EVolve, an Open Extensible Software Visualization Framework* back, ACM Symposium on Software Visualization, June 2003

Herramienta/API	Dirección
Fujaba	http://wwwcs.upb.de/cs/fujaba/
	http://wwwcs.upb.de/cs/fujaba/publications/index.html
Javavis	Rainer Oechsle, Thomas Schmidt, *JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface*, Software Visualization, May 2001.
BlueJ	http://www.bluej.org/
	http://www.bluej.org/help/papers.html
DrJava	http://drjava.sourceforge.net/
	http://drjava.sourceforge.net/publications.shtml
Fujaba life	http://wwwcs.upb.de/cs/fujaba/projects/education/main.html (inglés)
jGRASP	http://www.jgrasp.org/
	http://www.jgrasp.org/papers.html
ProfessorJ	http://www.cs.utah.edu/~kathyg/profj/
	http://www.cs.utah.edu/~kathyg/profj/#papers
API Reflections	http://java.sun.com/docs/books/tutorial/reflect/TOC.html
BCEL	http://jakarta.apache.org/bcel/
Javassist	http://www.javassist.org
PMD	http://pmd.sourceforge.net/
RECODER	http://recoder.sourceforge.net/
BARAT	http://barat.sourceforge.net/
OpenJava	http://openjava.sourceforge.net/
Eclipse JDT core	http://www.eclipse.org/jdt/
DynamicJava	http://koala.ilog.fr/djava/
BeanShell	http://www.beanshell.org/
JPDA	http://java.sun.com/products/jpda/
Eclipse JDT debug	http://www.eclipse.org/jdt/
JVMPI	http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html
STEP	http://www.sable.mcgill.ca/step/main.html

Tabla 3. *Direcciones de Internet y referencias bibliográficas con información sobre las herramientas y APIs*

3 Metodologías y Tecnologías

En el presente capítulo se va a justificar la metodología seguida en la definición e implementación de la API javaMod, así como las tecnologías software empleadas.

3.1 Metodología

Todo desarrollo de software tiene riesgos y es difícil de controlar, por eso, si no llevamos una metodología de desarrollo es posible que el software no se ajuste a las necesidades, funcionalidad y tiempo. A la hora de afrontar el desarrollo de este Proyecto Fin de Carrera, se han evaluado las dos metodologías más usadas actualmente para elegir la que más se ajusta a al desarrollo.

3.1.1 Proceso Unificado de Desarrollo

El Proceso Unificado de Desarrollo de Software (PUD) [Jacobson00] es una metodología para acometer el desarrollo de sistemas informáticos, propuesta por Ivar Jacobson, Grady Booch y James Rumbaugh. Tiene las siguientes características:

- Se trata de un estándar avalado por OMG, Object Management Group, consorcio de alrededor de 800 miembros (compañías de la industria del software), que busca el desarrollo de especificaciones para la industria del software que sean técnicamente excelentes, comercialmente viables e independientes del vendedor. OMG define object management como el desarrollo software que modela el mundo real mediante su representación como objetos. Estos objetos no son más que la encapsulamiento de atributos, relaciones y métodos de componentes software identificables;
- Recoge la experiencia de tres grandes metodologías anteriores:
  - OMT (Object Modeling Technique) de Rumbaugh
  - OOAD (Object-Oriented Analysis and Design) de Booch
  - Objectory de Jacobson
- Actualmente se usa en la mayoría de las empresas para grandes proyectos y se imparte por muchas instituciones.

El PUD es un proceso de desarrollo de software. Un proceso de desarrollo de software es el conjunto de actividades necesarias para transformar los requisitos de un usuario en un sistema software. Sin embargo, el proceso unificado es más que un proceso de trabajo, es un marco de trabajo genérico que puede especializarse para una gran variedad de sistemas software, para diferentes áreas de aplicación, diferentes tipos de organizaciones y diferentes niveles de aptitud.

El PUD está basado en componentes y utiliza UML [Booch99]. Está dirigido por casos de uso, porque con éstos se especifican las funcionalidades que el sistema proporciona al usuario. Los casos de uso representan los requisitos funcionales y fuerzan a pensar en términos de importancia para el usuario y no sólo en términos de funciones que sería bueno tener. Los casos de uso no sólo son una herramienta para especificar los requisitos del sistema, sino que también guían su diseño, implementación y prueba, es decir, guían el desarrollo software.

El PUD está centrado en la arquitectura, pues la manera en que se organiza el sistema depende de los casos de uso clave y debe tener en cuenta la comprensibilidad, la facilidad de adaptación al cambio y la reutilización. Los casos de uso clave son aquellos que dotan al sistema con la funcionalidad fundamental para los usuarios y sin los cuales, los demás casos de uso no tienen sentido.

El PUD es iterativo e incremental. El trabajo se divide en partes más pequeñas o llamadas iteraciones. En cada iteración se recorren los flujos de trabajo (*requisitos, análisis y diseño, implementación y pruebas*) y se obtiene una versión interna. En síntesis, en cada iteración se amplía el sistema con nuevos casos de uso, se identifican nuevos riesgos y se mitigan los ya conocidos. Las iteraciones se agrupan en fases, que por orden secuencial son las siguientes: *inicio, elaboración, construcción* y *transición.* Cada una centra sus esfuerzos más en unos flujos de trabajo que en otros. La etapa de inicio se centra en la captura de requisitos y el análisis. La etapa de elaboración lo hace con el análisis y el diseño. Las etapas de construcción y transición se centran en el diseño, implementación y pruebas.

El PUD es una metodología de desarrollo pensada para proyectos grandes, a largo plazo y con un equipo de desarrollo numeroso.

3.1.2 Programación extrema

La programación extrema (XP) [Beck99] es una de las metodologías de desarrollo de software con más éxito en la actualidad. Se utiliza en proyectos con equipo de desarrollo pequeños y con plazo de entrega corto. La metodología consiste en una programación rápida o extrema. Una particularidad es tener como miembro del equipo al usuario final. Esta metodología tiene las siguientes características:

- **Pruebas Unitarias:** Las pruebas se realizan a los principales procesos sistemáticamente durante todo el desarrollo.
- **Refactorización:** El código se cambia constantemente para que sea lo más reutilizable y flexible posible. La refactorización consiste en el cambio del código para añadir más calidad al mismo pero sin cambiar la funcionalidad de la aplicación.
- **Programación en pares:** Una particularidad de esta metodología es que propone la programación en pares, la cual consiste en que dos desarrolladores participen en un proyecto en un mismo puesto de trabajo. Cada miembro lleva a cabo la acción que el otro no está haciendo en ese momento. Es como el piloto y el copiloto: mientras uno conduce, el otro consulta el mapa.

XP propone que el desarrollo comienza con dotando a la aplicación de poca funcionalidad que va siendo aumentada a medida que avanza el desarrollo con retroalimentación continua. La gestión del cambio se convierte en parte esencial del desarrollo. El coste del cambio no depende de la fase o etapa. Por último no se introducen funcionalidades antes de que sean necesarias.

XP incorpora activamente al cliente en el proceso de desarrollo. El cliente decide qué se implementa. Sabe en todo momento el estado real y el progreso del proyecto. Puede añadir, cambiar o quitar requisitos en cualquier momento. Puede obtener un sistema funcionando en 3 ó 4 meses.

XP ofrece al desarrollador el poder de decidir como se implementan los procesos y crear el sistema con la mejor calidad posible. Puede disponer de cliente en cualquier momento para que le aclare algunos requisitos. Puede estimar el esfuerzo necesario para implementar el sistema y puede cambiar los requisitos en base a nuevos descubrimientos.

Lo fundamental de XP es:

- **La comunicación:** Entre los usuarios y los desarrolladores
- **La simplicidad:** Al desarrollar y codificar los módulos del sistema
- **La retroalimentación:** Concreta y frecuente del equipo de desarrollo, el cliente y los usuarios finales

3.1.3 Elección de la metodología

Después de estudiar dos de las metodologías más usadas actualmente en el desarrollo software, se ha elegido el uso de XP. Los principales motivos de esta elección han sido:

- Equipo de desarrollo formado por una persona.
- El proyecto construye una API que será usada por programadores, por tanto es más adecuado una metodología que integre al usuario final en el desarrollo.
- Un Proyecto Fin de Carrera está destinado a una sola personal y tiene una duración estimada de 150 horas de trabajo (15 créditos). El tamaño de la aplicación construida es muy inferior a la gran mayoría de las aplicaciones comerciales.
- Dado que se trata de construir una API es muy probable que se vayan incorporando nuevos requisitos a medida que se vayan necesitando. Por tanto habrá que volver a negociar los que se implementan.
- Es importante disponer de versiones funcionando a medida que se avanza en el desarrollo para realizar pruebas de viabilidad.

La metodología XP se complementará con UML para la descripción de los distintos diagramas.

3.2 Tecnologías software

A continuación se enumeran las tecnologías software empleadas en el desarrollo de este trabajo. En los siguientes párrafos se describen dichas tecnologías desde un punto de vista general y a continuación se presentan las características más relevantes de las herramientas concretas empleadas en la realización de este proyecto de fin de carrera:

- Orientación a objetos
- Java

- Procesadores de Lenguajes
- JavaCC
- Depuradores
- JPDA

3.2.1 Orientación a objetos

La orientación a objetos es el paradigma de programación más utilizado actualmente en el desarrollo de aplicaciones. Esto se debe a la existencia de lenguajes como Java, C# y otros que aportan otras muchas características que los hacen fáciles de usar, potentes, compatibles con muchas plataformas y sobre todo seguros y fáciles de depurar.

El paradigma de programación orientada a objetos establece una asociación entre la información y las sentencias encargadas de gestionar dicha información. Desde un punto de vista de la programación procedimental, la orientación a objetos se puede ver como la asociación entre funciones y procedimientos a los tipos de datos que manejan.

En programación orientada a objetos una **clase** es el conjunto formado por la definición de la estructura de la información y las sentencias que la gestionan. A los subprogramas que gestionan los datos, se les denomina **métodos**. A cada uno de los ítems de información que definen la estructura de la información se les denomina **atributos** o **campos**. En esta forma de organizar el código el módulo mínimo de programación es la clase.

Cuando ejecutamos un programa orientado a objetos, a la entidad que tiene los valores de los ítems definidos en una clase se le denomina **objeto**.

Además de esta organización del código existen primitivas de encapsulamiento, de forma que los atributos existentes en una clase quedan ocultos a los usuarios de esa clase. Para comunicar esa información se utilizan los métodos. Al conjunto de métodos de una clase que pueden ser usados por otras clases se le denomina **interfaz pública**.

Las principales aportaciones de este paradigma al desarrollo de sistemas software son la **herencia** y el **polimorfismo**.

La herencia es una característica que permite construir una clase (conocida como **clase hija**) basándose o teniendo como plantilla otra clase construida previamente (conocida como **clase padre**). Al utilizar una clase padre como base para definir una clase hija los atributos y métodos definidos en la clase padre están disponibles en la clase hija. Es decir, los objetos de la clase hija tendrán valores para los atributos definidos en la clase hija y también tendrán valores para los atributos definidos en la clase padre. Esta característica permite una mayor reutilización del código, ya que si dos clases de un programa tienen ciertas características en común, se puede construir una clase que sea clase padre de las dos clases que alberga los atributos y métodos que sean comunes a ambas.

El polimorfismo es una característica de la programación orientada a objetos, muy relacionado con la herencia, que solventa un conjunto de problemas típicos que aparecían en la programación procedimental. En multitud de aplicaciones es necesario gestionar información con distinta estructura de una forma homogénea. Por ejemplo, en un programa de diseño gráfico hay que gestionar figuras geométricas de distintos tipos como rectángulos, círculos, triángulos, etc... En algunos casos estas figuras han de gestionarse de forma homogénea, si se quiere cambiar su color o moverlas por el área de dibujo. En otras ocasiones, cada tipo de figura tendrá que responder de una forma particular. Si se quiere calcular su área, para cada tipo de figura hay que usar una expresión distinta. Para tratarlas de forma homogénea, se crea una clase padre, llamada `Figura`. En esta clase se definen las métodos comunes a todas las figuras: cambiar el color o cambiar su posición. En cambio, el área no se puede definir de una forma común, y por tanto, cada uno de los tipos de figuras definen su propio método para el cálculo del área. En la clase padre hay que definir un método que no tiene implementación, llamado **abstracto**. La implementación de este método será establecido en cada una de las clases hijas que representan a cada tipo de figura.

Si declaramos una variable de la clase `Figura`, puede contener objetos de cada una de las clases hijas. Cuando sobre el valor de esta variable se ejecuta el método de cálculo del área, se usa el polimorfismo y se ejecuta la implementación del método que corresponde a la clase concreta de la figura. Es decir, el código que se va a ejecutar cuando llamamos a un método, depende de la clase del objeto concreto que reciba el mensaje y no de la clase de la variable en que se llame.

3.2.2 Java

El lenguaje de programación utilizado en el desarrollo de este proyecto ha sido Java [Eckel98][Bloch01] y la API javaMod permite gestionar programas en el mismo lenguaje. A continuación se presentan algunas de las características que han llevado a la elección de este lenguaje:

- **Orientado a objetos:** Java trabaja con sus datos como objetos y con interfaces a esos objetos. Soporta las características propias del paradigma de la orientación a objetos: abstracción, encapsulamiento, herencia y polimorfismo.
- **Simple:** Posee una curva de aprendizaje muy rápida. Ofrece toda la funcionalidad de un lenguaje potente, pero sin las características menos usadas y más confusas de éstos.
- **Robusto:** Java realiza verificaciones en busca de problemas tanto en tiempo de compilación como en tiempo de ejecución. La comprobación de tipos en Java ayuda a detectar errores lo antes posible, en el ciclo de desarrollo. Java obliga a la declaración explícita de los tipos de los ítems de información, reduciendo así las posibilidades de error. Maneja la memoria para eliminar las preocupaciones por parte del programador de la liberación o corrupción de la misma.
- **Portable:** La indiferencia de la arquitectura representa sólo una parte de su portabilidad. Además, Java especifica los tamaños de sus tipos de datos básicos y el comportamiento de sus operadores aritméticos, de manera que los programas son iguales en todas las plataformas. Estas dos últimas características se conocen como la Máquina Virtual Java (JVM).
- **Orientado a aplicaciones:** Dispone de muchas bibliotecas orientadas al desarrollo de múltiples aplicaciones, por ejemplo, existen librerías para gestionar bases de datos, para XML, gráficos. . .

3.2.3 Procesadores de lenguajes

En este trabajo se van gestionar los ficheros fuente que definen un programa Java. La tecnología informática que estudia cómo gestionar los códigos fuente de un lenguaje de programación son los procesadores de lenguajes [Aho90]. Dentro de los procesadores de lenguajes se encuentran los traductores, compiladores, intérpretes, etc...

Para procesar un código fuente, se realizan una serie de fases: el análisis léxico, análisis sintáctico y análisis semántico. En los siguientes párrafos describiremos someramente las funciones de cada uno de ellos:

El **análisis léxico** es el primer análisis de la secuencia de caracteres de los ficheros del código fuente. Es un proceso de abstracción y simplificación de los caracteres de esas secuencias. Sin entrar en muchos detalles formales, se puede decir que en este proceso de análisis se agrupan las secuencias de caracteres en unidades de más alto nivel, llamadas **tokens**. Suele ser habitual eliminar los saltos de línea, tratar las palabras reservadas de una forma distinta a los identificadores, eliminar los comentarios del código fuente, etc.... El analizador léxico construye tokens a partir la cadena de entrada basándose en unos patrones llamados **expresiones regulares**. Durante este análisis se pueden producir errores si los caracteres presentes en el código fuente no son válidos para el lenguaje o bien no se corresponden a ninguna expresión regular. El resultado del análisis léxico es una secuencia de tokens, que son una abstracción de la secuencia de caracteres del código fuente. Cada token tiene un tipo y en algunas ocasiones tiene asociado la secuencia de caracteres original, llamada **lexema**.

El **análisis sintáctico** tiene como entrada la secuencia de tokens obtenida en el análisis léxico. El objetivo de este análisis es determinar si la secuencia de tokens se corresponde a una serie de patrones, denominados en conjunto **gramática**. Esos patrones determinan las construcciones válidas que pueden aparecer en los códigos fuente de un lenguaje de programación. El resultado de este análisis es un árbol sintáctico, cuyas hojas son los tokens, también llamados aquí **terminales** y cuyos nodos interiores corresponden a diversos elementos del patrón llamados **no terminales**. Si en algún punto concreto se esperaba un token determinado y éste no aparece, se produce un error sintáctico. El formato de especificación de la gramática no está estandarizado, y depende de muchos factores, pero existe una forma llamada **BNF** (Bakus-Naur Form) y otra extendida llamada EBNF (Extended BNF) comúnmente aceptadas. Además, dependiendo de las características de la gramática y de la facilidad para reconocer los códigos fuente que se acogen a ellas existen gramáticas LL[9], LR[10], etc...

El árbol sintáctico no tiene por qué ser una estructura de datos representada en memoria, de hecho, es habitual que se vaya comprobando que se puede construir aunque no se construya realmente. En cualquier caso, conceptualmente el árbol existe.

9 Left-to-right leftmost derivation
10 Left-to-right rightmost derivation

El **análisis semántico** es el último proceso que determina si un código es correcto y se ajusta a las reglas de un lenguaje. Este proceso está muy poco formalizado y existen muchas técnicas para llevarlo a cabo. En él se realizan los análisis de referencias, es decir, si un elemento al que se referencia está declarado, análisis de tipos, etc. Por último se analizan algunas características más particulares de cada lenguaje, por ejemplo, las variables que se referencian deben haberse inicializado previamente, a las constantes no se les puede asignar un valor distinto del inicial, etc... Algunas de estas comprobaciones se suelen realizar durante el proceso de análisis sintáctico y se denominan **acciones semánticas**.

La construcción de procesadores de lenguaje es una tarea que no suele realizarse manualmente. Habitualmente se definen las expresiones regulares del análisis léxico, la gramáticas del análisis sintáctico y las acciones semánticas del análisis semántico y es una herramienta, la que genera automáticamente el procesador del lenguaje o compilador. Posteriormente, este compilador será usado para procesar las ficheros fuente con los códigos del programa. A las herramientas que generan compiladores se las denomina **compiladores de compiladores** (compiler compiler) o **generadores de analizadores** (parser generators). Cada compilador de compiladores construye el compilador en un lenguaje de programación determinado y usando una técnica **ascendente** o **descendente**.

Existen multitud de generadores de analizadores, dependiendo del lenguaje de programación en el que se genera el compilador, las características que debe cumplir la gramática, las diversas facilidades que proporciona, etc...

3.2.4 JavaCC

En la implementación de la API javaMod se necesita gestionar códigos fuente del lenguaje Java. Para ello, se ha optado por utilizar un generador de analizadores para generar el analizador. El generador de analizadores elegido ha sido JavaCC [JavaCC]. JavaCC es una herramienta de código abierto bajo licencia BSD[11]. Sus principales características son:

- **Descendente:** Genera procesadores descendentes a diferencia de los procesadores ascendentes generados por YACC[12].

11 Berkley System Designs. Una de las muchas licencias de código abierto.
12 Yet Another Compiler-Compiler. Generador de compiladores típico.

- **Ampliamente usado por la comunidad:** Es de lejos el generador de analizadores usado en las aplicaciones Java más popular. En su web se indica que ha sido descargado cientos de miles de veces.
- **Especificación léxica y sintáctica en un fichero:** La especificación léxica (expresiones regulares) y la especificación sintáctica (gramática EBNF) está escrita junta en el mismo fichero. Esto hace la gramática más fácil de leer y más fácil de mantener.
- **Personalizable:** Ofrece muchas diferentes opciones para personalizar su comportamiento y el comportamiento de los procesadores que genera.
- **Generador de documentación:** Incluye una herramienta llamado JJDoc que convierta los ficheros de especificación del lenguaje en ficheros de documentación (opcionalmente en HTML).
- **Muchos ejemplos:** Incluye un amplio rango de ejemplos incluyendo gramáticas de Java y HTML. Los ejemplos y la documentación son una buena forma de aprender a usar la herramienta.
- **Internacionalizado:** El analizador léxico puede manejar una entrada en Unicode.
- **Especificaciones "lookahead" sintácticas y semánticas:** Por defecto, genera un analizador LL(1). En cualquier caso, puede haber porciones de la gramática que no sean LL(1). JavaCC ofrece la capacidad de un determinar la producción por la que seguir analizando de forma sintáctica o semántica. Esto permite resolver las ambigüedades localmente en los puntos donde se produzcan.
- **Permite especificaciones en BNF extendida:** Permite especificar la gramática en formato BNF extendido, es decir, especificaciones como `(A)*`, `(A)+`, etc.
- **Capacidades de depuración:** Usando algunas opciones se puede obtener mucha información del proceso de análisis léxico y sintáctico.
- **Informe de errores:** El informe de errores que proporciona muestra claramente el punto en el que se ha producido el error con información completa de diagnóstico.

3.2.5 Depuradores

Los depuradores son herramientas que permiten controlar la ejecución de un programa con el objetivo habitual de encontrar errores. La característica principal de este tipo de herramientas consiste en poder establecer puntos de ruptura en ciertas partes del código. Cuando se llegue a ejecutar esa sentencia en el proceso de ejecución, ésta se parará o suspenderá. Cuando la ejecución se encuentre suspendida, se podrán observar los valores que en ese instante tienen las variables, los subprogramas que se encuentran en la pila de ejecución, etc... Se podrá reanudar la ejecución del programa cuando el usuario lo considere oportuno, pudiendo ejecutar hasta el siguiente punto de ruptura o bien ir ejecutando sentencia a sentencia, es decir, paso a paso.

Habitualmente, los depuradores se encuentran en entornos de desarrollo integrado (IDE) para que puedan corregirse rápidamente los errores encontrados en el proceso de depuración.

La construcción de los depuradores está íntimamente relacionada con el lenguaje de programación empleado y la forma en que se ejecutan sus programas. Por ejemplo, en lenguajes compilados en código máquina las técnicas de depuración se suelen basar en facilidades proporcionadas por los propios procesadores para detener la ejecución. También suelen usarse otras técnicas como **instrumentación del código**, es decir, modificación automática del código para que sea semánticamente equivalente pero que comunique cierta información a la herramienta de depuración.

En las tecnologías de programación en las que los programas se ejecutan dentro de una máquina virtual, como Java, C#, etc... las técnicas de depuración en algunos casos son diferentes. En el caso de Java, la máquina virtual tiene un modo de ejecución especial, el modo de depuración. En este modo es posible comunicarse con la máquina virtual para consultar toda la información de la ejecución del programa, así como controlar la ejecución estableciendo puntos de ruptura, etc...

3.2.6 JPDA

JPDA (Java Platform Debugger Architecture) [JPDA] proporciona la infraestructura necesaria para construir herramientas de depuración para la plataforma Java. Incluye las siguientes APIs definidas en tres capas:

- **Java Debug Interface (JDI):** Interfaz de programación de alto nivel, que incluye soporte para depuración remota.

- **Java Debug Wire Protocol (JDWP):** Define el formato de la información y las peticiones transferidas entre la herramienta de depuración y la máquina virtual.
- **Java Virtual Machine Debug Interface (JVMDI):** API nativa de bajo nivel que define los servicios que la máquina virtual debe proporcionar para depuración.

En la realización de este trabajo, se ha usado la API JDI para implementar los servicios de depuración de javaMod. En los siguientes puntos se describen sus principales características:

- Es una API de alto nivel definida en Java.
- Proporciona acceso al estado de la máquina virtual que ejecuta el programa que se quiere depurar. Permite acceder a las clases, arrays, interfaces, tipos primitivos, e instancias de estos tipos.
- Proporciona control explícito sobre la ejecución de la máquina virtual: la capacidad de suspender y reanudar los hilos de ejecución, establecer puntos de ruptura, etc...
- Notifica las excepciones producidas, la carga de las clases, la creación de hilos, etc...
- Proporciona la capacidad de inspeccionar el estado de los hilos de ejecución, las variables locales, los marcos de la pila, etc...

La API JDI se encuentra implementada en el fichero `tools.jar` que se incorpora en las distribuciones del SDK de java de Sun Microsystems.

4 Especificación de requisitos

Durante el flujo de trabajo de los requisitos se identifican las necesidades de los usuarios como requisitos. Se le llama captura de requisitos al proceso de averiguar lo que se debe construir. Los desarrolladores de software profesionales habitualmente no crean código para sí mismos, sino para usuarios del software u otros desarrolladores. Los usuarios con frecuencia no saben cuáles son los requisitos ni tampoco cómo especificarlos de una forma precisa.

En este proyecto se han estudiado algunas de las herramientas más representativas de gestión de software, algunas de ellas están enfocadas a la visualización del software y otras son herramientas educativas. No existen clientes o usuarios con los que reunirse para la captura de requisitos, si no que los requisitos se obtuvieron partiendo de dichas herramientas.

En cuanto a los requisitos a considerar se puede distinguir a grandes rasgos entre: *funcionales*, aquellos que especifican las acciones que debe ser capaz de realizar el sistema; y *no funcionales*, aquellos que especifican propiedades del sistema.

En este capítulo se presentan aquellos requisitos que debería cumplir una API que pueda usarse para construir cualquiera de las herramientas estudiadas. Estos requisitos se han obtenido después de realizar un estudio exhaustivo de cada una de las herramientas. Debido a que la implementación de una API que los cumpla todos supondría un gran esfuerzo que se saldría del alcance de un único Proyecto Fin de Carrera no se han implementado todos ellos. No obstante, se definen en esta memoria por dos razones. En primer lugar, el conjunto total de requisitos se ha tenido en cuenta al definir la arquitectura de la API, de forma que su posterior ampliación para incorporarlos no sea costosa. En segundo lugar, como indica el proceso de desarrollo de programación extrema, durante el desarrollo de la API han ido apareciendo nuevos requisitos sobre nuevas necesidades, por lo que aquí se muestran tanto los iniciales como los que han ido apareciendo.

En los siguientes apartados se muestran los requisitos en un formato inspirado en el estándar IEEE 830-1998.

4.1 Descripción general

4.1.1 Perspectiva del producto

Interfaces del sistema

La definición de la API es totalmente independiente a cualquier otra parte del sistema. No obstante, las implementaciones posiblemente dependan de otras partes del sistema para su correcto funcionamiento.

Interfaces de usuario

La API debe proporcionar un componente de texto para la edición de códigos Java, este es el único requisito en el que se menciona el interfaz de usuario. Este componente textual debe estar definido usando el estándar para la construcción de interfaces gráficos en Java, Swing.

Interfaces hardware

La API se definirá usando la tecnología Java. No se consideran requisitos hardware adicionales a los que impone la propia tecnología Java. No obstante, las diferentes implementaciones de la API pueden considerar como requisito la existencia de cierto hardware.

Interfaces software

La API se definirá usando la tecnología Java. No se consideran requisitos software adicionales a los que impone la propia tecnología Java. No obstante, las diferentes implementaciones de la API pueden considerar como requisito la existencia de cierto software.

4.1.2 Funciones del producto

Las principales funciones de la API son las de gestionar en todos sus aspectos un programa Java, con diferentes niveles de detalle.

4.1.3 Características del usuario

Los usuarios que usen la API deberán conocer los principios básicos de construcción de procesadores de lenguajes y la especificación del lenguaje de programación Java.

4.2 Requisitos específicos

4.2.1 Interfaces Externos

La salida de la API serán programas escritos en el lenguaje Java en un formato ejecutable. No es un requisito de la librería, pero es razonable que las implementaciones de la API permitan acceder a los ficheros fuente en disco y a las librerías, en un formato estándar definido en la especificación de Java.

4.2.2 Requisitos funcionales

Los requisitos funcionales de esta API se pueden dividir en varios apartados dependiendo del aspecto del programa Java que se necesite manejar. En primer lugar, se especificarán aquellos requisitos relativos a la gestión de los elementos que forman la definición del programa. Por otro lado se especificarán todos los requisitos relativos al control de la ejecución del mismo y al registro de su traza.

Requisitos para gestionar la definición de un programa Java

RAProporcionar mecanismos para gestionar la definición de programas Java

RA.1:Manejar programas para el lenguaje Java formados por ficheros fuente y librerías.

RA.2:Proporcionar mecanismos para acceder a cada una de las partes que forman un fichero fuente del lenguaje Java.

RA.2.1:Representar cada una de las posibles partes que pueden formar un fichero fuente con interfaces, de forma que se pueda acceder a un fichero fuente como una estructura de objetos relacionados entre sí.

RA.2.2:Proporcionar el acceso a las partes y características de cada objeto que representa el código fuente, con métodos que sigan la forma y semántica habitual de los programas orientados a objetos.

RA.2.3:Permitir conocer la posición relativa de un elemento del código fuente en su contexto, por ejemplo, poder determinar en qué método se encuentra una sentencia, etc...

RA.2.4:Proporcionar mecanismos para tratar un fichero fuente como un árbol de sintaxis abstracta.

RA.2.5:Proporcionar una forma sencilla y rápida de obtener una representación textual de un objeto del código, que sea semánticamente equivalente a la parte del código fuente original del que se creó ese objeto.

RA.2.6:Proporcionar mecanismos que permitan obtener representaciones textuales de los ficheros fuente de una forma personalizada.

RA.2.7:Proporcionar mecanismos para asociar información a cualquier objeto del código fuente y poder recuperarla fácilmente cuando sea necesario.

RA.2.8:Proporcionar una forma de recorrer los objetos del árbol de sintaxis abstracta. Durante el recorrido de este árbol, se deben poder definir acciones que serán ejecutadas para cada uno de los objetos que lo forman. Se debe proporcionar una forma sencilla de especificar las sentencias que se ejecutarán, dependiendo de la clase del objeto.

RA.3:Ofrecer mecanismos para modificar la estructura de datos que representa el código fuente, desde métodos que se proporcionen en la API.

RA.3.1:Proporcionar mecanismos para comprobar los posibles errores que existan en el conjunto de ficheros fuente que forman el programa.

RA.3.2:Incorporar el patrón observador, para que se notifiquen eventos ante un cambio en la estructura de objetos que representa un código fuente. Estas notificaciones podrán ser usadas para actualizar las posibles visualizaciones.

RA.3.3:Proporcionar un componente textual en el que se puedan modificar los ficheros fuente del programa.

RA.3.3.1:Modificar la estructura de objetos que representa el código fuente cuando se produzcan modificaciones en el componente de texto.

RA.3.3.2:Reflejar en el componente de texto, si éste se utiliza, las modificaciones que se realicen en la estructura de objetos usando los métodos proporcionados en la API.

RA.4:Representar y gestionar la información de los ficheros fuente y de las librerías, de una forma homogénea y ortogonal, independientemente de donde se encuentre. La implementación de los métodos de las clases que estén en librerías no estará disponible en la API.

RA.5:Proporcionar una visión conjunta de un programa Java, formada por sus ficheros fuente y librerías, de forma que se reflejen las relaciones que aparecen entre ellos.

RA.5.1:Proporcionar una forma de acceder a un programa desde un punto de vista físico, es decir, poder gestionar la ubicación en disco de los ficheros fuente y las librerías que lo forman.

RA.5.2:Proporcionar una forma de acceder a un programa desde un punto de vista lógico, es decir, poder gestionar el conjunto de paquetes, con las clases y subpaquetes que los forman.

RA.5.3:Proporcionar mecanismos para determinar los ítems de información activos cuando la ejecución se encuentre en un punto determinado, es decir, se deberán resolver las reglas de ámbito para saber qué ítems de información son accesibles desde un punto del código determinado.

RA.5.4:Proporcionar, de una forma sencilla, relaciones entre elementos de la definición del lenguaje que puedan ser de interés para las visualizaciones.

RA.5.4.1:Proporcionar un mecanismo para acceder a los elementos del programa a los que se hace referencia desde otros elementos. Por ejemplo, desde una referencia a una variable debe ser posible acceder al objeto del código fuente que representa la definición de esa variable.

RA.5.4.2:Permitir determinar las referencias que se hacen a cualquier definición.

RA.5.4.3:Proporcionar un árbol con las llamadas que se realizan a un método determinado o a cualquiera de los métodos que redefine, conocido como árbol de llamadas.

RA.5.4.4:Proporcionar el árbol de herencia del que es padre una clase determinada.

RA.5.4.5:Proporcionar la lista de métodos que redefinen un método dado.

RA.6:Permitir la gestión de los elementos del lenguaje Java que no aparecen en los ficheros de código fuente pero representan conceptos del lenguaje.

RA.6.1:Permitir la gestión de los tipos predefinidos del lenguaje, de forma similar a cómo se gestionan las clases que se definen en código o librerías. Por ejemplo, debe representar los tipos primitivos, el tipo `null`, etc...

RA.6.2:Ofrecer la forma de gestionar el concepto de paquete, que no se define en los ficheros fuente, pero sí se hacen referencias al mismo.

RA.6.3:Proporcionar mecanismos para gestionar las operaciones que son referenciadas usando operadores. Esta gestión deberá ser similar a la forma en que se gestionan los métodos de las clases.

RA.7:Proporcionar mecanismos que permitan realizar consultas al estilo SQL sobre la definición del programa Java.

Requisitos para gestionar la ejecución de un programa Java y su traza

RBProporcionar mecanismos para ejecutar el programa Java.

RB.1:Proporcionar mecanismos para iniciar la ejecución del programa. Se indicará la clase que tiene el método `main` y los parámetros que recibirá la aplicación.

RB.2:Representar cada ejecución del programa por un objeto que permita gestionar esa ejecución.

RB.3:Proporcionar mecanismos para ejecutar un programa Java como se ejecutaría si se invocara desde el sistema operativo.

RB.4:Proporcionar mecanismos para ejecutar un programa Java en modo depuración.

RB.5:Proporcionar mecanismos para comenzar un sistema de ejecución que no inicie la ejecución del método `main` de ninguna clase. Este modo de ejecución estará basado en la idea de intérprete de comandos y estará destinado a la ejecución de sentencias que no están definidas, en principio, en el código fuente y se definen una vez comenzada la ejecución.

RB.6:Proporcionar mecanismos para asociar información a los objetos en la ejecución de un programa y así, poder recuperarla fácilmente cuando sea necesario. Este mecanismo debe estar definido de la misma forma que en el requisito RA.2.7.

RB.7:Controlar detalladamente la ejecución, tanto en modo ejecución de depuración como en modo de intérprete de comandos.

RB.7.1:Proporcionar mecanismos para asociar puntos de ruptura a los elementos del código fuente, de forma que si algún hilo ejecuta ese elemento, su ejecución se suspenderá.

RB.7.2:Proporcionar mecanismos para gestionar cada uno de los hilos de ejecución del programa de forma independiente.

RB.7.3:Suspender y reanudar la ejecución de un hilo de ejecución utilizando métodos en la API.

RB.7.3.1:Indicar la sentencia en la que se detendrá de nuevo la ejecución (al reanudarla): hasta el siguiente punto de ruptura, la siguiente sentencia, la siguiente unidad de ejecución (expresión o sentencia), la primera sentencia de un método o la siguiente sentencia después de la llamada al método o constructor en ejecución.

RB.7.3.2:Proporcionar mecanismos para que pueda realizarse una ejecución paso a paso de forma automática, en la que se configure el tiempo entre cada paso y las características del mismo.

RB.7.4:Determinar el estado de ejecución de un hilo: en espera, en ejecución, en modo zombie, etc...

RB.7.5:Consultar toda la información que caracteriza a la ejecución de un hilo, si éste se encuentra suspendido.

RB.7.5.1:Acceder a la pila de ese hilo, que estará representada como una pila de objetos, los cuales representan la ejecución de los métodos o constructores.

RB.7.5.2:Acceder al elemento de la definición del programa que se está ejecutando, de cada una de las ejecuciones de la pila.

RB.7.5.3:Acceder a la sentencia que se encuentra en ejecución, de cada una de las ejecuciones de la pila.

RB.7.5.4:Acceder a las locales y parámetros activos, en cada una de las ejecuciones de la pila.

RB.7.5.5:Acceder a la definición de cada ítem de información activo.

RB.7.5.6:Acceder al valor de cada ítem de información activo.

RB.7.5.7:Representar todos los valores que puede contener una variable, es decir, los valores de tipos primitivos, el valor `null` o referencias a los objetos, que pueden ser arrays o instancias de clases.

RB.7.5.8:Proporcionar el valor devuelto por las subexpresiones que se vayan ejecutando.

RB.7.5.9:Proporcionar mecanismos que permitan realizar consultas al estilo SQL sobre el estado actual de la ejecución del programa Java.

RB.7.6:Proporcionar mecanismos para interaccionar con los objetos y clases del programa en ejecución.

RB.7.6.1:Proporcionar mecanismos para gestionar los valores de los atributos estáticos de las clases.

RB.7.6.2:Consultar y modificar el valor de los atributos de un objeto.

RB.7.6.3:Invocar métodos en un objeto.

RB.7.6.4:Invocar métodos estáticos de una clase.

RB.7.6.5:Ejecutar dinámicamente sentencias correctas en el lenguaje Java, que hagan referencia a variables activas en una ejecución determinada dentro de la pila.

RB.7.6.6:Modificar el código de un método, constructor o inicialización, siempre que este método no esté ejecutándose.

RB.7.7:Registrar gestores de eventos en diversos elementos del programa, de forma que cuando se produzcan esos eventos en la ejecución, se notifique a los gestores.

RB.7.7.1:Determinar si al producirse un evento se suspende la ejecución del hilo en el que se ha producido o todos los hilos de la ejecución o ninguno.

RB.7.7.2:Notificar cuando se cambie el valor de un atributo cualquiera o de uno en concreto.

RB.7.7.3:Notificar cuando se ejecute un método o constructor cualquiera o un método o constructor concreto.

RB.8:Proporcionar mecanismos que permitan gestionar la entrada estándar y las salidas estándar y de errores, de una ejecución concreta del programa.

RB.8.1:Proporcionar un acceso de bajo nivel a la entrada y salidas en formato binario.

RB.8.2:Proporcionar un acceso de alto nivel a la entrada y salidas estándar en formato textual.

RB.9:Proporcionar mecanismos para crear una traza o registro con los eventos que se produzcan en una ejecución.

RB.9.1:Permitir definir qué eventos han de ser registrados de todos los que se produzcan.

RB.9.2:Proporcionar un mecanismo de ordenación temporal de eventos.

RB.9.3:Obtener mediciones realistas del momento en que se ha producido un evento.

RB.9.4:Registrar cada uno de los valores que van tomando cada una de las variables que se activan durante la ejecución.

RB.9.5:Registrar las sentencias que se han ejecutado.

RB.9.6:Conocer el valor de las subexpresiones y expresiones que se han ejecutado.

RB.9.7:Conocer los métodos y constructores que se han ejecutado y los parámetros con los que han sido llamados.

RB.9.8:Acceder a una estructura de objetos que muestre la evolución que ha seguido la pila de cada uno de los hilos de la ejecución.

RB.9.9:Conocer el ciclo de vida de cada objeto, cómo han ido cambiando los valores de sus atributos y los métodos que han sido invocados sobre él.

RB.9.10:Proporcionar mecanismos que permitan realizar consultas al estilo SQL sobre la traza de la ejecución del programa Java.

RB.9.11:Permitir realizar una ejecución simulada con la información que ha sido registrada en una ejecución anterior.

RB.9.11.1:Simular la ejecución paso a paso permitiendo controlarla con los mismos mecanismos que pueden darse en el control de la ejecución real.

RB.9.11.2:Notificar a los gestores de eventos cuando se simula la ejecución de igual forma a cuando se ejecuta realmente.

RB.9.11.3:Proporcionar los mismos mecanismos para acceder a la información que guarda la pila de ejecución de cada uno de los hilos, excepto para modificarla.

RB.9.11.4:Permitir avanzar hacia atrás en la ejecución simulada con opciones similares a como se avanza en la ejecución paso a paso.

RB.9.11.5:Permitir establecer el estado de la ejecución simulada en el momento en que se produjo un evento determinado. De forma que toda la información que caracteriza a la ejecución tenga los valores que se tenían en ese instante.

RB.10:Proporcionar un modo de ejecución que sea a la vez de depuración y de traza, de forma que a medida que se va ejecutando paso a paso se vaya registrando la traza.

RB.10.1:Proporcionar al usuario una forma de moverse hacia atrás en la ejecución, de manera que se realice una ejecución simulada con la información de la traza.

RB.10.2:Proporcionar mecanismos para avanzar paso a paso en la ejecución simulada, hasta que se llegue al instante en el que no hay más traza y tenga que realizarse una ejecución real.

RB.10.3:Registrar los eventos de forma transparente cuando se usen únicamente las posibilidades de control de la ejecución ofrecidas por la ejecución paso a paso hacia adelante.

RB.10.4:Registrar en la traza las modificaciones de los valores y las invocaciones de métodos realizadas desde la API, pero deberán ser identificadas como tales.

4.2.3 Requisitos no funcionales

Una vez especificados los requisitos funcionales del sistema, se especifican los no funcionales:

REQNF1 Fiabilidad: No existen atributos del sistema para este apartado.

REQNF2 Disponibilidad: No existen atributos del sistema para este apartado.

REQNF3 Mantenibilidad: La API debe ser suficientemente flexible para que se pueda mantener y mejorar de forma sencilla.

REQNF4 Portabilidad: Esta API debe ser portable a cualquier plataforma que soporte la tecnología Java.

REQNF5 Tiempo de respuesta: No existen atributos del sistema para este apartado.

REQNF6 Utilización de memoria: No existen atributos del sistema para este apartado.

4.3 Conclusiones

En este Proyecto Fin de Carrera se definirá la API con los más prioritarios y se desarrollará una implementación por defecto de los mismos. La ampliación de la API de forma que incorpore todos los requisitos y el desarrollo de implementaciones para ellos será llevada a cabo en posteriores Proyectos Fin de Carrera dentro de una línea de investigación del grupo ViDo.

Los requisitos que van a ser usados para definir la API, y para crear una implementación por defecto, son los requisitos RA y RB, pero con algunas excepciones: RA.3, RA.6.4, RB.3, RB.5, RB.7.1, RB.7.3.2, RB.7.5.8, RB.7.5.9, RB.7.6.5, RB.7.6.6, RB.8.2, RB.9, RB.10. La elección de los requisitos que se van a implementar se ha realizado atendiendo a los siguientes criterios:

- Se han elegido los más útiles para construir sistemas de visualización del software.

- Se han elegido los más importantes para determinar la arquitectura del sistema. Es decir, se han elegido aquellos que van a determinar la arquitectura del sistema y se no se han considerado aquellos requisitos que no van a influir en la arquitectura y su diseño e implementación se basa en la arquitectura existente.
- Se han ido eligiendo los requisitos que permiten una integración entre los diversos aspectos de la API, ya que la integración es uno de los objetivos de la misma.

5 Diseño de la API javaMod

En este capítulo se van a describir los diferentes aspectos que se han tenido en cuenta para diseñar la API javaMod. Hay que recordar que no se van a definir ninguna implementación de esta API, sino que simplemente se van a determinar las interfaces que proporciona, para obtener los servicios de la misma. En el siguiente capítulo se describirá una posible implementación para esta API.

En primer lugar se van a especificar las partes de la API que permiten manejar la definición de un programa Java y en segundo lugar, se especificará la parte de la API que permite gestionar una ejecución del mismo.

5.1 Definición de un programa Java

Los requisitos que han guiado el diseño de esta parte de la API están englobados en el requisito RA. Los programas Java van a estar representados como instancias del interfaz `JavaProgramCode`, y cada una de las propiedades que caracterízan a los mismos serán representadas como métodos de ese interfaz.

5.1.1 Elementos principales de un programa Java

Desde un punto de vista físico, un programa Java está formado por librerías y ficheros fuente y atendiendo al requisito RA.5.1 la API debe reflejar estos elementos. Desde un punto de vista lógico, un programa Java está formado por un conjunto de paquetes y subpaquetes. Teniendo en cuenta el requisito RA.5.2 la API debe proporcionar un mecanismo para acceder a todos estos paquetes de forma conjunta independientemente de si se encuentran en ficheros fuente o en cualquier librería. Para cumplir con estos requisitos, se han definido los interfaces `Directory`, `Source`, `LibraryRepository`, `Library`, `JarLibrary`, `ClassDirectoryLibrary`, `Package`, `ClassInterface`, `Class` e `Interface`, los cuales representan los elementos que forman un programa Java. En la Figura 1 se puede ver la relación existente entre estas interfaces.

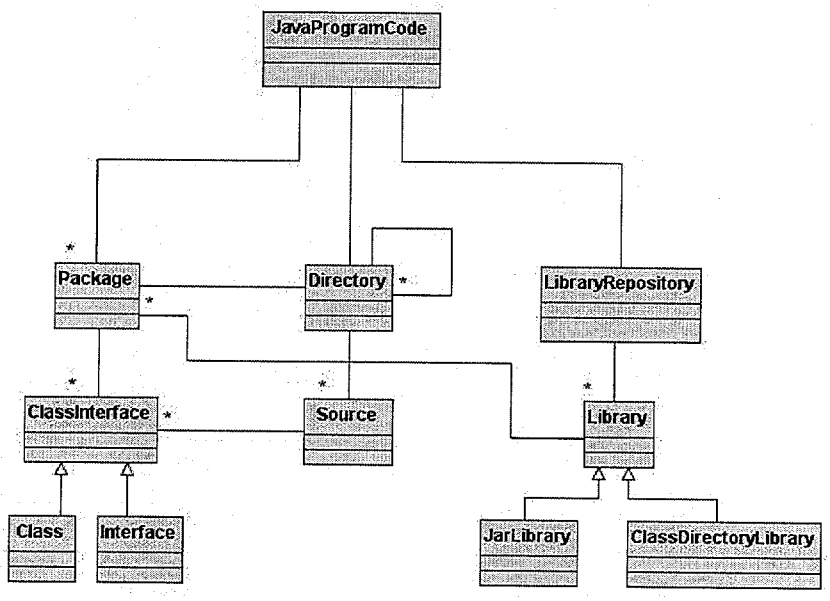

Figura 1. El programa java y sus componentes principales

5.1.2 Ficheros fuente

El requisito RA.2 implica que la API debe permitir el acceso detallado a cada una de las partes que componen un fichero fuente. Además, por el requisito RA.2.1, cada una de las partes que puedan formar un fichero fuente, han de estar representadas como una interfaz distinta. Por ejemplo, para representar una sentencia de asignación, existirá un objeto de la interfaz `AssignmentExpression`, para representar un literal de tipo `java.lang.String`, existirá un objeto de la interfaz `LiteralString`, para representar un *import* dentro de un fichero, existirá un objeto de la interfaz `Import`, etc... Por tanto, hay que identificar las diferentes partes que puede tener un fichero fuente y las relaciones que aparecen entre cada una de esas partes. Por ejemplo, es importante que la API refleje la relación entre un método, que podría estar representado por un objeto de la interfaz `Method` y sus parámetros, que podrían estar representados por una instancia de la interfaz `Parameters`. Por el requisito RA.2.2, la API debe proporcionar el acceso a las partes y características de cada objeto que representa el código fuente, con métodos que sigan la forma y semántica habitual de los programas orientados a objetos. Para cumplir con este requisito, la interfaz `Method` puede disponer del método `Parameters getParameters()` para acceder a los parámetros de dicho método.

Las interfaces que representen los elementos del código fuente deben seguir los principios de la orientación a objetos, por tanto, se debe hacer un uso adecuado de la herencia. Por ejemplo, `AssignmentExpression`, que representa a las sentencias de asignación, y `MethodRef`, que representa las llamadas a métodos, deben tener una interfaz padre común `Sentence`, que representaría a cualquier sentencia. Las interfaces padre servirán para contener aquellas propiedades comunes a todos sus interfaces hijos. Además, para especificar el cuerpo de una sentencia *for*, representada por `ForSent`, se usará `Sentence`, ya que cualquier sentencia puede ser cuerpo de un *for*.

Al observar estas primeras ideas generales, se puede ver que es necesario un profundo estudio de lenguaje Java para decidir cómo se representan, mediante interfaces, cada uno de los posibles elementos de un código fuente Java.

Como cualquier diseño, siempre estará sujeto a decisiones que se tienen que tomar en base a algunos criterios. El criterio seguido es que la estructura de objetos que representa al fichero fuente, debe mostrar lo que conceptualmente un programador tiene en mente cuando programa y además tiene que ser factible construir representaciones de esos objetos en su forma de texto convencional, según lo indicado en los requisitos RA.2.5 y RA.2.6.

Diseño orientado a objetos basado en la gramática

La gramática del lenguaje de programación estará íntimamente relacionada con las interfaces que representan los elementos de un fichero fuente. Por tanto, será útil ver algunos mecanismos para obtener las interfaces partiendo de las producciones de una gramática. En este apartado se establecen una serie de ideas generales, válidas para cualquier lenguaje, que permiten obtener interfaces de la gramática, las cuales siguen los principios orientados a objetos mencionados en el apartado anterior.

El punto de partida es crear una interfaz padre a todas las que representan elementos que aparecen en el código fuente, se denomina `CodeElement` y albergará aquellos servicios comunes a todos los elementos. También es importante que la API sea fácil de integrar en cualquier aplicación, por este motivo, todas las interfaces tendrán como paquete `es.urjc.escet.vido.javamod`, aunque es posible que existan subpaquetes para organizar todos los elementos de la API.

Obtención de propiedades desde los no terminales

El primer paso a la hora de definir las interfaces, partiendo de las producciones de la gramática, es crear una por producción y cada uno de los no terminales de la parte derecha, definirlos como propiedades. Para definir propiedades de lectura se usan métodos de acceso, es decir, un método cuyo nombre comienza con `get` y va seguido del nombre de la propiedad, o `is` seguido del nombre de la propiedad si ésta es booleana. Por ejemplo, partiendo de la producción de la sentencia *while*:

```
WHILE_SENT ::= "while" "(" EXPRESSION ")" SENTENCE
```

Obtenemos `WhileSent`, con dos métodos de acceso, `Expression getExpression()` y `Sentence getBody()`, como se puede ver en la Figura 2.

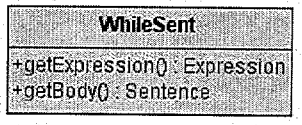

Figura 2. *Interfaz de la API en UML para representar la sentencia* while

A la propiedad que alberga el cuerpo de la sentencia *while*, se la ha llamado `Body` porque es un nombre intuitivo y representa mejor que `Sentente` lo que el programador tiene en mente cuando programa. Otro detalle importante es que la palabra reservada `while` no se refleja en ningún método de `WhileSent`, ya que es una palabra reservada fija, que siempre aparece en las sentencias *while* y por tanto puede ser regenerada cuando vaya a obtenerse una representación de texto de la sentencia.

Obtención de propiedades desde los terminales

No todas las propiedades de las interfaces se obtienen de los no terminales de la gramática. Existen una serie de propiedades que se obtienen de los terminales, por ejemplo, en la producción de los parámetros:

```
PARAMETER ::= [ "final" ] <RefItemType> <ID>
```

Existe un terminal `<ID>` y un terminal `"final"`. El terminal `<ID>` será representado como una una propiedad de la clase `java.lang.String`, llamada `Name`, en la interfaz `Parameter`. De forma análoga, también dispondrá de una propiedad llamada `Final` de tipo `boolean`, que será reflejada como el método `boolean isFinal()`. `Parameter` está representado en UML en la Figura 3.

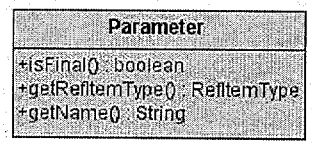

Figura 3. *Interfaz de la API para representar un parámetro*

Creación de las jerarquías de herencia

La jerarquía de herencia entre las interfaces se ve reflejada también en la propia gramática del lenguaje. Por ejemplo, en el caso de las sentencias, en la gramática se encuentra la siguiente producción:

```
   SENTENCE::= BLOCK | LABELED_SENT | EMPTY_SENT | EXPRESSION_SENT |
FLOW_CONTROL_SENT
```

Esto se materializará en la API como una jerarquía de herencia en la que `Sentence` es padre de `Block`, `LabeledSent`, `EmptySent`, `ExpressionSent` y `FlowControlSent`. La jerarquía de herencia de `Sentence` se puede ver en la Figura 4:

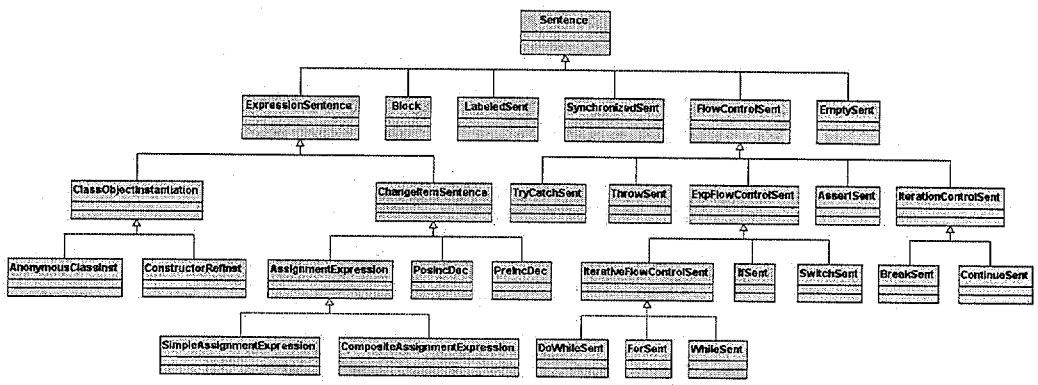

Figura 4. *Jerarquía de herencia de las sentencias*

Listas de elementos

En un principio, se podría pensar que se puede hacer una traducción directa desde la gramática para la obtención de las interfaces, no obstante, existen varios conceptos que aparecen en el lenguaje, que son *ocultados* por la gramática entre varios no terminales y producciones. Por ejemplo, las listas de elementos se ven reflejadas en la gramática dependiendo de las siguientes características:

- Si se necesita símbolo de separación o no entre los elementos.
- Si aparece un símbolo por cada elemento o sólo de separación entre ellos.
- Si existen elementos de la lista que requieren un símbolo de separación y otros elementos que no requieren ese símbolo, dependiendo de la naturaleza de ese elemento en concreto.
- Si el número mínimo de elementos es 0 o 1.

Estas son las características sintácticas que aparecen en la mayoría de las listas de los lenguajes de programación y se representan con las siguientes producciones gramaticales:

- Gramática para listas de elementos con carácter de separación, sin carácter al final, con número mínimo 0 elementos:

```
LISTA_1 ::= [ ELEMENTO (SEPARACION ELEMENTO)* ]
```

- Gramática para listas de elementos con carácter de separación, con carácter al final, con número mínimo de elementos 1 y en la que cada elemento determina si necesita o no el carácter de separación:

```
LISTA_2 ::= (ELEMENTO SEPARACION)+
ELEMENTO => ELEMENTO_A SEPARACION
                        | ELEMENTO_B SEPARACION
                        | ELEMENTO_C
                        | ELEMENTO_D
```

Como se puede apreciar, siempre que aparece el `ELEMENTO_A` en la secuencia aparece con el carácter de separación, pero el `ELEMENTO_C` no requiere de este carácter. Este tipo de situaciones se suele dar con las sentencias, donde algunas como la asignación y la llamada a un método, necesitan el símbolo se separación *";"* pero otras como la sentencia *if* o *while* no requieren de símbolo de separación si su cuerpo es un bloque de sentencias. Un concepto de alto nivel como son las listas, vendrá determinado por diversas construcciones sintácticas. Por tanto, la labor del diseñador será descubrir en la gramática donde se producen y describirlos como una lista de elementos.

En la API, las listas de elementos serán definidas como una interfaz llamada `CodeList`, la cual hereda de `java.util.List` y de `CodeElement`, ya que las listas del código fuente son otro elemento más.

Expresiones

Las expresiones son otros elementos de alto nivel, que a causa de la precedencia y asociatividad entre distintos grupos de operadores, requieren una descripción sintáctica con múltiples producciones y no terminales. Normalmente, esta descripción sintáctica no suele ser muy intuitiva y además es muy dependiente del tipo de gramática utilizada (LL o LR).

Dado un lenguaje con operadores de dos niveles de precedencia (+,- con menor precedencia y *,/ con mayor precedencia), con expresiones relacionales (<,>,==) y con asociatividad de izquierda a derecha, una posible gramática sería la siguiente:

```
EXP_RELACIONAL ::= EXP_ADITIV OP_REL EXP_ADITIV
EXP_ADITIV ::= EXP_ADITIV OP_ADITIV EXP_MULTIP
EXP_MULTIP ::= EXP_MULTIP OP_MULTIP EXP_PRIMARIA
EXP_PRIMARIA ::= "(" EXP_ADITIV ")" | ID | CTE
OP_MULTIP ::= "*" | "/"
OP_ADITIV ::= "+" | "-"
OP_REL ::= "<" | ">" | "=="
```

La gramática anterior es bastante sencilla, pero con gramáticas más complejas, la cantidad de no terminales y producciones se vuelve inmanejable. A la hora de diseñar la API, se ha definido una única interfaz, llamada `BinaryExpression`, que tiene dos operandos `Expression` y un operador infijo. Las reglas de precedencia y asociatividad no se van a reflejar en la API, es decir no existen las interfaces `ExpRelacional`, `ExpAditiv`, `ExpMultip`, `ExpPrimaria`.

Identificadores

Cuando en una gramática aparece un identificador, posteriormente en la fase de comprobaciones semánticas se usará el lexema de ese identificador para diversas tareas, dependiendo de en qué producción de la gramática se utilice. Por ejemplo, cuando aparece en una parte de definición ese identificador, es el nombre del elemento definido y cuando ese identificador se encuentra en otras partes, lo que se está haciendo es indicar que se quiere usar un elemento definido con ese nombre. Por tanto, la primera conclusión a la se puede llegar, es que los identificadores son semánticamente distintos si se encuentran en partes diferentes de la gramática.

Cuando se definan las interfaces partiendo de la gramática, hay que tener en cuenta esta diferencia. Cuando el identificador se encuentre en una parte de definición, habitualmente será representado por una propiedad llamada `Name` de la clase `java.lang.String`. En cambio, cuando un identificador aparezca como una referencia a un elemento, será representado como una interfaz que lo indique. Por ejemplo, cuando un identificador aparece en una expresión, en realidad es una referencia a un atributo, local o parámetro, por tanto, será representado como un objeto de `AttributeRef`, `LocalRef` o `ParameterRef`.

Árbol de sintaxis abstracta

Siguiendo las ideas propuestas en el apartado anterior, se definen una serie de interfaces que van a representar los elementos que aparecen en el código fuente. Esta estructura de objetos es similar al árbol de sintaxis abstracta obtenido después de un proceso de análisis sintáctico. No obstante, las relaciones entre el nodo padre y los nodos hijos en este árbol, están representadas mediante relaciones de composición y para acceder a los nodos hijos de un determinado nodo, hay que invocar cada uno de los métodos de acceso.

El requisito RA.2.4, establece que la estructura de objetos del código fuente pueda ser considerada como un árbol de sintaxis abstracta. Para conseguirlo en `CodeElement`, padre de todos los interfaces, se definen los siguiente métodos:

```
CodeElement getParent();
CodeElement getParent(Class parentClass);
CodeList getChildren();
```

Con estos métodos, es posible tratar la estructura de objetos como un árbol. Con ellos también se cumple con el requisito RA.2.3, que establece que la API debe permitir conocer la posición relativa de un elemento del código fuente en su contexto. Con los métodos `getParent(...)` se obtiene el objeto que contiene al preguntado en la relación de composición.

Además, se proporcionan otros métodos, que a parte de considerar la estructura de objetos como un árbol, permiten recorrer cada uno de sus nodos de una forma muy sencilla. Los métodos son:

```
void traverse(TraversalListener traversalListener);
void traverseVisitor(Visitor visitor);
```

Estos dos métodos inician el recorrido de los elementos del árbol de objetos. Cuando se invoca `traverse` con la instancia de `TraversalListener`, se recibe por cada nodo del árbol una notificación antes de recorrer sus hijos y otra notificación justo al finalizar el recorrido de los mismos. Cuando se invoca `traverseVisitor`, se realiza un recorrido en profundidad del árbol de objetos y se invoca por cada nodo el método correspondiente de la interfaz `Visitor`. Este último método atiende al requisito RA.2.8, que determina que la API debe proporcionar una forma de recorrer los objetos del árbol y durante el recorrido de este árbol se deben poder definir acciones que serán ejecutadas para cada uno de los objetos que lo forman.

Además, se implementa el patrón de diseño Visitador [Gamma95] usando el método:

```
void accept(Visitor visitor);
```

Al invocar este método en cualquier objeto, será invocado en el objeto `Visitor` el método correspondiente a la interfaz concreta de ese objeto.

Representación textual de los elementos del código fuente

El requisito RA.2.5 indica que la API debe proporcionar una forma sencilla y rápida de obtener una representación textual del objeto del código, que sea equivalente semánticamente a la parte del código fuente original del que se creó ese objeto. La forma más rápida y cómoda de ofrecer este servicio, es haciendo que el método `String toString()` de la clase `java.lang.Object` sea el que genere dicha representación.

El requisito RA.2.6 establece que la API debe proporcionar mecanismos que permitan obtener representaciones textuales de los ficheros fuente de una forma personalizada. Para conseguirlo, es necesario ofrecer al usuario de la API una representación textual por defecto que sea fácilmente adaptable a sus necesidades.

Todas estas facilidades se encuentran disponibles dentro del paquete `es.urjc.escet.vido.javamod.representer`. En primer lugar, se dispone de una clase, llamada `TokenManager` que declara constantes para identificar a cualquier token del lenguaje Java. Además, agrupa todos los tokens en palabras reservadas, identificadores, signos de puntuación, operadores y literales. Por último, dispone de un método que permite obtener una representación textual de un token, si es posible (algunos tokens como los literales o identificadores no tienen una única representación textual).

Se dispone de una clase llamada `CodeElementRepresenter`, ésta es una clase abstracta que tiene del siguiente método:

```
void representCodeElement(CodeElement codeElement)
```

Al invocar dicho método, se comienza a recorrer el árbol de objetos que parte de ese elemento y se van invocando los siguientes métodos abstractos y protegidos de la clase `CodeElementRepresenter`:

```
void newLine()
void incIndentation()
void decIndentation()
void representToken(int tokenType, CodeElement codeElement)
void representToken(int tokenType, String representation,
CodeElement codeElement)
```

Los métodos se irán invocando a medida que se recorre el árbol de objetos, para representar la lista de tokens semánticamente equivalente a la estructura de objetos. Para obtener esto, la clase `CodeElementRepresenter` implementa la interfaz `Visitor` y por cada interfaz diferente, implementa un método que genera las sucesivas llamadas. Aunque los detalles de implementación no se van a decidir hasta la fase de implementación, el método que representa la importación de un paquete podría estar implementado de la siguiente forma:

```
public void visitPackageImport(PackageImport codeElement) {
      representToken(TokenManager.IMPORT, codeElement);
      representCodeElement(codeElement.getPackageRef());
      representToken(TokenManager.DOT, codeElement);
      representToken(TokenManager.STAR, codeElement);
}
```

Existe una clase hija de `CodeElementRepresenter` llamada `PlainTextualRepresenter`, que implementa todos los métodos necesarios para generar una representación en texto plano. Para personalizar esta representación, basta crear una clase hija y redefinir el método de la interfaz `Visitor` del elemento que se desee adaptar. El diagrama de clases de la Figura 5 muestra estas relaciones.

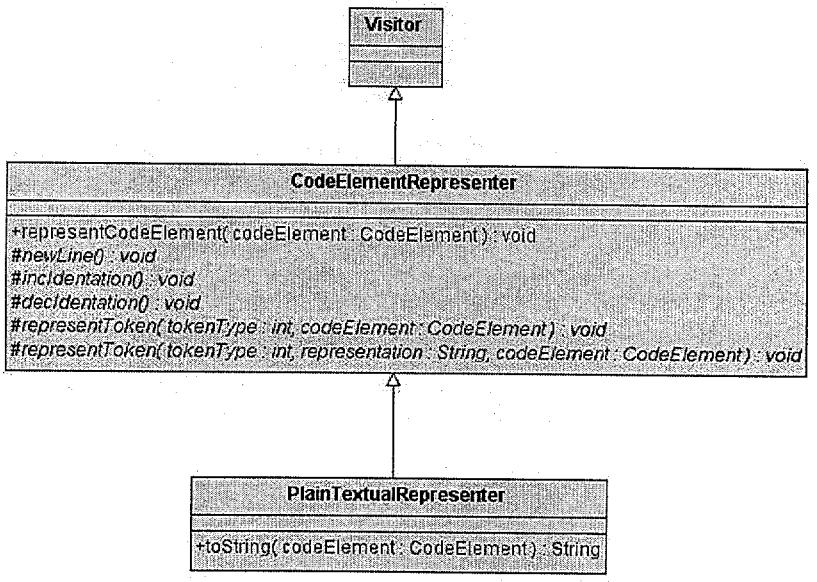

Figura 5. *Jerarquia de herencia de CodeElementRepresenter*

5.1.3 Elementos de Java que no aparecen en ficheros fuente

El requisito RA.6 establece que la API debe permitir la gestión de los elementos del lenguaje Java que no aparecen en los ficheros de código fuente pero representan conceptos del lenguaje. Por ejemplo, la API debe representar los tipos primitivos, el tipo del literal `null`, los paquetes, las operaciones, etc...

Estos elementos no aparecen estudiando la gramática ya que nunca aparecen en el código fuente. No obstante, se pueden definir por similitud con otros elementos que sí aparecen en la gramática. Por ejemplo, los tipos primitivos, los tipos de los arrays, el tipo de la constante `null` y el tipo especial `void`, pueden integrarse junto con las clases, tanto anónimas como nombradas, y las interfaces en la misma jerarquía de herencia. La jerarquía que se ha considerado más oportuna es la mostrada en la Figura 6.

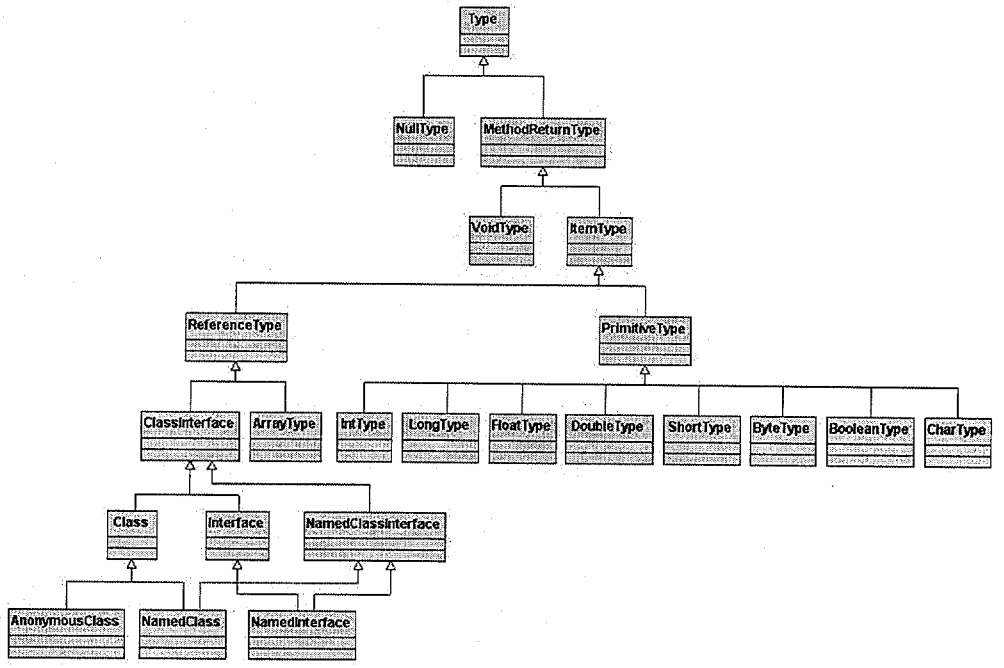

Figura 6. *Jerarquía de herencia de los tipos*

Para las operaciones de los tipos primitivos, también se ha decidido relacionarlas con las operaciones de las clases e interfaces, es decir, con los métodos. La jerarquía de herencia obtenida se muestra en la Figura 7.

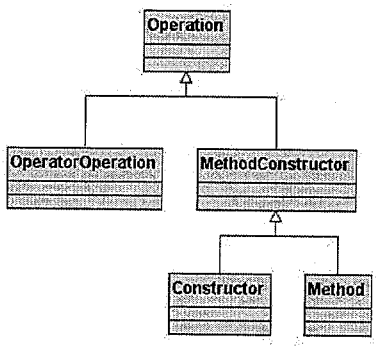

Figura 7. Jerarquía de herencia de las operaciones

5.1.4 Referencias y Definiciones

El requisito RA.5.4.1 indica que la API deberá proporcionar un mecanismo para acceder a los elementos del programa a los que se hace referencia desde otros elementos. En javaMod existen muchos elementos que realmente son referencias a otros elementos. Para ser ortogonales, se ha tomado como convenio que las referencias a una definición en el código fuente, tengan el mismo nombre que la definición concatenado con `Ref`. Por ejemplo, la referencia a un `Type` es un `TypeRef`, la referencia a una `Local` es una `LocalRef`, etc...

Además de esta convención de nombres, todas las referencias tienen como interfaz padre a `Reference` y todas las definiciones tienen como interfaz padre a `Definition`. En la interfaz `Reference` se define el método:

```
Definition getDefinition()
```

Este método devuelve la definición a la que apunta esa referencia, lo que permite considerar a todas las referencias polimórficamente de forma común y no hay que tratar cada una de ellas por separado.

Además del método genérico, cada referencia concreta define un método que devuelve al elemento referenciado con su interfaz concreta. Por ejemplo, `PackageRef` tiene el método:

```
Package getPackage()
```

De forma análoga `Definition` define el método:

```
CodeList getReferences()
```

Este método devuelve las referencias que apuntan a esa definición en concreto. Con esto queda definido el requisito RA5.4.2, el cual dice que la API deberá permitir determinar las referencias que se hacen a cualquier definición.

En el proceso de análisis semántico se enlazan las referencias con los elementos referenciados, para llevar a cabo esto se ha considerado que tanto las referencias como las definiciones, sean identificables con un identificador. De esta forma, una referencia se asociará a la definición si los identificadores son compatibles entre sí. Hay muchos tipos de identificadores, dependiendo de los elementos a los que estén identificando, por ejemplo, a una local la identifica el nombre, pero a un método lo identifican el nombre y el tipo de cada uno de sus parámetros.

Se ha decidido definir la interfaz `Identifier` que contiene del método:

```
boolen isCompatibleWith(Identifier id)
```

Además, `Definition` y `Reference` contienen el método:

```
Identifier getIdentifier()
```

En el diagrama de clases de la Figura 8 se muestran los interfaces `Definition`, `Reference` e `Identifier`.

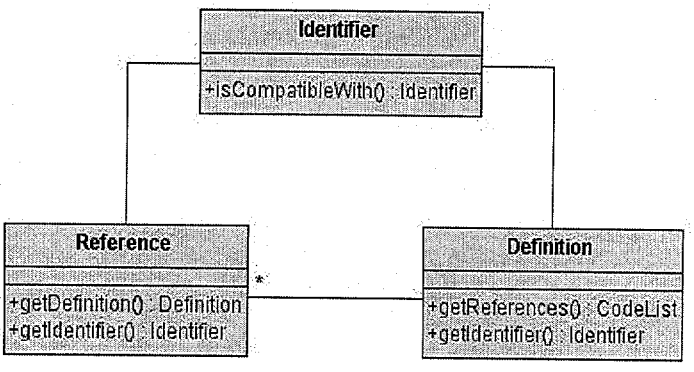

Figura 8. *Diagrama de clases de los interfaces* Definition, Reference *e* Identifier

5.1.5 Información semántica del programa

Algunos requisitos de la API requieren que se pueda acceder de una forma sencilla a las relaciones conceptuales que se producen entre los distintos elementos del programa. El requisito R5.4.3 dice que la API deberá proporcionar un árbol con las llamadas que se realizan a un método determinado o a cualquiera de los métodos que redefine, conocido como árbol de llamadas. El requisito RA.5.4.4 establece que la API deberá proporcionar el árbol de herencia del que es padre una clase determinada y por RA.5.4.5, la API deberá proporcionar la lista de métodos que redefinen a un método dado.

Por otro lado, el requisito RA5.3 dice que la API deberá proporcionar mecanismos para determinar los ítems de información activos cuando la ejecución se encuentre en un punto determinado.

La información necesaria para cumplir con los distintos requisitos está implícita en la propia semántica del lenguaje. Para hacerla explícita, será necesario realizar búsquedas por cada uno de los requisitos en los `CodeElements`. Cada interfaz define métodos semánticos que permiten acceder de una forma rápida y cómoda a toda esta información.

5.1.6 Modelado de las librerías

Dado el requisito RA.4, un programa Java habitualmente estará formado por ficheros fuente y por librerías y la información deberá ser representada y gestionada de una forma homogénea y ortogonal, independientemente de si se encuentra en ficheros fuente o librerías. Siguiendo este requisito, la forma de acceder a las clases e interfaces de las librerías ha de ser equivalente. La información no disponible, como la implementación de los métodos, no será reflejada. La información que se encuentre disponible será representada de la misma forma que en el código fuente. Esto permitirá, sin duda, que sea más cómodo de usar.

En cualquier caso, para diferenciar si una clase se encuentra en el código fuente o en una librería bastará determinar si el padre en el árbol formado por los elementos es un `Source` o una `Library`.

5.1.7 Asociación de objetos a los elementos

El requisito RA2.7 establece que la API debe proporcionar mecanismos para asociar información a cualquier objeto del código fuente y poder recuperarla fácilmente cuando sea necesario. Este servicio es muy útil cuando estamos construyendo visualizaciones de elementos. En estos casos, es habitual asociar el objeto que se encarga de visualizar una clase, por ejemplo, al propio elemento que representa a la clase. Cuando navegando por la definición del programa se encuentra dicha clase, se puede obtener muy rápidamente el objeto encargado de visualizarla.

Para poder definir ese requisito se ha considerado oportuno seguir la convención utilizada en los componentes gráficos de la librería estándar Swing [Swing] de Java. Para ello, el interfaz `CodeElement` define los métodos:

```
void putClientProperty(Object key, Object property);
Object getClientProperty(Object key);
void removeClientProperty(Object key);
void removeRecursiveClientProperty(Object key);
```

Con el método `putClientProperty` se asocia un objeto a un elemento junto con una clave. La clave es usada para poder gestionar posteriormente ese elemento. Con el método `getClientProperty` se obtiene el objeto al que se refiere la clave o `null` si no existe. Con el método `removeClientProperty` se elimina la asociación entre el objeto al que se refiere la clave y el `CodeElement`. Por último, para facilitar las tareas de gestión del árbol de sintaxis abstracta, el método `removeRecursiveClientProperty` elimina la asociación del subárbol que parte del `CodeElement`.

5.1.8 Repositorios de tipos

Dado un programa Java, es útil saber cuantas referencias al tipo primitivo entero se han realizado, los operadores del tipo carácter, etc... Es decir, los tipos primitivos junto con sus operaciones deben estar disponibles para ser consultados como cualquier otro tipo definido en una librería o definido por el usuario. Se ha considerado oportuno crear la interfaz `TypeRepository`, ésta tiene el método:

```
Type getType(TypeId type)
```

Este método devuelve el tipo del repositorio identificado por el identificador indicado o `null` si no existe en ese repositorio. En Java los tipos se pueden clasificar en tipos primitivos, clases o interfaces, arrays y los tipos especiales (el tipo `null` y el tipo `void`). Por ese motivo, se han creado subinterfaces de `TypeRepository` que albergarán cada uno de estos tipos, `ArrayTypeRepository`, `PrimitiveTypeRepository`, `ClassInterfaceRepository` y `SpecialTypeRepository`. En estos interfaces se definen métodos más adecuados para recuperar un tipo determinado dependiendo de su naturaleza.

Además se ha creado una interfaz llamada `AllTypesRepository` que permite gestionar todos los tipos de un programa Java. Por último, cabe mencionar que un paquete también es un repositorio de tipos y por tanto implementa el interfaz `TypeRepository`.

En la Figura 9 se muestra la jerarquía de herencia y los métodos más relevantes de cada interfaz. En la Figura 10 se muestran las relaciones de composición y asociación.

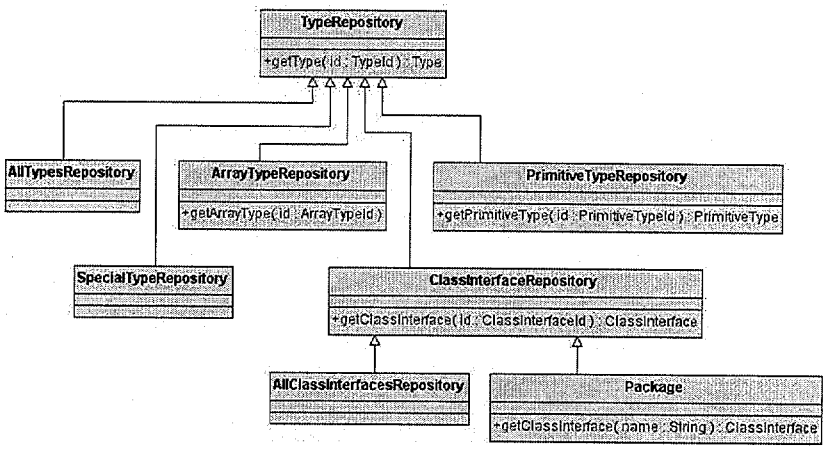

Figura 9. *Jerarquía de herencia de los repositorios de tipos*

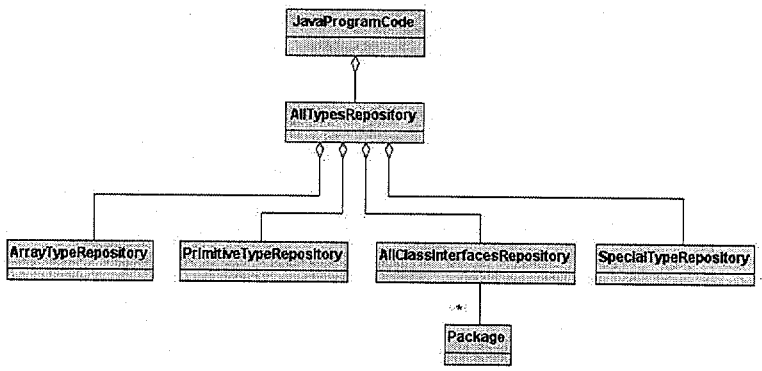

Figura 10. Relaciones de composición de los repositorios de tipos

5.2 Ejecución de un programa Java

Los requisitos que han guiado el diseño de esta parte de la API están englobados en el requisito RB. Las ejecuciones de los programas Java, están representadas como instancias de la interfaz `JavaProgramExec`, y cada una de las propiedades que caracterizan a las mismas como métodos de esa interfaz. La interfaz `JavaProgramExec` se define para cumplir con el requisito RB.2. Para obtener una instancia de ésta, se usará el método:

```
JavaProgramExec createJavaProgramExec()
```

Este método pertenece a `JavaProgramCode`. Todas las interfaces relativas a la ejecución del programa Java están situados en el paquete `es.urjc.escet.vido.javamod.execution`.

5.2.1 Inicio, finalización y control de la ejecución

Atendiendo al requisito RB.1, la API debe proporcionar mecanismos para iniciar la ejecución del programa. Se indicará la clase que tiene el método `main` y los parámetros que recibirá la aplicación. Por el requisito RB.4 se debe iniciar la ejecución en modo depuración y según el requisito RB.3 la API debe proporcionar mecanismos para ejecutar un programa Java como se ejecutaría si se invocara desde el sistema operativo. Para ello se han definido los siguientes métodos en `JavaProgramExec`:

```
void startExecution(Class mainClass, int mode)
```

```
void startExecution(Class mainClass, String arguments, int mode)
void startExecution(Class mainClass, String arguments, File
compilationDirectory, int mode)
```

Más adelante se verán todos los modos de ejecución posibles, donde el modo que permite que comience la ejecución de la aplicación, como si se tratara del sistema operativo, es `JavaProgramExec.RUN`. La invocación de estos métodos no bloquea el hilo de ejecución. La ejecución se finalizará normalmente cuando lo requiera el programa java que se está ejecutando, pero se puede forzar dicha finalización usando el método:

```
void finishExecution()
```

Se puede determinar si una instancia de `JavaProgramExec` está ejecutando el programa Java con el método:

```
boolean isExecuting()
```

También es posible detener momentáneamente la ejecución de todos los hilos del programa Java y volver a reanudar la misma cuando se desee con los siguientes métodos:

```
void suspendExecution()
void resumeExecution()
```

La ejecución de un programa Java está formada por un conjunto de hilos de ejecución. Un hilo de ejecución se crea y controla en este lenguaje como una instancia de la clase `java.lang.Thread`. Atendiendo al requisito RB.7.2 la API debe proporcionar mecanismos para gestionar cada uno de los hilos de ejecución del programa de forma independiente. Para ello, cada hilo de ejecución será representado como un objeto de la interfaz `ThreadObject`. Durante la ejecución de un programa Java podemos obtener los hilos de ejecución que se están ejecutando en `JavaProgramExec` con el método:

```
java.util.List getThreads()
```

La ejecución de cada hilo puede ser controlada independientemente. Para ello `ThreadObject` dispone de los siguientes métodos y atributos:

```
public static final int SUSPENDED
public static final int RESUMED

void suspendExecution()
void resumeExecution()
int getSuspendedOrResumed()
```

Con estos métodos, se puede detener y reanudar la ejecución de cada uno de los hilos que forman el programa Java en ejecución. El diagrama de clases de la Figura 11 muestra la relación entre los `ThreadObjects` y el `JavaProgramExec` y los métodos que permiten controlar su ejecución.

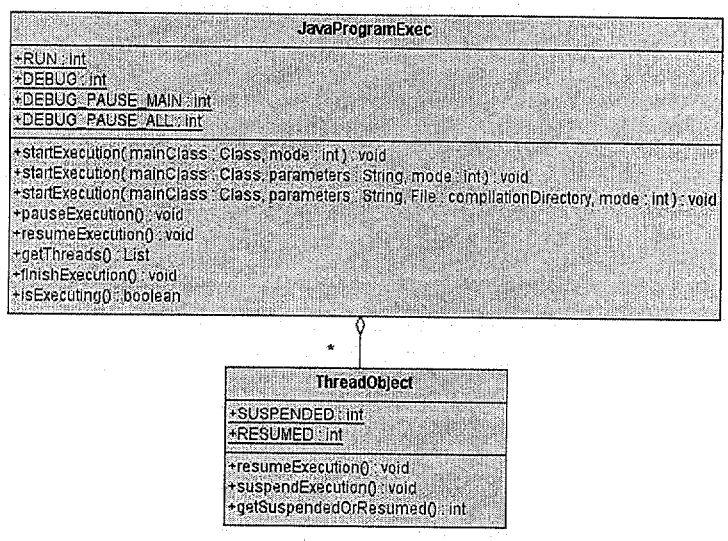

Figura 11. *Diagrama de clases que muestra los interfaces y métodos que controlan la ejecución*

Existen varios modos de iniciar la ejecución del programa, su comportamiento es el siguiente:

- `JavaProgramExec.RUN`: Inicia la ejecución sin ningún tipo de control, como si fuese iniciada desde el sistema operativo.

- `JavaProgramExec.DEBUG`: Inicia la ejecución en modo de depuración, pero al iniciar la ejecución todos los hilos de ejecución comienzan a ejecutarse.
- `JavaProgramExec.DEBUG_PAUSE_ALL`: Inicia la ejecución en modo depuración y todos los hilos están pausados.
- `JavaProgramExec.DEBUG_PAUSE_MAIN`: Inicia la ejecución en modo depuración con todos los hilos de ejecución en ejecución excepto el hilo de ejecución que ejecuta el método `main`, que detendrá su ejecución al llegar a la primera sentencia del método.

5.2.2 El estado de la ejecución

El estado de la ejecución se determina por el estado de cada uno de los hilos de ejecución. El estado de un hilo de ejecución sólo puede ser examinado usando lo métodos de `ThreadObject` cuando el hilo se encuentra suspendido. El requisito RB.7.4 establece que la API debe proporcionar mecanismos para determinar el estado de ejecución de un hilo, para ello existe el método:

```
int getState()
```

y devuelve alguna de las constantes siguientes:

```
ThreadObject.THREAD_STATUS_UNKNOWN
ThreadObject.THREAD_STATUS_ZOMBIE
ThreadObject.THREAD_STATUS_RUNNING
ThreadObject.THREAD_STATUS_SLEEPING
ThreadObject.THREAD_STATUS_MONITOR
ThreadObject.THREAD_STATUS_WAIT
ThreadObject.THREAD_STATUS_NOT_STARTED
```

Además se puede saber el estado de la pila de ejecución de un hilo según lo determinado por el requisito RB7.5.1. La pila de ejecución es una lista de los objetos que representan la ejecución de los métodos o constructores. La interfaz que representa la ejecución de un método o constructor se llama `MethodConstructorExec`. El método para obtener la pila de ejecución de un hilo es:

```
List getStack()
```

En los requisitos RB.7.5.2 y RB.7.5.3 se define que la API debe proporcionar mecanismos para determinar la última sentencia o expresión ejecutada por el hilo antes de pausar su ejecución, esta información se obtiene con el método:

```
Executable getLastExecuted()
```

En la Figura 12 se encuentra el diagrama de clases que muestra las relaciones entre el hilo de ejecución y los componentes de la pila.

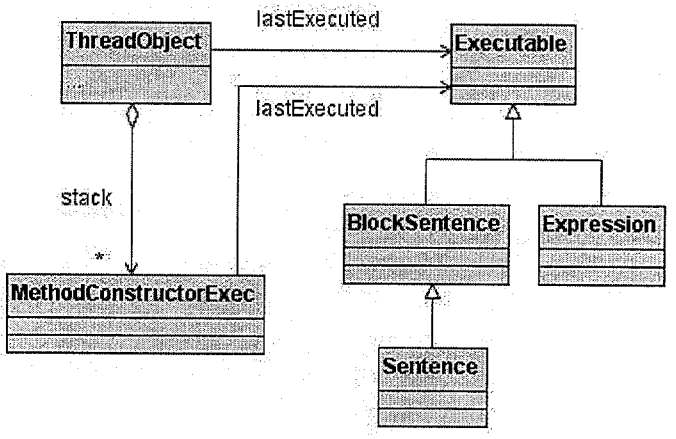

Figura 12. *Diagrama de clases que muestra los interfaces que representan el estado de la ejecución*

5.2.3 La información de un programa Java

La información de un programa Java está formada por los valores de las locales y parámetros de cada método en ejecución y por los atributos estáticos de las clases cargadas en memoria. Toda esta información debe ser accesible a los usuarios de la API, tal como indica el requisito RB.7.5.6. La información siempre está asociada a ítems de información activos, que serán las locales activas, los parámetros activos o los atributos en objetos. La Figura 13 muestra la jerarquía de herencia de los ítems de información.

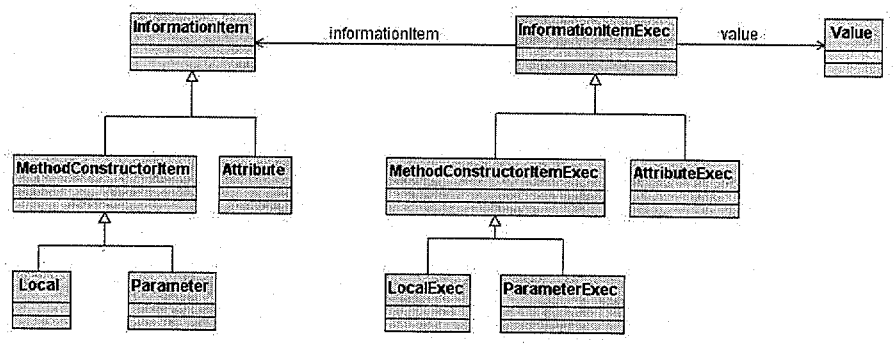

Figura 13. *Diagrama de clases que muestra los items de informacion*

El requisito RB.7.5.7 indica que la API deberá representar todos los valores que puede contener una variable, es decir, los valores de tipos primitivos, el valor `null` o referencias a los objetos, que pueden ser arrays o instancias de clases. Para ello, se ha definido la jerarquía de interfaces mostrada en la Figura 14.

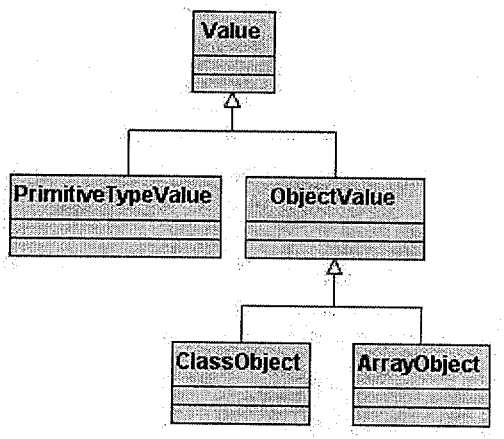

Figura 14. *Diagrama de clases que muestra la jerarquía de valores*

Para obtener las locales y parámetros activos en la ejecución de un método, se usa el método de `MethodConstructorExec`:

```
List getActiveMethodConstructorItemExecs()
```

Para acceder al valor de un ítem de información, basta invocar el método de `ItemInformationExec`:

```
Value getValue()
```

Para acceder a las clases e interfaces cargados en una ejecución, se usa el método de `JavaProgramExec`:

```
List getLoadedClassInterfaces()
```

5.2.4 Interacción con los objetos

Según el requisito RB.7.6 la API debe proporcionar mecanismos para interaccionar con los objetos y clases del programa en ejecución. Para ello, el interfaz `ObjectValue` dispone de los siguientes métodos:

```
public void setAttributeValue(Attribute attribute, Value value)
public Value getAttributeValue(Attribute attribute)
public Value invokeMethod(ThreadObject threadObject, Method method,
List parameters, int options)
```

5.2.5 Asociación de objetos a los elementos

La API debe proporcionar mecanismos para asociar información a los objetos en la ejecución de un programa y debe permitir recuperarla fácilmente cuando sea necesario (requisito RB.6). Este mecanismo debe estar definido de la misma forma que en el requisito RA.2.7. Por tanto, la interfaz `Value`, dispone de los métodos:

```
void putClientProperty(Object key, Object property);
Object getClientProperty(Object key);
void removeClientProperty(Object key);
void removeRecursiveClientProperty(Object key);
```

5.2.6 Ejecución paso a paso

El requisito RB.7.3.1 establece que al reanudar la ejecución se pueda indicar la sentencia en la que se detendrá de nuevo. Para poder cumplirlo `ThreadObject` contiene los siguientes métodos:

```
void stepOut()
void stepInto()
void stepOver()
```

Estos métodos hacen que el hilo reanude su ejecución, pero se detiene en una sentencia determinada. La semántica de estos métodos está basada en la semántica habitual de los depuradores pero en vez de estar basados en líneas, está basado en los elementos ejecutables del código fuente. Por ejemplo, si la próxima sentencia que hay que ejecutar es la llamada a un método, usando `stepOver`, la ejecución se irá deteniendo en la evaluación de cada parámetro.

5.2.7 Entrada y salidas del programa en ejecución

Se deberá proporcionar un acceso de bajo nivel a la entrada y salidas en formato binario (requisito RB.8.1). Para ello, `JavaProgramExec` dispone de varios métodos que sirven para tratar estos flujos como `InputStream` y `OutputStream` en la API. Los métodos son los siguientes:

```
void setStandardInput(InputStream stream)
void setErrorOutput(OutputStream stream)
void setStandardOutput(OutputStream stream)
```

5.2.8 Recibir notificaciones ante eventos de interés

Según el requisito RB.7.7 se podrán registrar gestores de eventos en diversos elementos del programa de forma que cuando se produzcan esos eventos en la ejecución se notifique a los gestores. Para definir la gestión de eventos se ha usado el sistema de eventos de la tecnología estándar de Java llamada JavaBeans. Usando este modelo, se crean los gestores de eventos implementando un interface `<EventGroup>Listener`, donde <EventGroup> es el nombre del grupo de eventos. Estos gestores se registran usando métodos del estilo:

```
void add<EventGroup>Listener(<EventGroup>Listener listener).
```

Cuando los eventos se produzcan, los métodos de la interfaz `<EventType>Listener` serán invocados. En esta API se han definido los siguientes grupos de eventos:

- Cuando se carga y se descarga una clase de memoria.

- Cuando se inicia y finaliza la ejecución de un método.
- Cuando se lee y escribe en un atributo.
- Cuando se inicia y finaliza la ejecución de un hilo.
- Cuando se pausa y reanuda la ejecución de un hilo.

Para poder gestionarlos se han definido los siguientes interfaces:

```
public interface ClassInterfaceLoadListener extends EventListener {
   void classInterfaceLoaded(ClassInterfaceLoadEvent e);
   void classInterfaceUnloaded(ClassInterfaceLoadEvent e);
}

public interface MethodConstructorListener extends EventListener {
    void startedMethodConstructorExec(MethodConstructorEvent e);
    void finishedMethodConstructorExec(MethodConstructorEvent e);
}

public interface AttributeListener extends EventListener {
    void readAttribute(AttributeEvent e);
    void writedAttribute(AttributeEvent e);
}

public interface SuspendResumeThreadListener extends EventListener
{
    void suspendedThreadExecution(SuspendResumeThreadEvent e);
    void resumedThreadExecution(SuspendResumeThreadEvent e);
}

public interface ThreadListener extends EventListener {
   void startedThreadExecution(ThreadEvent e);
   void finishedThreadExecution(ThreadEvent e);
}
```

Para crear un gestor de eventos hay que implementar cualquiera de estas interfaces. Para registrar el gestor de eventos, de forma que sus métodos sean invocados se usan los siguientes métodos de `JavaProgramExec`:

```
ClassInterfaceLoadListenerOptions addClassInterfaceLoadListener
(ClassInterfaceLoadListener listener)
MethodConstructorListenerOptions addMethodConstructorListener
(MethodConstructorListener listener)
ThreadListenerOptions addThreadListener(ThreadListener listener)
```

Con estos métodos se pueden registrar tantos gestores de eventos como se desee. Sus métodos serán invocados cuando se produzcan dichos eventos durante la ejecución del programa Java. `ThreadObject` también disponemos del método:

```
SuspendResumeThreadListenerOptions addSuspendResumeThreadListener
(SuspendResumeThreadListener listener)
```

Con las interfaces `ClassInterfaceLoadListenerOptions`, `MethodConstructorListenerOptions`, `ThreadListenerOptions` y `SuspendResumeThreadListenerOptions` se pueden configurar diversas opciones de la notificación de eventos. Por ejemplo se puede inhabilitar el gestor de eventos momentáneamente, se puede indicar que se notifiquen los eventos sólo cuando se produzcan en determinados métodos o clases, etc...

5.3 Distribución en paquetes

Todo el conjunto de clases e interfaces definidos en la API javaMod han sido distribuidos en paquetes. La división elegida es la siguiente:

- **es.urjc.escet.vido.javamod.code:** Paquete donde se encuentran las interfaces que representan la definición de un programa java. Se encuentran las interfaces `JavaProgramCode`, `Package`, `CodeElement`, etc...
- **es.urjc.escet.vido.javamod.code.types:** Paquete donde están todas aquellas interfaces que representan los tipos de un programa Java. Se encuentran las interfaces `Class`, `Interface`, `PrimitiveType`, `NullType`, etc...
- **es.urjc.escet.vido.javamod.code.expressions:** Paquete donde están todas aquellas interfaces que representan a las expresiones.
- **es.urjc.escet.vido.javamod.code.sentences:** Paquete donde se encuentran todas aquellas interfaces que representan a las sentencias.

- **es.urjc.escet.vido.javamod.execution:** Paquete donde están todos aquellos interfaces que representan a cada uno de los aspectos de la ejecución de un programa Java. Se encuentran los interfaces `JavaProgramExec`, `Value`, etc...
- **es.urjc.escet.vido.javamod.execution.events:** Paquete donde están todas las interfaces que permiten controlar los eventos en ejecución.
- **es.urjc.escet.vido.javamod.code.representer:** Paquete donde están todas las clases e interfaces involucradas en el proceso de representación textual personalizada.

6 Implementación y pruebas de la API

En este capítulo se va a mostrar una implementación de la API javaMod. Pese a que el objetivo de ésta es proporcionar una vista integrada de los aspectos de definición y de ejecución del programa Java, desde el punto de vista de la implementación se puede hablar de dos partes muy diferenciadas entre sí que además hacen uso de tecnologías muy diferentes. En los siguientes apartados se describen estas dos partes principales, se mencionan las pruebas efectuadas y por último se muestran diversas métricas realizadas sobre la implementación.

6.1 Definición de un programa Java

Para implementar la parte de la API en la que es necesario representar la definición de un programa Java, se realiza un análisis léxico, sintáctico y semántico de cada uno de los ficheros fuente que componen el programa Java. Además, se realiza un análisis de todas las librerías y con toda la información obtenida, se construye la estructura de objetos que implementan las interfaces de la API. La Figura 15 muestra este proceso.

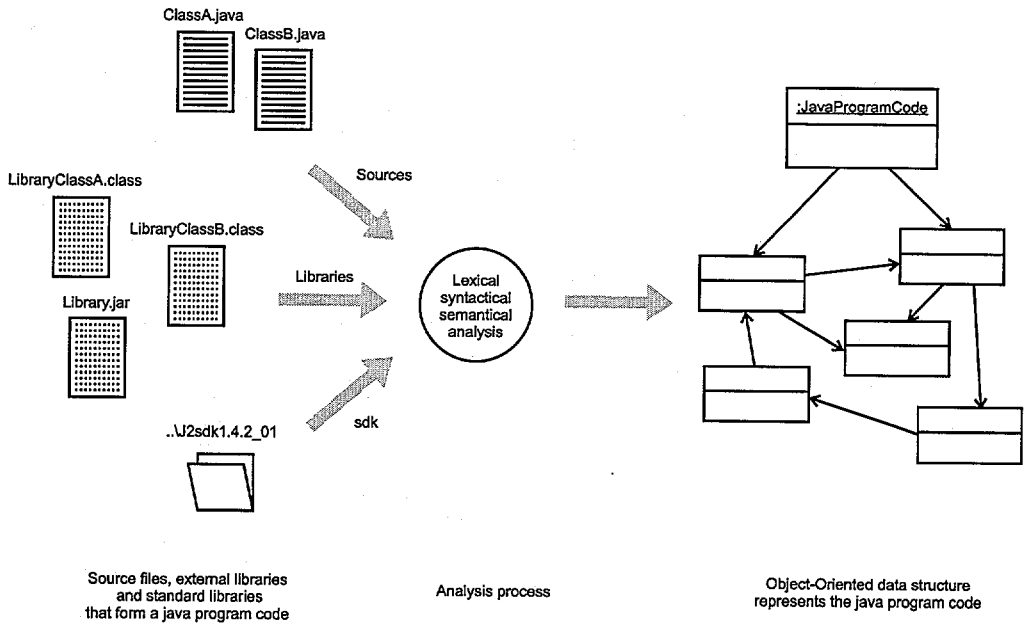

Figura 15. *Transformación de todos los ficheros que forman el programa Java en una estructura orientada a objetos*

Para obtener la estructura de objetos que representan toda la información de un programa Java se han de realizar una serie de pasos:

- **Primer paso:** Identificar todos los paquetes que se encuentran en las librerías, tanto la librería estándar como las externas. Esta información será útil cuando se analicen los `import` de un fichero o las referencias a clases o interfaces que incorporan el nombre del paquete. Es el primer paso, porque es necesario cargar las clases del lenguaje como `java.lang.Object` o `java.lang.String`.
- **Segundo paso:** Instanciar todos los objetos que representan los tipos primitivos del lenguaje Java y sus operaciones. Es necesario instanciar los tipos primitivos y las operaciones porque cuando en el código fuente se encuentre una referencia a los mismos, deberá crearse una relación entre ellos.
- **Tercer paso:** Identificar cierto tipo de clases de la librería estándar para que sean tratadas de una forma especial. Por ejemplo, la clase `java.lang.String` deberá ser tratada de forma especial ya que hay que incorporar la operación de concatenación. También tendrá un tratamiento especial la clase `java.lang.Object`, ya que todas las clases la tienen como padre.

- **Cuarto paso:** Construir el árbol de sintaxis abstracta de cada uno de los ficheros fuente que forman el programa.
- **Quinto paso:** Procesar cada uno de los ficheros fuente para asociar las referencias a sus respectivas definiciones. Algunas referencias se pueden resolver dentro del mismo fichero fuente y otras se resuelven en otros ficheros fuente, con las librerías o con los tipos primitivos y sus operaciones.

En los siguientes apartados se mostrará una descripción más detallada de cada uno de estos pasos y que técnicas se han empleado para su implementación.

6.1.1 Identificación de los paquetes de las librerías

El primer paso a la hora de representar un programa Java es la identificación de los paquetes de las librerías. Este proceso es crucial porque hay ciertas clases necesarias para el propio lenguaje.

En esta fase se construyen los objetos que van a representar a todos los paquetes como instancias de `PackageImpl`. Las clases de los paquetes no son cargadas en memoria porque sería muy costoso en espacio. Cuando las clases se necesiten se cargarán bajo demanda.

Para realizar el proceso de análisis de las librerías, se hace uso de la API BCEL. Esta API está desarrollada bajo código abierto por Apache. Con BCEL se pueden analizar los ficheros .class y .jar para determinar las clases e interfaces que definen y sus métodos y atributos. Toda la información proporcionada por la API es leída y convertida al formato definido por javaMod, para que el acceso a las clases o interfaces de las librerías siga los mismos convenios que el acceso a los ficheros fuente. Por ejemplo, en las librerías no se dispone del nombre de los parámetros de los métodos, por tanto, para que el acceso sea similar al acceso a ficheros fuente, estos nombres son generados automáticamente.

6.1.2 Inicialización de los tipos primitivos y operadores

El siguiente paso consiste en la creación de todos los tipos primitivos y la asociación de las correspondientes operaciones. Aunque este proceso pueda parecer trivial, en un primer momento existen muchos problemas con las dependencias circulares entre los distintos tipos primitivos y sus operaciones.

Para crear una implementación ortogonal del proceso de búsqueda de métodos y de operaciones en los tipos primitivos, se decidió asociar las operaciones con operador a los tipos primitivos de forma similar a la asociación de los métodos a las clases o interfaces. En el proceso de asociación, se considera el primer operando como el valor al que se le invoca el método. Según esta idea, el tipo primitivo debe disponer de una operación cuyo parámetro sea el segundo operando y cuyo tipo devuelto sea el que determina la operación. Esto permite modelar las operaciones de una forma más sencilla que la enumeración de las diversas posibilidades.

También se han implementado las conversiones implícitas como herencia, de forma que todos los tipos enteros tienen relaciones entre sí en una jerarquía de herencia de tipos. Esto evita tener que escribir todas las operaciones para todos los tipos primitivos si éstos tienen un tipo padre en común.

6.1.3 Tratamiento especial de algunas clases de las librerías

Existen clases en la librería estándar que son imprescindibles para cualquier programa Java. Además, tienen que ser tratadas de forma diferente a las demás debido a su naturaleza. Por ejemplo, la clase `java.lang.Object` es una clase que tiene que estar cargada en el sistema antes que cualquier otra porque cualquier otra, clase la tendrá como clase padre.

La clase `java.lang.String` es una clase muy especial porque es la única clase que tiene una operación con operador: el operador de concatenación. Además, el método `toString()` de la clase `java.lang.Object` devuelve un objeto de la clase `java.lang.String`.

6.1.4 Construcción del árbol de sintaxis abstracta

El siguiente paso que se lleva a cabo en esta implementación es analizar cada uno de los ficheros fuente que lo forman. Para ello, se ha usado el generador de analizadores JavaCC. Se ha escrito una definición léxico-sintáctica del lenguaje Java, que permite determinar si el fichero fuente es o no correcto.

Esta definición léxico-sintáctica ha sido completada con acciones semánticas, que van construyendo el árbol de sintaxis abstracta a medida que se va realizando el proceso de análisis. Para conseguir que el árbol se fuese construyendo de la forma adecuada, se ha seguido una determinada estructura. Consiste básicamente en instanciar un objeto cuando todos sus atributos o hijos en el árbol han sido analizados correctamente. Al instanciar el objeto se pasan como parámetros los atributos. La especificación sintáctica, junto con las rutinas semánticas de la especificación JavaCC para los parámetros es la siguiente:

```
ParameterImpl Parameter() :
{
    ItemTypeRefImpl itemTypeRef;
    String identifier;
    boolean finalParameter = false;
    int numDimensions = 0;
}
{
  [ "final" { finalParameter = true; } ]
  itemTypeRef = ItemTypeRef() identifier = Identifier()
  ( "[" "]"  { numDimensions++; } )*

  { return new ParameterImpl(
      identifier,itemTypeRef,finalParameter,numDimensions);
  }
}
```

No siempre es posible construir el árbol de sintaxis abstracta de forma directa partiendo de la gramática. Por ejemplo, cuando en un código fuente aparece una secuencia de identificadores separados por puntos, no se puede determinar qué clases instanciar. Una secuencia de identificadores separados por puntos puede representar la referencia a una local y referencias encadenadas a los atributos públicos. Pero también puede representar la referencia a una clase, cualificada con el nombre del paquete y posteriormente un acceso encadenado a los atributos. De igual forma, un identificador en una expresión puede representar el acceso a un atributo, el acceso a una local o el acceso a un parámetro. En ninguno de estos casos es posible determinar la naturaleza de los elementos en el proceso de análisis léxico-sintáctico.

Para solucionar este problema, lo que se hace es instanciar objetos de una serie de clases especiales que indican la naturaleza ambigua de esas construcciones sintácticas. Posteriormente, en la etapa en la que se van enlazando las referencias a las definiciones, se identifican estos elementos dentro del árbol de sintaxis abstracta y se resuelven las ambigüedades que se hayan encontrado. Las reglas de resolución de este tipo de ambigüedades están bien definidas en la especificación del lenguaje.

6.1.5 Enlaces de las referencias a las definiciones

Dentro de las librerías, todas las referencias están enlazadas a sus definiciones. Lo mismo ocurre con los objetos que representan a los tipos primitivos. En cambio, en los ficheros fuente es necesario establecer explícitamente el enlace entre las referencias y las definiciones. Desde un fichero fuente se pueden hacer referencias a elementos definidos en diversas partes de la definición del programa, las posibilidades son:

- **Referencias a definiciones dentro del mismo fichero fuente:** Las referencias a locales o parámetros siempre referencian a elementos definidos previamente en el código fuente.
- **Referencias a definiciones en otro código fuente:** Si en dos ficheros fuente se definen dos clases y una es la clase hija de la otra. Si desde la clase hija se hace referencia a un atributo de la clase padre se estará creando una referencia entre dos ficheros fuente.
- **Referencias a definiciones de las librerías:** Cuando desde un código fuente se hace referencia a una clase de la librería estándar o cualquier otra.
- **Referencias a tipos primitivos u operaciones:** Cuando se hace referencia a un tipo primitivo o a una operación del mismo se han de establecer las relaciones entre ellos. Los tipos primitivos estarán representados también en la definición del programa Java.

Para establecer los enlaces se recorre el árbol de sintaxis abstracta que representa al código fuente y cuando se encuentra una referencia se comienza un proceso de búsqueda para enlazarla. Por ejemplo, en el caso de las referencias a métodos, se obtiene la clase o interfaz del ítem de información sobre el que se invoca el método, se obtienen todos sus métodos y posteriormente, se busca dentro de los métodos usando el nombre del método y los tipos de los parámetros. En el caso de los enlaces de los ítems de información el proceso es similar, cuando se encuentra un ítem de información se busca en los ámbitos léxico que se encuentran activos y si se encuentra un ítem de información cuyo nombre es igual al nombre de la referencia, se produce el enlace.

Como se ha comentado anteriormente, las clases e interfaces de las librerías no se cargan en memoria a no ser que sean referenciadas desde el código fuente. Esta decisión se ha tomado en base a que lo habitual es analizar el código fuente del programa y no las librerías. En cualquier caso, si a un paquete se le piden todas las clases o interfaces que contiene, también se cargarán.

6.2 Ejecución de un programa Java

Para controlar la ejecución de un programa Java se ha usando la plataforma JPDA a través de la interfaz JDI (Java Debugger Interface). Con esta interfaz se puede controlar la ejecución de un programa Java de una forma muy parecida a la requerida por javaMod. En cualquier caso, existen algunas funcionalidades que JPDA no proporciona y es necesario implementarlas. Uno de los aspectos que JPDA no permite es controlar la ejecución paso a paso basándose en los elementos del código fuente, tan sólo permite controlar la ejecución por líneas de código. Por este motivo, hay que realizar un preproceso de los ficheros fuente que forman el programa Java para que, usados con JPDA, se pueda obtener la funcionalidad deseada. Para controlar la ejecución del programa Java se realizan las siguientes fases:

- Se construyen los ficheros fuente adaptados a JPDA.
- Se compilan los ficheros fuente usando el compilador de Java proporcionado por Sun.
- Se gestiona la ejecución del programa usando la plataforma JPDA a través de la interfaz JDI (Java Debugger Interface).

En los siguientes apartados se describen cada una de estas fases.

6.2.1 Construcción de los ficheros fuente adaptados a JPDA

JPDA permite controlar la ejecución paso a paso de un programa Java basándose en las líneas del fichero fuente. Debido a esto, no es posible controlar la evaluación individual de los parámetros de una llamada a un método si éstos se encuentran en la misma línea en el código. Es un requisito de javaMod poder controlar la ejecución paso a paso por cada unidad de ejecución, es decir, que cuando se vaya ejecutando paso a paso, se pueda controlar parámetro a parámetro independientemente de que se encuentren en la misma línea.

Para poder implementar los requisitos de javaMod, usando JPDA, se construyen unos ficheros fuente con todos sus elementos ejecutables divididos en diferentes líneas. Estos ficheros fuente son difíciles de leer por las personas, pero permiten controlar la ejecución paso a paso. Para generar estos ficheros se crea un representador textual modificado que va partiendo en líneas todos los componentes de un fichero fuente. Además, va asociando a cada `CodeElement` la línea del código fuente en la que se encuentra. De esta forma, cuando JPDA informa de que se ha detenido la ejecución en una línea determinada, se puede obtener el `CodeElement` de esa línea. Hay que resaltar que los ficheros fuente generados para la depuración son iguales que los originales con la única diferencia de que introducen saltos de línea para separar los elementos, por tanto, hay que tener claro que no se incorporan sentencias adicionales.

6.2.2 Compilación de los ficheros fuente

Después de generar los ficheros fuente es necesario compilarlos. Para ello se utiliza el compilador incluido en el SDK de Sun. Este compilador se encuentra en la librería `tools.jar`, la misma que incorpora JDI. Para invocar al compilador se utiliza un código como el siguiente:

```
boolean errorCompilacion;
// Parámetros del compilador como los usados en la línea de
comandos
String[] params = new String[] {...};
ByteArrayOutputStream erroresStream =
   new ByteArrayOutputStream();
sun.tools.javac.Main compiler =
   new sun.tools.javac.Main(erroresStream,"javac")
compiler.compile(params);
```

```
String errores = erroresStream.toString();
errorCompilacion = (errores.indexOf("error") != -1);
```

6.2.3 Gestión de la ejecución con JDI

Existen algunos aspectos de JDI que han de ser adaptados para ser usados con la API definidos en javaMod. A continuación se describen los aspectos más significativos de cada uno de ellos.

Adaptación del modelo de eventos de JDI al modelo de javaMod

En JDI los eventos se gestionan de una forma completamente diferente a como se gestionan en el modelo JavaBeans[13]. En JDI si se está interesado en un evento hay que crear una petición e indicárselo a un gestor global de eventos. Las notificaciones de los eventos producidos se reciben como una lista de eventos, la cual es necesario leer constantemente.

Por cada evento producido en JDI, se identifica su naturaleza y el elemento de javaMod involucrado en él. A ese elemento se le indica que se ha producido el evento y él es el encargado de notificar a los gestores de eventos que se hayan registrado.

Control paso a paso de la ejecución basado en elementos del código

El sistema de ejecución paso a paso de JDI está basado también en eventos, pero en javaMod se ha decidido que sea implementado en `ThreadObject` con los métodos `stepOut()`, `stepIn()` y `stepOver()`. Por tanto se han tenido que gestionar internamente todos los eventos que permiten dicha funcionalidad.

Cuando un hilo suspende su ejecución, informa del fichero de código fuente y de la línea en la que se encuentra suspendido. Al usuario de la API hay que indicarle la sentencia o expresión donde se encuentra detenida la ejecución. Para dar esa información se utiliza una estructura de datos que permite conocer el elemento del código fuente que se encuentra en dicha línea. Como se ha mencionado anteriormente, esta estructura de datos se construye en el proceso de generación de los ficheros fuente adaptados.

---

13 Modelo de componentes estándar de Java. http://java.sun.com/products/javabeans/

Gestión eficiente de los valores de la ejecución

Cuando se controla la ejecución de un programa Java uno de los mayores problemas es la forma en que se tratan los objetos de la misma. Es deseable que un mismo objeto del programa en ejecución que esté representado por el mismo objeto de la API javaMod. Por ejemplo, si asociamos información a ese objeto, esa información tiene que permanecer asociada a él durante toda la vida del objeto de ejecución. Una posible implementación consiste en instanciar un objeto de javaMod por cada objeto en ejecución y mantenerle en memoria hasta que el objeto en la ejecución desaparezca. Pero esta solución no es muy buena ya que es posible que muchos objetos no sean necesarios y mantenerlos en memoria ocuparía un espacio muy preciado.

En la implementación realiza el objetivo ha sido ofrecer la misma funcionalidad pero manteniendo en memoria aquellos objetos estrictamente necesarios. La idea principal es que no es necesario que el objeto de ejecución esté representado siempre por el mismo objeto javaMod todo el tiempo, si no que la funcionalidad sea similar a esta idea pero implementada de forma algo más eficiente. Para conseguirlo se ha implementado un sistema de caché de objetos. En este sistema, el objeto javaMod se mantiene en memoria siempre que esté siendo referenciado por el usuario de la API y siempre que tenga información asociada. En el momento en que el objeto javaMod deje de estar referenciado o no tenga información asociada podrá ser recolectado por el recolector de basura. Si se vuelve a necesitar ese objeto javaMod, se instanciará de nuevo. Desde el punto de vista del usuario de la API, el funcionamiento obtenido con este sistema es similar al obtenido si fuese el mismo objeto, porque siempre que él tenga una referencia, obtendrá el mismo objeto, y si ya no tiene referencias, puede obtener otro objeto javaMod, pero no se notará la diferencia.

Para implementar este sistema de caché se han usado las clases de java `java.lang.ref.WeakReference` y `java.util.WeakHashMap`. Estas dos clases permiten modificar el comportamiento general del recolector de basura de Java. Se mantiene una referencia débil (*weak*) a un objeto siempre y cuando existan referencias normales a ese objeto, pero cuando no exista ninguna, automáticamente la referencia débil se establece a `null` y el objeto es recolectado por el recolector de basura.

6.3 Pruebas

A medida que se avanzaba en el desarrollo se iban realizando constantemente pruebas. Las pruebas consistían en gestionar programas Java correctos que tuvieran la funcionalidad implementada hasta el momento. Esto se repetía iterativamente hasta que se tuviera implementada de forma completa toda la API, momento en el que se comienza a realizar pruebas con programas reales.

Desde un punto de vista de más alto nivel, se ha probado la implementación construyendo un depurador gráfico de Java. En este depurador se han construido representaciones textuales personalizadas del código fuente, mostrando su versatilidad. Además, se han integrado los aspectos de definición y de ejecución mostrando un correcto funcionamiento.

6.4 Instalación y uso de la implementación

Para poder usar la esta implementación de javaMod hay que incluir en el CLASSPATH las siguientes librerías:

- **javaModImpl.jar:** Contiene las clases e interfaces de javaMod y esta implementación.
- **bcel5.1.jar:** Contiene las librerías BCEL necesarias para procesar los ficheros .class y .jar.
- **tools.jar:** Esta librería, situada en el directorio lib del SDK de java de Sun Microsystems, contiene el compilador y la librería JDI.

Las clases de la implementación que deben usarse para cargar el `JavaProgramCode` se encuentran en el paquete `es.urjc.escet.vido.javamod.implementation`. Se utiliza el método estático de la clase `JavaProgramCodeLoader`:

```
public static JavaProgramCode load(
        File sourceDirectory, List classPath, File sdkDirectory)
        throws LoadException
```

7 Conclusiones y trabajos futuros

En este capítulo se evalúan los resultados del trabajo realizado en este Proyecto Fin de Carrera y se divide en dos partes. Por un lado, se enumeran las principales aportaciones de javaMod, tanto en la construcción de sistemas de visualización del software, como en herramientas educativas. Por otro lado, se enumeran los posibles trabajos futuros.

Cabe mencionar que este proyecto se ha publicado como código abierto bajo licencia GPL y se encuentra disponible en la dirección http://vido.escet.urjc.es/javamod/.

7.1 Principales aportaciones de javaMod

Como ha podido verse a lo largo de esta memoria, la gestión de todos los elementos del desarrollo software no es una tarea trivial. La mayor dificultad se observa cuando se tienen que sincronizar los elementos de código y de ejecución. Se han encontrado muchas dificultades durante el desarrollo de javaMod, por tanto, es razonable pensar que proporcionar una API que oculte todos estos detalles al programador, y que ofrezca una interfaz homogénea, integrada y sencilla, facilitará la construcción de herramientas de gestión de programas.

Algunas de las principales aportaciones de este trabajo se han enviado a congresos internacionales. En los siguientes apartados se muestra una breve descripción de las mismas.

7.1.1 javaMod: An Integrated Java Model for Java Software Visualization

Las aportaciones de este trabajo en el campo de la visualización del software se presentaron en el *Third Program Visualization Workshop* (PVW'04), organizado por la Universidad de Warwick. La referencia del artículo es:

Gallego Carrillo, M., Gortázar Bellas, F., Velázquez Iturbide, J.Á.: javaMod: An integrated Java model for Java software visualization. Proc. 3rd Program Visualization Workshop (aceptado).

En este artículo se muestran las características que hacen de javaMod una herramienta para la construcción de aplicaciones de visualización del software. Se definen sus características principales y se muestra, a modo de ejemplo, un prototipo de depurador en el que múltiples visualizaciones se encuentran sincronizadas, tanto las que visualizan código, como las que visualizan los valores de las variables durante la ejecución del programa. En la Figura 16 se muestra la arquitectura del depurador y en la Figura 17 se muestra una captura de pantalla del mismo.

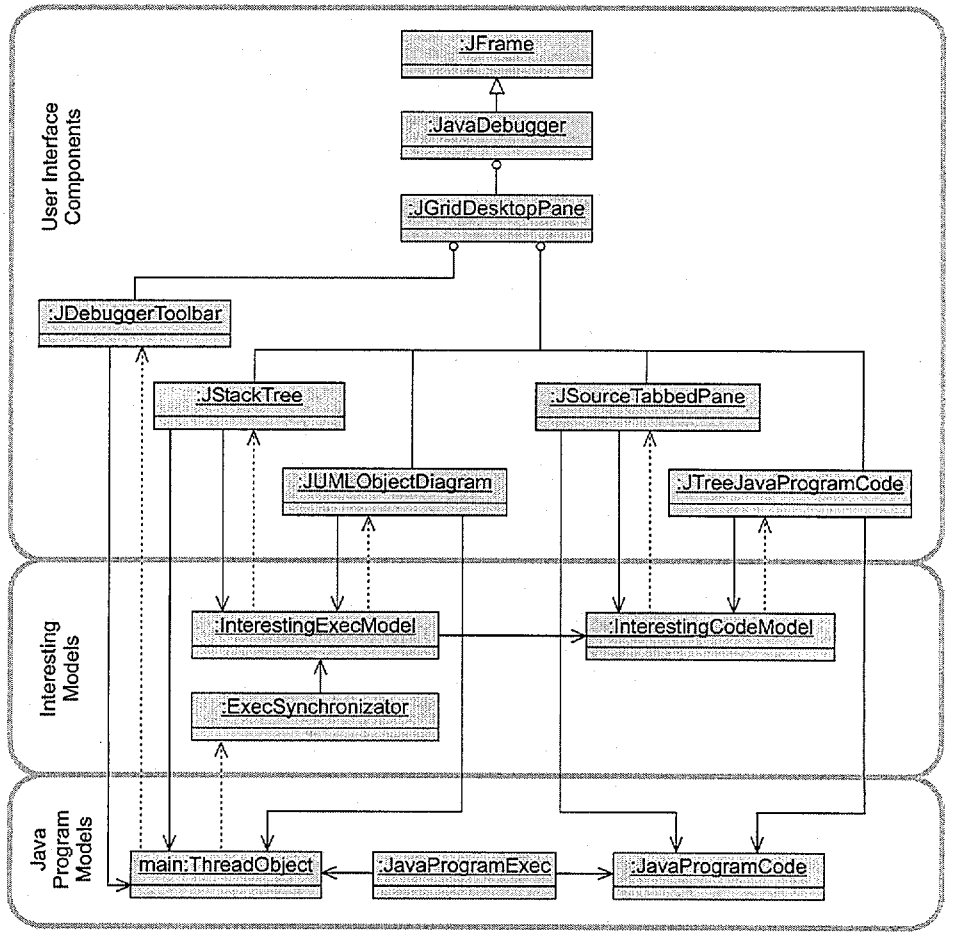

Figura 16. *Arquitectura de la aplicación JavaDebugger usando javaMod*

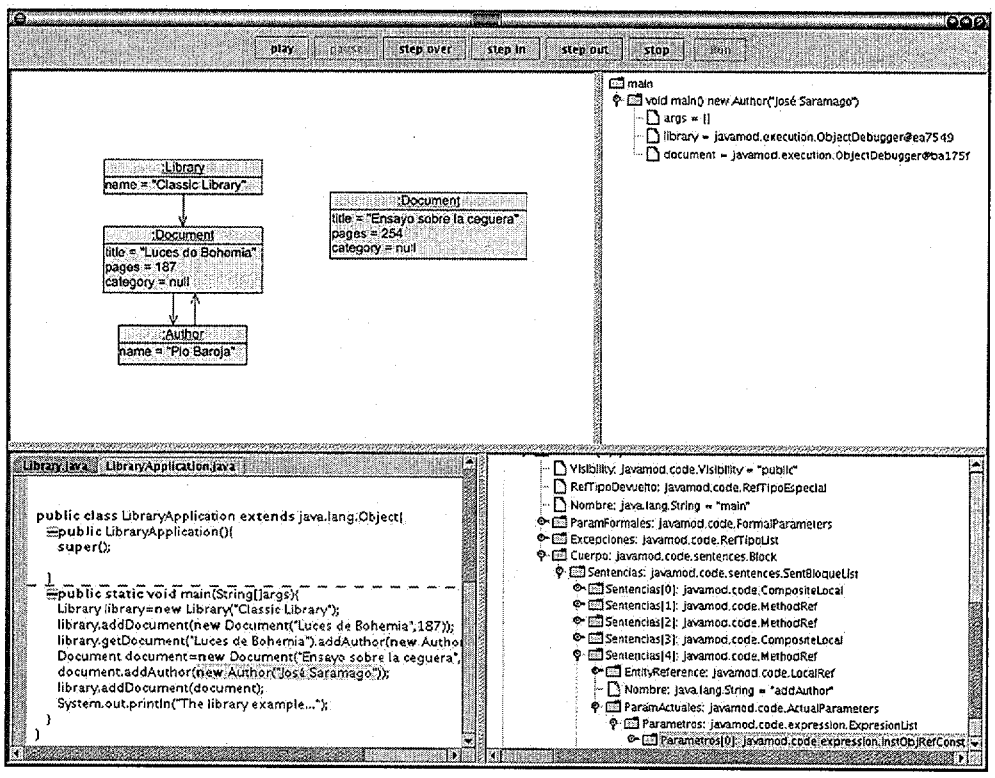

Figura 17. Captura de pantalla de JavaDebugger en la que se muestran las diferentes visualizaciones sincronizadas de los distintos aspectos del programa

7.1.2 Depuración Estructural: Acercando la Práctica a la Teoría de la Programación

Una de las aportaciones de este trabajo en el campo de las herramientas para la enseñanza de la informática se enviaron al 6° Simposio de Informática Educativa (SIIE'04) organizado por la Universidad de Extremadura. La referencia del artículo es:

Gallego-Carrillo M., Gortázar-Bellas F., Velázquez-Iturbide J.Á.: Depuración Estructural: Acercando la Práctica a la Teoría de la Programación. 6° Simposio de Informática Educativa (en proceso de revisión)

En este artículo, se describe la depuración estructural que ofrece javaMod frente a la depuración, basada en líneas de código, que ofrecen la mayoría de los entornos de desarrollo. Esta depuración es beneficiosa para la enseñanza de la programación, ya que los conceptos explicados en las clases de teoría aparecen reflejados en las herramientas de una forma mas clara y directa.

7.2 Trabajos futuros

Por supuesto, tanto la definición de la API javaMod como la implementación por defecto se pueden mejorar en muchos aspectos. Este Proyecto de Fin de Carrera es sólo el comienzo de una serie de proyectos que están siendo desarrollados dentro de ViDo. Dichos proyectos están destinados a la construcción de una API con toda la funcionalidad necesaria para construir cualquier tipo de herramienta que requiera la gestión de programas Java, tanto en definición, como en ejecución del mismo. Los trabajos realizados con esta API pueden servir incluso de base para la definición y creación de APIs similares para otros lenguajes. Por tanto, javaMod seguirá desarrollándose hasta que se consiga proporcionar todas las funcionalidades requeridas. En los siguientes apartados se describen cada una de las líneas por las que va a seguir trabajando en javaMod.

7.2.1 Ampliación de la API para cubrir todos los requisitos

Por supuesto, los siguientes trabajos con javaMod deberían ir encaminados a la ampliación de la API para cubrir todos los requisitos mencionados. La incorporación de estos nuevos requisitos deberá seguir los mismos principios que han condicionado el diseño de la API: deberán estar integrados todos los aspectos relacionados y deberán ocultarse al usuario los detalles de implementación.

7.2.2 Implementación de la API como plugin para los entornos de desarrollo

Uno de los mayores objetivos en la construcción de software es la reutilización, la cual es necesaria para no tener que *reinventar la rueda* cada vez que se necesite introducir alguna funcionalidad adicional. En nuestro caso, no tiene sentido la construcción de un nuevo IDE si queremos construir sistemas de visualización o sistemas de programación visual; lo ideal es utilizar cualquiera de los IDE que existen actualmente, comerciales o de código libre, y desarrollar un plugin para ellos. De esta forma, tan sólo tenemos que desarrollar una mínima parte de los sistemas para conseguir una visualización, o programación visual, en nuestro entono. El problema es que cada uno de estos IDE tiene una API distinta y además, la gestión del código y de la ejecución no están integradas entre sí, lo que dificulta la creación de visualizaciones. Por este motivo, uno de los trabajos futuros que se pueden llevar a cabo consistiría en implementar la API javaMod como un plugin de un entorno de desarrollo, de forma que las aplicaciones que usen javaMod puedan integrarse en dicho entorno de desarrollo con mucha facilidad. En concreto, se está pensando en implementar javaMod como plugin para el entorno de desarrollo de código abierto Eclipse[14].

7.2.3 Adaptación de la API javaMod a la versión del lenguaje incluido en J2SE 1.5

JavaMod es una API diseñada para la versión del lenguaje de programación Java usada en la versión 1.4 de la tecnología Java 2 Standard Edition. No obstante, cuando se escribe este Proyecto de Fin de Carrera la nueva versión 1.5 está en fase beta. Esta nueva versión incorpora modificaciones en el lenguaje de programación, por lo tanto, es muy recomendable incorporar esas nuevas características a javaMod. Entre otras incorporaciones al lenguaje, destacan los *generics*, enumerados, autoboxing/unboxing de tipos primitivos, metadata, for mejorado, etc...

14 http://www.eclipse.org

7.2.4 Incorporación los comentarios convencionales y de JavaDoc a javaMod

JavaMod no tiene soporte para gestionar los comentarios de ninguna forma; éstos simplemente se ignoran. Para construir sistemas de visualización sería muy interesante disponer, en la propia API, de mecanismos para acceder a los comentarios convencionales, así como a los comentarios JavaDoc de los ficheros fuente. Los comentarios JavaDoc pueden ser tratados estructuralmente como se trata el código fuente. La integración de la documentación en javaMod permite que una forma de visualizar un método, en formato reducido, sea mostrar su descripción. Además, se podría integrar la API con la documentación de JavaDoc generada en HTML.

7.2.5 Automatizar la construcción de APIs con los conceptos de javaMod para otros lenguajes

Los principios que se han seguido para la construcción de javaMod se pueden aplicar a la construcción de APIs para manejar los aspectos relacionados con otros lenguajes. La idea es ampliar las capacidades de un generador de analizadores como JavaCC, para que además de generar el analizadores, construya todas las clases necesarias para gestionar ese lenguaje. Esto permitiría que los usuarios de una API como javaMod pudiesen pasar con mucha facilidad a cualquier otra API que siga los mismos principios.

Por otro lado, se podrían construir herramientas que utilizasen cualquier modelo construido con esta herramienta, por ejemplo, un editor con resaltado de sintaxis podría ser usado para editar códigos de cualquier lenguaje.

7.2.6 Detección de patrones de diseño en programas Java

Un programa de tamaño medio, escrito en lenguaje Java, puede estar formado por cientos de clases. En la fase de ejecución, miles de objetos pueden estar en memoria en un momento dado. En programas grandes estas cifras llegan a niveles de órdenes de magnitud superior, siendo necesario utilizar algún tipo de técnica que nos permita reducir el número de elementos de las visualizaciones. La identificación de los patrones software [Gamma95] que se utilizan en un programa ayudará a reducir la cantidad de elementos que se van a visualizar, y permitirá que las representaciones sean semánticamente más ricas, indicando el patrón que se sigue, y ofreciendo una representación del mismo. Es interesante que la API javaMod pueda disponer de rutinas, o utilidades que analicen el código de un programa y la ejecución del mismo, en busca de los patrones de diseño usados.

Estas utilidades pueden ser tan sencillas como buscar métodos en una lista predefinida, o se pueden complicar haciendo uso de detallados análisis del código y la ejecución.

Bibliografía

[Aho90] : Aho, A; Sethi, R; Ullman, J, *Compiladores. Principios, técnicas y herramientas*, 1990

[Beck99] : Beck, K., *eXtreme Programming eXplained*, 1999

[Bloch01] : Bloch, J, *Effective Java Programming Language Guide*, 2001

[Booch99] : Booch, G; Jacobson, I; Rumbaugh, J, *El lenguaje Unificado de Modelado*, 1999

[Eckel98] : Eckel, B, *Thinking in Java*, 1998

[Gamma95] : Gamma, E; Helm, R: Johnson, R; Vlissides, J, *Design Patterns - Elements of Reusable Object-Oriented Software,*, 1995

[Jacobson00] : Jacobson, I; Booch, G; Rumbaugh, J, *El Proceso Unificado de Desarrollo de Software*, 2000

[JavaCC] https://javacc.dev.java.net/

[JPDA] http://java.sun.com/products/jpda/

[Mehner00] : Mehner, K and Wagner, A, *Visualizing the Synchronization of Java-Threads with UML*, 2000

[Myers86] : Myers, B.A., *Visual programming, programming by example, and program visualization: a taxonomy*, 1986

[Stasko98] : John T. Stasko, John B. Domingue, Marc H. Brown and Blaine A. Price, *Software Visualization*, 1998

[Swing] http://java.sun.com/products/jfc/

[Systä00] : Systä, T, *Understanding the Behavior of Java Programs*, 2000