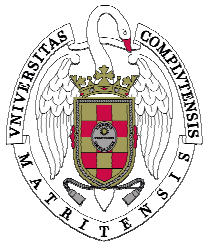


Universidad Complutense de Madrid
Facultad de Informática

Depurador Declarativo de programas JAVA

Memoria de proyecto Sistemas Informáticos

Francisco González-Blanch Rodríguez
Reyes de Miguel Roses
Susana Serrano Soria

Director: Rafael Caballero Roldán

2006

ÍNDICE

RESUMEN DEL PROYECTO

PROJECT SUMMARY

Palabras clave

Depurador, Depurador declarativo, Depuración en Java, SWT, JPDA, JDI, motor de depuración, árbol de cómputo.

Resumen del proyecto

El objetivo del proyecto que presentamos a continuación es el desarrollo de un depurador declarativo para programas que estén implementados en Java. Recoge las ideas de la depuración declarativa para obtener una herramienta intuitiva y de fácil manejo que consiga encontrar los errores en el código de un programa orientado a objetos.

La depuración declarativa es un método de búsqueda de errores en el código de un programa basado en la semántica del lenguaje en el que está escrito dicho código. Es decir, se centra en el significado del código que estamos ejecutando, no en su estructura ni tampoco en la manera en la que está implementado.

La aplicación tiene cómo objetivo el de obtener un árbol de ejecución a partir del cómputo del programa y depurar dicho árbol. En el árbol, los nodos representan las llamadas a funciones que encontramos en el código. El proceso de depuración se lleva a cabo evaluando cada nodo del árbol. Éstos se validan o invalidan en función de si el comportamiento de la llamada asociada es el pretendido o no.

Hemos obtenido una herramienta útil e innovadora que permite depurar programas orientados a objetos y que facilita la incorporación de nuevas estrategias de recorridos de árboles.

Key words

Debugger, Declarative Debugger, Java debugging, SWT, JPDA, JDI, debug engine, execution history.

Project summary

The goal of the project we are presenting in this document is the development of a declarative debugger for programs that are implemented in Java. It gathers the ideas of the declarative debug process to obtain an intuitive and easy managed tool which finds code errors in an object orientated program.

The debugging declarative process is a method to find mistakes on a program. It is based on the semantic of the language. This means that, it looks for the meaning of the code that we are executing, instead of looking for the way that it is implemented.

Our application obtains an execution tree from the program trace, and debugs it. In the tree, the nodes represent the calls to functions that we can find in the code. The declarative debug process consists on evaluating these tree nodes. These nodes are correct or incorrect depending on the result that we obtain from the function call.

Finally, we have implemented a useful and innovative tool that allows debug object orientated programs or even the incorporation of new navigator strategies for trees.

ESTRUCTURA DEL DOCUMENTO

Con este documento pretendemos dar una idea detallada de cómo hemos desarrollado el proyecto. En primer lugar, explicaremos, en el apartado de *Introducción,* la importancia de una herramienta depuradora, el contexto en el que surgieron los depuradores declarativos y algunos de los ejemplos encontrados que implementan de una u otra forma la depuración declarativa. A partir de esto, podemos definir detalladamente la funcionalidad, ventajas e inconvenientes de este tipo de depuradores. Estos puntos, se encuentran en la sección de *Objetivos*.

Una vez se tiene clara la idea del depurador declarativo, podemos pasar a explicar una aplicación concreta que representa esta funcionalidad. Nos centraremos entonces, en el apartado de *Diseño* en el desarrollo y posterior implementación de nuestra herramienta. En este capítulo, abordaremos con detalle en los requisitos que se pretendían con la aplicación así como la forma de conseguirlos, explicando los diferentes pasos que hemos seguido hasta llegar a la versión final. Como en todo proceso de diseño, aparecen problemas en este caso, tanto lógicos como gráficos, que comentaremos en este aprtado.

Al diseño de la aplicación, podrían añadírsele nuevas mejoras que, por falta de tiempo, no han llegado a implementarse pero que se explican con detalle en la sección de *Ampliaciones*. En esta sección, no sólo se comentarán las posibles mejoras sino que se detalla la manera de incluirlas en la aplicación final.

En proyectos de estas características, es importante un apartado de pruebas (Ver *Pruebas de la aplicación*) con las que el usuario podrá comprender mejor el funcionamiento del depurador.

Concluiremos haciendo un análisis de la utilidad de la aplicación (*Conclusiones y resultados*) de forma que el usuario tenga una visión general de la finalidad del proyecto.

En toda aplicación orientada al usuario, es necesario un manual de uso que le explique como realizar cada una de las acciones posibles por ejemplo, iniciar una depuración, cambiar el estado de un nodo, etc (Ver apartado de *Manual de Uso*).

INTRODUCCIÓN

En este apartado se intenta hacer una revisión de los depuradores a lo largo de la historia. Se analiza el porqué de los depuradores declarativos, y se citan brevemente sus ventajas frente a los depuradores tradicionales. A continuación se habla de los depuradores en nuestros días, y se describen aquellos depuradores de los que hemos recogido ideas para llevar a cabo el desarrollo de nuestra herramienta.

La depuración ha supuesto siempre una parte muy costosa del desarrollo del software. La técnica de **depuración declarativa**, introducida por E. Y. Shapiro [0], establece la primera aproximación teórica a la depuración de programas. Un método para localizar errores en programas lógicos basado en la semántica declarativa de estos lenguajes. En esta técnica, el programador aporta parte de la especificación del programa durante el proceso de localización del error, mediante la respuesta a preguntas que le hace el sistema de depuración.

Podemos describir a grandes rasgos el proceso de depuración propuesto en este trabajo, ya que contiene gran parte de las ideas claves de la actual depuración. La depuración comienza cuando el usuario observa que un programa lógico produce un resultado inesperado como resultado del cómputo de un objetivo. El depurador entonces repite el cómputo paso a paso, pero preguntando (normalmente al usuario) si los resultados intermedios obtenidos en cada paso son los esperados. De esta forma, comparando el modelo representado por el programa con el conocido se puede localizar el error. Dicho modelo pretendido es, en caso del trabajo de Shapiro, un subconjunto de la base de Herbrand, es decir un conjunto de átomos sin variables.

Posteriores trabajos de G. Ferrand y J.W.Lloyd continúan y amplían la línea propuesta por Shapiro. En G. Ferrand se extiende la noción de modelo pretendido a conjuntos de átomos con variables, mientras que en sus trabajos J.W.Lloyd en lugar de tratar con programas formados por cláusulas de Hörn considera programas lógicos más generales formados por fórmulas con un átomo y una fórmula de primer orden general.

Una generalización de la depuración declarativa descrita por Pereira (1986) [1] es la denominada “Rational Debugging” (depuración racional). En esta última se pierde la propiedad que caracteriza a la depuración declarativa, debido a que las preguntas que se le hacen al usuario no sólo están referidas a los valores de entrada y salida de las llamadas a los métodos y procedimientos del programa.

Una desventaja de la depuración declarativa, son las múltiples interacciones con el usuario a lo largo del proceso de depuración. Así, una importante mejora consistiría proveer al sistema de depuración de cierta información, con el fin de reducir este el número de preguntas.

En resumen, la depuración declarativa definida en un principio por Shapiro es un proceso interactivo, en el que el sistema de depuración obtiene información acerca de lo que se espera en la ejecución de un programa, y utiliza esta información para localizar errores. Esta información se adquiere, como dijimos anteriormente a partir de una serie de preguntas de respuesta si/no que se le hacen al usuario.

En 1988, es Drabent [2], quien introduce una generalización en el lenguaje que el depurador utiliza para comunicarse con el usuario. Además de las preguntas de respuesta si/no, introduce afirmaciones. Estas proporcionan una especificación más formal acerca del programa que se está depurando. Estas especificaciones pueden ser programas lógicos, y con ellas se consigue reducir, considerablemente el número de preguntas que se le hacen al usuario.

La depuración declarativa en nuestros días

La localización de errores en cualquier aplicación que estemos desarrollando, parece algo inevitable a la hora de desarrollar software. Las técnicas que han aparecido a lo largo de la historia, intentan reducir el tiempo y esfuerzo empleado en el proceso de depuración.

La depuración tradicional está basada en la ejecución paso a paso del programa que se está depurando, a nivel de instrucción. Sin embargo, esta técnica no funciona bien para lenguajes de alto nivel como por ejemplo los lenguajes de programación lógica, en los que la secuencia de ejecución paso a paso puede llegar a ser muy compleja. Además, este tipo de programas, no sólo especifican qué se está representando en cada paso de ejecución, sino también la lógica que existe detrás de cada paso. Esto último es sobre lo que se basa la depuración declarativa.

A continuación vemos una breve introducción acerca de los depuradores que se utilizan hoy en día, que incluye tanto los tradicionales como los declarativos. El motivo de esta pequeña descripción se debe a que el depurador que hemos implementado, recoge características de varios de ellos.

Un ejemplo de depurador declarativo es el implementado para el lenguaje Haskell 98, conocido como **buddha** [3]. Buddha es un depurador declarativo que presenta la evaluación de un programa escrito en Haskell como una serie de equivalencias, en lugar de la tradicional presentación de pasos de ejecución, propia de los lenguajes imperativos. Una sesión de depuración típica involucra un número de preguntas y respuestas. Las preguntas las hace el depurador y las respuestas la da el usuario. Las preguntas están destinadas al fin de evaluar el funcionamiento de la función durante la ejecución del programa. La evaluación se presenta de la siguiente forma:

```
Prelude 35 map
arg 1  = fst
arg 2  = [(True,1),(False,2)]
result = [True,False]
```

El significado estas instrucciones es el siguiente. Aplicamos la función map a la función fst y [(True,1),(False,2)], y el resultado de la evaluación es [True,False] . La línea "Prelude 35 " nos indica la línea en la que está definida la función map en el módulo Prelude. El usuario es ahora quien debe determinar si el resultado es el esperado o no, para ello hemos de suponer que el usuario conoce el comportamiento de antemano.

En este depurador, el programa que se depura es "transformado" y compilado. Esta transformación consiste en la aplicación de una serie de reglas sobre el código depurado de tal manera que el "programa transformado" constituye la evaluación del original más un árbol cuyos nodos contienen la misma información que la que acabamos de presentar. La depuración consiste en recorrer ese árbol, encontrando aquellos nodos cuyo resultado no coincida con el esperado.

La ventaja de este depurador radica en que el usuario no tiene que pensar en cómo está implementado su programa, simplemente debe pensar en términos de la especificación lógica (semántica declarativa). Algo especialmente útil en lenguajes de programación como Haskell.

El depurador que hemos implementado recoge alguna de estas ideas. Una de ellas es la de representar la ejecución del programa en forma de árbol. Además, la información que recopilamos en los nodos del árbol es prácticamente la misma. La diferencia fundamental con nuestro depurador es, al estar hecho para programas escritos en Java, debe mostrar información adicional como por ejemplo la del objeto invocador del método. Igualmente, consideramos útil que el depurador indique si la llamada al método ha modificado alguno de sus argumentos.

Hasta ahora, no hemos encontrado ningún depurador declarativo que esté pensado para lenguajes de programación orientados a objetos. A mitad de desarrollo de nuestra aplicación descubrimos que no estábamos solos. Un grupo del departamento de Informática e Ingeniería de la universidad de Búfalo en Nueva York [4], ha implementado una herramienta similar a la nuestra. En este proyecto la depuración declarativa se basa en un catálogo de consultas almacenado en una base de datos. Las entradas de esta base de datos corresponden a eventos de ejecución. La figura 1 define las relaciones básicas presentas en el esquema de la base de datos.

Relation	Fields
methodcall	location, thread, instance-class, name, arguments
methodexit	location, thread, instance-class, name, returned-value
setfield	location, thread, instance-class, name, value
getfield	location, thread, instance-class, name
datastructure	location, thread, contents
exception	location, thread, instance, message, caught-uncaught
step	location, thread, variables
classprepare	thread, class, member-fields
threadstart	thread, thread-group
threaddeath	thread, thread-group

Figura 1

El depurador de Java JDB, es un depurador de línea de comandos, similar al que Sun proporciona en sus Sistemas, dbx. Es complicado de utilizar y un tanto críptico, por lo que, en principio, tiene escasa practicidad y es necesaria una verdadera emergencia para tener que recurrir a él. Sin embargo, con el fin de dotar a nuestra herramienta con la mayor funcionalidad posible, decidimos inspirarnos en éste para implementar nuestro depurador de línea de comandos. Muchos de los comandos que podemos encontrar en el JDB aparecen también en nuestro depurador (por ejemplo: where, list, help..etc), aunque tienen distinto significado.

Para determinar la ejecución de los eventos que se dan durante la ejecución de un programa tales como la llamada a un método, el valor que devuelve, la creación de un objeto, etc, hemos utilizado los métodos que proporciona la interfaz de la **JPDA** (Java Platform Debugger Architecture).

Esta consiste, a su vez, en tres interfaces diseñadas para ser utilizadas por los depuradores en los entornos de desarrollo. Estas interfaces son la **JVMI** (Java Virtual Machine Interface), que define los servicios que la máquina virtual de Java debe ofrecer al proceso de depuración. La **JDWP** (Java Debug Wire Protocol) que define el formato de información y la transferencia de peticiones entre el proceso que se está depurando y el front end del depurador, que está implementado por la **JDI** (Java Debugger Interface). Y, por último, la JDI, que define la información y peticiones del el usuario a nivel de código. Es una API de Java con información útil para depuradores y sistemas similares que necesitan tener acceso al estado de ejecución de la máquina virtual.

Alguno de los problemas que se nos presentaron a la hora del desarrollo de nuestra herramienta, fue la obtención de valor de retorno de un método. En la interfaz JDI que estábamos utilizando, no figuraba el método que nos permitía obtener el valor de retorno de un método, para ello tuvimos que utilizar una versión de Java (Java 1.6, únicamente disponible en la versión beta). Este

problema lo explicamos más adelante en el apartado de problemas de este documento.

Por último, hemos de referirnos al depurador **DDD** (Data Display Debugger). El GNU DDD presenta un front-end gráfico para depuradores de líneas de comandos como pueden ser el GDB, DBX, WDB…etc. La característica fundamental de este tipo de depurador consiste en disponer de una representación gráfica para estructuras complejas (estructuras de datos) mediante un grafo. Imaginemos que el objeto que queremos evaluar estructura de datos compleja, por ejemplo una lista. Cada nodo de la lista dispone de referencias al nodo siguiente en la lista, además de una así mismo. Esta idea se ilustra en la siguiente figura.

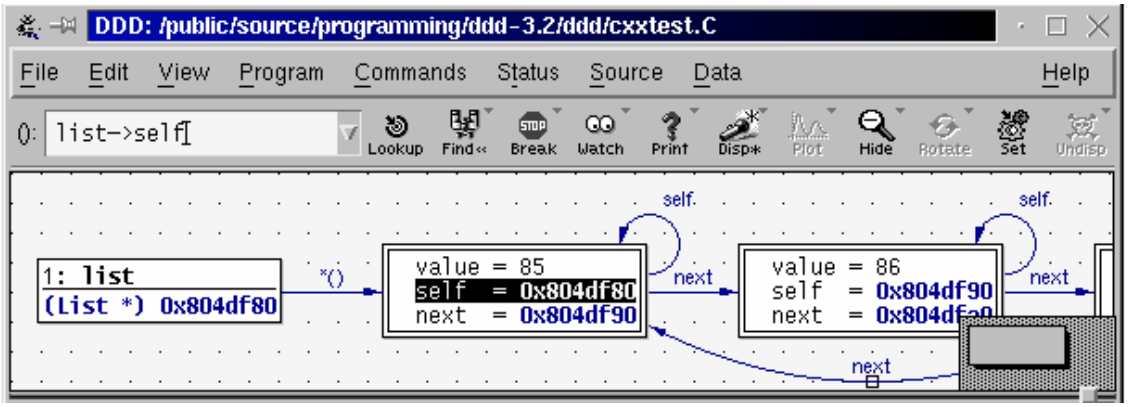


Figura 2

Como muestra la imagen, de cada nodo sale una flecha apuntando al siguiente nodo de la lista (next) y otra apuntando así mismo (self). Este tipo de representación resulta mucho más clara que si optásemos por un árbol. Por ello, como posible ampliación de nuestro proyecto proponemos adoptar un tipo de visualización similar basada en un grafo (ver apartado de ampliaciones de este documento).

OBJETIVOS

¿Qué es un depurador declarativo?

El objetivo de cualquier depurador, es ayudar al desarrollador a encontrar la parte del código responsable de un error. El depurador declarativo pretende facilitar dicha tarea comparando el comportamiento real del programa con la semántica pretendida de éste.

Para ello, al ejecutar el programa creará un árbol de ejecución que almacene toda la información relevante de los métodos: llamadas a otros métodos, valor de argumentos, el valor que devuelven, etc. De esta forma, el usuario podrá ir validando la corrección de cada llamada a un método. Con esto, se consigue localizar el método concreto que produce el error o uno de los errores. Si después de revisar el método en cuestión el programa sigue dando un resultado erróneo, se ejecutará de nuevo el depurador para encontrar un nuevo fallo.

El depurador declarativo permite ahorrar trabajo al desarrollador puesto que no es necesario que se revisen todas las líneas de código sino sólo en las pertenecientes al método causante del error. En caso de partir de un programa de miles de líneas de código, esta estrategia facilita el trabajo considerablemente. De ahí la importancia de este tipo de depuradores ya que el hecho de que sea declarativo implica que el usuario sólo debe conocer el significado de cada método y no el cómo está implementado.

Esta aplicación concreta basada en la técnica de depuración declarativa está desarrollada en Java aprovechando las posibilidades que ofrece este lenguaje para la ejecución de programas, como la Java Platform Debugger Architecture (JPDA) para la repetición del cómputo y la generación del árbol.

¿Cómo funciona?

Cuando tenemos un programa que compila (su sintaxis es correcta) pero no se comporta adecuadamente (no devuelve el valor esperado) debemos depurarlo para encontrar el código responsable del fallo. Es en este momento cuando se invoca al depurador.

Lo primero que hace el depurador declarativo es ejecutar el programa. Al hacerlo va anotando internamente qué funciones o métodos se han llamado, así como los atributos y valores de cada uno de ellos. Con toda esta información se construye un árbol, que representa la ejecución del programa erróneo. Cada nodo, se corresponde con la llamada a un método o función y contendrá el valor de sus parámetros (si los tiene) así como el valor que devuelve. Si el método no devuelve ningún resultado, modificará un argumento o un atributo del objeto en que se encuentran por lo que el nodo también

recoge información de estos cambios de estado, anotando tanto el valor anterior como el posterior a la llamada de los argumentos o atributos modificados.

Los nodos estarán unidos mediante relaciones padre-hijo, donde un nodo Y es hijo de un nodo X, si el método representado por X hace una llamada al método Y.

Una vez construido el árbol, el depurador lo presentará al usuario mediante un interfaz gráfico. Dicho interfaz, consta de un navegador que recorrerá el árbol preguntando al usuario por el estado de cada nodo, de forma que se vayan validando cada uno de los nodos. Cuando hablamos de estado, nos referimos a las 4 posibles situaciones en las que puede encontrarse un nodo depurado. Si la llamada al método es correcta, se marcará el nodo como válido; de lo contrario, de marcará como no válido. Si el usuario no sabe si la llamada es correcta, puede ponerlo como desconocido. Y ya por último, si el usuario "confía" en el nodo, es decir, lo quiere marcar como correcto y además, cree que son correctas las llamadas que se hagan al mismo método, marcará el nodo como "de confianza".

Se considera que se ha localizado el método asociado a un nodo que es el causante del error cuando se producela siguiente situación:

- Un nodo padre es marcado como no válido y todos sus descendientes están marcados como válidos. Un caso particular de esta situación, aparece cuando un nodo hoja (sin hijos) se marca como no válido. Esta situación, equivalente a la anterior, también concluye con la localización del error.

Si se produce esta situación, se detiene la ejecución del depurador y se informa al usuario sobre qué método debe revisar para encontrar un fallo del programa. Si una vez resuelto el error indicado el programa sigue sin comportarse como se espera, se ejecutará de nuevo el depurador creando un nuevo árbol de ejecución.

¿Por qué es útil?

La principal ventaja del depurador declarativo es su eficiencia. Se reduce considerablemente el espacio de búsqueda del error ya que se validan métodos enteros sin necesidad de recorrer cada una de las líneas de código que lo componen. Así conseguimos reducir el tiempo empleado en depurar un programa.

La representación del programa ejecutado en forma de árbol, nos permite aplicar sobre éste las distintas estrategias de búsqueda. Por ejemplo, si aplicáramos el algoritmo de "Divide y vencerás", el espacio de búsqueda se reduciría, como mínimo a la mitad del árbol tras cada pregunta al usuario. Esto supone menor coste en tiempo a la hora de encontrar el método erróneo, ya que una mitad del árbol ha sido podada.

Las diferentes estrategias de búsqueda hacen que el número de llamadas a métodos que hay que examinar sea menor que en un depurador de traza.

Además, este tipo de depurador permite al usuario concentrarse en el significado lógico del programa dejando aparte los detalles de implementación.

¿Qué ventajas tiene frente a otros depuradores?

La depuración de traza se basa en un recorrido paso-a-paso y objeto-a-objeto del programa para descubrir el origen del error. Para facilitar este proceso, en las últimas décadas, se han desarrollado técnicas y herramientas como es el caso de los break points (puntos de ruptura), object inspectors, etc. Algunas plataformas como Eclipse, NetBeans y Visual Studio también proporcionan estas posibilidades.

En la mayoría de los casos los programas constan de miles de líneas, por lo que un método de depuración como el que acabamos de explicar puede resultar tedioso si para encontrar un error hay que recorrer una gran cantidad de métodos.

Frente a esto, la depuración declarativa propone reducir el tiempo empleado en la depuración basando la corrección de un programa en la corrección de cada uno de sus métodos. Este tipo de depuración, crea un historial de la ejecución que se almacena, en nuestro caso, en una estructura de árbol. Dicha estructura permite al programador tener una visión general de la estructura de su código, ya que a simple vista podemos conocer las llamadas que hace cada método. Además, el uso de un navegador que guíe al usuario en la depuración reduce el número de preguntas necesarias para la localización del error.

¿Qué desventajas tiene?

Frente a todas las ventajas enunciadas, el uso del depurador declarativo presenta las desventajas siguientes:

• Si el programa tiene una gran cantidad de métodos, el tamaño del árbol de ejecución puede ser considerable, con lo que también aumenta el tiempo requerido para validar todas las llamadas a los métodos.

• Como a la hora de depurar debemos validar llamadas a métodos a partir de la información mostrada, puede que esta sea insuficiente para determinar la corrección o no de un nodo. Esto aumenta la complejidad para determinar la validez de algunos nodos.

Sin embargo, estrategias como implementar diferentes técnicas de búsqueda contrarrestan la primera de las desventajas ya que usando la técnica adecuada en cada ocasión el árbol puede reducirse considerablemente. Con esto concluimos que las ventajas se imponen en este tipo de depuradores, por lo que son de gran utilidad.

DISEÑO

Comentaremos a continuación el proceso de desarrollo de nuestro proyecto, centrándonos, principalmente, en su especificación, diseño y posterior implementación.

Una visión general

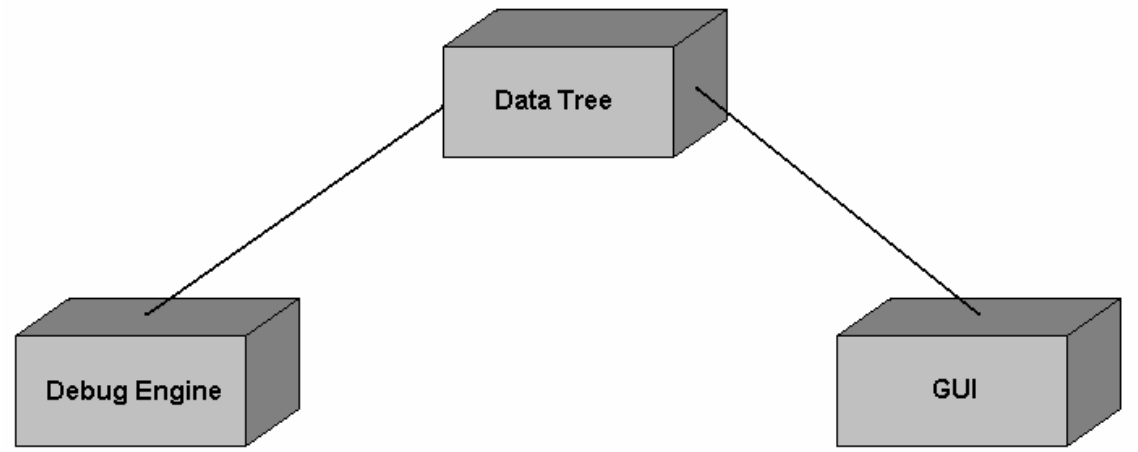
Intentando conseguir la mayor modularidad posible en el proyecto, definimos dos módulos independientes, lo que además permite el desarrollo en paralelo. Así podemos distinguir entre: el **módulo gráfico** y el **módulo lógico**.

El módulo gráfico, abarca la representación gráfica del depurador declarativo, es decir, el aspecto que muestra el depurador ante el usuario.

El módulo lógico, implementa el depurador en sí, tanto el motor de depuración como la estructura de datos general que luego será representada por el módulo gráfico. Implementa a su vez la navegación por los nodos del árbol de depuración así como la interfaz de usuario en modo texto (consola).

Ambos están relacionados, siendo uno el que representa gráficamente al otro. Por ello, es necesario definir una interfaz común a los dos, que facilite la posterior integración de ambas partes sin demasiados problemas. La estructura que se definió era completada por el módulo lógico, construida a partir del árbol de ejecución y visualizada por el módulo gráfico.

A partir de esto, decidimos dividir la aplicación en distintos módulos, que a la hora de implementar se convirtieron en paquetes. Cada uno de los cuales incluía clases con funcionalidades similares. El esquema que aparece a continuación muestra esta estructura:



Para poder representar la información relevante en un nodo y útil para el proceso de depuración, como es: el nombre del método, sus argumentos de entrada, el valor de retorno o el objeto invocador, existe el paquete **dataTree.**

El paquete **gui,** es el que incluye las clases que implementan las funcionalidades del módulo gráfico. Las distintas interfaces gráficas y sus componentes (árbol gráfico con sus nodos gráficos, el object inspector, la barra de herramientas, etc)

Por último el paquete **debugEngine**, consiste en el motor de depuración y es el encargado de ejecutar los programas que se quieran depurar, y extraer de ellos la información necesaria para la depuración, la información obtenida de la ejecución será almacenada por el paquete **dataTree**, Esto es, en cierta manera se encarga de rellenar las estructuras de datos del **dataTree.**

Módulo lógico. Desarrollo

1.- Especificación

El módulo lógico es el encargado de implementar tanto el motor de depuración como la estructura de datos básica sobre la que se desarrollarán las depuraciones.

1.1.- Requisitos de la aplicación

Como primer requisito fundamental, y dado que este módulo debe realizar dos funciones completamente disjuntas, el módulo debe estar comprendido en dos partes:

- El motor de depuración (paquete debugEngine): Encargado de recuperar el programa java a depurar, compilarlo y extraer la información necesaria para la depuración. Al ser un depurador off-line, toda la información extraída debe ser persistente durante toda la sesión de depuración, es ahí donde entra en juego la segunda parte.

- Estructura de Datos (paquete dataTree): En ella incluiremos toda la información referente a las llamadas a los métodos y las variables visibles en cada momento.

Ahora pasaremos a describir los requisitos de cada una de las partes de manera detallada, empezaremos por el motor de depuración:

- El motor de depuración debe ser capaz de admitir un fichero de código fuente java compilarlo y ejecutarlo, según un directorio de clases (classpath) determinado, y recoger a partir de ahí toda la secuencia de acciones efectuadas durante la ejecución, almacenándolas en la base de datos.

- El motor debe rellenar la estructura de datos con la información necesaria para la depuración al tiempo que ejecuta el programa a depurar (depuración off-line).

- El usuario debe poder filtrar la información que no quiera que aparezca en la depuración, para ello, excluirá los paquetes que el considere correctos.

- El motor debe ofrecer a los programadores una interfaz de control de la depuración, proporcionando métodos de control referentes a los posibles comandos del depurador.

- El motor debe proporcionar una interfaz básica de usuario en modo texto.

- El motor debe ser independiente del modulo grafico, pero no de la estructura de datos, que será la interfaz común entre el motor de depuración y el modulo grafico.

Los requisitos para la estructura de datos son:

- La estructura de datos contendrá de forma eficiente la información necesaria a lo largo de la sesión de depuración.

- La estructura de datos ha de ser única y persistente durante la ejecución de un programa.

- Al ser el puente común entre el módulo gráfico y el módulo lógico, debe ser accesible tanto por el motor de depuración como por el módulo gráfico, posiblemente en algunos casos concurrentemente.

- Los datos conjuntamente con cierta información del módulo gráfico, deben ser serializables para permitir almacenar (y posteriormente recuperar) de forma persistente.

A partir de estos requisitos básicos, pasamos a explicar el diseño e implementación de los módulos que más correcta y eficientemente se adapta a ellos.

2.- Diseño e implementación

En este apartado explicaremos como ha sido el proceso de diseño y posterior implementación del módulo lógico.

Como se dijo en la especificación, este módulo está dividido en dos partes, el motor de depuración y la estructura de datos. Que, aún siendo separadas gran parte del diseño es común a las dos.

Primero describiremos la estructura de datos (DataTree) pues es la parte sobre la que se sustenta el resto.

2.1.- La Estructura de Datos (paquete dataTree)

El paquete dataTree representa a la estructura de datos donde se almacena toda la información de la sesión de depuración, que se este llevando a cabo en ese momento.

Dado que nuestro depurador es un depurador declarativo, toda esta información será la referente a las llamadas a métodos a lo largo de la ejecución del programa, así como toda la información del entorno en el momento de realizar estas llamadas.

Esta información ha de ser almacenada persistentemente durante la sesión de depuración dado que nuestro depurador es un depurador off-line, esto es, en lugar de depurar al mismo tiempo que se ejecuta el programa, la depuración se lleva a cabo una vez el programa haya terminado, o haya ocurrido algún error parando la ejecución. De tal manera que debemos almacenar toda la secuencia de ejecución del programa a depurar.

En cierta manera podríamos decir que la labor del depurador es almacenar la traza de ejecución del programa a través de las llamadas a los métodos, para después ser procesada y así encontrar el error en el código.

A partir de estas premisas, decidimos cual podría ser el tipo de nuestra estructura de datos, de tal manera que fuese correcta y eficiente, eligiendo al final una estructura de datos en forma de árbol. Nuestra elección se basa principalmente en dos razones. Por una parte, por su similitud con los datos a representar (un programa Java puede ser considerado como un árbol de llamadas entre métodos). Y por otra, la eficiencia que ofrece a la hora de recorrer la estructura, en busca de los errores del programa a depurar. Así como la cantidad de recorridos del árbol posibles, tanto los básicos (pre-orden, in-orden, anchura…) como otros más sofisticados (estrategias de divide y vencerás).

Otra decisión importante fue la de aplicar un par de patrones de diseño sobre la estructura. Teniendo en cuenta que la estructura tiene que ser común a todos los módulos del programa, decidimos aplicar una pequeña variación del patrón singleton, de tal manera que todos los módulos actúen sobre la misma instancia de los datos. Y por otra parte, dado que las operaciones sobre los datos van a ser comunes, también utilizamos el patrón fachada, de tal manera que una clase (DataTree) hace de interfaz de acceso a los datos, para el resto de clases.

Pasamos entonces a describir de manera concreta, la implementación del paquete dataTree.

Para la implementación de la estructura, se optó por una estructura de datos de tipo árbol. Antes de empezar a codificar la estructura, sopesamos las opciones que nos daba la API de Java a la hora de implementar una estructura similar. Después de ver las opciones disponibles optamos por la clase DefaultMutableTreeNode dentro del paquete javax.swing.tree.

Esta clase nos ofrece la implementación de una estructura de tipo árbol, además de diversas funciones útiles para su posterior recorrido.

Para terminar de establecer el nivel de abstracción entre la estructura y los datos que nos interesan, hicimos que la información a almacenar en los nodos heredase de esta clase DefaultMutableTreeNode consiguiendo de esta manera la abstracción que nos interesaba.

Bajo esta decisión y tal como vemos en la figura 1, el paquete dataTree acabó estando compuesto por tres clases:

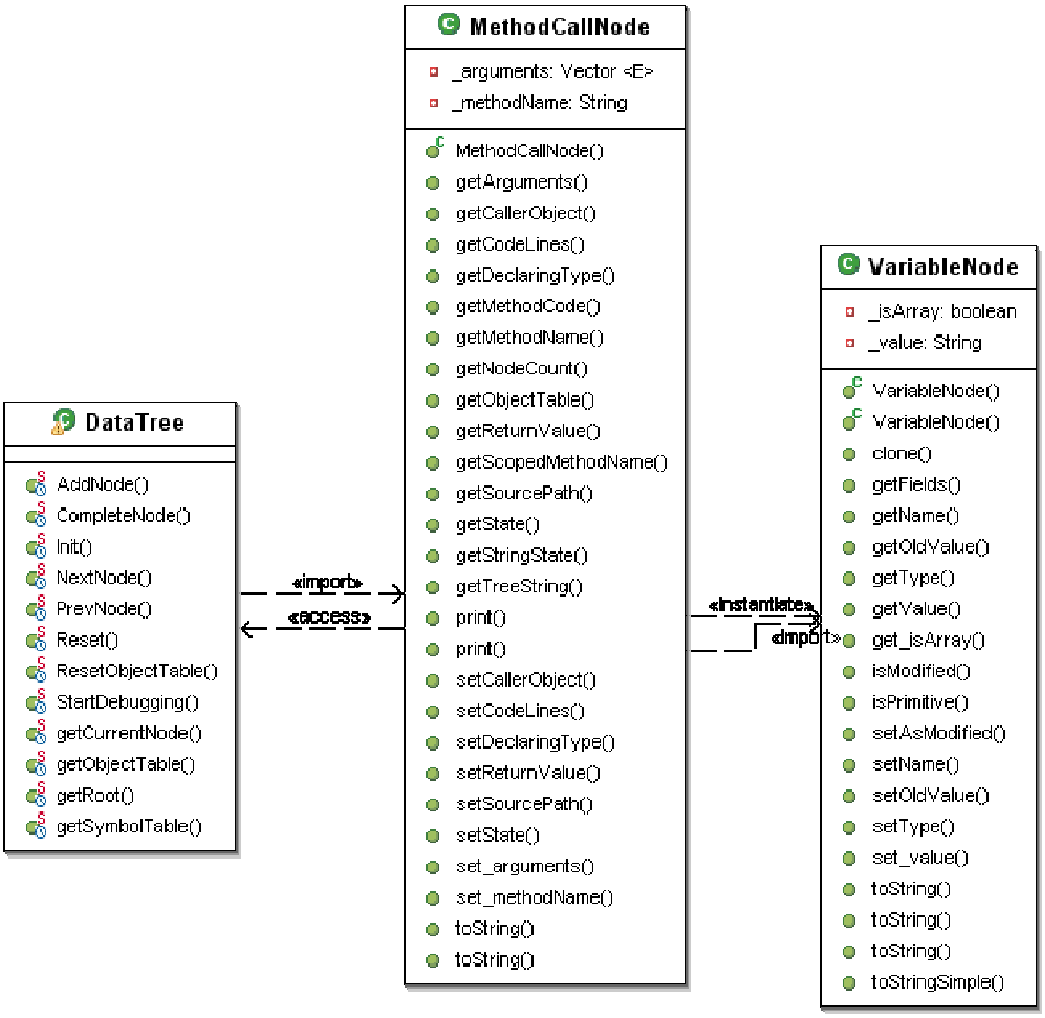


Figura 1: Diagrama de clases del paquete DataTree

2.1.1.- La Clase MethodCallNode

La primera clase que explicaremos es la clase MethodCallNode. Cada objeto de esta clase representa una invocación a un método durante la ejecución del programa. Es por ello que dentro del depurador declarativo podríamos considerarla como la unidad de datos básica. Además es la clase que oculta por debajo la estructura de datos real, pues es la clase que hereda de DefaultMutableTreeNode y por tanto implementa los nodos del árbol.

En ella almacenamos toda la información referente a la invocación de un método. Esta información que en un principio podría resultar trivial, es sin

embargo, una de las decisiones más importantes y que más ha ido cambiando a lo largo del proyecto.

En un primer momento, consideramos que la información suficiente sería la siguiente:

- Los argumentos con los que se llama al método.
- Los argumentos modificados después de la ejecución del método y
- El valor de retorno del método

Sin embargo, nos equivocamos en diversos puntos.

Por una parte algo bastante importante que se nos pasó, fue el objeto invocador. El objeto invocador es como decimos importante por varias razones: Una es que, como su propio nombre indica, invoca al método, así que salvo con los métodos estáticos, es el que nos da toda la información del entorno de la invocación conjuntamente con los argumentos. Y la otra razón, es que además y debido al paradigma de la programación orientada a objetos, suele (con bastante probabilidad) ser el elemento que cambia después de la invocación al método. Buena muestra de ello son los métodos mutadores, prácticamente omnipresentes en todos los programas.

Otra mala elección en el diseño que posteriormente fue subsanada, fue la de distinguir entre argumentos y argumentos modificados como elementos diferentes. Básicamente los argumentos en Java no deberían poder ser modificados, además, por otra parte, si algún objeto (argumento u objeto invocador) ha sido modificado, consideramos responsabilidad de la clase que implemente la representación del objeto (en nuestro proyecto la clase VariableNode) de indicar que ese objeto ha sido modificado; aunque ese es un punto que explicaremos cuando expliquemos la clase VariableNode más adelante.

Por último, otro elemento importante, y que en nuestro prototipo no ha sido implementado pero si contemplado (contenido en las ampliaciones) es el de las excepciones. Si dentro del código de un método se lanza una excepción, el objeto que represente a esa excepción debe estar representado al igual que el valor de retorno o los argumentos dado que nos ofrece también información sobre el entorno de la invocación en ese instante.

Como hemos dicho antes, los nodos de la estructura de datos es de las cosas que más ha evolucionado a lo largo del desarrollo del proyecto, en parte por la constante aparición de nuevas necesidades.

Finalmente los miembros en la versión definitiva son los siguientes:

- El objeto invocador: sobre el que se realiza la invocación.

- Los parámetros de la invocación definidos en el prototipo del método
- El valor de retorno del método si lo tiene.

Aparte, también encontramos información adicional, que, aún no siendo imprescindible, es útil para la depuración. Información útil como es el fragmento de código fuente del método que representa el nodo así como el estado de depuración del nodo durante la sesión de depuración (el sentido y funcionamiento del estado, es algo que se explicará en el modulo grafico).

2.1.2.- La Clase VariableNode

La siguiente clase que nos encontramos dentro del paquete DataTree es la clase VariableNode. Esta clase ha sido sin lugar la más complicada a definir, o más bien encontrar un diseño óptimo que se adaptase a nuestras necesidades.

El requisito imprescindible de que el depurador sea off-line, implica que además de la traza de los métodos por los que pasa la depuración, necesitemos almacenar el entorno de las invocaciones, y esto implica a su vez almacenar el valor de las variables visibles en el momento de la ejecución así como cuidar de que esa información sea correcta y corresponda a ese momento justo, dado que el valor de estas variables varía entre invocación e invocación.

Por otra parte, otro gran problema es el de cómo almacenar toda la información de esas variables, teniendo en cuenta que el entorno de depuración de Java JPDA, está orientado a la depuración on-line y no sólo no trabaja con los datos “absolutos” (realmente trabaja con referencias (copias espejo) de lo que esta pasando en la máquina virtual) sino que además, no nos proporciona esos datos de manera concreta sino a través de interfaces.

Es por ello por lo que nos vimos en la necesidad de crear una clase que meta-representase esas variables, y las convirtiera en persistentes durante la sesión de depuración. Pudiendo hacer así que nuestro depurador sea off-line aún basado en una arquitectura de depuración on-line como es JPDA.

En una primera aproximación, la meta-representación de las variables consistía en tratar como cadenas de texto el nombre de la variable, su tipo y su valor. Esta estrategia presentaba sin embargo, varios inconvenientes:

- Esta representación era hasta cierto punto correcta para los tipos básicos, en cambio si la variable correspondía a un objeto o un vector/matriz, esta interpretación no era suficiente.

Para solventarlo dentro de la misma clase hemos hecho distinción si la variable corresponde a un tipo básico o, por el contrario, es un objeto (que es la opción más común en el lenguaje Java) Y una vez dentro de esta última opción, si corresponde a un vector/matriz (array) o si es un objeto propiamente dicho. Para meta-representar estos tipos, dentro de cada VariableNode encontramos un vector que en el caso de los vectores/matrices incluye a sus elementos representados por otros objetos del tipo VariableNode, y en el caso de que sea un objeto propiamente dicho ese vector incluirá también objetos del tipo VariableNode que representen a los atributos del objeto. Estos últimos, obtenidos haciendo uso de la API de **Reflection** de Java, que nos permite hacer introspección sobre los objetos sin saber el tipo hasta no estar en tiempo de ejecución.

Ya sólo quedaba resolver otro gran problema, y es si un objeto o vector/matriz incluye referencias cruzadas con otros objetos, o de igual manera una referencia a él mismo. Cuando esto ocurría, nuestra aplicación entraba en un bucle infinito de inclusiones.

Este problema entrañó una gran dificultad debido a la inexistencia de punteros en Java, y por tanto no saber como identificar si algún objeto está en el ámbito actual en un momento dado. Y así referenciarlo y no copiarlo, que es lo que se estaba haciendo. La solución final fue la de incluir en cada nodo de invocación a método (MethodCallNode) una tabla de referencias, donde a medida que algún objeto era meta-representado en el ámbito del método, este era almacenado dentro de la tabla, de tal manera que si el objeto ya había sido meta-representado lo único que había que hacer era referenciarlo en lugar de recrearlo. En la tabla de referencias a objetos, utilizamos como clave el identificador único usado por Java para identificar a los objetos dentro de la máquina virtual.

Otro punto a tener en cuenta respecto a la tabla de referencias es que debía ser única para cada nodo de invocación a método (MethodCallNode). Pues un mismo objeto puede tener diferentes estados a lo largo de una ejecución.

- Otro problema que hubo que solucionar fue la ineficiencia en memoria de la meta-representación de las variables, pues hay casos en los que para almacenar un entero (int) como mínimo son utilizadas 3 cadenas de texto (String) con su coste correspondiente. La solución aplicada en este caso no es absoluta, pues la meta-representación de las variables sigue siendo un inconveniente en el rendimiento sobre todo para programas grandes. Aun así mejora el rendimiento. La solución consiste en usar una tabla de símbolos global accesible por todos los métodos donde se almacenan las cadenas de texto que van apareciendo en tipos de variables identificadores y nombres, de tal manera que cuando sea posible esas cadenas sean referenciadas en lugar de copiadas. Aportando así en cierta medida una mejora en el rendimiento.

- Por último, y tal como se comentó en la descripción de la clase MethodCallNode, dentro de la clase VariableNode debe contemplarse si esta ha sido modificada dentro del cuerpo del método al que pertenece, y si es así referenciar su antiguo valor.

2.1.3.- La Clase DataTree

Finalmente nos encontramos la clase DataTree. Esta clase es la interfaz de acceso a la estructura de datos. En ella, se concentran tanto el acceso a los datos, como las operaciones sobre ellos.

En ella es donde ponemos en práctica el patrón fachada y una variación del patrón singleton. El primero lo encontramos al hacer las operaciones publicas y estáticas, esto es, accesibles desde cualquier otra parte del código y evitando de esta forma problemas de referencias incorrectas a la estructura de datos. Esta estrategia, sin embargo, es arriesgada, pues hay que tener cuidado con los problemas de concurrencia que pueda ocasionar, así como asegurar la coherencia de los datos en todo momento.

El segundo patrón nos lo encontramos al hacer que los propios datos también sean estáticos. Consiguiendo de este modo trabajar siempre sobre la misma instancia, de la misma manera que describe el patrón singleton.

Hay que añadir también que en esta clase encontramos funciones referentes a la depuración, como son el establecer estado o navegar por los nodos. Estas funciones son únicamente utilizadas por la interfaz de consola.

También encontraremos la tabla de símbolos explicada en el apartado anterior referente a la clase VariableNode. Pues como dijimos, esta tabla debe ser global a todas las llamadas.

2.2.- El motor de depuración (paquete debugEngine)

El motor de depuración tiene por función, recuperar el programa a depurar, ejecutarlo y obtener de él toda la información necesaria para luego ser utilizada durante la sesión de depuración.

Dado que el motor de depuración fue, en un primer momento, el paquete básico sobre el cual se realizaron las primeras pruebas de concepto. También implementa una interfaz en modo texto (Consola) que proporciona las funciones básicas de depuración.

Desde el principio se optó por desarrollar este módulo haciendo uso de la API JPDA, aunque luego hubo que decidir a qué nivel usaríamos ésta. Después de conocer las opciones disponibles y contrastarlas con nuestras necesidades, optamos por utilizar JDI (Java Debug Interface). En este nivel, nosotros únicamente debemos encargarnos de programar las consultas de datos así como el código a nivel usuario. Por tanto, nuestra aplicación es un front-end de la arquitectura de depuración ofrecida por Java.

En la primera etapa del desarrollo, y para familiarizarnos con la API de JPDA, en especial la parte de JDI, utilizamos los ejemplos proporcionados por Sun. Uno de ellos fue el jdb, que es la implementación del depurador de modo texto básico de Java. Con él comprendimos conceptos relacionados con el manejo de los marcos de pila (StackFrames) así como la obtención de valores de variables en el entorno.

Aunque la complejidad de su código hizo que eligiéramos como código base otro ejemplo diferente. El ejemplo sobre el cual realizamos las primeras pruebas fue “Trace”, un programa encargado de únicamente imprimir por pantalla la traza de los métodos por los que pasa la aplicación, a partir de una ejecución.

Dado que existía cierto grado de similitud con nuestra idea, decidimos partir de él e ir modificándolo en función de nuestras necesidades. Incluso ahora en la versión final, se conserva alguna parte (aunque mínimas) de el ejemplo original.
Una característica común de los dos, es la estructura de clases mediante la cual se hace uso de JDI. Esto es realmente apreciable en el diagrama de clases del paquete:

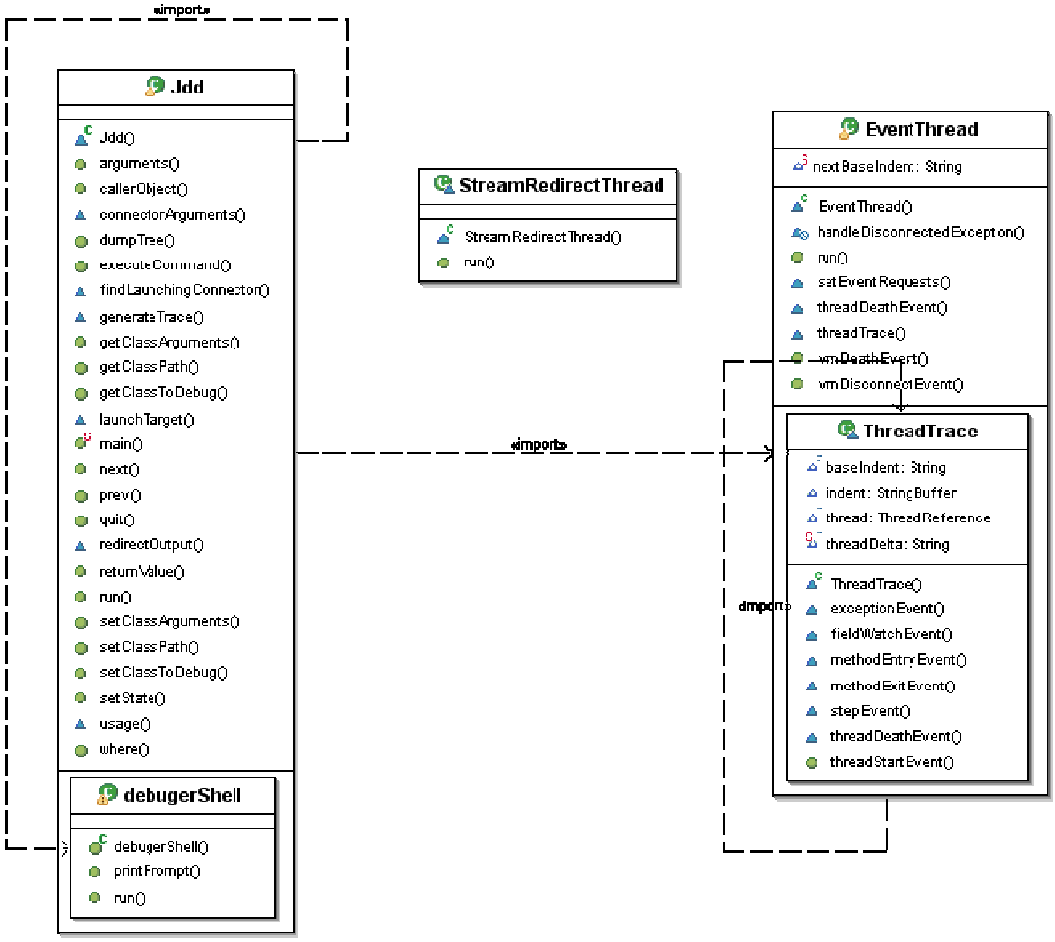


Figura 2: Diagrama de clases del paquete DebugEngine

Dado que este paquete hacemos uso de la API JDI, las clases incluidas en él tienen una relación directa con el funcionamiento de la JPDA.

A continuación, explicaremos el esquema básico de uso de la JPDA aplicado en el motor de depuración. Tal y como se describe en el artículo "Using the Java Debugger Plattform Architecture" [5].

El primer paso es ejecutar el programa a depurar. Para ello, ejecutaremos el programa en una máquina virtual aparte y después conectaremos nuestro front-end a ella.

Existen diversas formas de conectar el front-end con la máquina virtual donde se encuentra el programa en ejecución. Nuestra opción ha sido la de conectarlos mediante el conector "Sun Command Line Launching Connector", el cual arranca el programa a depurar en una nueva máquina virtual y lo ejecuta bajo línea de comandos, aunque cualquier otro tipo de conector podría ser igualmente usado con un cambio mínimo.

El siguiente paso es el de registrar los eventos que deseamos capturar en la máquina virtual. Para ello, obtenemos la instancia del gestor de eventos de la máquina virtual donde se ejecuta el programa, y registramos la escucha de esos eventos. Además, y dado que en ciertos eventos queremos obtener información de los marcos de la pila, indicamos que cada vez que un evento ocurra, los threads de ejecución sean interrumpidos.

Para nuestra aplicación los eventos que queremos escuchar son los siguientes:

- Entrada a los métodos
- Salida de ámbito de un método
- Modificación de un atributo
- Lanzamiento de una excepción

Cada vez que el gestor de eventos capture un evento de este tipo este debe ser enviado al método que nosotros indiquemos, para que el evento sea procesado.

En el caso de que no queramos que algunos eventos sean lanzados para determinadas clases existe la opción de incluir filtros de exclusión. En nuestra herramienta la manera de proporcionar las clases o paquetes a excluir es por medio de la opción "-excludes" cuando llamemos al programa desde la línea de comandos.

A continuación, mostramos un esquema que reproduce la secuencia de acciones que ilustra los pasos explicados anteriormente:

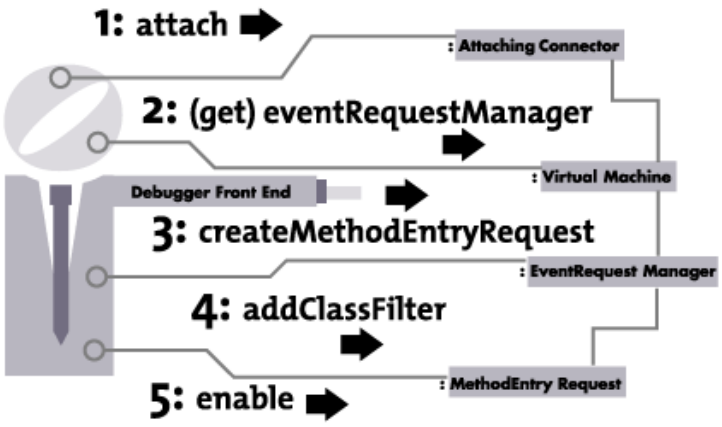


Figura 3: Esquema JPDA

A la vista del diagrama de clases del paquete en la figura 2, ahora pasamos a explicar la morfología y funcionamiento de cada una de ellas por separado.

2.2.1.- La Clase Jdd

Es la clase principal de la aplicación. En ella se implementa el método main, y es en ella donde procesamos los parámetros pasados por línea de comando, los cuales proporcionan las opciones de configuración del depurador.

Una vez tenemos esos parámetros, registramos los eventos que queremos capturar en la máquina virtual y lanzamos los hilos que capturarán la salida estándar del programa a ejecutar, la cual se mostrará más tarde. Por último, se configura y se lanza la máquina virtual donde ejecutaremos el programa a depurar.

Es la clase madre de la aplicación, pues es la que contiene y lanza el módulo gráfico además de implementar el módulo lógico.

Dado que fue de la primera pieza que implementamos, para poder realizar las pruebas desarrollamos una sencilla interfaz de control de usuario por comandos. A partir de esa interfaz implementamos una interfaz de texto (Consola).

2.2.2.- La Clase EventThread

Podríamos considerarla como la clase más importante del motor de depuración, pues si bien toda la estructura de la ejecución del programa está implementada en la clase Jdd, es en EventThread donde se encuentra el comportamiento de la captura de eventos.

El comportamiento específico para cada uno de los eventos es el siguiente:

- Entrada a un método (MethodEntryEvent): Creamos un nuevo nodo MethodCallNode y completamos parte de la información contenida en el (Objeto Invocador y argumentos), sacando parte de la información de el último marco de pila (referente al método invocado) y parte del penúltimo marco de pila (donde está el entorno del objeto invocador). También extraemos el código del método invocado.

- Salida de un método (MethodExitEvent): Al recibir este evento lo que hacemos es cerrar el entorno del método que se esté ejecutando actualmente y pasar al de su padre. Antes de salir del método actual rellenamos el valor de retorno del método actual.

- Modificación de un atributo (FieldWatchEvent): Cuando encontramos este evento buscamos dentro del entorno del método

actual cual ha sido el elemento modificado, marcándolo como modificado y almacenando su valor anterior para la posterior visualización.

- Inicialización de clases (ClassPrepareEvent): Cuando una clase se inicializa, registramos todos sus campos para que estos lancen el evento de modificación cuando sean modificados.

2.2.3.- La Clase StreamRedirectThread

Es una clase auxiliar utilizada para redireccionar y formatear mediante tabuladores, la salida del depurador en modo texto.

Módulo gráfico. Desarrollo

1.- Especificación

El módulo gráfico es el que se encarga de implementar la interfaz que el usuario maneja en todo momento. Por eso, es imprescindible que sea lo más intuitiva posible y de fácil accesibilidad. Hemos desarrollado la interfaz con la tecnología SWT [6] que facilita la integración con Eclipse debido a que comparten el entorno gráfico.

1.1.- Requisitos de la aplicación

Los requisitos enumerados a continuación consiguen intuitividad y accesibilidad en la interfaz. Están orientados a facilitarle al usuario, en lo posible, el manejo del depurador:

- Menú con las distintas opciones que se ofrecen: cargar un nuevo programa a depurar, visualizar cualquiera de las ventanas del shell, iniciar el proceso de depuración, pararlo, salir..etc.

- Visualización de los distintos elementos de nuestra aplicación que consideramos importantes en el proceso de depuración.
  Estos son:

  - Ventana que visualiza el árbol de depuración. Denominada como **DebugTreeWindow.**
  - Ventana que visualiza el objeto que estamos explorando. Es decir, aquel sobre el que queremos conocer el valor de sus atributos. Denominada **Tree Object Inspector.**

  - Ventana de código. Permite al usuario consultar el código del programa, a partir del cual se ha construido el árbol de ejecución que se muestra en la ventana DebugTreeWindow.
  - Ventana de consola. Permitiremos la depuración por consola.

- Además de poder acceder a todas las opciones que implementamos a través del menú, se incluye la posibilidad de hacerlo de forma más directa con una barra de herramientas. En ella, aparecen las opciones de uso más frecuente: navegación por el árbol (siguiente y anterior nodo), inicio del proceso de depuración, parada del mismo, elección del estado del nodo seleccionado sobre cuatro posibilidades (**válido**, **no válido**, **desconocido** y **de confianza**), cargar un nuevo programa para depurarlo, etc.

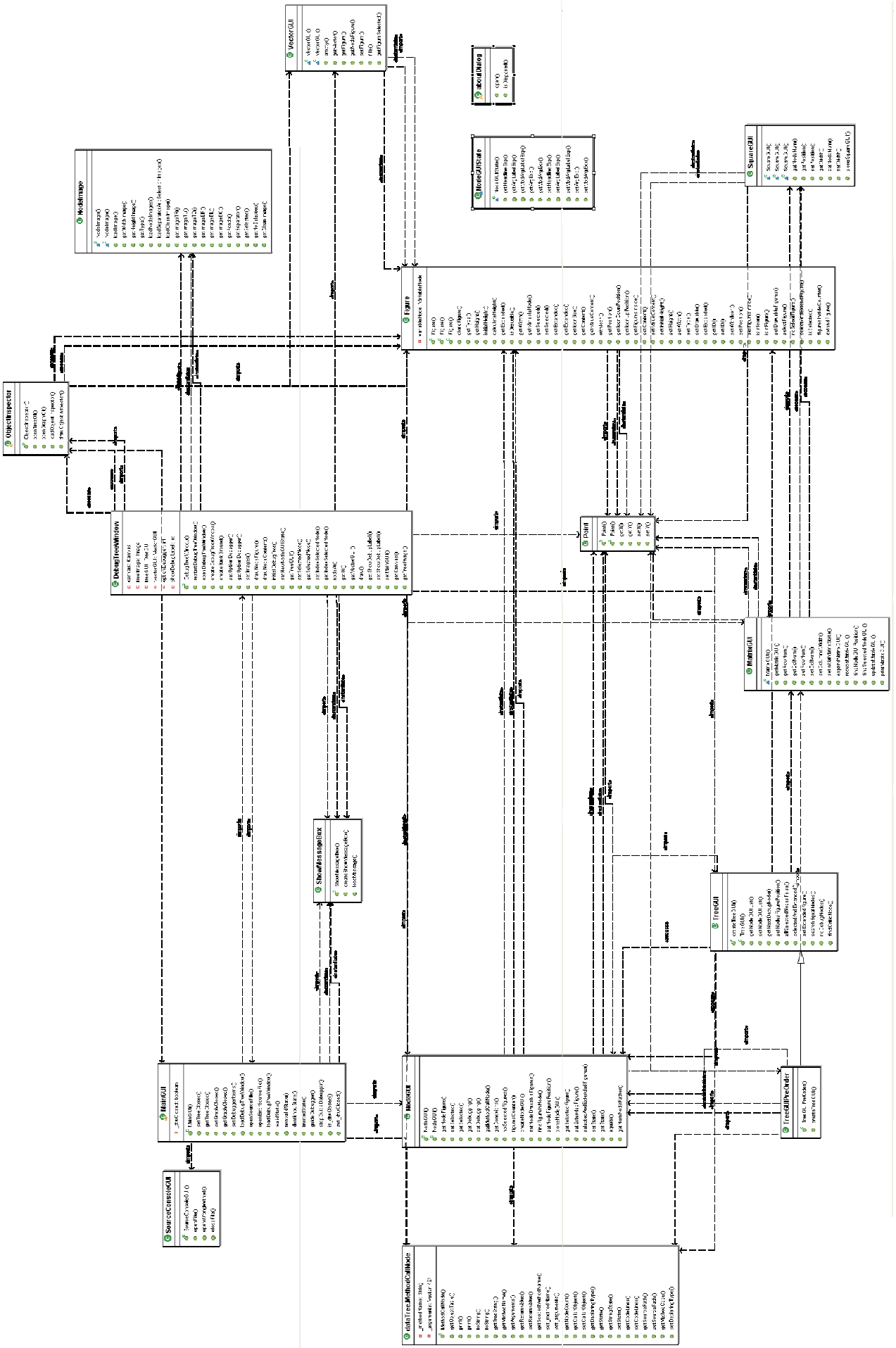
- Dos tipos de depuración definidos: la **depuración automática o guiada** y la **depuración manual.** En la primera de ellas, es el propio depurador el que, de manera automática navega por los nodos del árbol con la estrategia seleccionada; mientras que en la segunda, es el usuario quien selecciona los distintos nodos y establece su estado eligiendo entre los cuatro posibles.

- Posibilidad de elegir el tipo de estrategia de depuración. El árbol de ejecución puede ser recorrido de maneras distintas cuando hayamos seleccionado la opción de depuración guiada. Se puede recorrer en pre-orden, in-orden o post-orden, o implementar estrategias más sofisticada como la del algoritmo de divide y vencerás.

- Permitir cambiar de un tipo de depuración a otro, generando en cada caso árboles de depuración diferentes, uno por tipo de estrategia.

- Posibilidad de guardar las sesiones de depuración y cargarlas posteriormente.

Todo esto hace que el depurador sea más manejable e intuitivo, y por tanto más accesible al usuario. Seleccionamos un diseño gráfico de los nodos que resultara vistoso y profesional, para hacer la interfaz más atractiva y descriptiva al usuario.



1.2.- Simulación

A continuación, comentaremos, mediante diagramas de secuencia, el funcionamiento de nuestro depurador. Primero, mostraremos un diagrama de secuencia que visualice las acciones que se desencadenan en la creación y visualización del árbol de ejecución. Después las acciones que se llevan a cabo al iniciar una sesión de depuración.

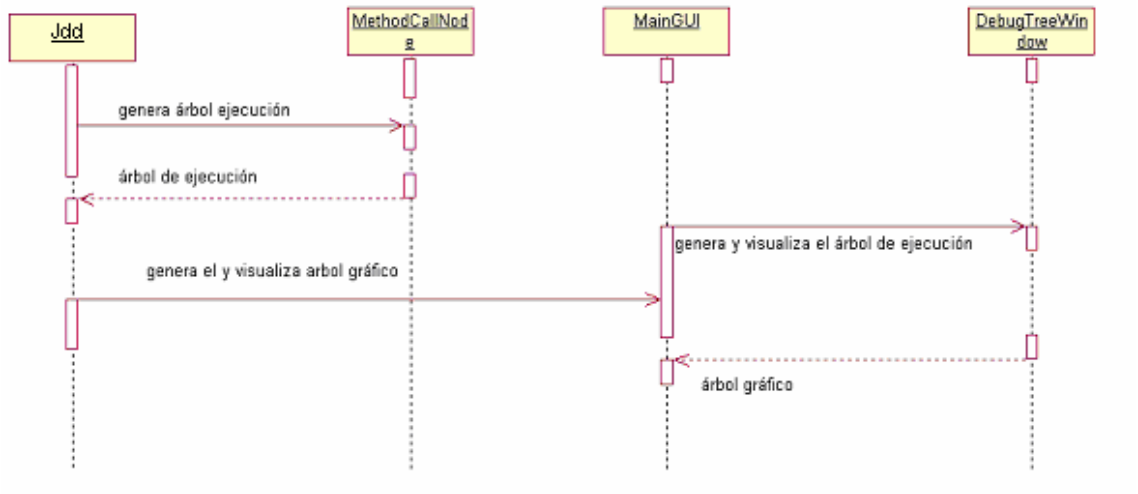


Figura 4: Diagrama de secuencia del proceso de creación del árbol de depuración, y posterior visualización

Desde la clase Jdd generamos el árbol de ejecución, obteniendo toda la información almacenada en un objeto de la clase **MethodCallNode**, clase perteneciente al paquete **dataTree.**

A partir de este objeto, invocaremos a la clase **DebugTreeWindow**, que es la que se encarga de generar el árbol gráfico y de su consiguiente visualización.

En el siguiente diagrama mostramos la secuencia de acciones implicadas en la depuración guiada de un programa:

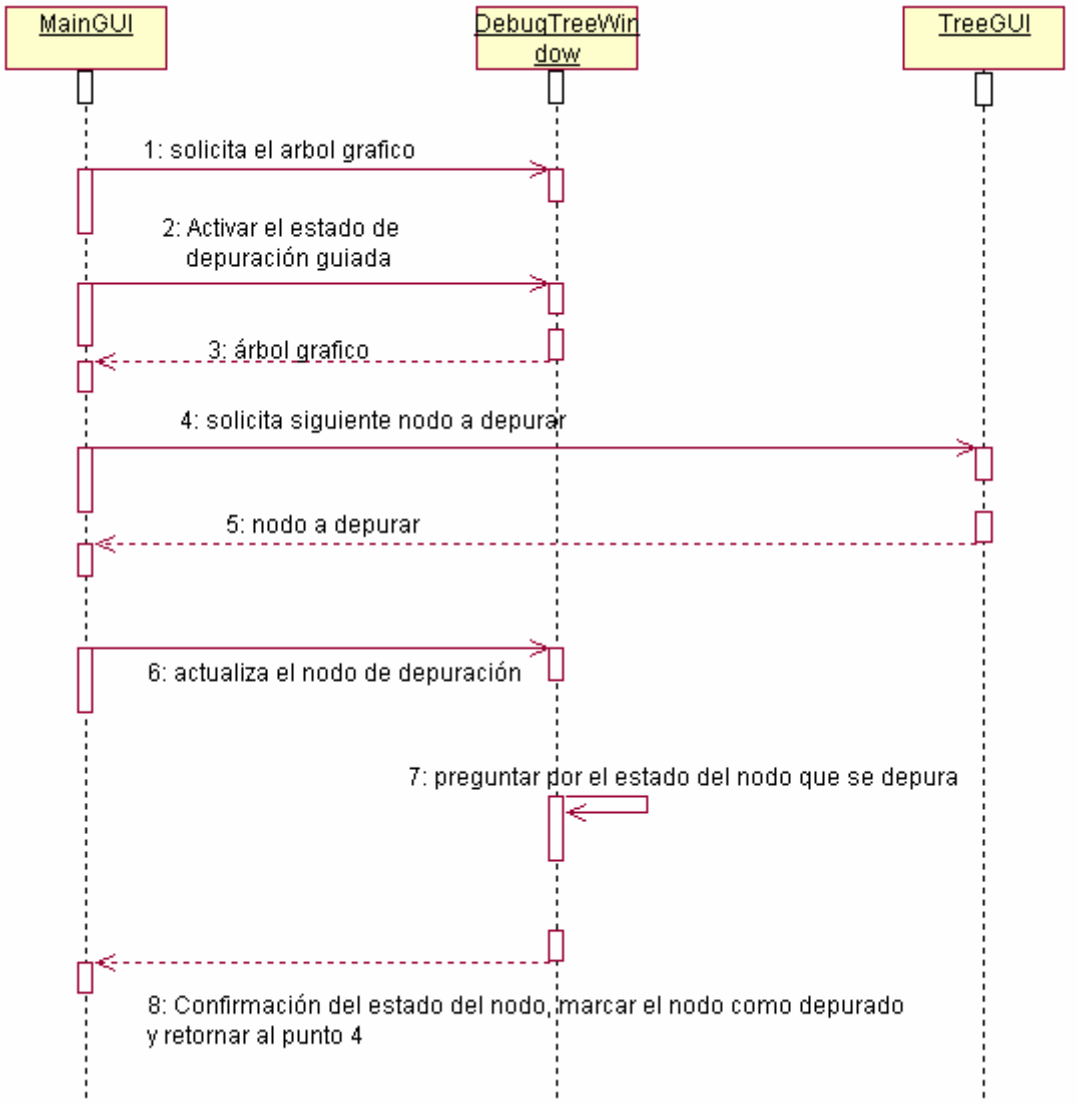


Figura 5: Diagrama de secuencia obtenido al iniciar una sesión de depuración

En primer lugar, MainGUI solicita a DebugTreeWindow el árbol gráfico y establece el estado de depuración guiada. A continuación, se irá recorriendo el árbol según la forma en la que esté almacenado (pre-orden, in-orden,..) de manera que se pregunta al usuario (a través de DebugTreeWindow) por el estado del nodo a depurar en ese momento. Esta secuencia finalizará cuando se haya recorrido todo el árbol, o bien se haya encontrado el método erróneo.

¿Cuándo finaliza el proceso de depuración?

El proceso de depuración termina cuando se marca a un nodo padre como no válido y, sin embargo, todos sus hijos son correctos.

2.- Diseño e implementación

Uno de los fines que se persiguen a la hora de diseñar cualquier aplicación es el de disponer de un paso intermedio entre lo que especificamos, nuestras ideas y el código que finalmente implementa la aplicación. Igualmente, debe ser lo más modular posible, y flexible a futuros cambios.

En este apartado se comentará el diseño correspondiente al paquete gráfico (**gui**), que incluye todas aquellas clases que implementan funcionalidades del módulo gráfico.

2.1.- La estructura del árbol gráfico

La estructura que se define debe ser consecuente con el uso que vamos a hacer más tarde de ella, e igualmente, constituir un reflejo de la estructura de datos que vamos a representar, en este caso el árbol de ejecución que recibimos del módulo lógico.

Para tener una visión general de la distribución del módulo gráfico presentamos el siguiente esquema:

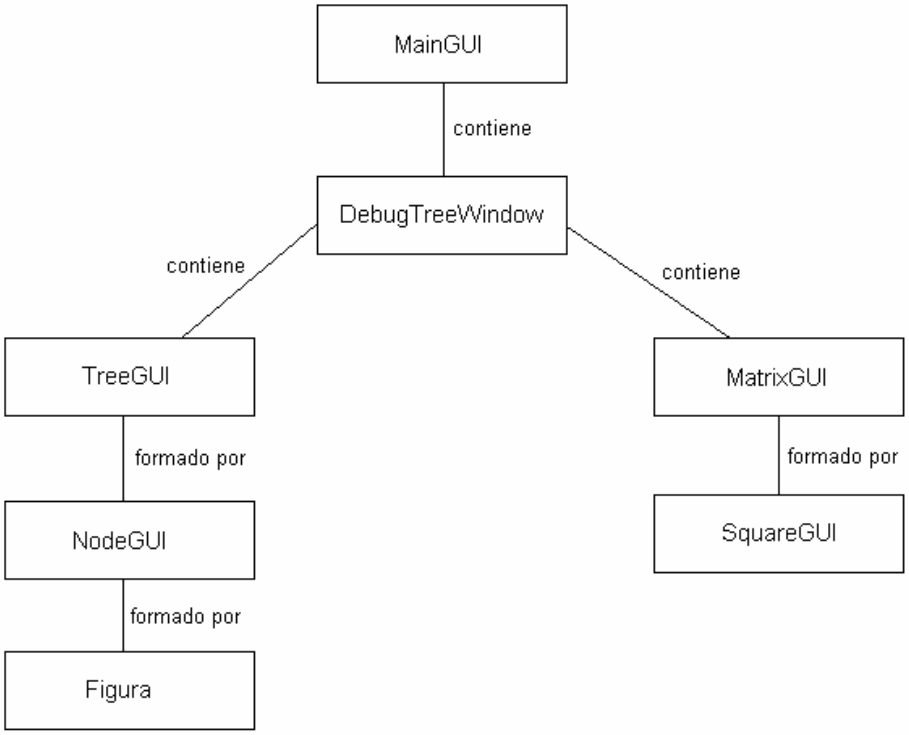


Figura 6: Distribución del módulo gráfico

La clase principal, MainGUI, se encargará de crear el DebugTreeWindow pasándole el árbol lógico que debe mostrar.

Por su parte, DebugTreeWindow construirá, a partir del árbol lógico, una representación gráfica de éste. Dicha representación está almacenada en una estructura de tipo TreeGUI, que contiene un vector de nodos gráficos (NodeGUI). Para representar cada nodo, usaremos figuras que contendrán cada uno de los elementos representativos del nodo: una figura para los argumentos, otra figura para el caller object, etc. De esta forma, la representación del árbol gráfico se reduce a la representación de las figuras que forman cada nodo.

Además, DebugTreeWindow contiene una matriz (MatrixGUI) que facilita la gestión de los nodos y establece la posición de cada uno. Dicha matriz está formada por casillas (SquareGUI) de forma que cada casilla contiene un nodo del árbol. Esto nos permitirá, sabiendo en qué casilla estamos, conocer el nodo seleccionado por el usuario.

Definimos más detalladamente dos conceptos claves en el diseño del módulo gráfico: el **árbol gráfico** y el **nodo gráfico**.

El árbol gráfico se obtiene recorriendo el árbol de ejecución y guardando los nodos en un vector. La forma de almacenar los nodos en el vector depende de la estrategia de navegación que se elija. De manera que para obtener el siguiente nodo o el anterior a uno ya seleccionado simplemente tengamos que acceder a la posición siguiente o a la anterior del vector respectivamente. Se reduce así el coste de este tipo de operaciones.

Por defecto, los nodos se guardan en pre-orden, pero únicamente redefiniendo el método de creación del árbol podríamos conseguir una estrategia de búsqueda distinta.

A raíz de esta idea, el árbol gráfico estará definido por la clase **TreeGUI,** de la que heredarán distintas clases que redefinan el método de construcción del árbol. Dependiendo del tipo de estrategia que seleccionemos tendremos los nodos del árbol almacenados de una forma u otra.

El nodo gráfico, representa una llamada a un método del programa que estamos ejecutando. Toda la información que debemos conocer de la llamada a un método determinado la contiene la clase **MethodCallNode**, definida en un apartado anterior. Por tanto, un nodo gráfico debe disponer de una referencia al nodo del árbol de datos correspondiente, es decir un objeto de la clase MethodCallNode. Con este objeto, tenemos acceso a toda la información que vamos a visualizar (la cabecera del método, compuesta por nombre del método, argumentos, nombre del objeto invocador, y los tipo de todos ellos). El problema es el de definir la apariencia del nodo.

2.1.1.- Representación del nodo gráfico

Comenzamos por organizar la información presentada en los argumentos. En realidad, en una llamada a un método no disponemos de información acerca de cómo está implementado ese método. Por ello, es necesario tener una idea del resultado que esperamos en ese método recibiendo unos parámetros de entrada determinados. A lo mejor resulta interesante conocer qué valores toman los argumentos de entrada, y si estos son modificados durante la ejecución del método, para determinar si éste es correcto o no. Una buena opción es, por tanto, agrupar los argumentos en dos clases: los modificados y los no modificados.

A partir de esta idea, podemos ver el nodo gráfico como un conjunto de dos **figuras**, la de los argumentos modificados y la de los argumentos no modificados. Cada figura engloba, a su vez, las figuras correspondientes a cada argumento. Hay que tener en cuenta, cómo representar los argumentos, en función de su tipo. Un argumento puede ser de tipo simple (int, boolean, char,...) o puede ser un objeto. En este último caso se presenta el problema de visualización de los atributos. Se optó por la solución de asignar a cada atributo una figura, de manera que la figura asociada al argumento puede constar a su vez de figuras.

Una primera aproximación de la representación del nodo fue la de definir distintos tipos de figuras. Un tipo para el nodo en sí, otro para los argumentos modificados y no modificados y otro para los atributos de los argumentos, en el caso de que éstos fueran objetos. La figura que aparece a continuación representa la primera representación que diseñamos del nodo gráfico.

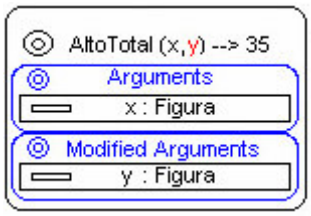


Figura 7: Primera aproximación al nodo desplegado

Como podemos observar en la Figura 7 se distinguen perfectamente los distintos tipos de figuras que definen un nodo gráfico. La figura del nodo es un rectángulo redondeado de color negro, la de los argumentos modificados y no modificados uno azul, y los atributos “x” e “y”, que son objetos de una hipotética clase Figura, son simples rectángulos de color negro.

Los símbolos que aparecen en la esquina superior izquierda de cada figura representan iconos. Se utilizarán para poder desplegar y plegar el nodo.

En ocasiones, puede que no queramos consultar el valor de los parámetros de entrada del método, bien porque no existan, o bien porque la información que

aparece en la cabecera nos sea suficiente. En este caso, no desplegaríamos el nodo. El aspecto que presentaría el nodo plegado es el de la Figura 8. De esta manera conseguimos que el árbol de ejecución que se construya ocupe el menor espacio posible.

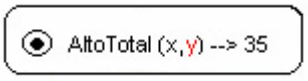


Figura 8: Primera aproximación al nodo plegado

Para que el usuario pueda obtener la mayor información posible rápidamente, aparecerá el argumento cuyo valor ha sido modificado en rojo. Con esto, conseguimos saber si en un método un parámetro de entrada ha sido modificado o no sin tener que desplegarlo. Igualmente, si estamos buscando sólo los parámetros que se han modificado dentro del método, y en nodo no encontramos ningún parámetro en rojo, no perderemos tiempo en buscar dentro de dicho nodo.

El problema que se plantea en este momento es el de la definición de un **objeto modificado**. Un objeto está modificado cuando alguno de sus campos se modifica. Para ello, es necesario tener acceso a esos campos. La visualización del objeto se hará en una ventana independiente a la del árbol de ejecución, para obtener mayor claridad. Surge entonces la ventana que denominamos **Tree Object Inspector**. En ella se debe tener acceso a cada uno de los valores de los atributos de la clase a la que pertenece el objeto.

Desde el punto de vista de la implementación, se definen varias clases que en conjunto representan el árbol gráfico, formado por nodos gráficos y las figuras que componían a estos últimos. Estas son, respectivamente, **TreeGUI, NodeGUI, Figure.**

Sin embargo, si volvemos a la representación del nodo gráfico, nos damos cuenta de que en ella falta información. Generalmente, cuando se escribe un programa, la invocación de los métodos, a no ser que estos sean estáticos, se realiza a través de objetos de la clase en la que están definidos e implementados dichos métodos. Este objeto lo denominados **objeto invocador.** Este objeto no es ni un argumento modificado ni uno no modificado, necesariamente debe tener una clasificación diferente. Lo que implica una figura distinta y al mismo nivel que la de los argumentos modificados y no modificados. Análogamente, el valor de retorno de un método no tiene porqué ser siempre un tipo simple o primitivo; puede que el método devuelva un objeto, del que también queramos saber el valor de sus campos.

Todas estas observaciones nos llevan a una nueva y más completa representación del nodo gráfico, que incluya toda la información que se acaba de comentar. El aspecto del nuevo nodo es el de la Figura 9.

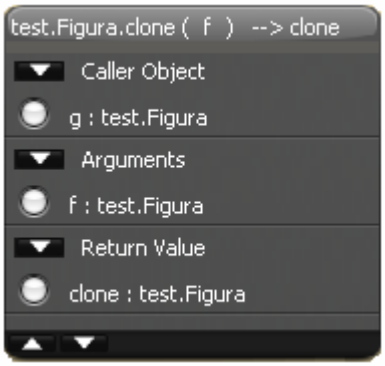


Figura 9: Representación del nodo final

En esta nueva estructura clasificaremos la información en tres tipos:

Caller Object: es el objeto invocador del método. Puede ser modificado por el método. Por ejemplo, en el caso de que el método sea un mutador, uno de los campos del objeto se verá obligatoriamente modificado. Por tanto, debemos poder consultar su nuevo valor.

Arguments: agrupa tanto a los argumentos modificados como a los argumentos no modificados, dibujando en rojo a los primeros, junto con su valor antiguo y nuevo.

Return Value: El valor de retorno del método. En caso de que el método devuelva un tipo simple esta figura no se incluye, simplemente se escribe dicho valor en la cabecera del método. Si lo que devuelve es un objeto, entonces aparecerá como un objeto más del que podremos conocer el valor de sus atributos.

Esta nueva interfaz resulta mucho más sofisticada y profesional que la anterior. Sin embargo, lo único que ha cambiado ha sido las figuras que componen a los nodos. En lugar de ser rectángulos ahora estamos cargando imágenes, pero la estructura es la misma. Una figura de nodo sigue conteniendo figuras en su interior. Únicamente ha variado el número de figuras, antes eran dos y ahora son tres. Y cada una de ellas tiene figuras en su interior, lo que correspondería a los propios objetos. Esta es la ventaja de definir una estructura lo suficientemente flexible, como para soportar cualquier cambio en el tipo de visualización (antes rectángulos ahora imágenes), sin tener que cambiar la estructura por completo.

2.1.2.- Representación del árbol gráfico

Una vez definida la estructura del nodo, debemos organizar el diseño del árbol gráfico para que se muestre la información de la forma más sencilla posible.

Se representará en la ventana DebugTreeWindow (anteriormente mencionada) y debe presentar al usuario la mayor cantidad de información posible. Para ello, en el instante inicial, todos los nodos del árbol permanecen plegados. De esta forma, será el usuario el que, en base a sus intereses, despliegue un nodo u otro.

Para dibujar el árbol, basta con dibujar en orden cada uno de los nodos, es decir, recorrer el vector de nodos. El problema radica en dónde dibujar cada nodo. Debemos por tanto asignar a cada nodo una posición que lo sitúe respecto al resto de nodos.

Una primera aproximación nos llevó a asignar una posición fija al primer nodo (el nodo raíz) y a partir de ésta, y teniendo en cuenta la altura de los nodos dibujar el resto del vector. Sin embargo, el árbol gráfico es dinámico en cuanto a que sus elementos se pliegan y se despliegan a gusto del usuario y, por tanto, con esta primera aproximación los nodos acabarían montándose unos sobre otros. Esta implementación, también complicaba la forma de averiguar sobre qué figura de qué nodo se había pinchado para proceder al cambio pertinente.

Por todo esto, decidimos reorganizar todo el diseño gráfico en base a una nueva idea: la matriz gráfica.

2.1.2.1.- La matriz gráfica

La idea básica de la matriz gráfica (implementada en la clase **MatrixGUI**) es la de mantener una cuadrícula de forma que para conocer sobre qué nodo hemos pinchado, basta con calcular en qué casilla de la matriz nos encontramos.

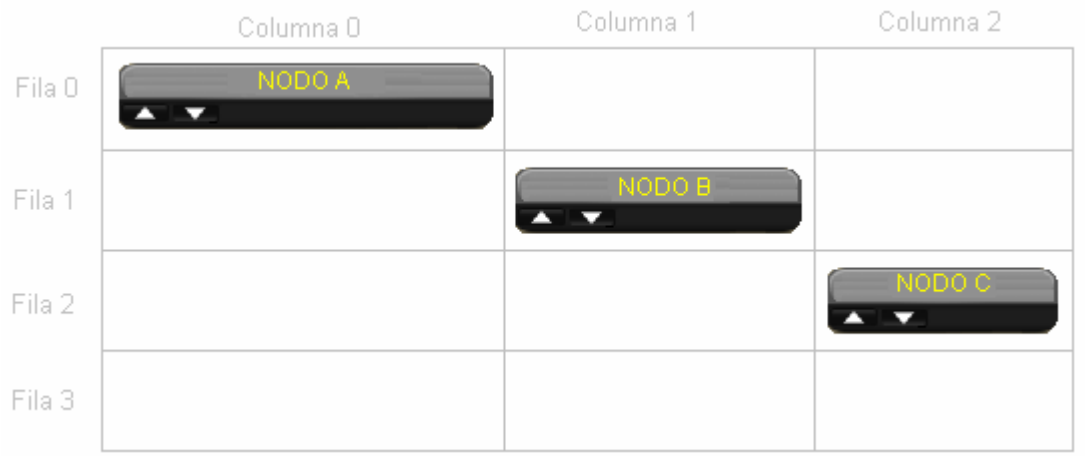
Definimos la matriz como una estructura con un altura de fila constante, pero un ancho de columna variable. De esta forma, el ancho de cada columna se adaptará a la anchura máxima de los nodos de la columna. Así, se hace un uso eficiente del espacio de representación disponible.

La matriz estará formada por un array bidimensional (número de filas x número de columnas) de casillas (implementadas en la clase **SquareGUI**) con la restricción de que cada casilla sólo puede ser ocupada por un nodo.

Inicialmente, se crea una matriz vacía con un número de filas igual al número de nodos del árbol, y un número de columnas equivalente a la profundidad del mismo. A medida que recorremos el árbol, completamos la

matriz de forma que cada nodo ocupa la casilla correspondiente a la primera fila libre y la columna siguiente a la columna de su padre.

Ejemplo: Para un árbol de cuatro nodos y una profundidad de 3, ¿dónde situaríamos el nodo D?



El nodo D irá en la casilla de fila 3 y columna 1. Como se ve, la primera fila libre es la tercera, y la columna que le corresponde es la siguiente a la de su padre: (0) +1 = 1.

Las casillas de la matriz, poseen el número de nodo (posición en el vector de nodos) que contienen, de forma que si sabemos que hemos pinchado sobre la casilla (2,2) podemos acceder a dicha casilla e identificar que el nodo que contiene es el nodo número 2, es decir, el nodo C.

Una vez resuelto el problema de asignar posiciones únicas a cada nodo, nos queda por tratar el despliegue de los nodos. Pero con la nueva estructura, el problema desaparece.

Consideramos la matriz como una estructura dinámica que se ampliará o se reducirá si es necesario.

A la hora de desplegar un nodo pasará lo siguiente:

- Si la altura del nodo desplegado es de *x* casillas y la altura del nodo plegado era de *y* casillas, calculamos la diferencia de ambas. Llamamos *z* a dicha diferencia ($z = y - x$). Si el número de casillas libres por debajo del nodo es mayor o igual a *z*, desplegamos el nodo sin modificar la matriz.

Ejemplo: Desplegamos el nodo B. No es necesario añadir filas.

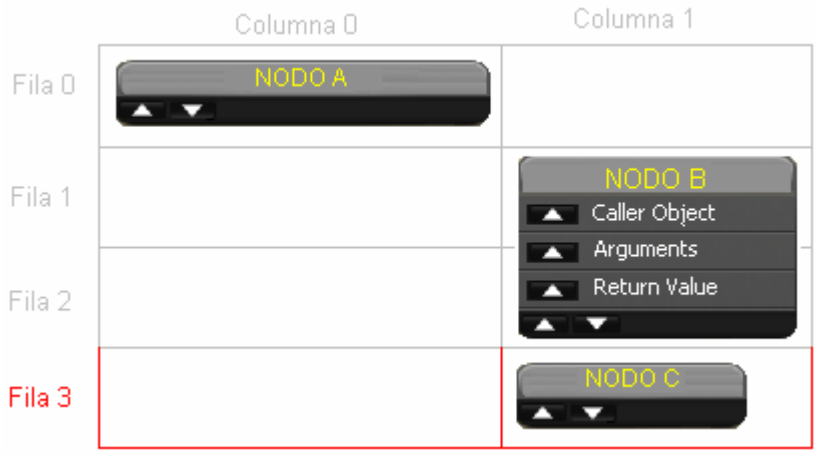
En el ejemplo, desplegamos el nodo B que pasa a ocupar las casillas (1,1) y (2,1), pero ni el nodo D ni el nodo C sufren modificaciones en su posición.

• Si la altura del nodo desplegado es de *x* casillas y la altura del nodo plegado era de *y* casillas, calculamos la diferencia de ambas. Llamamos *z* a dicha diferencia (*z = y - x*). Si el número de casillas libres por debajo del nodo es menor que *z*, la matriz debe actualizarse añadiendo las filas necesarias. Para que después se simplifique el hecho de reducir la matriz, ésta contendrá un flag de expansión para cada nodo, de forma que si la matriz se expande por un nodo i, el flag para dicho nodo estará activo.

Ejemplo: Partimos de la siguiente matriz:

Desplegamos el nodo B. La matriz debe actualizarse. El nodo B desplegado ocupa 2 casillas (x = 2) y la altura del nodo plegado es de 1 casilla (y = 1). z = 2-1 = 1. Como el número de casillas libres bajo B es menor que z (0<1), debemos

añadir a la matriz 1 nueva fila. Dicha fila se añadirá al final de la matriz (fila 3). Los nodos que queden por debajo de B, en este caso sólo C, deben desplazarse una fila hacia abajo.



A la hora de plegar un nodo, la situación es la contraria:

- Si el flag de expansión para dicho nodo está inactivo, significa que la matriz no se actualizó al expandir dicho nodo por lo que al plegarlo, no debemos reducirla.

- Si el flag de expansión para dicho nodo si está activo, significa que la matriz se expandió al desplegar el nodo y, por tanto, ahora debemos reducirla. Para ello, basta con eliminar las filas que se hayan quedado vacías, es decir, aquellas filas en las que todas sus casillas no contienen ningún nodo.

Es importante señalar que el hecho de incluir los flags de expansión para cada nodo permite mantener la estructura de árbol en todo momento sin que confunda al usuario sobre los hijos de cada padre.

Con todo esto, la representación del árbol gráfico es sencilla. Basta con recorrer la lista de nodos del treeGUI (objeto de la clase TreeGUI) y dibujarlos en la posición correspondiente. Cada nodo se pintará según su estado (válido, no válido, desconocido o de confianza) y situación (plegado, desplegado, seleccionado,…).

2.2.- Estructura del Tree Object Inspector

Con el fin de conseguir una interfaz lo más organizada posible, optamos por representar los objetos que el usuario desea evaluar en una ventana aparte (Tree Object Inspector). De esta forma conseguimos también que, si un objeto tiene múltiples atributos, el despliegue del nodo ocupe demasiado espacio en la ventana del árbol de ejecución.

Para representar los objetos elegimos el componente JTree que nos permite mostrar el contenido de un objeto en un espacio relativamente reducido.

En el momento en que el usuario selecciona un objeto en el DebugTreeWindow, en el Tree Object Inspector aparecerá, tanto la cabecera del método al que pertenece, como la información de dicho objeto. Los atributos del mismo podrán desplegarse y evaluarse en esta misma ventana.

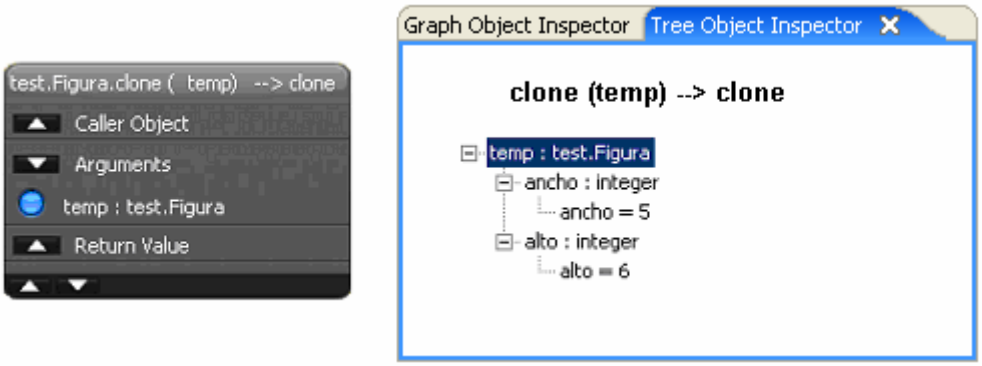


Figura 10: Representación de la información de un objeto en el Tree Object Inspector

En la figura 10, mostramos cómo se representa la información, en el Tree Object Inspector, de un objeto seleccionado en la ventana DebugTreeWindow.

2.3.1.- ¿Y si el objeto está modificado?

Cuando el valor de un objeto se modifica en un método resulta interesante en el proceso de depuración el poder consultar el valor antiguo y el nuevo valor de dicho objeto.

Para ello, además de mostrar el objeto en color rojo en el DebugTreeWindow, especificaremos en el Tree Object Inspector qué atributo o atributos de dicho objeto son los que se han modificado. Un ejemplo de dicho comportamiento se muestra en la Figura 11:

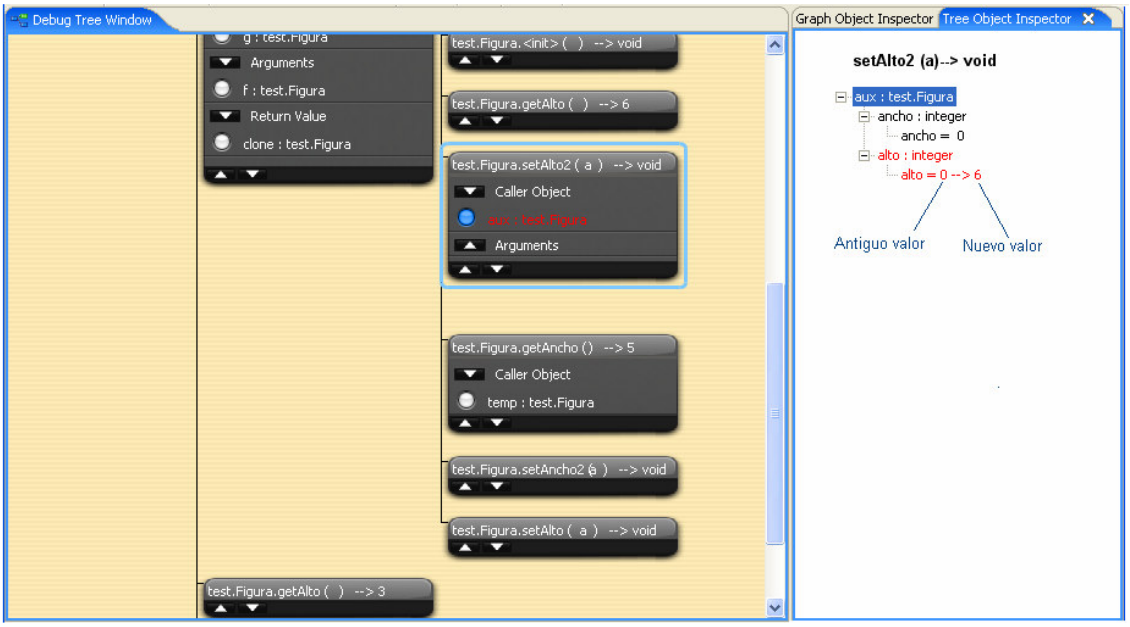


Figura 11: Representación en el Tree Object Inspector de un objeto modificado

2.3.2.- ¿Qué ocurre con las referencias?

Si implementamos referencias cruzadas, debemos encontrar una forma de representación sencilla que no contenga bucles infinitos. Por ejemplo, partamos del código siguiente:

```
class A{
      public A a;

      public A dameA(){
            return a;
      }

      public void ponA(A mia){
            a = mia;
      }
}

public class test4 {
      public static void main(String[] args) {
            A uno = new A();
            uno.ponA(uno);
            A dos = new A();
            dos.ponA(uno.dameA());
      }
}
```

Una vez detectada una referencia, dejamos de descender en la estructura de JTree del Tree Object Inspector. Dicha situación se muestra con un ejemplo en la figura 12.

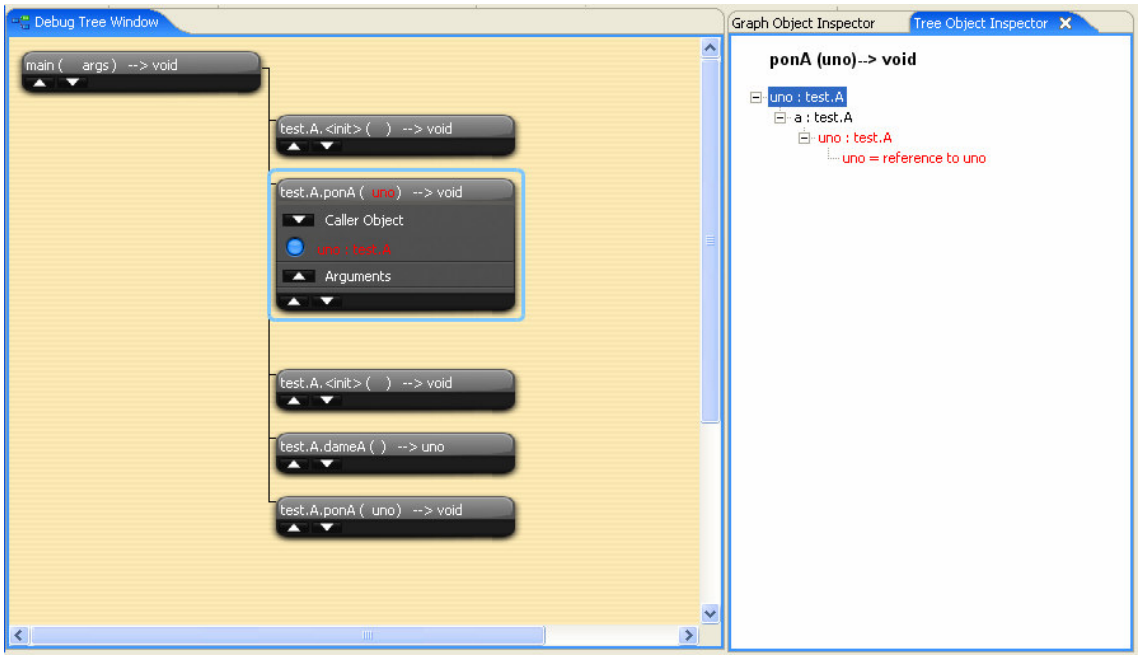


Figura 12: Representación de un objeto que hace referencia a sí mismo

2.3.- Visualización del código

El depurador declarativo, se basa en la semántica del programa, es decir, a la hora de depurar basta que el usuario conozca el significado de cada método para que pueda asignarles un estado u otro.

Sin embargo, a menudo puede que el usuario no recuerde el significado de cada método por lo que puede serle útil tener una ventana de visualización del código. Así, en la aplicación, si hacemos doble clic sobre algún nodo, aparecerá en una pestaña aparte el código de la clase a la que pertenece el método asociado a dicho nodo. Para que resulte aún más fácil localizarlo, se sombreará el método en azul. La figura 13 muestra esta utilidad:

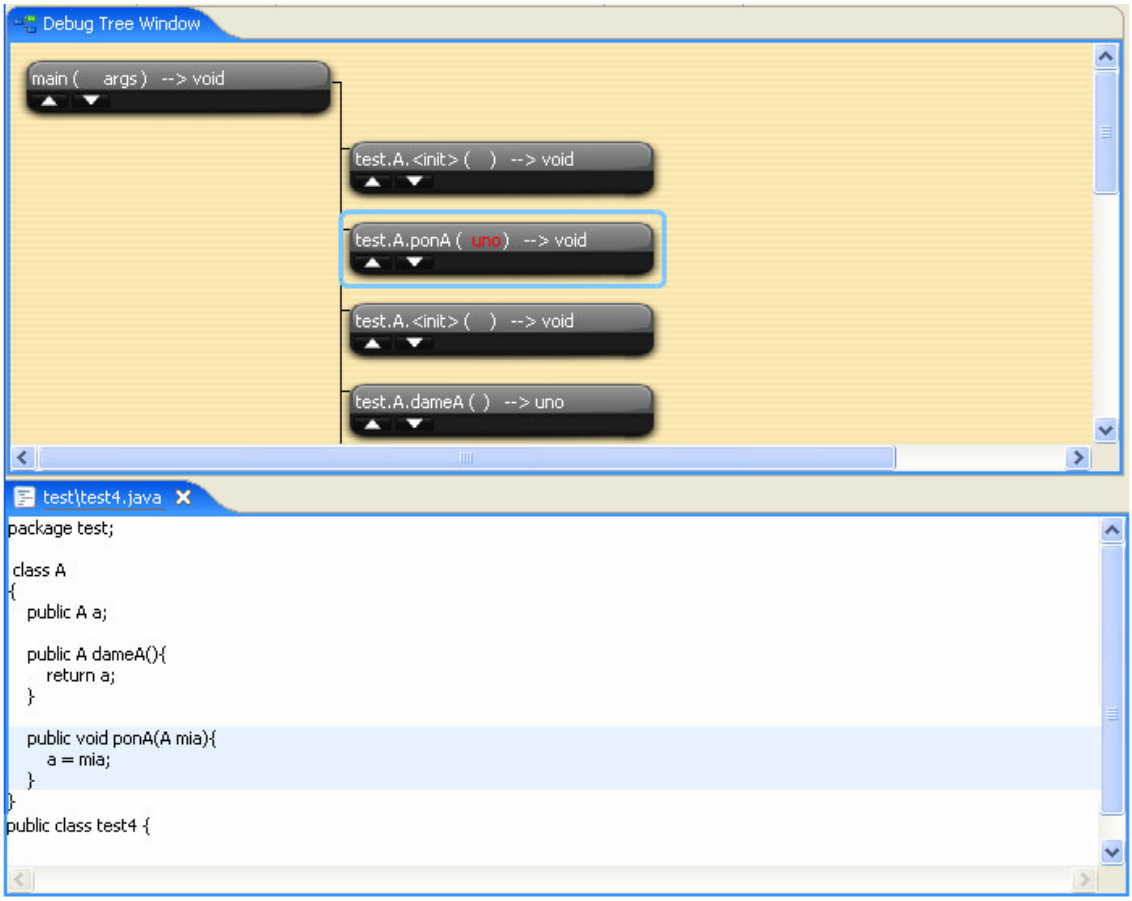


Figura 13: Visualización del código fuente de un método

Además, cuando se detecta un error en el proceso de depuración, se indica al usuario el método asociado al nodo erróneo y se ofrece la posibilidad de ver el código de dicho método. Éste código se muestra sombreado en rojo para una rápida localización. En la figura 14 se ilustra un ejemplo de esta situación.

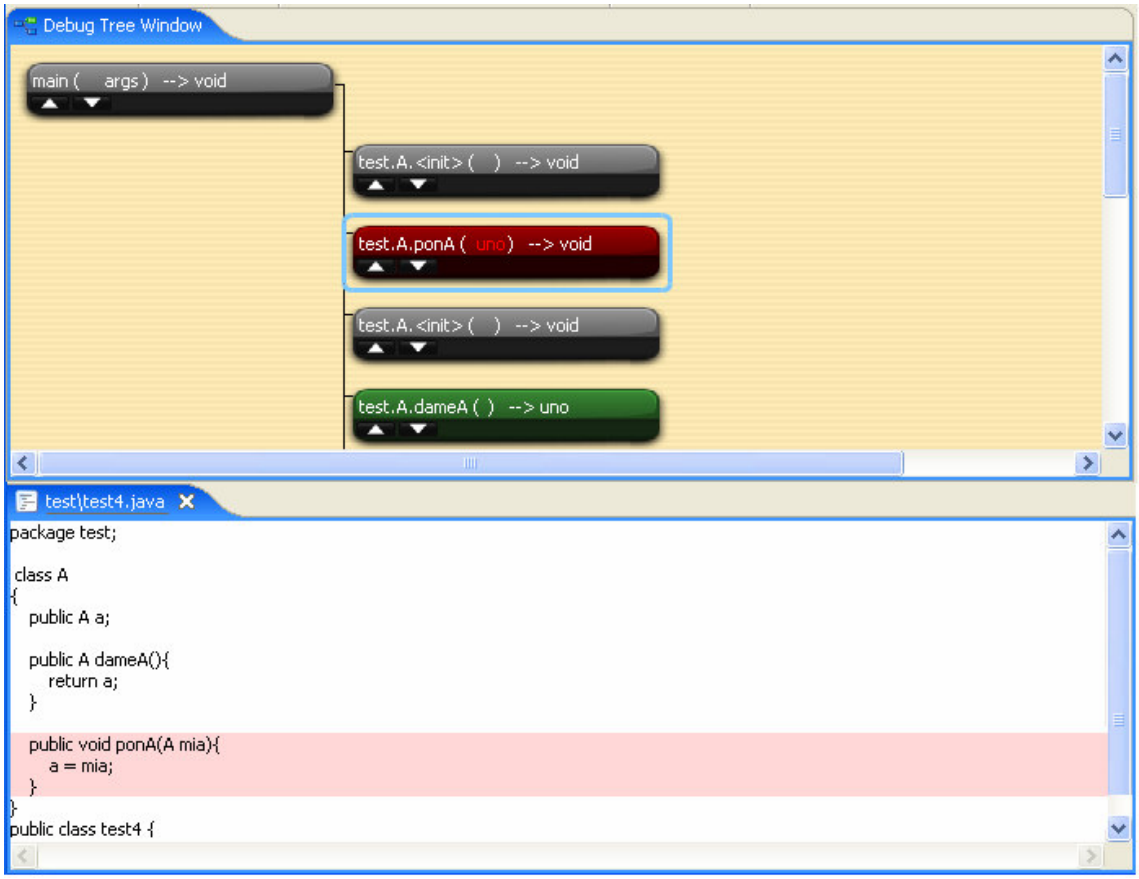


Figura 14: Visualización del código fuente de un método erróneo

2.4.- Consola

Una vez que el usuario se ha familiarizado con la aplicación, puede resultarle más sencillo realizar el proceso de depuración por línea de comandos. Para ello, hemos implementado un shell con el que pueden realizarse las mismas operaciones que a través de la interfaz gráfica.

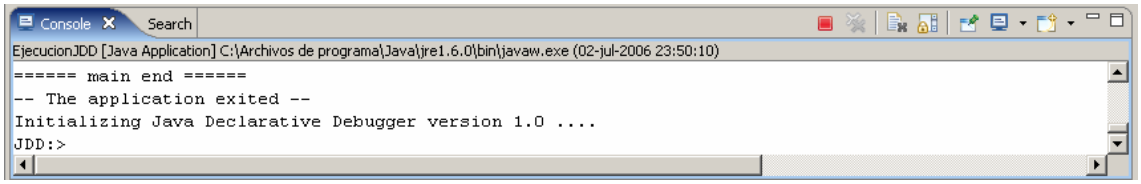


Figura 15: Aspecto de la consola

El listado de comandos permitidos en el interfaz de modo texto es el siguiente:

- **run <class> <args>** : ejecuta la clase especificada
- **classpath**: muestra el classpath
- **list** : muestra el código asociado al nodo actual
- **where**: muestra el nodo actual
- **where <levels>**: muestra el árbol desde el nodo actual hasta la profundidad indicada en <levels>
- **dumpTree**: muestra el árbol de depuración
- **callerObject**: muestra el caller object del nodo actual
- **arguments**: muestra los argumentos del nodo actual
- **returnValue**: muestra el return value del nodo actual
- **set <state>**: asigna al nodo actual el estado de depuración especificado en <state>
- **next** :siguiente nodo a depurar
- **prev**: nodo previo al actual
- **version**: muestra la versión de la aplicación
- **exit/quit** :cierra la aplicación
- **help**: muestra al usuario los posibles comandos ejecutables en la consola.

3.- Conclusiones

En este apartado, ofrecemos una visión detallada de la estructura y diseño de nuestra aplicación, así como una idea de la implementación y organización del código.

Hemos conseguido un diseño modular y flexible a futuras ampliaciones (ver apartado de Ampliaciones) definiendo un estructura de paquetes que resumen las distintas funcionalidades del proyecto.

Además, a la hora de programar hemos intentado buscar un compromiso entre el coste y la memoria empleando estructuras que, a pesar de añadir algo de coste en espacio, reducían considerablemente el coste en tiempo (por ejemplo, el uso de una matriz que me permita localizar fácilmente los nodos).

Finalmente, conseguimos un diseño que reúne tanto la sencillez e intuitividad para el usuario, como claridad en el código para el programador.

AMPLIACIONES

A la funcionalidad descrita a lo largo de este documento, podemos añadir algunas más que estaban previstas pero, por falta de tiempo, no se han podido llevar a cabo.

La estructura del proyecto facilita la incorporación de ciertas ampliaciones debido ya que se ha diseñado en base a esta idea. Además de las que podría demandar el usuario, nosotros proponemos algunas mejoras que solventan las limitaciones de nuestro proyecto.

1.- Estrategias de recorrido

Como ya hemos mencionado anteriormente, el uso del depurador declarativo podría ser más completo y eficiente con la incorporación de nuevas estrategias de recorrido. El espacio de búsqueda del error se reduciría considerablemente empleando, por ejemplo, una técnica de Divide y vencerás.

1.1.- ¿Cómo introducir esta nueva funcionalidad?

Es sencillo. Como la única diferencia de usar una técnica de recorrida u otra radica en la forma de almacenar el árbol de ejecución, bastará redefinir el método de creación de dicho árbol. Es por esto por los que hemos incluido la clase abstracta TreeGUI de la que heredarán las clases que implementen las distintas estrategias.

En la aplicación, ya está incluida la clase **TreeGUIPreorden** que guarda los nodos del árbol gráfico en preorden. Si se quisieran incluir nuevas estrategias, se seguirán los siguientes pasos:

1) Crear una nueva clase TreeGUINuevaEstrategia que herede de la clase TreeGUI.

2) TreeGUINuevaEstrategia debe redefinir el método *createTreeGUI*, único método abstracto de la clase padre.

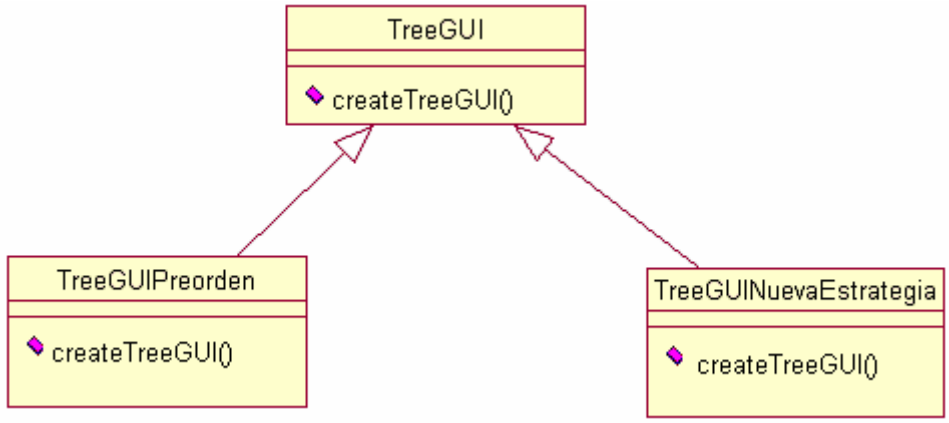


Figura 16: Inclusión de una nueva estrategia de recorrido

1.2.- ¿Cómo influyen estos cambios en la interfaz gráfica?

Este nuevo cambio debe ser reflejado en la interfaz gráfica de forma que se permita al usuario escoger entre las estrategias de búsqueda implementadas.

Una posibilidad sería crear un nuevo submenú que, nada más iniciar el proceso de depuración, permitiera el usuario escoger la estrategia. Según la opción elegida, se creará un objeto de la clase correspondiente.

2.- Documentación de los métodos

Una de las principales características del depurador declarativo es que el usuario sólo debe conocer el significado de cada método, no siendo necesario que conozca la implementación del mismo. Pero, es posible, que en un programa con un número elevado de métodos, el usuario no recuerde con detalle la funcionalidad de cada uno de ellos. Por ello, otra posible ampliación de la aplicación sería ofrecer al usuario la posibilidad de consultar información relativa al método que desee (javadoc).

Así, bastaría con seleccionar el nodo asociado a la llamada a un cierto método y escoger la opción de consulta de documentación. Dicha documentación, mostraría los siguientes aspectos:

- Funcionalidad del método: explica qué hace el método sin entrar en detalles de implementación.
- Parámetros: muestra los argumentos que recibe el método y lo que representa cada uno de ellos.
- Valor devuelto: informa sobre el tipo del valor devuelto por el método y lo que representa.

Esta información facilitará la tarea de depuración ya que, si en un momento dado, el usuario no sabe qué estado asignar a un método porque no recuerda la función de éste, podrá consultar la documentación anterior para que le ayude a decidir en el proceso.

2.1.- ¿Cómo introducir esta nueva funcionalidad?

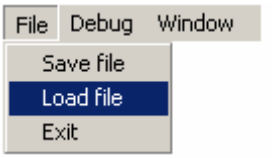
Basta con añadir una opción “Ver documentación” en la barra de herramientas o en un submenú asociado a algún evento de ratón, de forma que al seleccionar un método, se muestre un panel al lado de este con la información detallada anteriormente.

3.- Guardar sesiones de depuración

A la hora de depurar programas de gran tamaño, puede ser de utilidad para el usuario tener la posibilidad de salvar la depuración realizada hasta el momento para continuarla en otro momento. Así, se guardaría el estado actual del árbol, es decir, qué nodos se han depurado, cuáles son sus estados, etc. También podrían guardarse las sesiones de depuración a través de la consola, de forma que podrían insertarse comentarios en el archivo, que más tarde pudiesen servir de ayuda para el mismo usuario o incluso para otro.

3.1.- ¿Cómo introducir esta nueva funcionalidad?

En primer lugar, añadimos opciones al menú que permitan, tanto guardar la sesión de depuración como abrir una ya creada. Quedaría de la siguiente manera:



Para guardar la sesión de depuración, debemos llevar a cabo un proceso de serialización de los datos. Es decir, debemos encontrar un formato de representación de la información de aquellos objetos que queramos guardar. Para lograr este propósito, implementaremos la interfaz *Serializable* definiendo los métodos para serializar y deserializar.

La serialización permite crear a partir de las variables y propiedades públicas de una clase un objeto serializado. La deserialización permite crear objeto con la información deserializada proveniente de un objeto ya serializado. Nosotros proponemos almacenar los datos de la sesión de depuración en formato XML.

Por ejemplo, en nuestro caso, deseamos guardar el árbol de ejecución en el estado actual. Para ello, debemos definir como serializables, tanto el árbol como los nodos, de forma que la serialización del árbol, incluirá la llamada a la serialización de los nodos:

```
<tree>
        <node>
                <state value = st/>
        </node>
</tree>
```

4.- Crear patrones de error

Otra posible mejora que se podría implementar en la aplicación, sería incluir patrones de error, de forma que los errores se clasificasen según su tipo. Se podría seguir la clasificación de Knuth [7] modificada para abarcar los errores de los programas orientados a objetos y de los programas concurrentes:

1.- Programas orientados a objetos: OO
2.- Programación concurrente: C
3.- Anomalías de diseño: DA
4.- Anomalías algorítmicas: A
5.- Equivocaciones: B
6.- Fallos en los datos: D
7.- Falta de memoria: F
8.- Language lossage: L
9.- Mismatches: M
10.- Falta de robustez: R
11.- Typographic trivia: T

5.- Incluir el depurador como una vista de Eclipse

Debido a la utilidad de la aplicación, resulta interesante la posibilidad de incluir el depurador declarativo como una herramienta más de Eclipse. Hemos implementado la interfaz gráfica con la tecnología SWT de forma que es sencilla la integración con Eclipse puesto que es el mismo entorno gráfico.

5.1.- ¿Cómo introducir esta nueva funcionalidad?

Para añadir una nueva vista al área de trabajo basta con seguir los siguientes pasos [8]:

1) Crear un plug-in [9]
2) Añadir una extensión de la vista al fichero plugin.xml.
3) Definir una nueva clase vista.

Un plugin es un programa que interactúa con otro para aportarle una función o utilidad específica, generalmente muy específica. Podríamos decir que es un "parche" que añade características nuevas. El PDE (Eclipse Plug-in Development Environment) proporciona un buen entorno para la creación de plug-ins y su integración con la plataforma de Eclipse.

Necesitamos crear un proyecto para el plugin y crear el fichero plugin.xml. Dicho fichero contiene una declaración del identificador del plugin, nombre, pre-requisitos, etc.

El fichero plugin.xml tiene la forma siguiente:

```
<?xml version="1.0" encoding="UTF-8"?>
	<plugin
		name="Views Plugin"
		id="org.eclipse.ui.articles.views"
		version="1.0.0"
		provider-name="OTI">

		<requires>
			<import plugin="org.eclipse.core.boot"/>
			<import plugin="org.eclipse.core.runtime"/>
			<import plugin="org.eclipse.core.resources"/>
			<import plugin="org.eclipse.swt"/>
			<import plugin="org.eclipse.ui"/>
		</requires>

		<runtime>
			<library name="views.jar"/>
		</runtime>

	</plugin>
```

Una vez tenemos el plugin, definimos la extensión de la vista. El XML siguiente se añade al fichero plugin.xml después del elemento *runtime* y contiene los atributos básicos para la vista: identificador, nombre, icono y clase.

```
<extension point="org.eclipse.ui.views">
	<view id="org.eclipse.ui.articles.views.labelview"
		name="Label View"
		class="org.eclipse.ui.articles.views.LabelView"
		icon="icons\view.gif"/>
</extension>
```

Por último, debemos definir la clase vista. Para ayudarnos la plataforma contiene una clase abstracta denominada org.eclipse.ui.part.ViewPart la cual implementa la mayoría del comportamiento que debe tener por defecto.

6.- Graph Object Inspector

A la hora de representar objetos hemos optado por una estructura JTree que mostramos en el Tree Object Inspector. Sin embargo, si quisiéramos mostrar estructuras de datos más complejas, la representación escogida no es adecuada. Esta es la principal limitación de nuestro proyecto.

Por ejemplo, si queremos visualizar una lista donde cada elemento apunta al siguiente lo más apropiado sería visualizar el objeto como un grafo. Esta utilidad aparece ya implementada en depuradores como el DDD [9] y tiene la siguiente forma:

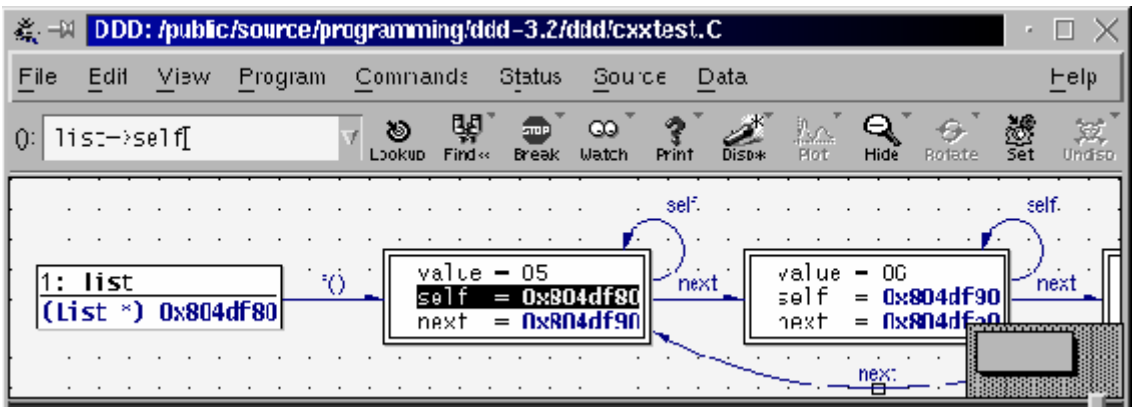
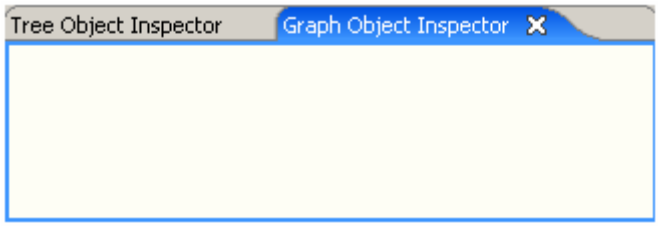


Figura 17: Representación de un objeto en forma de grafo (DDD)

6.1.- ¿Cómo introducir esta nueva funcionalidad?

Para solventar la limitación descrita podemos implementar una estructura de grafo similar a la anterior en una pestaña ya creada: Graph Object Inspector.



De esta forma, lo único que diferenciaría al Tree Object Inspector del Graph Object Inspector es la forma de representar los datos.

7.- Optimización del código

Una mejora siempre deseable a la hora de desarrollar un proyecto es conseguir un código lo más eficiente posible de forma que el uso de memoria sea reducido. El uso de la memoria constituye una limitación en el proyecto ya que el diseño de la interfaz ha requerido de la utilización de muchas imágenes.

7.1.- ¿Cómo introducir esta nueva funcionalidad?

En el código de nuestra aplicación podría mejorarse la eficiencia si se liberasen todos los objetos una vez dejan de ser usados. Este es el caso de las imágenes cargadas en la interfaz: se haría el *dispose* de cada una de las imágenes.

En la práctica, hacer el *dispose* de los objetos es delicado y puede llevar a resultados imprevisibles siempre que otro objeto intente tener acceso a un objeto ya liberado.

Otra optimización posible sería mejorar la meta-representación de las variables.

8.- Control de excepciones

En la aplicación que hemos desarrollado no mostramos en el árbol lógico las excepciones lanzadas por los métodos. Como es un depurador off-line, la ejecución del programa ha terminado al iniciar la sesión de depuración por lo que se sería conveniente mostrar esta información.

En nuestro caso, no está implementado pero el funcionamiento de la herramienta en este caso sería aceptable. Si existe un método en el programa que en la ejecución lanza una excepción, el árbol se generaría hasta la llamada a método anterior a la llamada que lanza la excepción. De esta forma, podría identificarse al instante que método en cuestión.

8.1.- ¿Cómo introducir esta nueva funcionalidad?

Deberíamos registrar la excepción en el momento que capturamos todos los eventos del programa (clase eventThread) y añadirla como un elemento más en la información del nodo.

Depurador Declarativo para Java

PRUEBAS DE LA APLICACIÓN

Depurador Declarativo para Java

En todo proceso de desarrollo de software es necesario probar la aplicación que se está implementando, para poder conseguir el correcto funcionamiento del programa en cuestión. Para ello se definen unos ejemplos o pruebas que intentan recoger todas las posibles situaciones en las que nuestra herramienta pueda fallar, o se comporte de manera inesperada.

En este documento incluimos todas las pruebas que hemos realizado a lo largo del desarrollo de la herramienta. Comenzaremos por pruebas muy sencillas que consisten en programas con métodos estáticos, y cuyos argumentos de entrada son de tipo primitivo, al igual que el valor devuelto. Continuaremos con pruebas, algo más complejas que incluyan objetos, y en las que las funciones sean métodos que pertenezcan a una clase. El valor devuelto puede ser tanto un objeto de una clase como un tipo simple. Además, en estas pruebas incluiremos estructuras como arrays. Para finalizar, probaremos nuestra herramienta con programas típicos como el que implementa el algoritmo de las Torres de Hanoi, para ver su funcionamiento con programas recursivos; y haremos diversas pruebas del comportamiento de la aplicación con objetos y referencias entre ellos (referencias cruzadas).

Al disponer de una estructura modular, deben existir pruebas independientes para cada componente. Por ello, en lo que se refiere al módulo gráfico fue necesario definirse una estructura de datos de prueba para representar el árbol de ejecución. Ésta equivaldría, en un futuro, a la estructura que el módulo gráfico recibe del módulo lógico (representada por el paquete **dataTree**). Consistía en una representación basada en cadenas de caracteres.

Aparece, por tanto, un nuevo componente en la estructura del proyecto: el paquete **dataTreeTest**. Está organizado de la misma forma que el dataTree. En él se definen las mismas clases, con los mismos métodos. La única diferencia está en que el árbol de ejecución no es un árbol real, sino uno de prueba. Esto nos permitió probar el módulo gráfico antes de integrarlo al lógico. Además, al mantener la estructura presente en el paquete dataTree la posterior integración resultó muy sencilla.

Igualmente ocurrió con el módulo lógico. A diferencia del gráfico este sí contaba con un árbol real. Recordemos que éste es el módulo que se encarga de construir el árbol de ejecución que se desprende directamente del código del programa que se quiere depurar.

Una vez llevado a cabo el proceso de integración de los dos módulos principales que describen la estructura de nuestro proyecto, fue preciso definir los ejemplos de ejecución que lo probaran. Las distintas pruebas descritas a continuación incluyen el código del programa que se va a ejecutar, una breve explicación de lo que hace el programa, lo que estamos probando en cada uno de ellos, y, por último el árbol de ejecución que obtenemos con cada una de las pruebas.

Prueba 1

Programa básico. Este ejemplo simplemente prueba si el árbol de ejecución se genera adecuadamente. Todos sus métodos son estáticos, y por tanto, exceptuando la llamada a la constructor, el resto de las llamadas a los métodos no se hacen mediante ningún objeto invocador (no existe “caller object”).

Aunque esta prueba resulte muy básica, en realidad se puede comprobar la mayor parte de la funcionalidad de la herramienta. Lo vemos por secciones.

DebugTreeWindow

Aquí se puede comprobar si el árbol se genera correctamente, es decir si aparece un nodo por cada una de las llamadas a los métodos que presenta el programa, y si la jerarquía de los nodos es la correcta. Es decir, si las llamadas a los métodos que se hacen en un método concreto aparecen, en el árbol, como hijos de dicho método.

Object Inspector

Debido a la sencillez el funcionamiento del object inspector no puede comprobarse en esta prueba. Por el contrario, podremos hacerlo en pruebas posteriores.

Funcionalidades del menú y de la barra de herramientas

Es la más interesante de todas las secciones, ya que permite probar el depurador en sí. Tanto desde la barra de herramientas, como desde el menú podemos iniciar una sesión de depuración guiada o bien establecer el estado de un nodo (valid, non-valid, don´t know, trusted) de forma manual.

También, podemos observar si la ejecución guiada recorre el árbol en el orden que corresponde con la estrategia de depuración que hemos elegido, (pre-orden por defecto). Igualmente si un nodo ya tiene asignado un estado, el depurador debe saltarlo, ya que se supone que el usuario ha seleccionado ya el estado de ese nodo y no quiere cambiarlo. Opciones tales como parar una ejecución, y pasar de un nodo a otro desde cualquier nodo, son más funcionalidades que se pueden probar con este ejemplo.

Por último, y más importante, conseguimos en esta prueba descubrir si el depurador es capaz de detectar un error o no. En cuyo caso deberá abortarse el proceso de depuración, es decir, en el caso de encontrar un nodo incorrecto, cuyos nodos hijos son todos correctos.

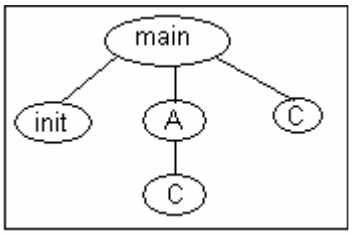
Ventana del código fuente

En la ejecución de nuestra aplicación, al hacer doble-clic sobre alguno de los nodos del árbol, debe mostrarse el código fuente del programa en la ventana que se encarga de ello. Aparecerá resaltado en azul, el método que corresponde a la llamada que está representada en el nodo seleccionado. Si al marcar un nodo como erróneo y confirmar su estado el depurador encuentra un posible error, en el código fuente aparecerá resaltado el método erróneo, esta vez en color rojo.

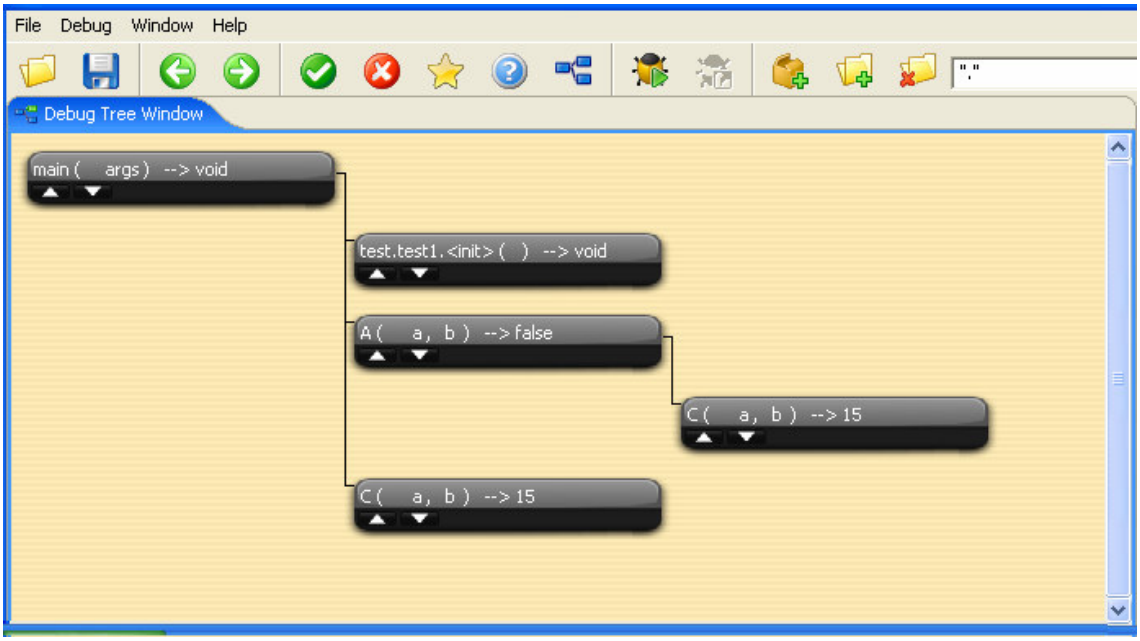
A continuación, mostramos el código fuente que corresponde a esta prueba, y construiremos el árbol de ejecución que debería mostrarse, para su posterior comprobación con lo que se obtiene en la ejecución de la herramienta.

```
public class test2 {

    public test2() {
        super();
    }
    public static boolean A(int a, int b){
        return C(a,b)==0;
    };

    public static int C(int a, int b){
        return a*b;
    };
    public static void D() throws Exception{
        throw new Exception("Salta!!!!!!!!");
    };
    /**
     * @param args
     * @throws Exception
     */
    public static void main(String[] args) throws Exception {
        test1 prim = new test1();

        boolean falso = A(3,5);
        int quince = C(5,3);
        //prim.primo(quince);

    }
```

El árbol que deberíamos obtener es el siguiente:



Comprobamos, finalmente, que es el obtenemos en la ejecución del programa.



Prueba 2

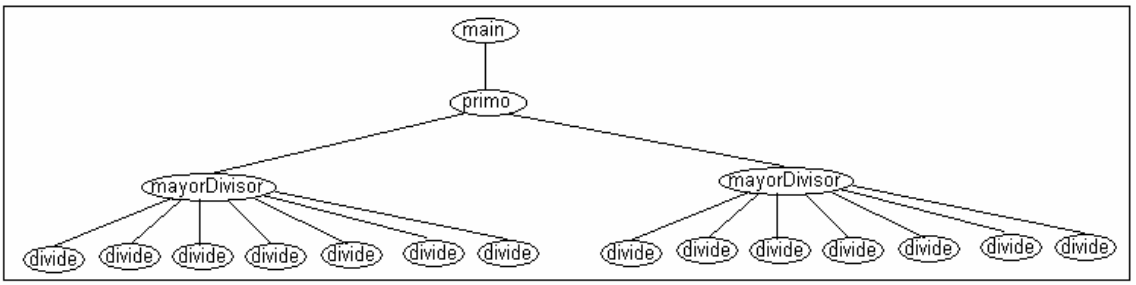
La prueba 2, es similar a la prueba 1 en cuanto a las características del programa fuente que se va a ejecutar, la diferencia fundamental se encuentra en que es un que genera un árbol con muchos más nodos, ya que hay muchas más llamadas al métodos.

Este tipo de prueba nos permite corroborar que el árbol se genera bien para programas algo más complejos. Además, al no incluirlo en la prueba anterior, visualizaremos en esta la pantalla de código, y cómo se muestra en rojo l método en el que el sistema de depuración ha encontrado un error.

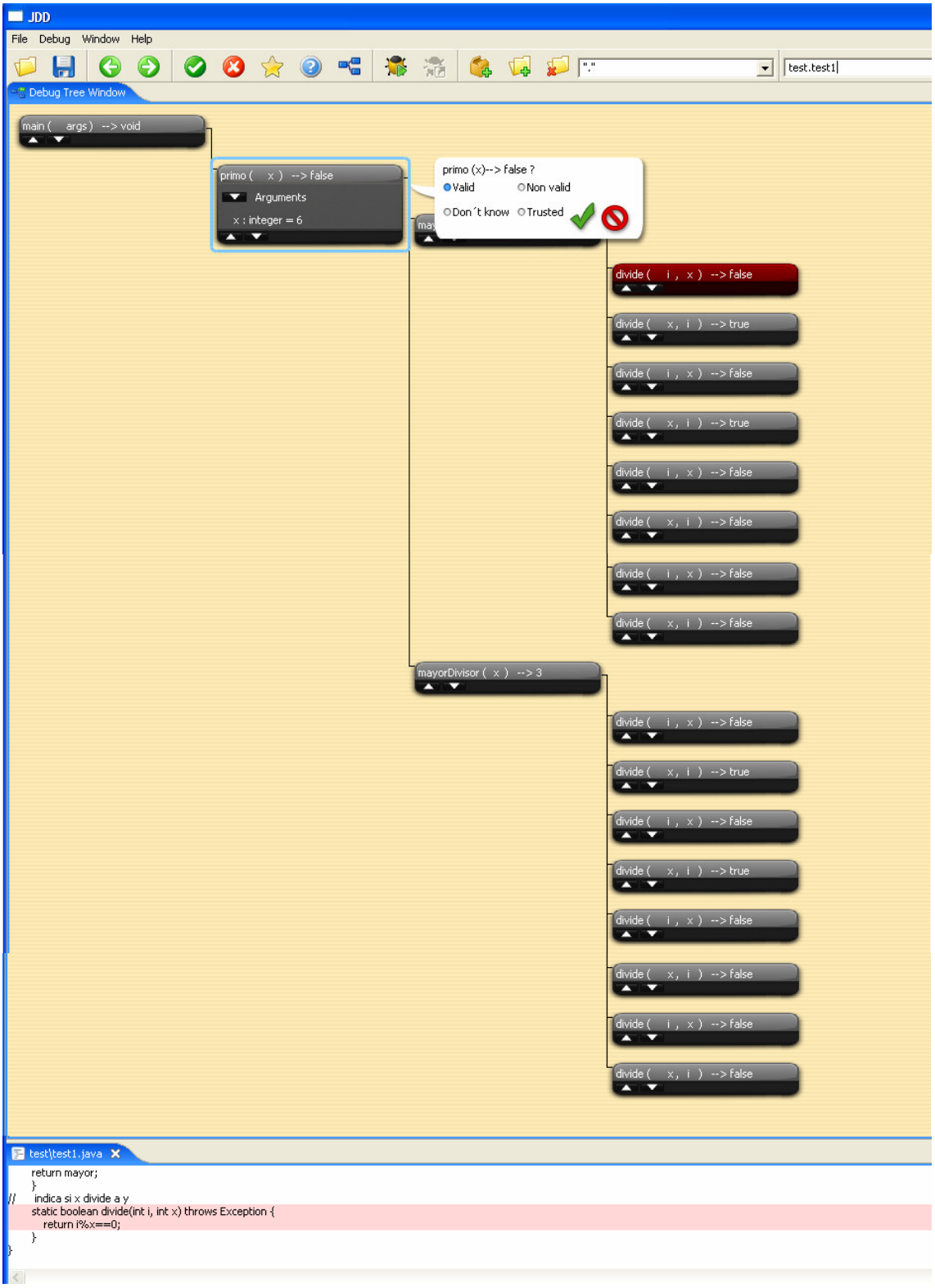
El programa que se está ejecutando es el que se muestra a continuación:

```
package test;
public class test1 {

    public static void main(String []args) throws Exception{
        test1 a;
        int x=6; // para probar
//       probamos si x es primo


        if (primo(x)) System.out.println(x+"no es primo");
        else System.out.println(x+" es primo");
        }
        static boolean primo(int x) throws Exception {
//       es primo si el mayor divisor menos que el mismo es 1
        System.out.println("mayordivisor("+x+") -->"+ mayorDivisor(x));
        return (mayorDivisor(x)==1);
        }
//       debe devolver el mayor divisor de x (distinto del propio x)
        static int mayorDivisor(int x) throws Exception {
        int mayor=1;
//       i representa los posibles divisores
//       empezamos en dos porque el 1 divide con seguridad
        for (int i=2; i<x; i++) {
        System.out.println("divide("+x+", "+i+") --> " +divide(i,x));
        if (divide(x,i) && i>mayor) // tenemos un divisor mayor que los encontrados hasta ahora
        mayor = i;
        }
        return mayor;
        }
//       indica si x divide a y
        static boolean divide(int i, int x) throws Exception {
            return i%x==0;
        }
```

Como podemos ver en el código, la llamada a la función “divide” está dentro de un bucle for, lo que generará un gran número de llamadas. El árbol que deberíamos obtener en la ejecución del programa es el siguiente:



Efectivamente, es lo que obtenemos al ejecutar este mismo programa en nuestra aplicación:



No debemos olvidar que un ejemplo de este estilo es idóneo para probar el funcionamiento del estado “trusted”. Cuando establecemos el estado de un

nodo del árbol a “trusted” indicamos que confiamos en ese método, es decir, que todas las llamadas que se hagan a ese método las consideraremos

correctas. Una llamada es igual a otra llamada si el nombre del método coincide, el tipo de los argumentos de entrada es el mismo, al igual que el tipo del valor de retorno del método. En el caso de esta prueba si pusiésemos a trusted una de las llamadas al método divide, automáticamente se validarían el resto de los nodos con esa misma llamada.

Prueba 3

Este ejemplo se diseñó para poder probar la ventana del object inspector de la aplicación. Esta ventana tiene la cualidad de permitir al usuario evaluar el contenido de un objeto que ha seleccionado en el árbol de ejecución, ya que puede ver los distintos atributos del objeto correspondiente y el valor que toman éstos. Además en la ventana deberá aparecer el nombre método del que proviene ese objeto.

En el object inspector deben aparecer objetos de todo tipo, tanto si constituyen los argumentos de entrada de la llamada al método, como si se trata del objeto invocador del método o del propio valor de retorno.

El programa que elegimos a continuación debe tener las características que acabamos de citar. Es decir, debe constar de un método que devuelva un objeto como valor de retorno, al igual que sus argumentos deben ser objetos de una clase determinada. En esta prueba se suprimen los métodos estáticos que aparecían en las anteriores, de manera que cada llamada a un método se hace a partir de un objeto.

A demás en esta prueba se definen métodos mutadores que modifican atributos del objeto que los invoca. Según, la especificación de nuestro proyecto, un objeto está modificado cuando alguno de sus atributos se modifica. Cuando el depurador se encuentra con un objeto de esta clase, debe marcarlo en rojo. Igualmente debe ser posible conocer el valor antiguo y el valor nuevo de ese atributo, esto se mostrará en la ventana de visualización del object inspector. En caso de que el objeto modificado sea uno de los argumentos de entrada, este deberá aparecer en rojo, en la propia cabecera. Esta última situación, no aparece en la prueba 3, pero si en las definidas más adelante.

Después de todas estas consideraciones, el ejemplo que correspondería a esta prueba deberá contener dos clases distintas. La clase test3, que es la principal, que contiene el método main. Y la clase figura. El código del programa de prueba sería el siguiente:

Código de la clase test 3:

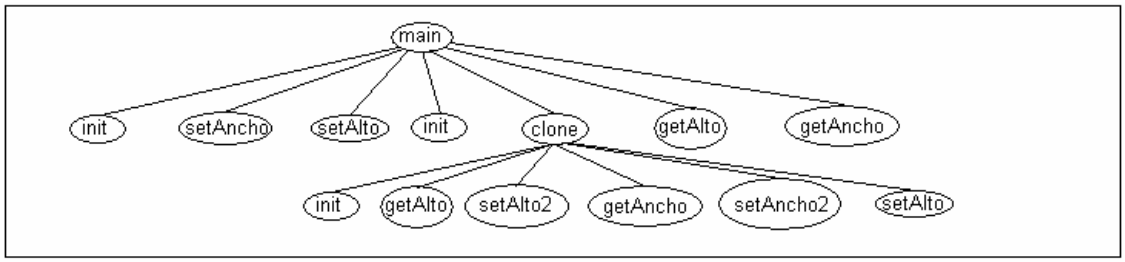
```
public class test3 {

    /**
    * @param args
     */
    public static void main(String[] args){
        Figura f = new Figura();
        f.setAncho(5);
        f.setAlto(6);
        Figura g = new Figura();
        g = g.clone(f);
        System.out.println(f.getAlto());
        System.out.println(f.getAncho());
    }

    public static int Area(Figura f){
        return f.area();
    };
}
```

Código de la clase Figura:

```
public class Figura {

    /**
     * @param args
     */
    int ancho;
    int alto;

    public Figura(){
        ancho = 0;
        alto = 0;
    }

    public Figura(int _ancho,int _alto){
        ancho = _ancho;
        alto = _alto;
    }

    public int getAncho(){
        return ancho;
    }

    public int getAlto(){
        return alto;
    }
```

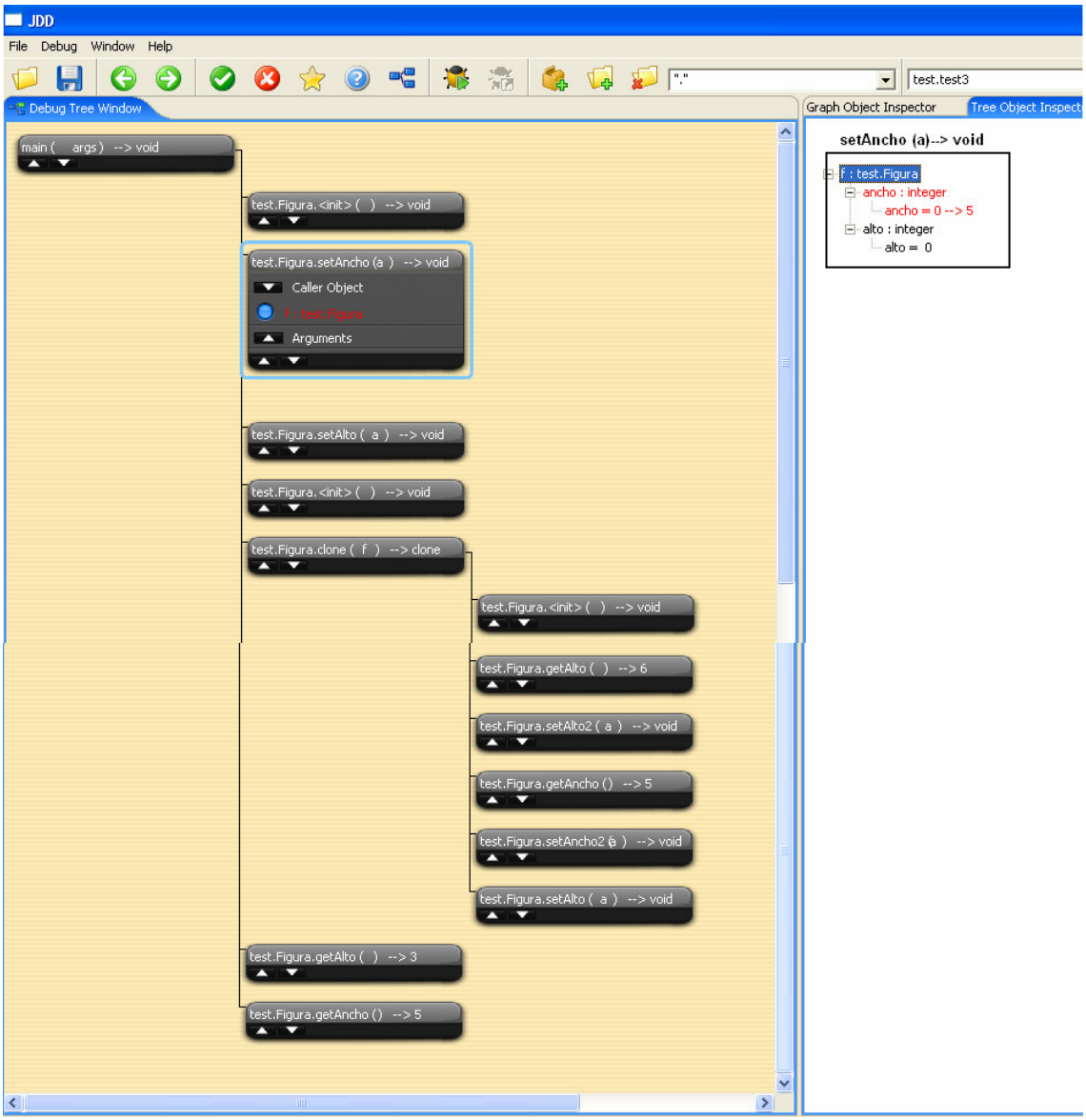
```
public int area(){
    return ancho*alto;
}

public void setAncho(int a){
    ancho = a;
}

public void setAlto(int a){
    alto = a;
}
private void setAncho2(int a){
    ancho = a;
}

private void setAlto2(int a){
    alto = a;
}
public Figura clone(Figura temp)
{
    Figura aux = new Figura();
    aux.setAlto2(temp.getAlto());
    aux.setAncho2(temp.getAncho());
    temp.setAlto(3);
    return aux;
}
```

El árbol que deberíamos obtener al ejecutar nuestro programa con el test3 debería ser el siguiente:



Como podemos observar en la imagen anterior ante una llamada del estilo: “setAlto2 (temp.getAlto)”, se deben crear dos nodos a la misma altura ya que ambos métodos de invocan en el método clone, no uno dentro del otro.

El árbol que obtenemos al ejecutar la aplicación lo mostraremos en la siguiente figura. Recalcamos la visualización del objeto seleccionado en el object inspector. Mostramos un objeto en el que se ha modificado su atributo ancho, ha pasado de tener valor 0 a tener valor 5.



Prueba 4

En esta prueba comprobamos el funcionamiento de nuestra herramienta ante las referencias cruzadas. Vemos lo que entendemos por referencia cruzada con un ejemplo:

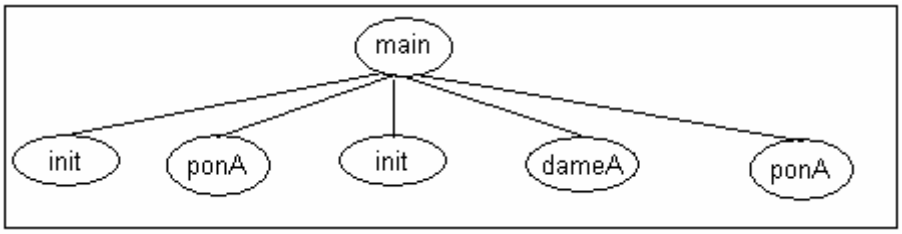
```
class A{
      public A a;

      public A dameA(){
            return a;
      }
      public void ponA(A mia){
            a = mia;
      }
}
public class test4 {
      public static void main(String[] args) {
```

```
                A uno = new A();
                uno.ponA(uno);
                A dos = new A();
                dos.ponA(uno.dameA());
}
```

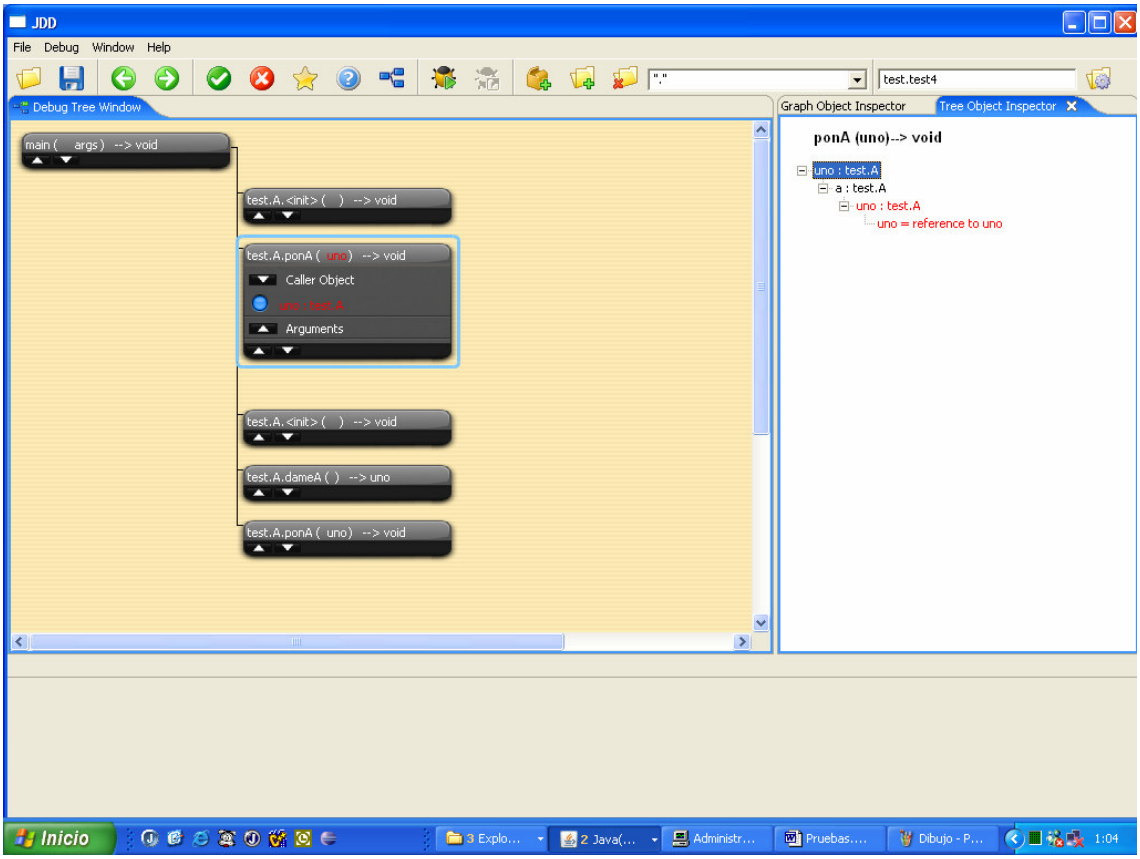
Creamos una clase (class A), que tiene un atributo de esta misma clase (a). En el código de la función main, nos declaramos un objeto de la clase A, y asignamos el valor de su atributo a continuación. De esta manera estamos creando una referencia así mismo. Esto lo denominamos **referencia cruzada**. A continuación, declaramos otro objeto (dos), al que asignamos el valor del objeto declarado anteriormente.

Como el objeto se referencia así mismo a la hora de representarlo podríamos obtener un bucle infinito. Para evitar esta situación, en cuanto el depurador detecta una referencia cruzada, asigna al objeto el valor *"reference to object"*, donde "object" es el objeto que tiene una referencia así mismo.

Con este código, el árbol que deberíamos obtener es el siguiente



Comprobamos que el resultado que obtenemos es el mismo al ejecutar nuestra aplicación:



Prueba 5

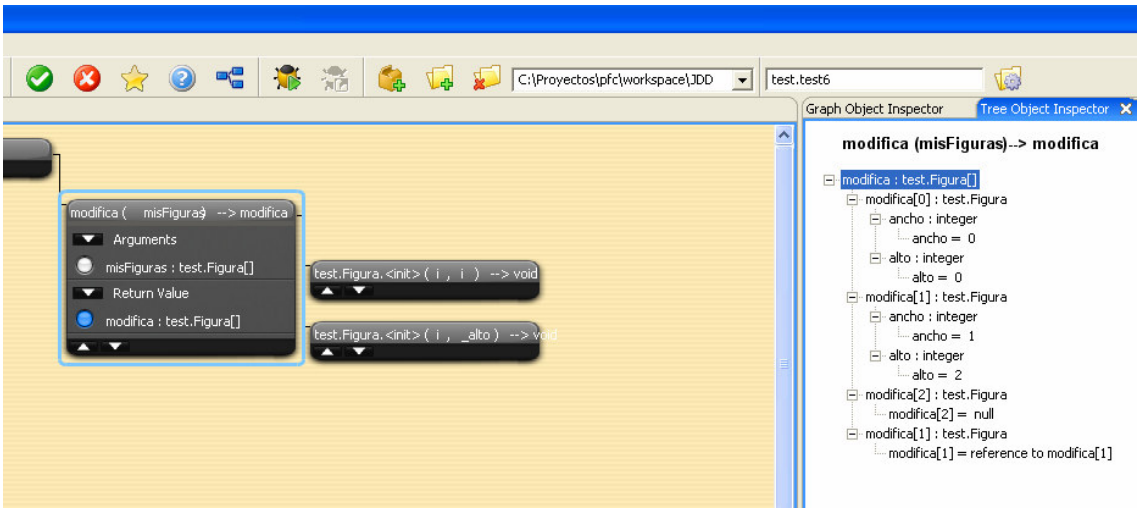
En esta prueba se introducen los arrays. Hasta ahora los argumentos y valores de retorno representados eran o bien de tipo simple o bien objetos de una clase determinada. Por ello, resultaría interesante comprobar cómo funciona nuestra herramienta cuando nos declaramos un array.

La prueba consiste en declararnos un método que devuelve como valor de retorno un array con un par de posiciones inicializadas a valores de tipo primitivo. En la ejecución que mostramos, seleccionamos el array en cuestión, y desplegamos sus elementos en el object inspector, para poder conocer los valores que va tomando.

El código a ejecutar es el siguiente:

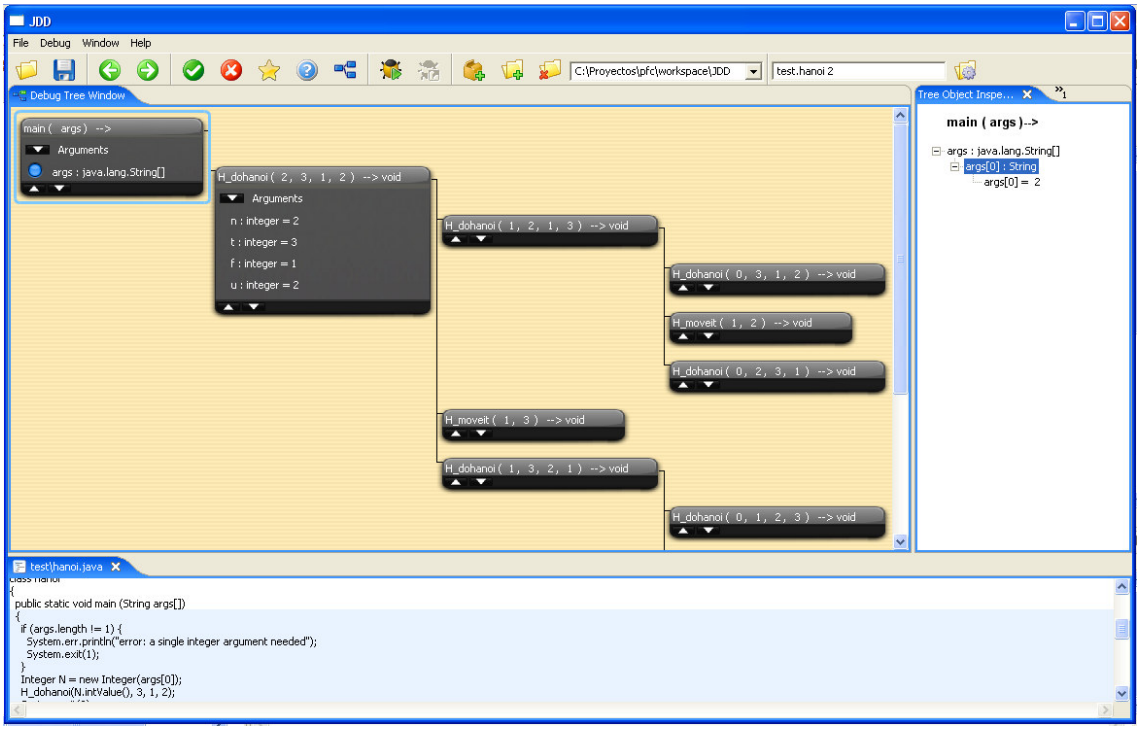
```
package test;

public class test6 {

   public static Figura[] modifica(Figura[] b)
   {
      Figura[] misFiguras = new Figura[b.length];
      for(int i = 0;i<misFiguras.length/2;i++)
      {
         Figura f = new Figura(i,i*2);
         misFiguras [i] = f;
         misFiguras [b.length-1]= misFiguras[i];
      }
      b = misFiguras;
      return b;
   }



   /**
    * @param args
    */
   public static void main(String[] args) {
      // TODO Auto-generated method stub

   Figura[] misFiguras = new Figura[4];

   modifica(misFiguras);



   }

}
```

El resultado de en la ejecución de nuestra herramienta es el siguiente:



Prueba 6

Para finalizar decidimos comprobar el comportamiento de nuestro programa ante un test que no ha sido escrito por nosotros. Para ello, probamos el depurador con el código del programa que implementa el conocido algoritmo de las Torres de Hanoi. La prueba la realizamos para 5 discos y lo que obtuvimos fue el resultado que aparece en la siguiente figura.



Con esta prueba demostramos que nuestro depurador puede funcionar correctamente para un programa algo mayor, ya que el coste de este algoritmo es exponencial, y por tanto se genera un árbol con un número considerable de nodos.

Depurador Declarativo para Java

Depurador Declarativo para Java

MANUAL DE USUARIO

Depurador Declarativo para Java

Este manual servirá al usuario para familiarizarse con el depurador. El diseño del interfaz de usuario permite un manejo sencillo de la herramienta. Está implementado con la tecnología SWT de forma que facilita la integración con Eclipse ya que comparten el entorno gráfico.

Introducción al interfaz gráfico

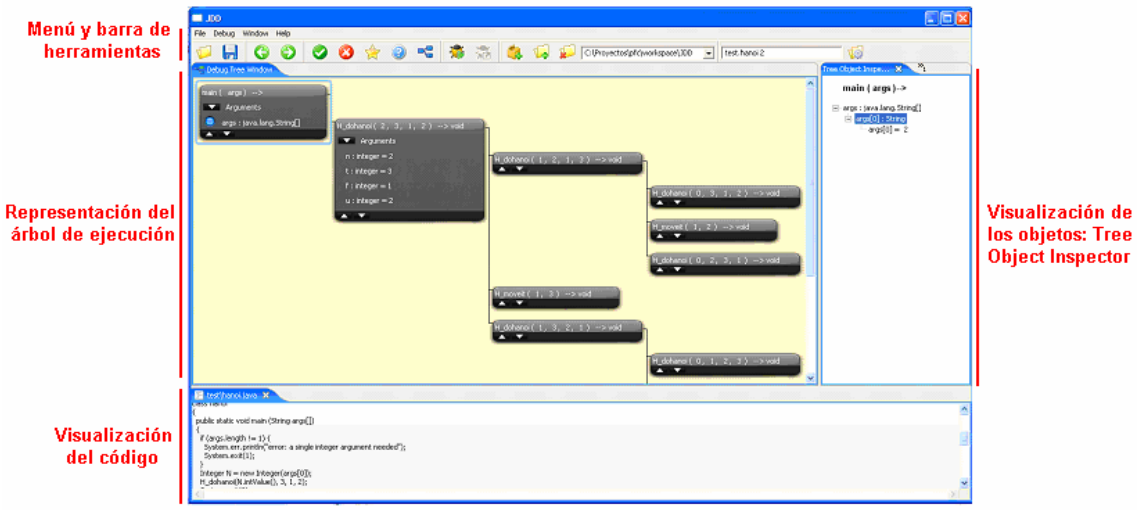
Con el fin de mostrar la información lo más sencillamente posible, dividimos la interfaz de usuario en 3 paneles básicos:

- *Ventanas de visualización del código fuente*: haciendo doble-clic sobre un nodo, mostramos en una ventana situada en la parte inferior del interfaz, el código de la clase a la que pertenece el método asociado al nodo. Dicho método, se mostrará sombreado en azul.

- *Ventana para la inspección de objetos:* el objeto seleccionado en el árbol de ejecución se mostrará en el Tree Object Inspector (situado en la parte derecha del interfaz) con el fin de que el usuario pueda evaluar cada uno de sus atributos.

- *Ventana de visualización del árbol de cómputo:* el árbol de ejecución se mostrará en la DebugTreeWindow, de forma que cada uno de sus nodos puedan ser desplegados para su inspección. El proceso de depuración se realizará sobre dicho árbol de forma que se identificará gráficamente (por colores) el estado de cada nodo.

Además, se proporciona una barra de herramientas y un menú que permiten realizar cada una de las acciones tanto de la depuración manual como de la guiada.



Seleccionar un Classpath

Para modificar un classpath previamente definido o para seleccionar uno nuevo, basta con añadirlo en la barra de herramientas:

Existen dos posibilidades:

- Añadir un archivo .jar. Para ello, se pulsa sobre  . A continuación se abrirá un cuadro de diálogo que nos permitirá especificar la ruta donde está localizado el archivo.
- Para añadir una ruta pulsar sobre  . En el cuadro de diálogo que se abre, seleccionar el directorio que se desee.

El classpath elegido se mostrará en el cuadro de texto de la barra de herramientas.

Eliminar un Classpath

Si se desea eliminar un classpath previamente seleccionado, debemos colocar dicho classpath como primera opción del cuadro de texto:

Una vez marcado el classpath que se desea eliminar, se pulsa sobre  .

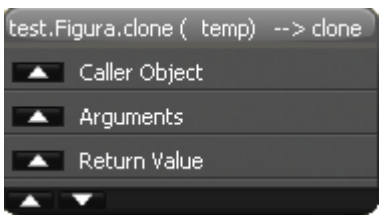
Crear un nuevo árbol de ejecución

Para crear un nuevo árbol de ejecución, es necesario especificar el fichero .java que se desea visualizar y depurar. Para ello, escribimos el nombre de la clase en el campo de texto marcado a continuación así como los parámetros que recibe dicha clase. Una vez hecho esto, pulsamos sobre  en la barra de herramientas.

Visualización de los nodos

Los nodos, por defecto, vienen plegados, es decir, únicamente muestran la información referente al nombre del método, los argumentos que recibe, y el valor de vuelto:

Si se desea desplegar el nodo para ver información más detallada acerca del método, basta con pulsar sobre  , con lo que obtenemos:



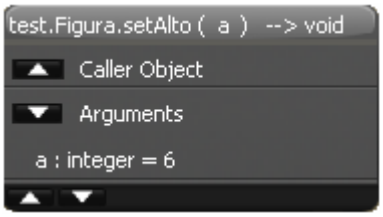
Si quisiéramos volver a plegar el nodo, pulsamos sobre  .

Visualización de la información

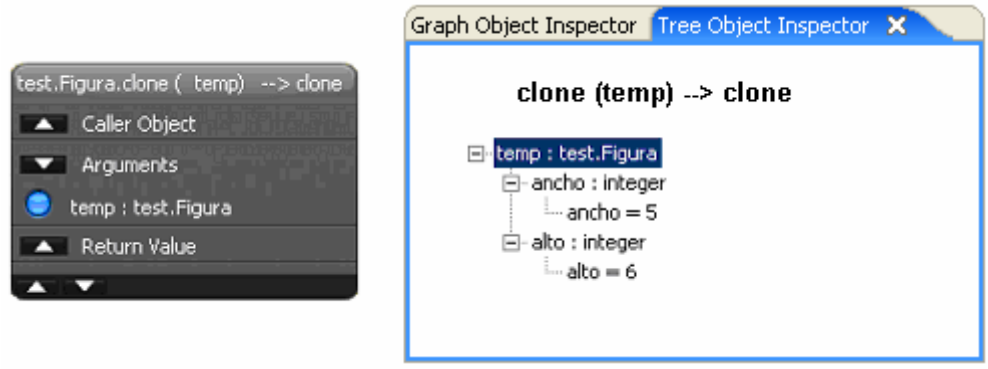
Una vez desplegado el nodo *(Ver Visualización de objetos)*, podemos desplegar cada una de sus componentes mediante el botón  .

A la hora de mostrarse la información pueden suceder dos cosas:

• Si el parámetro es de un tipo primitivo (real, entero,...), se mostrará la información en el mismo nodo especificando el nombre de la variable, su tipo y su valor



• Si el parámetro es un array o un objeto, deberemos seleccionar dicho objeto pulsando sobre su icono. Su información se verá entonces en el Tree Object Inspector:

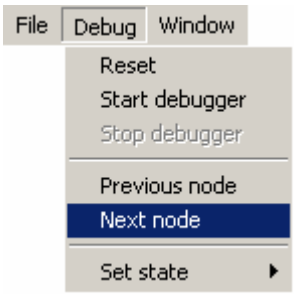


Recorrer el árbol

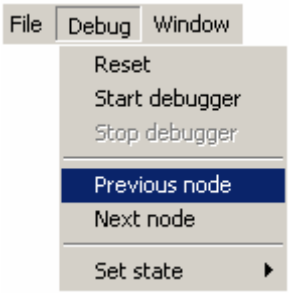
Podemos recorrer el árbol en pre-orden tanto hacia delante como hacia atrás.

En primer lugar, y sino hay ningún nodo seleccionado, debemos marcar uno. Para ello, basta con pinchar sobre el nodo que se desee. El nodo seleccionado aparecerá distinguido como a continuación:

Una vez seleccionado, podemos pasar al siguiente nodo pulsando sobre  , o bien:



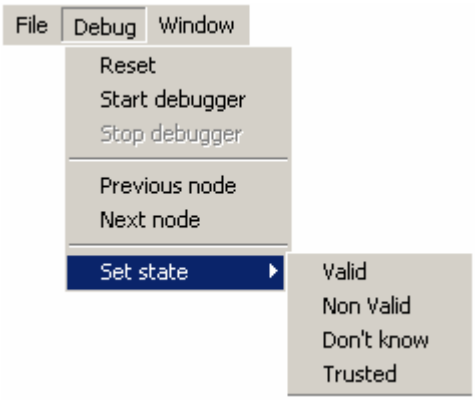
Si por el contrario, deseamos pasar al nodo anterior, pulsamos sobre  , o bien:



Depuración manual

Por defecto, la aplicación está en modo depuración manual, es decir, podremos modificar el estado de los nodos simplemente seleccionando un nodo y marcando su estado. Para ello, una vez seleccionado el nodo a validar, existen tres alternativas para asignarle estado:

- A través del menú Debug, seleccionando la opción Set State:



• A través de la barra de herramientas seleccionando el estado que se desee:

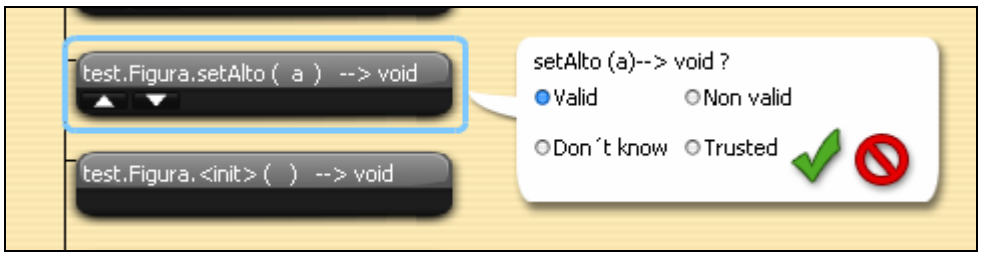
 Si el nodo es válido

 Si el nodo no es válido

 Si con la información de la que disponemos no sabemos si el nodo es válido o no

 Si "confiamos" en el nodo. Es decir, Todos los nodos que sean iguales al seleccionado, serán considerados también válidos.

• Una vez encima del nodo, pulsamos sobre el botón derecho del ratón. Aparecerá un diálogo donde se pregunta al usuario por el estado del nodo.



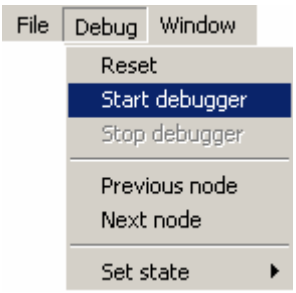
Seleccionamos el estado del nodo y pulsamos sobre  para confirmar, o  para cancelar.

Iniciar depuración guiada

La depuración guiada se diferencia de la manual en la forma de recorrer el árbol. Si en la manual, era el usuario el que escogía en cada momento el nodo al que asignar estado, en la depuración guiada será la propia aplicación la que recorra el árbol preguntando al usuario por el estado de los nodos. Al igual que en la depuración manual, la ejecución terminará cuando se encuentre el nodo responsable del error o cuando el usuario decida finalizar.

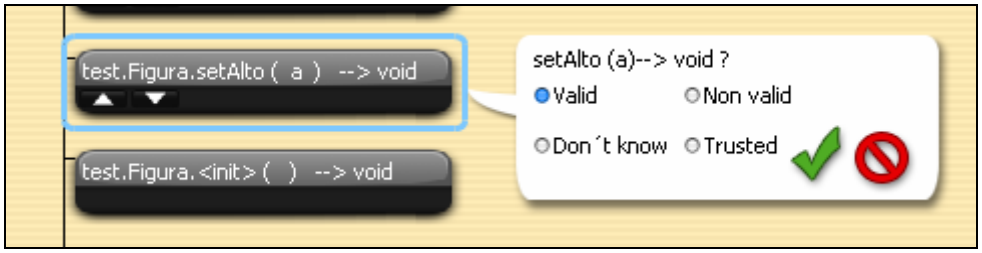
En primer lugar, se escoge el nodo a partir del cual queremos comenzar a depurar. Si no se especifica ninguno, se comenzará por el primer nodo.

Para iniciar la depuración guiada puede hacerse a través del menú o a través de la barra de herramientas:



o bien, pulsar sobre  .

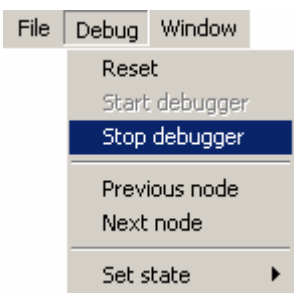
A continuación, se señalará el nodo a validar y se preguntará al usuario por su estado:



Una vez seleccionado el nuevo estado se confirma la elección pulsando  para confirmar, o cancelamos con  . La aplicación, pasará entonces a preguntar por el siguiente nodo del árbol.

Finalizar depuración guiada

Si estamos en modo depuración guiada y no se ha encontrado aún el nodo causante del error (la aplicación continúa depurando), podemos parar la depuración a través del menú o a través de la barra de herramientas:



o bien, pulsando sobre  .

Reset del árbol de ejecución

Si se desea desmarcar los nodos ya validados, es decir, partir del árbol inicial en el que ningún nodo tiene asociado estado se puede hacer de dos formas:

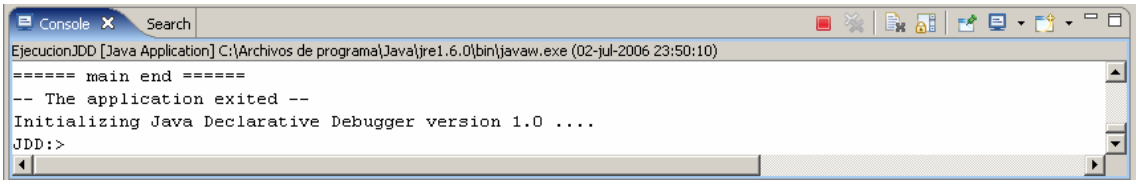
- Seleccionar en el menú Debug la opción Reset (Debug→ Reset), o bien
- Pulsar sobre el icono  .

Salir del programa

Para salir del programa, seleccionar en el menú File la opción Exit (File→ Exit).

Manejo de la consola

La aplicación cuenta también con un shell con el que podrán hacerse las mismas operaciones que a través de la interfaz.



A continuación, se explican los comandos implementados en la aplicación:

- **run <class> <args>** : ejecuta la clase especificada
- **classpath**: muestra el classpath
- **list** : muestra el código asociado al nodo actual
- **where**: muestra el nodo actual
- **where <levels>**: muestra el árbol desde el nodo actual hasta la profundidad indicada en <levels>
- **dumpTree**: muestra el árbol de depuración
- **callerObject**: muestra el caller object del nodo actual
- **arguments**: muestra los argumentos del nodo actual
- **returnValue**: muestra el return value del nodo actual

- **set <state>**: asigna al nodo actual el estado de depuración especificado en <state>
- **next** :siguiente nodo a depurar
- **prev**: nodo previo al actual
- **version**: muestra la versión de la aplicación
- **exit**/**quit** :cierra la aplicación
- **help**: muestra al usuario los posibles comandos ejecutables en la consola.

Depurador Declarativo para Java

CONCLUSIONES Y RESULTADOS

Depurador Declarativo para Java

Una vez explicado en profundidad tanto el concepto de depuración declarativa como el funcionamiento concreto de nuestra aplicación, podemos establecer una serie de conclusiones que nos permitan tener una idea clara de ambos aspectos.

La depuración ha supuesto siempre una parte muy costosa del desarrollo del software. Por ello, es lógico que se investiguen procedimientos que permitan realizar el proceso de depuración de la forma más eficiente y rápida posible. Probablemente, el tipo de depuración más extendido sea la depuración de traza o imperativa. Consiste en recorrer el programa erróneo línea a línea hasta que se encuentra el error que provoca que el programa no termine de la forma esperada. Es obvio suponer, que en programas de gran tamaño el proceso de depuración puede resultar tedioso.

En contraposición a la depuración de traza, aparece la depuración declarativa. Esta técnica pretende simplificar la búsqueda del error ahorrando tiempo al usuario.

La depuración declarativa es un método de búsqueda de errores en programas basado en la semántica de los lenguajes. De esta forma, el proceso de depuración no recorre el programa línea a línea sino que sólo se interesa por las llamadas a los métodos. Con estas llamadas creará un árbol de ejecución que será el que se examine para encontrar el error. El proceso a seguir es el siguiente:

1. Crear el árbol de ejecución a partir de las llamadas a los métodos y estableciendo relaciones padre-hijo donde un nodo Y es hijo de un nodo X, si el método representado por X hace una llamada al método Y. Cada llamada a un método almacenará los argumentos que recibe, el objeto que lo llama y el método que devuelve. También mostrará el cambio que se producen en los objetos.

2. Recorrer, de forma guiada o manualmente el árbol asignando un estado a cada nodo. Denominamos estado a la evaluación que el usuario hace de la llamada a un método. Es decir, si el usuario observa que la llamada a un método es correcta marcará el nodo asociado como válido. Si por el contrario, el comportamiento de la llamada no es el pretendido, marcaremos el nodo como no válido. El proceso termina cuando se encuentra la llamada errónea.

3. Una vez localizado el error, hay que repasar el código del método (aquí puede utilizarse el depurador de traza) para resolverlo.

Como puede verse, el uso del depurador de traza se limita a recorrer un sólo método, aquel que le ha indicado el depurador declarativo. Se demuestra entonces que el tiempo empleado en la depuración con la técnica declarativa es menor que usando únicamente la depuración paso-a-paso.

Otra ventaja que añade el depurador declarativo es el hecho de que el usuario que depura el código, no tiene porqué ser el mismo que lo desarrolló, ya que basta con que el usuario que pretende depurar el código conozca el significado de cada método, dejando a un lado los detalles de implementación.

La aplicación JDD (Java Declarative Debugger) está basada en las ideas expuestas proporcionando una herramienta eficiente para el usuario. Nuestra aplicación aporta:

- *Intuitividad y sencillez*: el diseño de la interfaz permite al usuario un fácil manejo del depurador.

- *Claridad*: la representación de árbol elegida así como su visualización por pantalla hacen que el usuario tenga una visión clara de la estructura del programa a depurar.

- *Organización*: la estructura de la herramienta agrupa la visualización de objetos de una sección aparte para que puedan examinarse sin agrandar el esquema árbol. Además, ayuda al usuario ofreciéndole la posibilidad de repasar el código si así lo desea.

A la vista del resultado final, consideramos que hemos conseguido lo siguiente:

- Aplicar la técnica de depuración declarativa a la programación orientada a objetos.

- Desarrollo de una interfaz gráfica intuitiva que facilite al usuario el proceso de depuración.

- Establecer una plataforma para probar diversas estrategias de depuración declarativa.

- Ofrecemos otra herramienta innovadora y útil para depurar programas.

BIBLIOGRAFÍA

Depurador Declarativo para Java

[0] SHAPIRO, E. Y. 1983. Algorithmic Program Debugging. MIT Press, Cambridge, Mass.

[1] PEREIRA, L. M. 1986. Rational debugging in logic programming. In Proceedings of the 3rd Logic Programming Conference (London, England, July). 203-210.

[2] DRABENT, W., NADJM-TEHRANI, S., AND MALUSZYNSKI, J. 1988. The use of assertions in algorithmic debugging. In Proceedings of the FGCS Conference (Tokyo), 573–581.

[3] Bernie Pope. Department of Computer Science at the university of Melbourne.

[4] HANI Z. GIRGIS, BHARAT JAYARAMAN. JavaDD: A Declarative Debugger for Java. University of Buffalo, New York (2006)

[5] LOTON, TONY. Using Platform Debugger Architecture (2001): www2.sys-con.com/ITSG/virtualcd/Java/archives/0603/loton/index.html

[6] SWT: Para conocer el funcionamiento de la tecnología nos hemos ayudado de las siguientes referencias:

www.eclipse.org/swt.

http://www.cs.umanitoba.ca/~eclipse/ (University of Manitoba, Winnipeg, Manitoba, Canada)

[7] Knuth, D.E. The Errors of TeX. Software, Practice & Experience, 19(7): 607-685, 1989.

[8] http://www.eclipse.org/articles/viewArticle/ViewArticle2.html

[9]http://www.eclipse.org/articles/ArticleYour%20First%20Plugin/YourFirstPlugin.html

[10] DDD (Data Display Debugger): http://www.gnu.org/software/ddd

Depurador Declarativo para Java