UNIVERSIDAD
POLITECNICA
DE VALENCIA

Escuela Técnica
Superior de Ingeniería
Informática

PROYECTO FINAL DE CARRERA:

Una implementación de *Subterm Dependency Tracking* para Java.

Autor: María Ortega López
Directores: David Insa Cabrera
Josep Silva Galiana

Valencia, Septiembre 2011

Índice de contenidos

Índice de ilustraciones

1 Introducción

Este documento presenta la implementación de la estrategia *Subterm Dependency Tracking* [1] en el *Depurador Declarativo para Java (*DDJ) [2]. Esta estrategia se encuentra enmarcada dentro de una técnica de depuración llamada *Depuración Algorítmica*, que fue propuesta por Ehud Saphiro en 1982 [3], y que desde entonces se ha ampliado y se aplica en prácticamente todos los paradigmas de depuración. En la práctica, estas técnicas han evolucionado hasta producir depuradores maduros para varios lenguajes de programación.

La depuración algorítmica consiste en localizar errores en programas mediante una serie de preguntas al programador. De esta manera toda la técnica recae sobre el programador y la interpretación que tiene acerca de lo que tiene que hacer el programa, es decir, hay computaciones que son correctas y otras incorrectas según la semántica pretendida por el programador.

Mediante las respuestas obtenidas, el depurador descartará las partes correctas del programa, dejando aquellas que pueden ser las causantes del error.

En esencia, podemos identificar 2 fases secuenciales en la depuración declarativa: en la primera fase, se crea la estructura de datos que representa el *Árbol de Ejecución*[5]. Esta estructura de datos contiene nodos que son una representación de las sub-computaciones del programa. Por tanto, si nos centramos en los lenguajes orientados a objetos, podemos identificar los nodos de esta estructura como la ejecución de un método, incluyendo la llamada al método con sus parámetros y el resultado, y los valores de los atributos al alcance del método antes y después de su ejecución. El nodo raíz es la ejecución del método principal del programa, y cada invocación a un método es añadido como un hijo de éste.

Seguidamente se plantea un ejemplo Java para aclarar el concepto de árbol de ejecución generado en la primera fase de la depuración declarativa:

Ejemplo 1: ejemplo Balance.

El siguiente programa Java implementa el cálculo de un balance contable, según una serie de movimientos contenidos en una colección.

Dada la colección de movimientos, el programa realiza la suma de los gastos y de los ingresos, comparando si hay beneficios (devolviendo true) o no (devolviendo false). Se ha introducido un error en el método **obtenerIngresos**.

```
public class ejemploBalance {

	public static void main(String[] args) {
		ArrayList<Integer> listaMovimientos = new ArrayList<Integer>();
		listaMovimientos.add(new Integer(-120));
		listaMovimientos.add(new Integer(160));
		listaMovimientos.add(new Integer(-160));
		listaMovimientos.add(new Integer(-120));
		listaMovimientos.add(new Integer(120));

		Integer[] gastosIngresos = obtenerGastosIngresos(listaMovimientos);
		boolean hayBeneficio = balance(gastosIngresos);
	 }

	private static Integer[] obtenerGastosIngresos(ArrayList<Integer> listaMovimientos) {
		Integer[] resultado = new Integer[2];
		resultado[0] = sumGastos(listaMovimientos);
		resultado[1]  = sumIngresos(listaMovimientos);
		return resultado;
	}

	private static Boolean balance(Integer[] gastosIngresos) {
		return (gastosIngresos[1]+gastosIngresos[0])>0;
	}

	private static Integer sumIngresos(ArrayList<Integer> listaMovimientos) {
		ArrayList<Integer> ingresos = obtenerIngresos(listaMovimientos);
		Integer resultado = sumar(ingresos);
		return resultado;
	}

	private static ArrayList<Integer> obtenerIngresos(ArrayList<Integer> listaMovimientos){
		ArrayList<Integer> ingresos = new ArrayList<Integer>();
		for(int i = 0; i< listaMovimientos.size(); i++)
			if(listaMovimientos.get(i)<0) ingresos.add(listaMovimientos.get(i));
		return ingresos;
	}

	private static Integer sumGastos(ArrayList<Integer> listaMovimientos) {
		ArrayList<Integer> gastos = obtenerGastos(listaMovimientos);
		Integer resultado = sumar(gastos);
		return resultado;
	}

	private static ArrayList<Integer> obtenerGastos(ArrayList<Integer> listaMovimientos) {
		ArrayList<Integer> gastos = new ArrayList<Integer>();
		for(int i = 0; i< listaMovimientos.size(); i++)
			if(listaMovimientos.get(i)<0) gastos.add(listaMovimientos.get(i));
		return gastos;
	}

	private static Integer sumar(ArrayList<Integer> lista) {
		Integer resultado = new Integer(0);
		for(int i = 0; i< lista.size(); i++)  resultado += lista.get(i);
		return resultado;
	}
}
```

Ilustración 1: Programa de ejemplo

Para este ejemplo el árbol de ejecución generado es el siguiente:

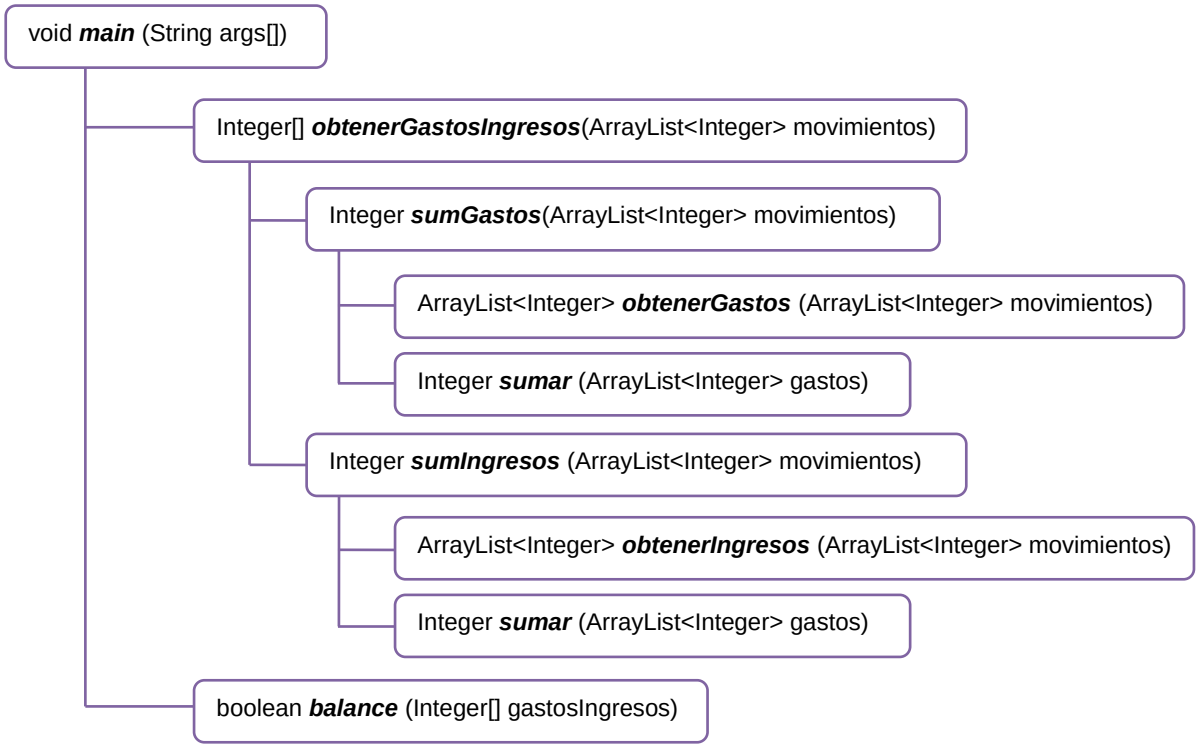


Ilustración 2: Árbol de ejecución asociado al Ejemplo1

Durante la segunda fase, el depurador utiliza una estrategia para recorrer el árbol, y va seleccionando nodos para preguntarle al programador. Utilizando las respuestas el depurador trata de encontrar cuál es el nodo erróneo. Si un nodo es marcado como incorrecto, puede deberse a que:

1) Éste contiene un error, o
2) Alguna de sus sub-computaciones contiene un error y éste se refleja en él.

Por ello, un nodo erróneo es aquel en el cual todas sus sub-computaciones se comportan correctamente y él se comporta de manera incorrecta.

Ejemplo 1:

Una de las estrategias implementadas en DDJ es Top-down [6], que realiza un recorrido de arriba hacia abajo y de izquierda a derecha del árbol de ejecución, preguntando siembre por un hijo o hermano del anterior nodo. En la siguiente sesión de depuración realizada seleccionamos Top-down como estrategia de depuración, siendo X e V las respuestas dadas por el usuario (X para indicar una computación incorrecta y V para una correcta).

main(String args[])	X
obtenerGastosIngresos(ArrayList<Integer> movimientos) → Integer[] resultado	X
sumGastos(ArrayList<Integer> movimientos) → Integer gastos	V
sumIngresos(ArrayList<Integer> movimientos) → Integer ingresos	X
obtenerIngresos(ArrayList<Integer> movimientos) → ArrayList<Integer> ingresos	X

Error encontrado en el método:
ArrayList<Integer> ***obtenerIngresos***(ArrayList<Integer> listaMovimientos)

Ilustración 3: Sesión de depuración del árbol de ejecución del Ejemplo1.

Esta técnica garantiza que, si se detecta una computación incorrecta, el error será encontrado siempre y cuando el programador responda correctamente a las preguntas que realiza el depurador [3]. Además, los depuradores declarativos modernos, como DDJ, permiten la exploración libre del árbol de ejecución mediante una *Interfaz Grafica de Usuario* (IGU).

Desgraciadamente, con programas reales, el árbol de ejecución puede ser enorme, y éste es el principal problema de esta técnica de depuración. Por ello el Depurador Declarativo para Java propone una nueva arquitectura, en la cual los depuradores declarativos no se ejecutan en las dos fases secuenciales comentadas anteriormente, sino en dos fases concurrentes, solucionando de este modo los 3 problemas de escalabilidad: memoria, tiempo y visualización gráfica del árbol de ejecución.

Las estrategias implementadas en DDJ determinan distintos caminos para recorrer el árbol de ejecución según las respuestas aportadas. Por ejemplo, la idea de Divide & Query es realizar una pregunta en cada paso que divida el árbol en dos partes con un número de nodos similares. Mientras que Top-Down, realiza un recorrido de arriba hacia abajo y de izquierda a derecha del Árbol de Ejecución [5] preguntando siembre por un hijo o hermano del anterior nodo, y así buscar el error desde la raíz del árbol al nodo erróneo. Cada estrategia tiene sus propias ventajas, por ejemplo Top-Down tiene la ventaja de que las preguntas realizadas al usuario están más relacionadas, mientras que Divide & Query hace en total menos preguntas al usuario.

Estas estrategias realizan una serie de preguntas acerca de si la ejecución de ciertos nodos han ido correctamente o no. Sin embargo, aceptando sólo estas dos respuestas, dejamos aparte cierta información que podría mejorar significativamente la búsqueda. Cuando el usuario dice que cierto nodo es incorrecto, es porque conoce, al menos implícitamente, el conjunto de soluciones correctas para la llamada, y puede observar que los argumentos de salida del nodo que se está computando difieren de los argumentos de salida de las soluciones correctas.

Con frecuencia, la salida de la computación actual es casi correcta, sólo un subconjunto de las estructuras de datos de salida (o de los objetos modificados durante la ejecución del método en el caso de Java) es incorrecto. Sin embargo, a menos que el depurador permita especificar exactamente qué partes de qué argumentos de salida son incorrectos, la búsqueda en la computación representada por el nodo no será capaz de establecer el foco en el conjunto de nodos que ha computado la parte incorrecta.

Por tanto, uno de los objetivos es implementar en DDJ la estrategia *Subterm Dependency Tracking*, incluyendo un mecanismo que permita al usuario marcar subtérminos de los argumentos del nodo por el que se está preguntando. Así, si se marca un subtérmino de una computación, el sistema utilizará la información sobre la identidad del subtérmino erróneo para guiar la búsqueda del error. Concretamente, el sistema deberá empezar a preguntar sobre los nodos que han generado el subtérmino marcado, ya que es probable que sean éstos quiénes tienen errores dentro de su árbol de llamadas, o se les dio información incorrecta a ellos mismos.

De este modo, centrar la búsqueda en un subtérmino incorrecto, puede resultar muy beneficioso. Si el subárbol a explorar es grande (es decir, representa una computación con muchas invocaciones a métodos) y sólo una parte pequeña es incorrecta, entonces no explorar las partes que han generado las partes correctas del nodo evita un gran número de preguntas innecesarias. Asimismo, éste comportamiento hace que el depurador sea más comprensible, ya que este es el comportamiento que el usuario intuitivamente esperaría.

Ejemplo 1:

Dado el ejemplo 1, en la depuración del nodo *ObtenerGastosIngresos*, sólo una de las partes del valor de retorno es errónea (el valor de los ingresos). Por ello resultaría beneficioso poder marcar este valor como incorrecto, y que DDJ dirigiera la depuración a las computaciones que crearon este valor.

Aquí vemos una sesión de depuración usando Subterm Dependency Tracking:

main(String args[])	X
obtenerGastosIngresos(ArrayList<Integer> movimientos) → marcamos *resultado[1]*	
sumar(ArrayList<Integer> ingresos) → Integer ingresos	V
sumIngresos(ArrayList<Integer> movimientos) → Integer ingresos	X
obtenerIngresos(ArrayList<Integer> movimientos) → ArrayList<Integer> ingresos	X

Error encontrado en el método:
ArrayList<Integer> ***obtenerIngresos***(ArrayList<Integer> listaMovimientos)

Ilustración 4: Sesión de depuración para el Ejemplo1 usando SDT

De esta manera vemos como, mediante el marcado de una parte del valor de retorno, hemos podado las computaciones correctas del árbol (*sumGastos* y *balance*) y se ha empezado preguntando por el método *sumar*, que es el creador del valor, y posteriormente por *sumIngresos*. Esto ocurre porque SDT crea un camino entre el nodo creador del subtérmino marcado como incorrecto y el nodo que contiene éste subtérmino, centrando la depuración en los nodos presentes en éste camino.

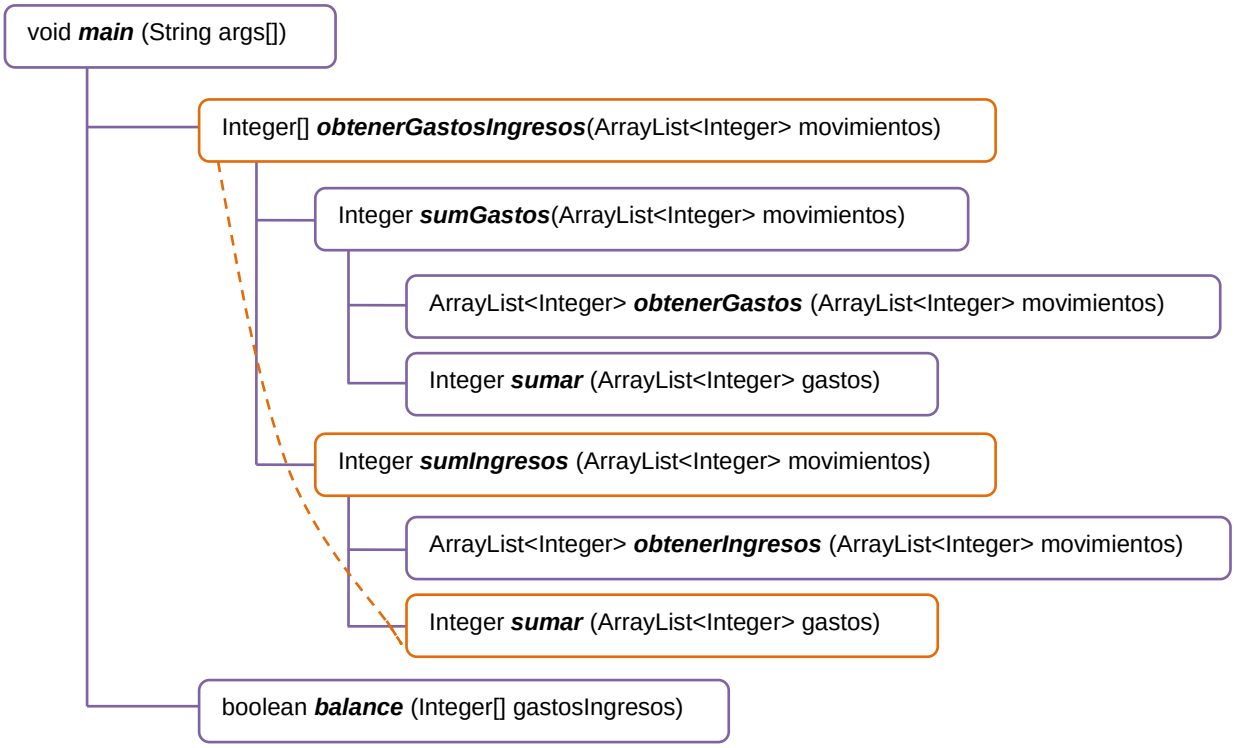


Ilustración 5: Camino generado por SDT para el Ejemplo1

Este proyecto va a ser desarrollado en el marco del grupo de investigación de la UPV, MIST (Multi-paradigm Software Technology)[9]. El objetivo común de este grupo es el desarrollo de técnicas para ayudar a la construcción de software fiable mediante todas las fases de su ciclo de vida: las especificaciones, la depuración, verificación y optimización; que proporcionan una sólida base formal para los distintos desarrollos que permiten formalmente demostrar la exactitud y eficacia de las técnicas. El grupo mantiene un equilibrio entre los desarrollos teóricos y prácticos.

El siguiente documento expone el desarrollo de las ampliaciones presentadas en DDJ, analizando el funcionamiento de la herramienta y considerando la implementación necesaria para su realización. El objetivo de esta memoria es servir de ayuda para la comprensión y evaluación del proyecto, y está dividida en 4 partes: la introducción, donde se introducen los conceptos clave para la realización del proyecto y se analiza qué se quiere implementar; en la especificación de requerimientos software, se explican las partes a implementar del sistema y su funcionalidad; en el siguiente apartado, se realizan las tareas de análisis y diseño propias de un ciclo de desarrollo de

software, empezando por un análisis del sistema (DDJ) y terminando con la identificación de los cambios a realizar en el mismo; para terminar se explica la implementación realizada y se propone un caso de estudio para mostrar el funcionamiento obtenido.

2 Especificación de requerimientos software

2.1 Introducción

2.1.1 Propósito

El propósito del presente apartado es definir los requerimientos software que debe tener la implementación de la técnica Subterm Dependency Tracking para la herramienta de depuración declarativa para Java DDJ. Se abordará el estudio y realización de los cambios necesarios en DDJ para que implemente dicha estrategia de depuración.

2.1.2 Ámbito

El producto a desarrollar forma parte de las aplicaciones de depuración declarativa. Concretamente el propósito de este proyecto es implementar en DDJ una estrategia de recorrido del árbol de ejecución.

Por tanto, la aplicación podrá dar soporte al uso de esta estrategia, permitiendo el marcado de subtérminos como correctos o incorrectos. Y, en respuesta a esto, el depurador redirigirá la búsqueda del error.

2.1.3 Definiciones, Acrónimos y abreviaturas

- Acrónimos:

DDJ: Depurador Declarativo para Java.

SDT: Subterm Dependency Tracking.

2.1.4 Referencias

Para la preparación de este texto se han tenido en cuenta los siguientes documentos:

1. IEEE Std 830- IEEE Guide to Software Requeriments Specifications. IEEEStandars Board.[4]

2.1.5 Visión global

Este documento consta de 3 partes: la primera sección explica el propósito de la especificación de requisitos, identifica el producto a desarrollar, así como define su aplicación, indicando beneficios, objetivos y metas. La segunda parte nos da una perspectiva del producto, un resumen de sus funciones principales, indica las características del usuario e indica restricciones y dependencias generales. Por último, la tercera parte incluye todos los requisitos funcionales del producto software.

2.2 Descripción General

A continuación vamos a ver los factores que afectan al producto y a sus requerimientos.

2.2.1 Perspectiva del producto

El producto a desarrollar formará parte de un sistema mayor, conocido como DDJ [2] y que es un depurador declarativo para Java. Por tanto el lenguaje de programación usado viene impuesto por DDJ y es Java. Lo mismo ocurre con la base de datos, que por defecto es ACCESS en DDJ.

La depuración declarativa, y por tanto DDJ, se basa en 2 fases diferenciadas: en la primera de ellas se genera el árbol de ejecución del programa a depurar, y en la segunda fase, el depurador utiliza una estrategia de búsqueda, seleccionando nodos del Árbol de Ejecución para preguntar al programador si la computación asociada al nodo ha funcionado correctamente. Es aquí donde, en este proyecto, desarrollaremos una estrategia de recorrido, SDT. Esta estrategia, como ya se ha comentado, se basa en marcar como correctos o incorrectos los subtérminos de las computaciones presentes, redirigiendo la búsqueda del error hacia los nodos responsables del subtérmino marcado como incorrecto.

Por tanto SDT será implementada en DDJ como una nueva estrategia, y ésta se ejecutará al marcar algún subtérmino como incorrecto. Cuando la estrategia termine, DDJ se encargará de volver a la estrategia activada anteriormente.

2.2.2 Funciones del producto

- Activar/Desactivar Subterm Dependency Tracking.
- Marcar Subtérmino de una computación como Válido / No válido.
- Ejecución del algoritmo de recorrido de SDT.

2.2.3 Características del usuario

Esta aplicación va dirigida a programadores expertos, que deben tener conocimientos acerca de la semántica del programa que están depurando, para responder con certeza a las cuestiones planteadas por el depurador.

2.2.4 Restricciones generales

Las restricciones generales son las propias de DDJ:

- Puede trabajar con cualquier base de datos, pero por defecto se utiliza ACCESS, por tanto será necesario tener instalado el driver ODBC para Access 2007.
- Opcionalmente, podemos instalar *Standard Widget Toolkit (*SWT)[7].
- Será necesario tener instalado el componente *Java Developement Kit* (JDK)[8].

2.2.5 Supuestos y dependencias

SDT estará integrado en DDJ, y por tanto tendrá las mismas dependencias de hardware y de software que DDJ. Dichas dependencias así como una descripción completa del sistema DDJ antes de la realización de este proyecto puede encontrarse en [11].

2.3 Requisitos específicos

2.3.1 Requerimientos funcionales

- **Activación de Subterm Dependency Tracking:**

Este apartado añade SDT a la lista de estrategias implementadas en DDJ, permitiendo su activación para ser usada en combinación con otra de las estrategias existentes. Aunque por defecto la estrategia estará activa.

- **Selección de subtérminos:**

La característica principal de la estrategia que se va a implementar es permitir marcar subtérminos como correctos o incorrectos. Éstos pueden ser marcados tanto en la representación del árbol de ejecución como en el Object Inspector presente en DDJ. Por ejemplo, ante la ejecución del siguiente método: *crearPunto(int x, int y) -> Point resultado*, el usuario podrá marcar tanto los argumentos de entrada como el valor de retorno del método haciendo clic sobre estos en la cabecera del nodo. DDJ permitirá el marcado de subtérminos siempre que SDT esté activa.

- **Uso de la estrategia:**

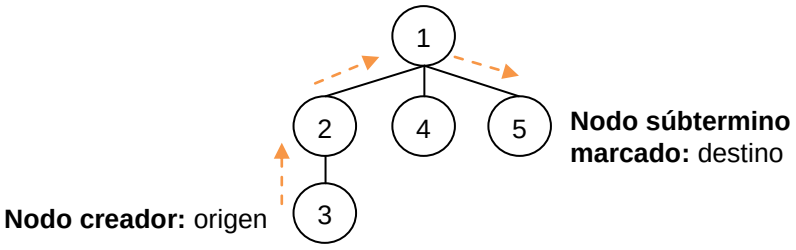
Ante un subtérmino marcado como incorrecto, DDJ deberá interrumpir la estrategia activa e iniciar SDT. Y del mismo modo, cuando ésta finalice, deberá volver a la estrategia activa.

Según las respuestas dadas por el programador, el algoritmo de SDT determinará cuando debe terminar, es decir, cuando debe dejar de usarse esta estrategia. Los casos en que el algoritmo termina vienen determinados por el estado de los nodos presentes en el camino entre el nodo donde se marca el subtérmino incorrecto y el nodo creador de éste.

- Al encontrar/marcar un nodo incorrecto en el camino, que tenga en su subárbol el nodo creador. Con esto, DDJ termina la ejecución de SDT e inicia la estrategia seleccionada para buscar el error en el subárbol definido por éste nodo.
- Al encontrar/marcar un nodo correcto que tenga en su subárbol el nodo donde se ha marcado el subtérmino. En este caso se poda todo el subárbol determinado por éste, incluyendo el nodo con el subtérmino incorrecto.
- En el caso de que el nodo tenga en su subárbol los dos nodos, el creador y el que contiene el subtérmino marcado, se adopta el comportamiento explicado en el ejemplo 2.

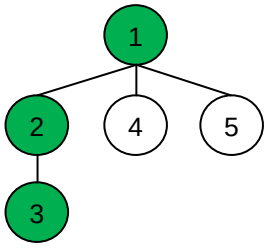
Ejemplo 2:

Suponemos que en una sesión de depuración utilizando Top-Down[6], el usuario marca como incorrecto un subtérmino del nodo 5 que ha sido creado en el nodo 3. En este momento DDJ inicia SDT, en el cual se construye un camino desde el nodo creador (nodo 3) hasta el nodo que contiene el subtérmino incorrecto (nodo 5).



En este caso tenemos dos sesiones de depuración que muestran el comportamiento de este nodo:

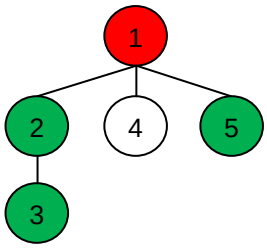
- Ante el marcado del nodo 1 como correcto:



1. Nodo 3: Valido.
2. Nodo 2: Valido.
3. Nodo 1: Valido.

DDJ depura todo el árbol determinado por el nodo 1, con ello termina SDT y continua con la estrategia activa.

- Ante el marcado del nodo 1 como incorrecto:



1. Nodo 3: Valido.
2. Nodo 2: Valido.
3. Nodo 1: No válido.

DDJ continua el camino determinado por SDT.

4. Nodo 5: Valido

Se ha terminado el camino, y DDJ continua con la estrategia activa, preguntando por el nodo 4.

2.3.2 Requisitos de interfaces externas

2.3.2.1 Interfaces de usuario

El producto debe ampliar la interfaz gráfica de usuario de DDJ para que permita al usuario marcar subtérminos como correctos o incorrectos, mostrando de manera visual este estado.

Los elementos visuales como botones o barras de herramientas introducidos en el sistema deberán respetar los códigos de colores y tamaños de los ya existentes.

2.3.2.2 Interfaces hardware

No se han descrito.

2.3.2.3 Interfaces software

Es imprescindible que el usuario disponga de Java Development Kit (JDK).

2.3.3 Requerimientos de eficiencia

Las implementaciones llevadas a cabo en todas las extensiones propuestas no deberán suponer en ningún caso un deterioro de la eficiencia del depurador. La respuesta durante la ejecución de SDT deberá percibirse como instantánea por parte del usuario.

2.3.4 Restricciones de diseño

No se han descrito.

2.3.5 Atributos

No se han descrito.

3 Análisis y diseño de la solución

3.1 Estudio previo

Con el objetivo de entender mejor el funcionamiento de DDJ y realizar una búsqueda más eficiente de los cambios requeridos para la implementación de SDT, se realizará un estudio general acerca del funcionamiento de DDJ, centrándose en la nueva arquitectura que propone.

DDJ propone una nueva arquitectura en la cual, las dos fases en las que están basados los depuradores tradicionales, la construcción y la exploración del árbol de ejecución, se ejecutan en paralelo, de manera que se solventan los problemas de escalabilidad de éstos.

DDJ utiliza una arquitectura de tres capas, con el objetivo de mantener independientes los algoritmos de búsqueda del árbol de los problemas de caché de la base de datos. En esta arquitectura todos los componentes tienen acceso a un *árbol de ejecución virtual*, una estructura de datos similar al árbol de ejecución, pero éste puede contener nodos que están incompletos. Por tanto, las estrategias pueden recorrer el árbol virtual ya que representa un subárbol del árbol de ejecución final.

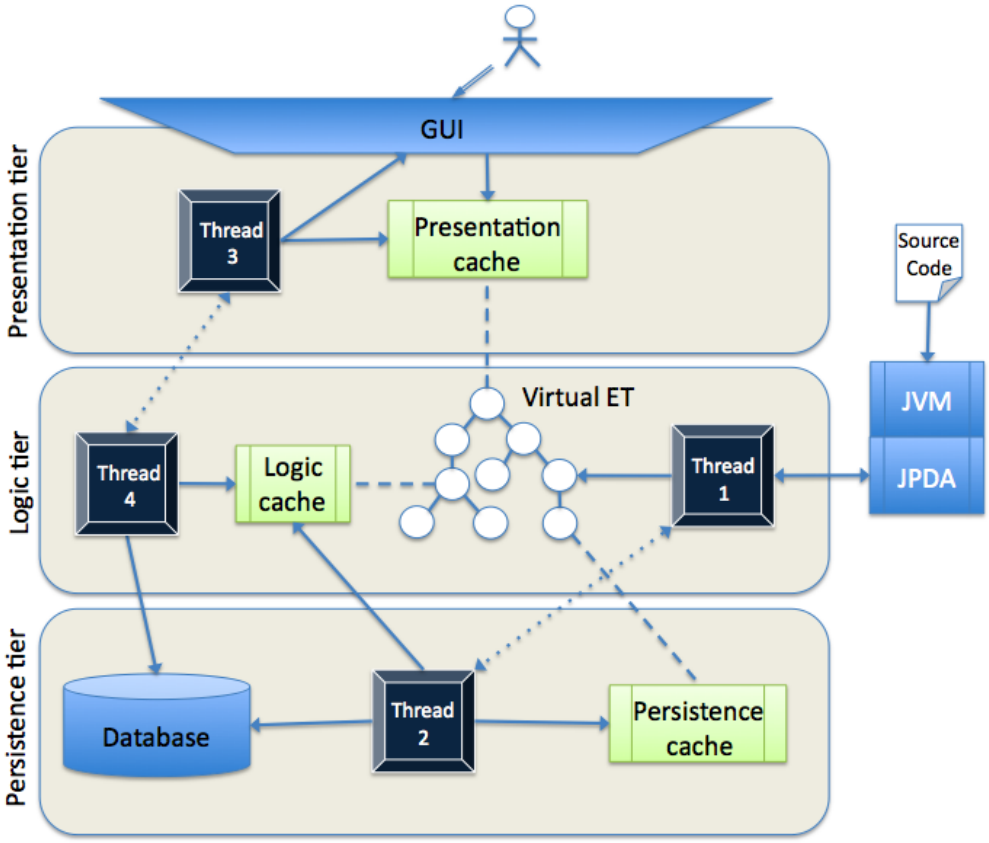


Ilustración 6: Arquitectura de DDJ.

En la Ilustración 8 se presenta la arquitectura de DDJ, donde podemos observar que cada una de las capas tiene una caché que pueden verse como una vista del árbol de ejecución y se utilizan para fines diferentes:

- Caché de persistencia: contiene los nodos que deben ser almacenados en la base de datos, y determina cuál es el máximo número de nodos completados que pueden ser mantenidos en el árbol de ejecución (límite de persistencia) asegurando que la memoria principal nunca será desbordada.

- Caché de lógica: contiene una cantidad limitada de nodos del árbol de ejecución virtual (límite de lógica) y son aquellos con más probabilidad de ser preguntados, y por ello deben ser recuperados de la base de datos.

- Caché de presentación: contiene un subárbol dentro de la caché de lógica, siendo éstos los nodos que se muestran al usuario en la IGU. Esté subárbol se define seleccionando un nodo como raíz y una profundidad (límite de presentación).

El comportamiento del depurador se controla con cuatro hilos que trabajan en paralelo:

- Hilo 1. Construcción del árbol de ejecución.
  Este hilo ejecuta un programa y para cada invocación a un método, genera un nuevo nodo. Cuando la cantidad de nodos completados se acerca al límite de persistencia, este hilo despierta al hilo 2, que mueve algunos nodos a la base de datos. Si el límite se alcanza, el hilo 1 se duerme hasta que se borran los nodos suficientes.

- Hilo 2. Control del tamaño del árbol de ejecución virtual.
  Este hilo controla qué información debe ser almacenada tanto en memoria como en la base de datos y cuál solamente en la base de datos. El hilo 2, al ser despertado, borra la información de algunos nodos del árbol de ejecución virtual copiándola a la base de datos, almacenando la mayor cantidad de información que se encuentren en los nodos que no se encuentren en la caché de lógica.

- Hilo 3. Comunicación con la interfaz.
  Este hilo controla la información mostrada en la IGU con la caché de presentación. Mediante las respuestas del usuario y el límite de presentación, éste selecciona cual es el nodo raíz de la caché de presentación. Esto se realiza tras cada pregunta, asegurando que el nodo de la pregunta realizada, su padre y tantos descendientes como el límite de presentación permite se muestren en la IGU.

- Hilo 4. Selección de preguntas.
  Este hilo se encarga de seleccionar la siguiente pregunta, teniendo en cuenta la estrategia seleccionada. Con el nodo seleccionado, la caché de lógica se actuali-

za y se obtienen los nodos que forman parte de la nueva caché de lógica. Esto se hace utilizando el padre del nodo seleccionado y el límite de lógica. De manera que los nodos que forman parte de esta nueva cache de lógica y no estaban en la anterior deben ser cargados en memoria desde la base de datos.

3.2 Identificación de los cambios a realizar

Dentro del contexto de las estrategias implementadas en DDJ, en este proyecto nos proponemos la implementación de la estrategia Subterm Dependency Tracking, que debido a sus características requiere la realización de ciertos cambios tanto en la interfaz gráfica de usuario, como en la lógica del programa.

3.2.1 Capa presentación

Primero nos centraremos en los cambios que requiere la IGU de DDJ para la implementación de SDT.

La IGU disponible en DDJ nos muestra una representación gráfica del árbol de ejecución, de manera que el usuario puede navegar por los nodos que representan las computaciones y ver la información acerca de los parámetros de entrada y salida de éstos.

DDJ dispone de dos modos de depuración: manual y automática. De manera que en la depuración manual DDJ permite seleccionar nodos libremente, mostrando una ventana de depuración con las opciones posibles para responder a la pregunta asociada al nodo. De la misma manera, DDJ puede trabajar en modo automático, seleccionando los nodos automáticamente según la estrategia seleccionada.

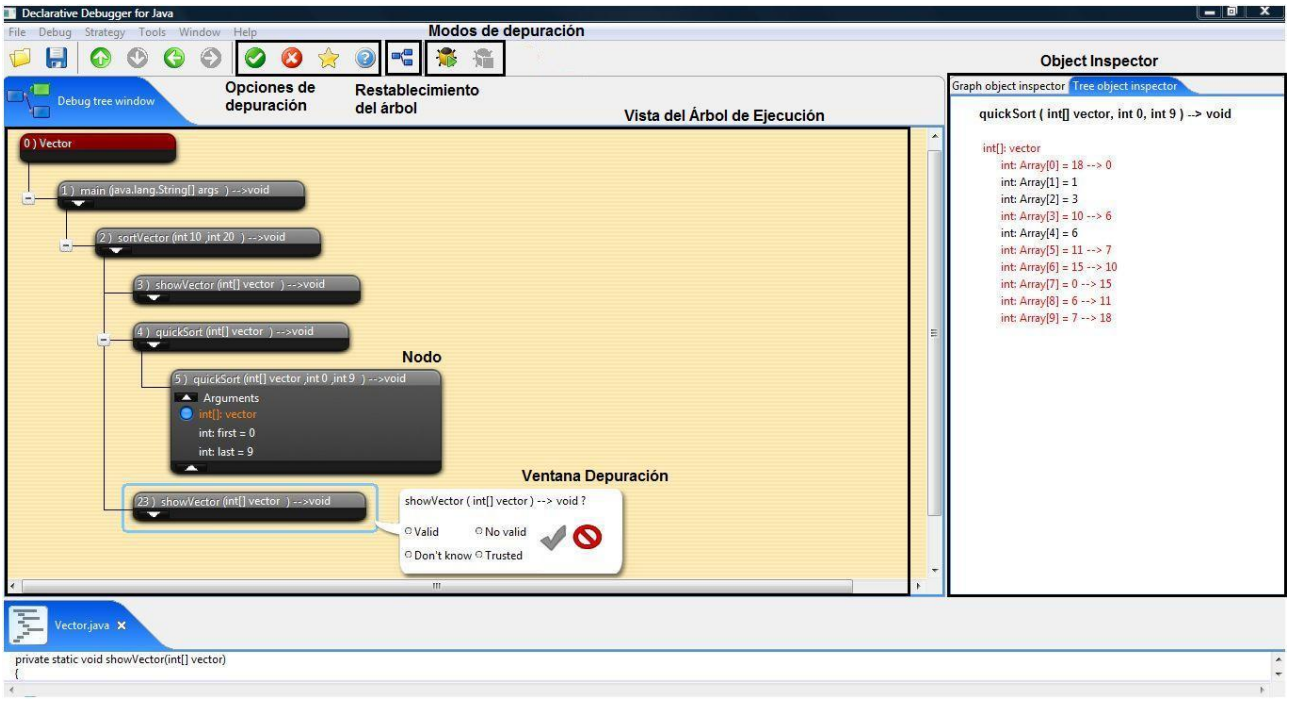


Ilustración 7: Componentes de la interfaz de DDJ

DDJ también dispone de un *Object Inspector* que permite la inspección de los objetos en el árbol de ejecución, es decir, al abrir un nodo para ver la información de la computación (tanto argumentos como valor de retorno), DDJ permite la selección de elementos, que se abren en el *Object Inspector*. Esta herramienta resulta muy útil ya que podemos inspeccionar los valores de los elementos de las colecciones y los atributos de los objetos.

Los cambios que requiere la implementación de Subterm Dependency Tracking son los siguientes:

- En el menú de Estrategias se podrá activar/desactivar SDT.

- Deberemos tener en cuenta el concepto de Subtérmino, que abarca los argumentos de entrada y salida de las computaciones. De manera que la IGU de DDJ capturará los eventos de ratón (clic) sobre la representación gráfica de éstos en las cabeceras de los nodos.

- Ante un clic en un subtérmino se mostrará una ventana similar a la de depuración en el caso de los nodos. Esta ventana permitirá al usuario seleccionar las opciones Valido y No valido.

- Después de responder una pregunta sobre un subtérmino, éste se marcará de manera gráfica según la respuesta.

- Asimismo, DDJ permitirá seleccionar elementos del Object Inspector, obteniendo un comportamiento igual al de los subtérminos de las cabeceras de los nodos.

- En DDJ podemos restablecer el árbol de ejecución, es decir, devolverlo a su estado original (restableciendo el estado de los nodos), por tanto también tendrán que restablecerse los estados de los subtérminos.

En el siguiente diagrama vemos la relación entre los 3 elementos más significativos de la representación del árbol de ejecución en la IGU del programa. Vemos que el árbol está formado por nodos, y a su vez cada nodo está compuesto por una colección de subtérminos que representan los argumentos del método y otro subtérmino que representa el valor de retorno. Asimismo el árbol contiene una referencia al último subtérmino seleccionado, con el objetivo de gestionar la ventana de depuración de subtérminos que veremos más adelante.

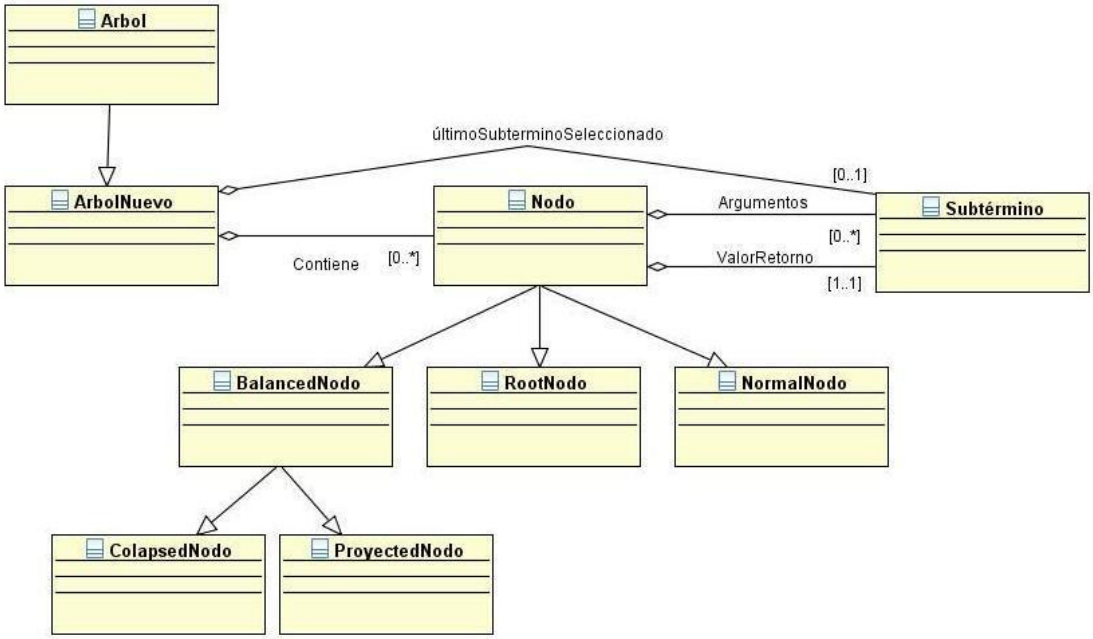


Ilustración 8: Diagrama de clases: Capa Presentación (incompleto)

3.2.2 Capa Lógica

Tomando como referencia la arquitectura en 3 capas de DDJ, nos disponemos a analizar los cambios que requiere la implementación de SDT en la capa de lógica del programa.

DDJ implementa las siguientes estrategias de depuración:

- Divide & Query (by Shaphiro).
- Divide & Query (by Hirunkitti).
- Divide by rules.
- Top-Down – Heaviest First.
- Top-Down – More Rules First.
- Top-Down – Left to Right.
- Single Stepping (Post-orden).
- HatDelta – More Wrongs.
- HatDelta – Less Rights.
- HatDelta – Best Division.

Una de las tareas a realizar es la implementación del algoritmo de Subterm Dependency Tracking como una estrategia más en DDJ.

La implementación de la estrategia se ha realizado siguiendo los patrones definidos por las estrategias ya implementadas en DDJ. En el siguiente diagrama de clases

se representa la jerarquía de clases de DDJ formada por las estrategias implementadas, y donde hemos añadido una nueva clase *SubtermDependencyTracking*.

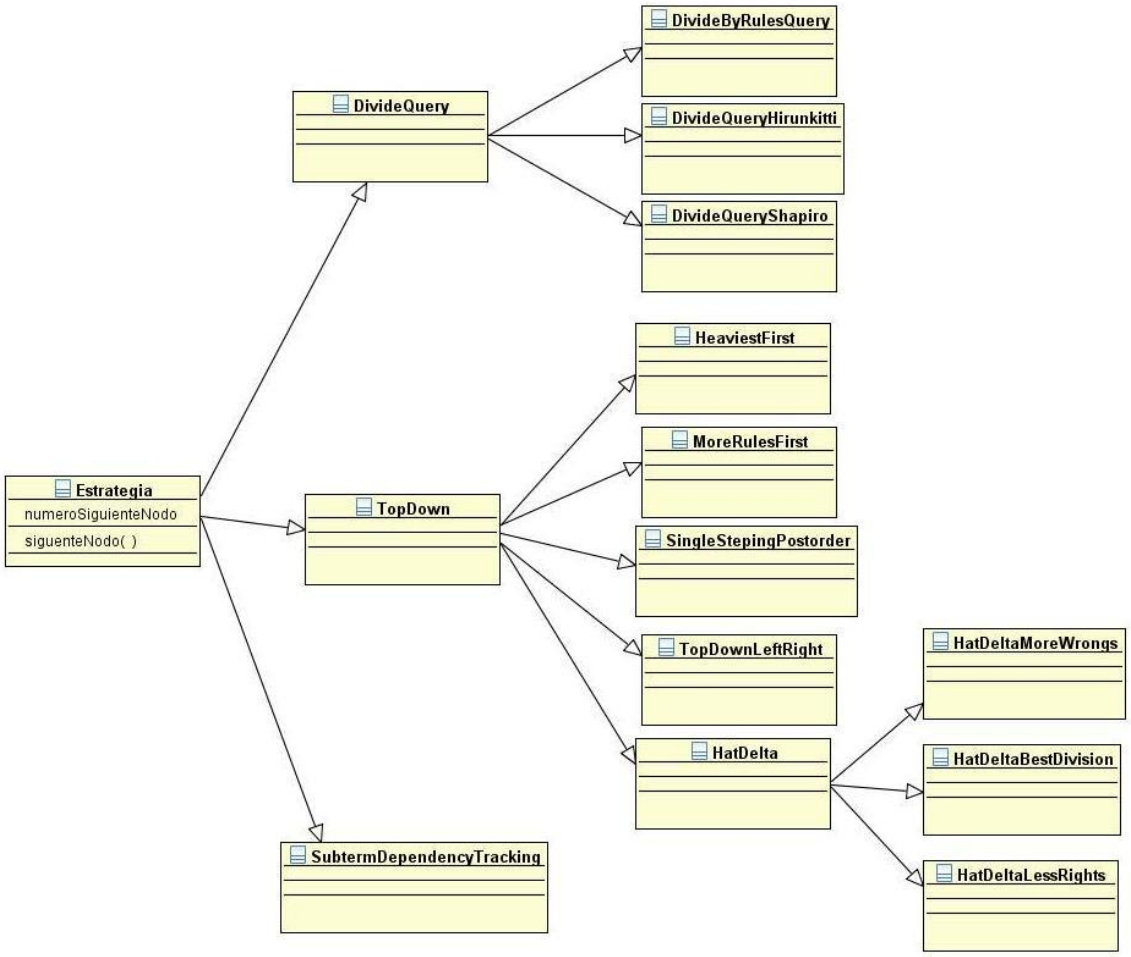


Ilustración 9: Diagrama de clases: Estrategias

En el diagrama podemos ver que la clase *SubtermDependencyTracking* hereda de *Estrategia*, e implementa el método abstracto *siguienteNodo()*, que contiene el algoritmo de la estrategia, y cuyo objetivo es dar valor al atributo *numeroSiguienteNodo*. El resto de métodos no se implementan en SDT, así se toma el comportamiento por defecto heredado de la clase Estrategia.

DDJ se inicia sin una estrategia seleccionada, utilizando Top-Down por defecto. Al seleccionar una estrategia en el menú Estrategias, DDJ utiliza ésta para recorrer el árbol de ejecución (en modo automático) realizando las preguntas asociadas a cada nodo. Esta selección puede realizarse mientras se depura, de modo que en la siguiente búsqueda del nodo a depurar, DDJ cambiará la estrategia que se utilizará por la estrategia seleccionada.

La implementación de Subterm Dependency Tracking cambia este comportamiento, ya que ante la selección de un subtérmino, y su marcado como erróneo, DDJ debe iniciar el algoritmo de SDT. Y al terminar, bien porque lo indica el algoritmo, o bien porque el usuario desactiva SDT, volver a la estrategia seleccionada.

El comportamiento que se ha implementado se representa mediante el siguiente diagrama de secuencia:

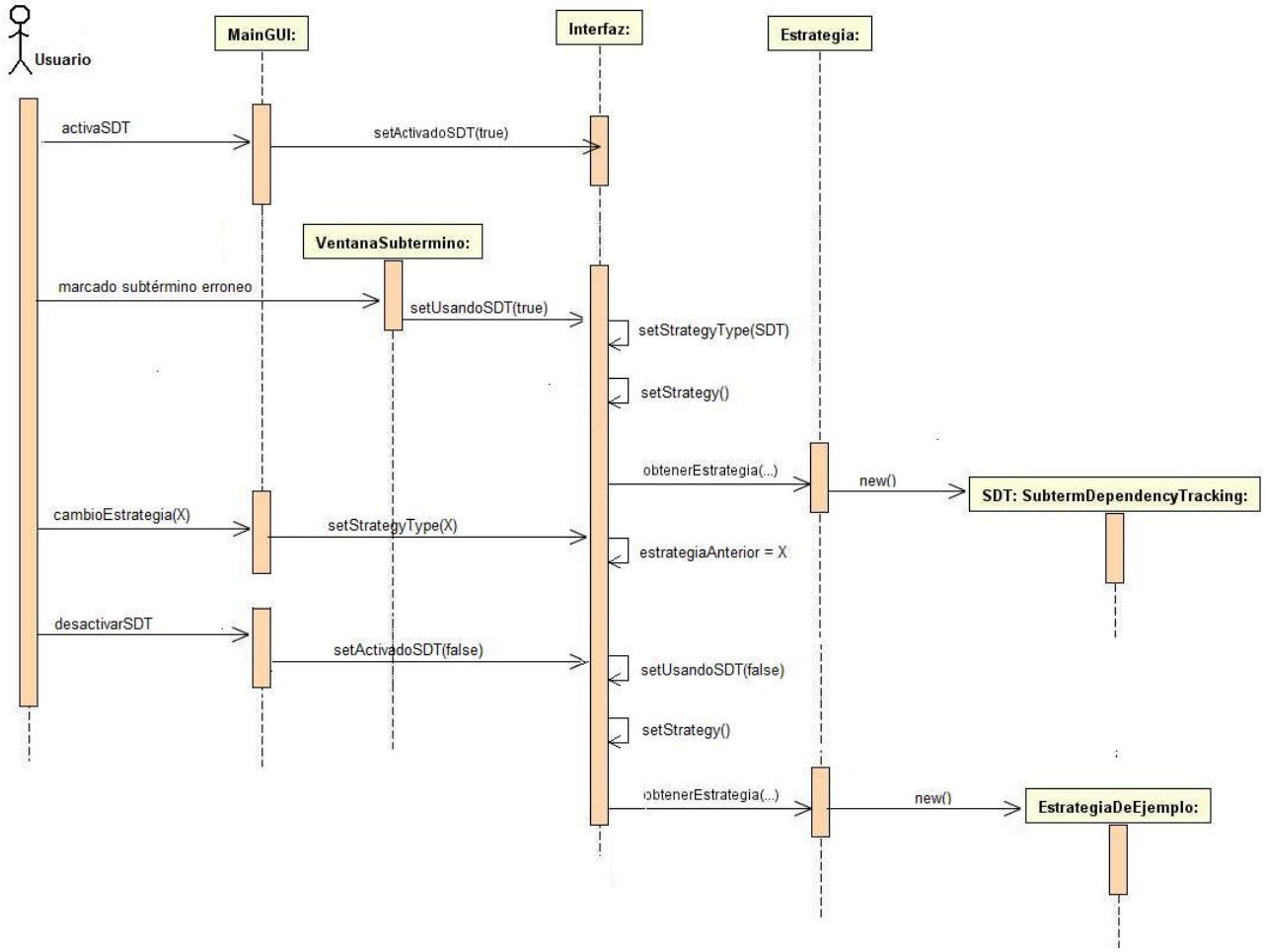


Ilustración 10: Diagrama de secuencia (comportamiento SDT)

Nótese que la estrategia seleccionada antes de ejecutar el algoritmo de SDT puede ser distinta de la seleccionada al terminar, de manera que durante la ejecución del algoritmo, la lógica deberá atender a los cambios de estrategia en el menú Estrategias, pero sin detener el algoritmo de SDT.

Por tanto los cambios que debemos realizar en la lógica de DDJ para la implementación de SDT son los siguientes:

- Implementación del algoritmo de Subterm Dependency Tracking.

- Controlar que mientras SDT se está ejecutando, los cambios en la estrategia seleccionada no detienen el algoritmo, pero sí que cambian la estrategia que DDJ ejecutará al terminar.

- Controlar que si el usuario desactiva SDT o éste termina, DDJ continua con la estrategia seleccionada.

- Ante el restablecimiento del árbol de ejecución, DDJ volverá éste a su estado inicial. Y por tanto se terminará el algoritmo de SDT.

4 Implementación realizada

En este apartado nos disponemos a analizar cómo se han implementado los cambios requeridos en DDJ.

4.1 Cambios en la interfaz gráfica de usuario

4.1.1 Representación de los subtérminos

Para la representación gráfica de los subtérminos de las computaciones en las cabeceras de los nodos, ha sido necesaria la implementación de una clase Java llamada *Subtérmino*

Ilustración 11: Nodo con Subtérminos

Las características más significativas de esta clase son:

- La clase contiene los atributos de tipo GC y Display de SWT[7] inicializados en su constructor con los valores pasados por parámetro.

- Asimismo dispone de un texto correspondiente con el subtérmino que representa. También contiene la posición relativa del subtérmino, de manera que, el objeto GC es el encargado de dibujar el texto en esa posición del Display.

- El evento clic en esta clase se representa con un método llamado *comprobarClic*, que comprueba si el clic se encuentra en el rectángulo definido por la posición del subtérmino, devolviendo true en caso afirmativo.

- Por otra parte hay que destacar el atributo *Estado* del subtérmino, que determina con un entero si el subtérmino es válido, no valido o todavía no se ha determinado. El atributo *Color* se determina a partir de éste, y el *atributo colorOriginal* determina el color inicial del atributo.

- Para terminar, el subtérmino contiene un método *mostrarVentana*, que crea la ventana de depuración para el subtérmino y la dibuja.

4.1.2 Ventana de depuración de los subtérminos

Para el marcado de subtérminos como correctos o incorrectos se ha implementado una clase, *ventanaSubtermino*, muy similar a la ventana de depuración de nodos.

Ilustración 12: Ventana de depuración de Nodos

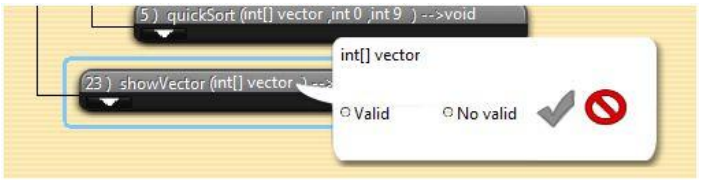


Ilustración 13: Ventana de depuración de Subtérminos

Esta clase sigue el patrón *singleton*, es decir, existe una instancia única de la clase y un punto de acceso global a ésta. Por ello dispone del método *crearVentanaSubtérmino* que crea una instancia del objeto si todavía no existe ninguna, un constructor privado, y un método de acceso, *obtenerVentanaSubtermino*.

En el método clic de esta clase, dada la opción seleccionada y una referencia al subtérmino, cambia el estado del subtérmino. Además, si la opción seleccionada es "No valido", se hace la llamada a la lógica del programa para iniciar Subterm Dependency Tracking. Si la opción es "Valido", no se inicia ningún comportamiento.

Como hemos visto anteriormente, el árbol contiene una referencia al último subtérmino seleccionado, de manera que:

- En el método dibujar del árbol se comprueba el valor de éste y si es distinto al valor por defecto, se dibuja la ventana mediante la llamada: *ultimoSubterminoSeleccionado.mostrarVentana(x,y).*

- En la clase Nodo, se captura el clic en los subtérminos, de manera que, ante la selección de otro subtérmino, se llama al método modificador del *ultimoSubterminoSeleccionado* de la clase árbol y de los atributos *numSubtermino* y *numNodo* de la *ventanaSubtermino*.

- Al cerrar la ventana del subtérmino, el *ultimoSubterminoSeleccionado* toma el valor por defecto.

4.1.3 Object Inspector

Para el control del subtérmino seleccionado en el Object Inspector, hemos necesitado la declaración de 2 nuevas variables de manera que guardamos una referencia al subtérmino seleccionado en el Object Inspector, mediante su número de variable y el número del nodo donde se encuentra.

Esta variable es excluyente con respecto a la variable existente en ArbolNuevo, que hace referencia al último subtérmino seleccionado en la representación gráfica del árbol de ejecución. De manera que al dar valor a una de ellas, damos valor nulo a la otra.

El object inspector es una clase compuesta de una estructura gráfica, *Tree*, existente en SWT[7]. De este modo se ha implementado un *SelectionListener* de manera que al seleccionar uno de los *TreeItem* nos guardamos la referencia al subtérmino seleccionado como se ha comentado anteriormente y realizamos la llamada necesaria para dibujar las opciones de depuración.

4.1.3.1 Marcado Subtérminos en el Object Inspector

Para el marcado de subtérminos en el Object Inspector se ha implementado una clase java llamada *VentantaSubterminoOI*, que contiene características comunes a la *VentanaSubtermino* (del árbol de ejecución), como que sigue el patrón *Singleton* y contiene los atributos *Display* y *GC* para su representación gráfica.

Esta ventana consiste en dos botones que representan las opciones de depuración de subtérminos, “Valid” y “No Valid”, y éstos aparecen al seleccionar un elemento *TreeItem* en la estructura *Tree* del Object Inspector.

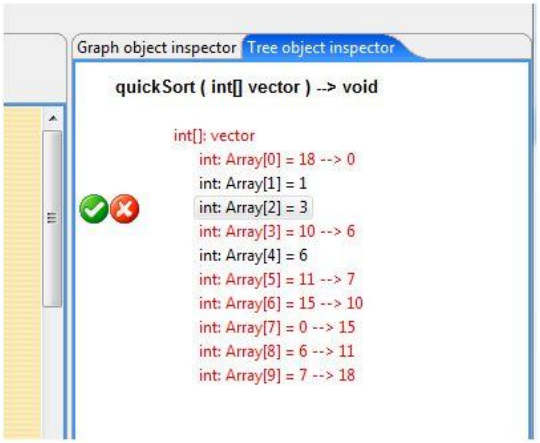


Ilustración 14: Opciones Depuración en Object Inspector

Para el marcado de subtérminos se ha necesitado la implementación de dos oyentes en la clase *ObjectInspector*, referentes al Canvas que éste contiene. El primero de ellos un PaintListener, que comprueba si hay un ítem seleccionado, y en ese caso dibuja la *VentanaDepuracionOI.* El segundo es un MouseListener, que comprueba si se

ha hecho clic en alguna de las dos opciones, y en este caso se hacen las llamadas a la lógica del programa correspondientes.

Además de esto también pueden marcarse los subtérminos mediante los botones valid / no valid de la barra de herramientas de DDJ, de manera que al seleccionar un subtérmino, tanto en el árbol de ejecución como en el Object Inspector, podemos marcarlo como correcto o incorrecto.

4.2 Cambios en la lógica del programa

En este apartado veremos cómo se han realizado los cambios requeridos en la lógica del programa.

4.2.1 Algoritmo

El algoritmo definido en la clase *SubtermDependencyTracking* es el siguiente:

Entradas:

Origen: Nodo del subtérmino seleccionado.
Destino: Nodo creador del subtérmino seleccionado.

Salidas:

NumeroSiguienteNodo (heredado de Estrategia).

```
Inicio
Actual = Anterior = Origen;
Mientras (Destino no está en el subárbol definido por Actual)
(1)   Si (Actual es Wrong || Actual es I Don't Know)
(2)      Actual = Anterior e  ir a (19)
(3)   Si no
(4)      Si (Actual es Right || Actual es Trusted)
(5)         Terminar SDT
(6)      Si no
(7)         Anterior = Actual; Actual = Padre del nodo Actual;
Fin Mientras
Mientras (Actual distinto de Destino)
(8)   Si (Actual es Right || Actual es Trusted|| Actual es I Don't Know)
(9)      Actual = Anterior e  ir a (19)
(10)  Si no
(11)     Si (Actual es Wrong)
(12)        Terminar SDT
(13)     Si no
(14)        Anterior = Actual;
            Actual = Hijo del Actual que tiene en su subárbol al nodo destino;
Fin Mientras
(15)  Si (Actual == Destino && Actual esta depurado)
```

```
(16)        Actual = Anterior
(17)    Si(Actual == Origen && Actual esta depurado)
(18)        Terminar SDT
(19)    NumeroSiguienteNodo = Actual;
Fin
```

4.2.2 Ejecución de la estrategia

En el siguiente punto veremos la implementación asociada al cambio de estrategia, es decir, cómo se cambia de la estrategia activa a SDT y cómo al terminar ésta, se vuelve a la estrategia activa.

4.2.2.1 Activación de la estrategia

Por defecto, SDT está activado en el menú Estrategias, de manera que se gestionan los eventos clic sobre los subtérminos.

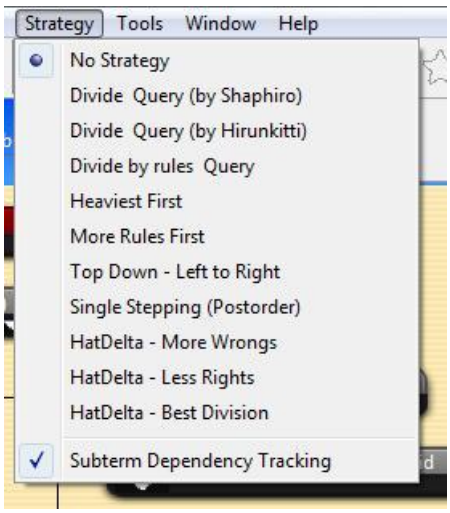


Ilustración 15: Menú Estrategias

4.2.2.2 Inicio algoritmo SDT

El algoritmo de SDT se inicia al hacer clic en un subtérmino y marcarlo como incorrecto. En este caso la *ventanaSubtérmino* hace una llamada al método *setUsandoSDT(true, numero de la variable (subtérmino), numero del nodo).* Esta llamada se encarga de crear una instancia de SDT cuando se haga la llamada al método estático de *Estrategia*, *obtenerEstrategia()*, de manera que se establece esta instancia de la estrategia como la que debe usar el encargado de lógica.

4.2.2.3 Ejecución de SDT

Como ya se ha comentado, durante la ejecución del algoritmo se puede cambiar la estrategia seleccionada en el menú estrategias. Cuando se produce este cambio, el método *setStrategyType(nuevaEstrategia*) es el encargado de cambiar la estrategia. Pero para la implementación de SDT ha sido necesario realizar cambios en este método para que no cambie la estrategia al seleccionar otra en el menú.

A continuación se muestra el método *setStrategyType(estrategiaSeleccionada)* en pseudocódigo:

```
/*Si SDT esta ejecutándose*/
Si ( estrategiaActual == SDT )

        /*Si se hace clic en otro subtérmino y se marca como erróneo, ejecutamos SDT con
        nuevos parámetros*/
        Si ( estrategiaSeleccionada ==  SDT (nueva) )
                estrategiaActual = SDT (nueva);

        /*Nos guardamos la estrategia seleccionada en el menú, para ejecutarla cuando termi-
        ne SDT*/
        Si no
                estrategiaAnterior = estrategiaSeleccionada;

/*Si se está ejecutando otra estrategia */
Si no
         Cambiamos estrategiaActual a estrategiaSeleccionada;
```

Nótese que *estrategiaAnterior* es el atributo que contendrá la estrategia que debe ejecutarse al terminar SDT.

4.2.2.4 Retorno a la estrategia activa.

Cuando se termina el algoritmo de SDT, se desactiva SDT en el menú de estrategias o se resetea el árbol, DDJ ejecuta la estrategia que tiene seleccionada en el menú. Esto se realiza mediante el método *setUsandoSDT(false),* que realiza el cambio a la estrategia que tiene guardada en *estrategiaAnterior*.

5 Caso de estudio

5.1 Enunciado del problema

El programa que planteamos como caso de estudio consiste en el cálculo mediante 3 técnicas del cuadrado de un número, y la posterior comparación de resultados, siendo ésta la respuesta del programa.

Métodos del programa:

- ***main***: inicializa los enteros que sumados serán el número del cual sacaremos el cuadrado.
- ***sqrtest***: dada una lista de enteros obtiene el cuadrado de la suma de éstos, mediante 3 métodos distintos, y devuelve si los 3 cuadrados obtenidos son iguales.
- ***obtenerSqrt***: devuelve la lista de enteros con los 3 cuadrados obtenidos.
- ***listsum***: realiza la suma de los elementos de un ArrayList de enteros.
- ***computs***: realiza las 3 computaciones correspondientes para obtener el cuadrado y las devuelve en un array.
- ***computs1***: es el primer método para obtener el cuadrado.
- ***cuadrado***: dado un entero devuelve su cuadrado multiplicando éste por si mismo.
- ***computs2***: es el segundo método para obtener el cuadrado. Dado un entero n, creamos una lista de longitud n, y cuyos elementos son todos igual a n.
- ***computs3***: es el tercer método para obtener el cuadrado.
- ***partialsums***: realiza el cálculo del cuadrado por sumas parciales, obtiene los resultados de sum1 y sum2, y los suma.
- ***sum1***: dado un entero n realiza la operación: n*(n-1)/2.
- ***sum2***: dado un entero n realiza la operación: n*(n+1)/2.
- ***incr***: dado un entero devuelve su incremento en uno.
- ***decr***: dado un entero devuelve su decremento en uno.

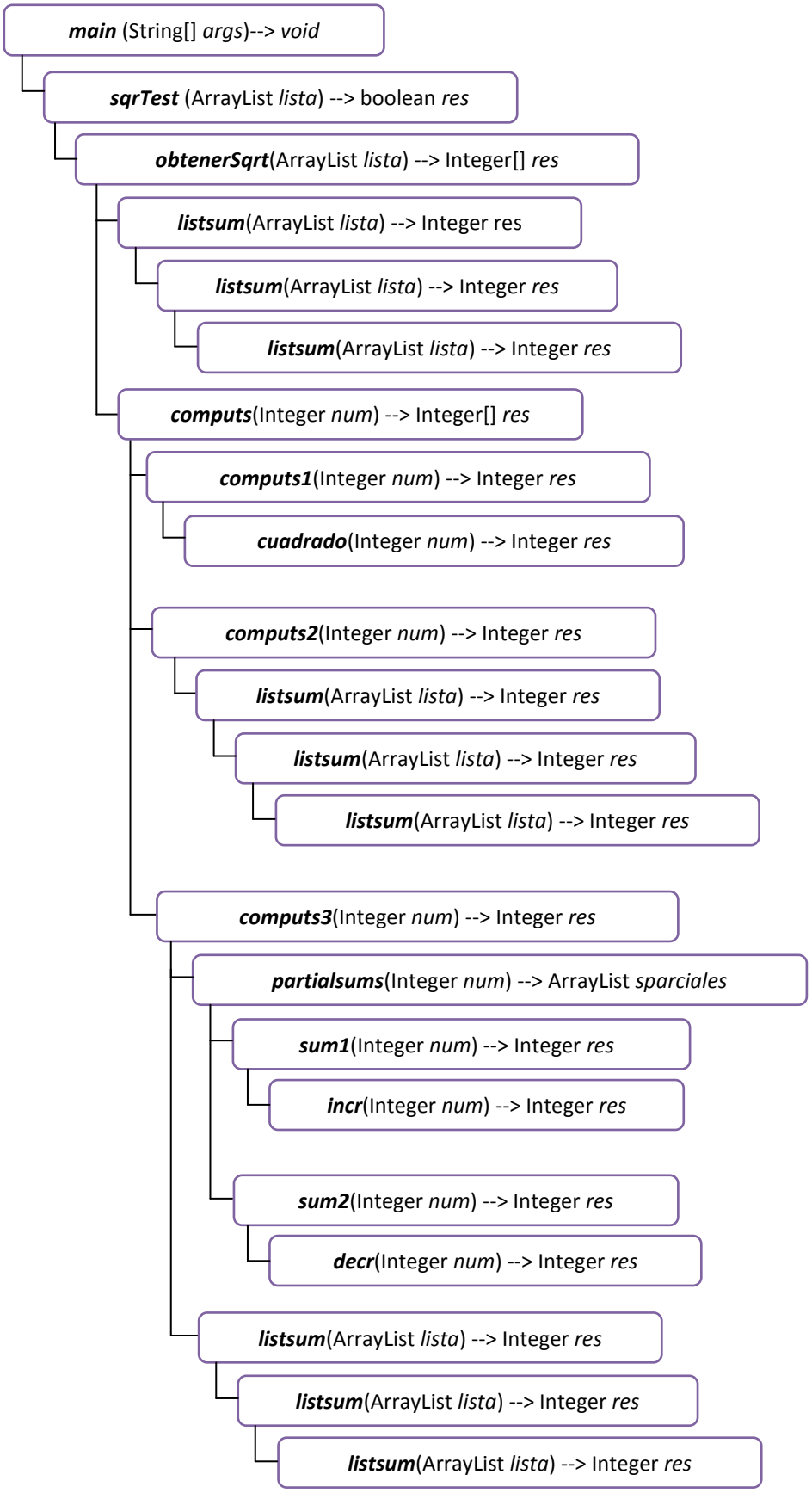
Para la realización de la sesión de depuración se ha introducido un error en el método sum2, de manera que obtiene el resultado de la operación: (n+(n+1))/2.

5.2 Implementación del programa

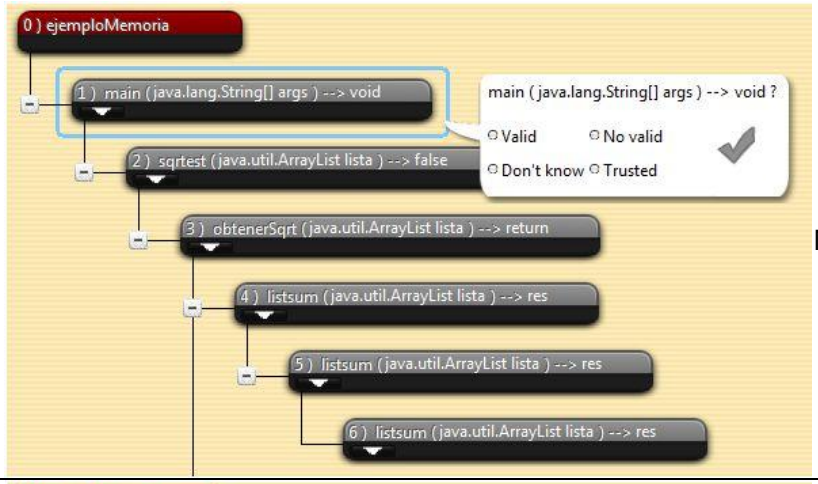
```
public class ejemploMemoria {

	public static void main(String[] args)
	{
		ArrayList<Integer> lista = new ArrayList<Integer>();
		lista.add(new Integer(1));
		lista.add(new Integer(2));
		boolean resultado = sqrtest(lista);
	}

	public static boolean sqrtest(ArrayList<Integer> lista)
	{
		Integer [] arrayComputs = obtenerSqrt(lista);
		return
		arrayComputs[0]== arrayComputs[1] && arrayComputs[1]== arrayComputs[2];
	}

	public static Integer[] obtenerSqrt(ArrayList<Integer> lista)
	{
		return computs(listsum(lista));
	}

	public static Integer listsum(ArrayList<Integer> lista)
	{
		Integer res;
		if(lista.size() == 0) res = new Integer(0);
		else
		{
			Integer aux = lista.remove(0);
			res = aux + listsum(lista);
			lista.add(aux);
		}
		return res;
	}

	private static Integer[] computs(Integer resultado)
	{
		Integer arrayComputs [] = new Integer[3];
		arrayComputs[0] = computs1(resultado);
		arrayComputs[1] = computs2(resultado);
		arrayComputs[2] = computs3(resultado);
		return arrayComputs;
	}

	private static Integer computs1(Integer resultado)
	{
		return cuadrado(resultado);
	}
```

```
	private static Integer cuadrado(Integer resultado)
	{
		return resultado*resultado;
	}

	private static Integer computs2(Integer resultado)
	{
		ArrayList<Integer> lista = new ArrayList<Integer>();
		for(int i = 0; i< resultado; i++) lista.add(resultado);
		return listsum(lista);
	}

	private static Integer computs3(Integer resultado)
	{
		return listsum(partialsums(resultado));
	}

	private static ArrayList<Integer> partialsums(Integer resultado)
	 {
		ArrayList<Integer> sparciales = new ArrayList<Integer>();
		sparciales.add(sum1(resultado));
		sparciales.add(sum2(resultado));
		return sparciales;
	}

	private static Integer sum2(Integer resultado)
	{
		return (resultado+decr(resultado))/2;
	}

	private static Integer incr(Integer resultado)
	{
		return resultado = new Integer(resultado+1);
	}

	private static Integer sum1(Integer resultado)
	{
		return (resultado*incr(resultado))/2;
	}

	private static Integer decr(Integer resultado)
	{
		return resultado = new Integer(resultado-1);
	}

}
```

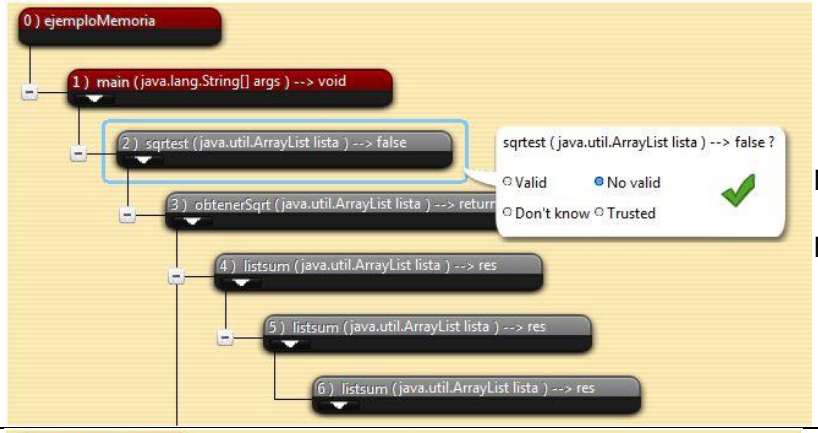
5.3 Árbol de ejecución



5.4 Sesión de depuración

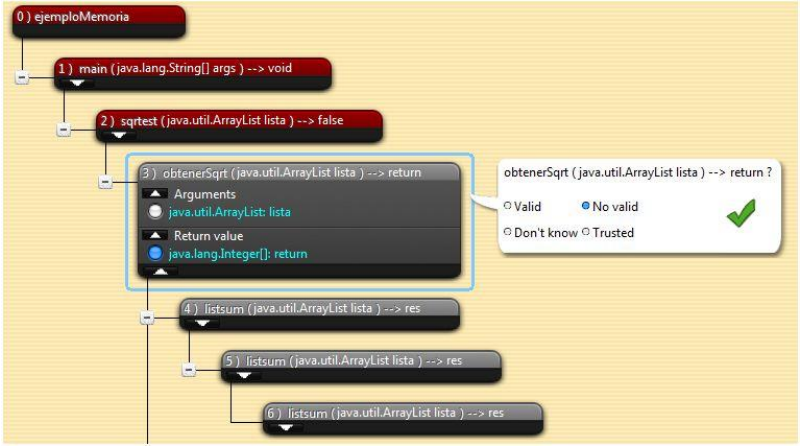


Nodo 1: ¿?



Nodo 1: Valid

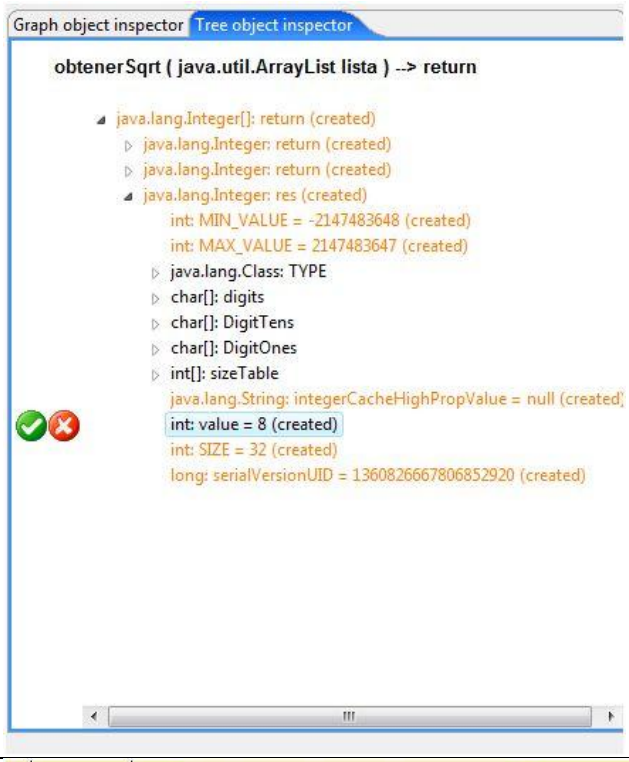
Nodo 2: ¿?



Nodo 1: Valid

Nodo 2: No Valid

Abrimos en el object inspector el valor de retorno del nodo 3.



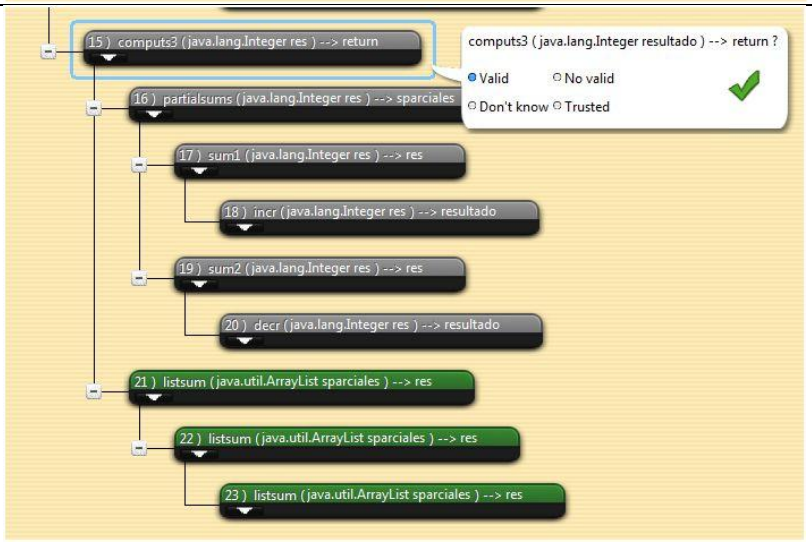
En el object inspector observamos que una de las 3 componentes del array devuelto es diferente.

Según la computación realizada en el nodo 3, se debería devolver el mismo valor en las 3 componentes (el cuadrado del número según 3 métodos diferentes)

Por ello marcamos el valor de esta componente como incorrecto.

Esto inicia SDT, llevando la depuración al nodo 21 (creador del súbtermino marcado)

Nodo 21: ¿?



Nodo 21: Valido

Nodo 15: ¿?

Nodo 15: No valid
Termina SDT, y se inicia la búsqueda del error según la estrategia activa.

Nodo 16: ¿?

Abrimos el object inspector del valor de retorno del nodo 16.

Observamos que la segunda componente del array devuelto no se corresponde con la lógica pretendida, por ello marcamos su valor como incorrecto.

Esto inicia SDT, llevando la depuración al nodo 19 (creador del súbtermino marcado)

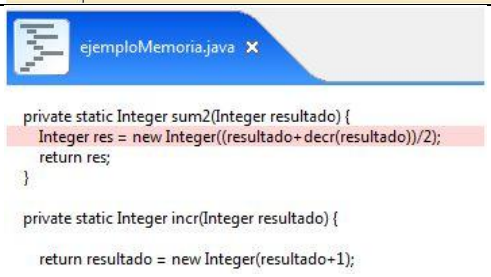
Nodo 19: ¿?

Nodo 19: No valid

Nodo 20: ¿?

Nodo 20: Valid

Error encontrado en el nodo 19.

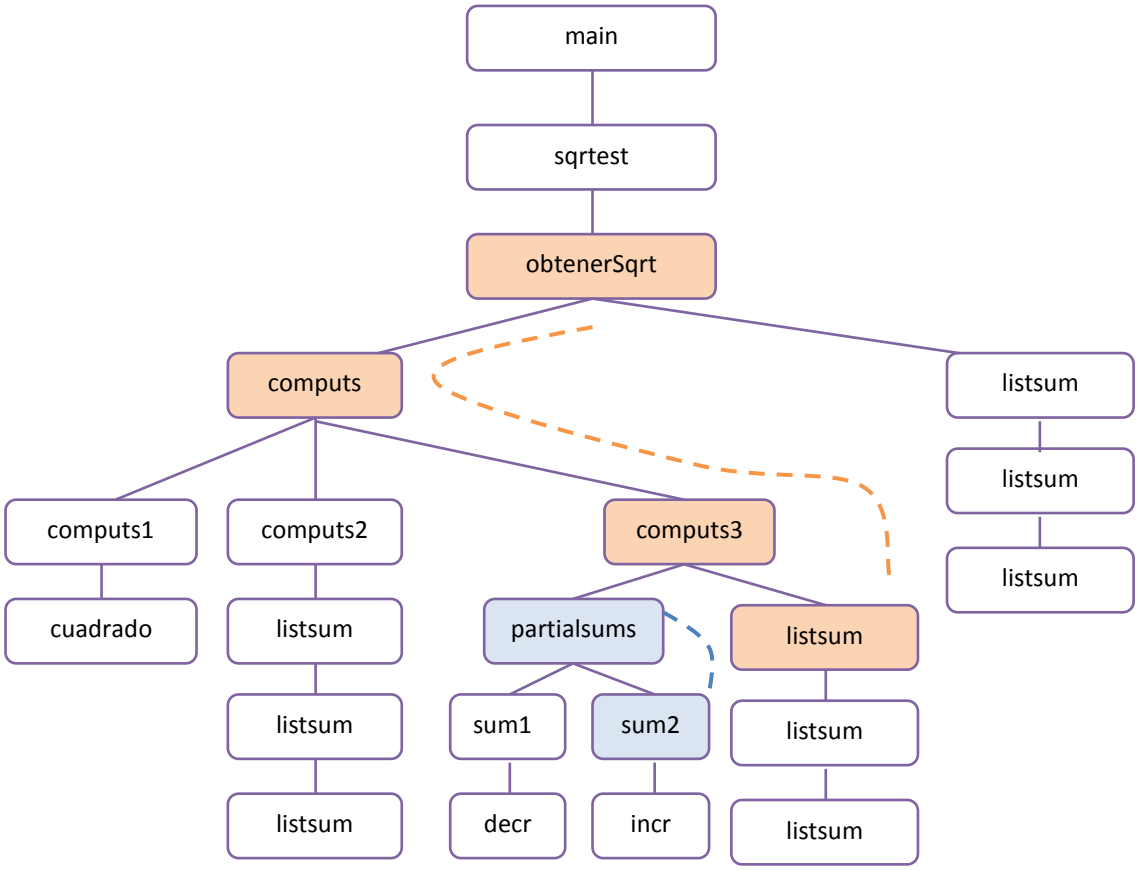


Código del error en el nodo 19.

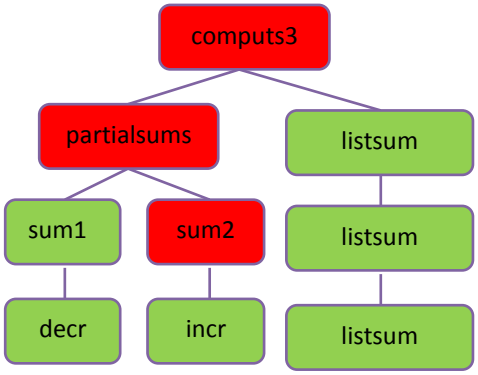
5.5 Resumen de la depuración

En esta sesión de depuración vemos como DDJ sigue el comportamiento esperado ante el marcado de un subtérmino como incorrecto.

Como hemos visto DDJ ha redirigido la depuración al nodo creador del subtérmino, y seguidamente a dirigido las cuestiones mediante un camino entre este nodo, y el que contiene el subtérmino marcado.



1º Marcado del subtermino en obtenerSqrt (Inicio algoritmo SDT)
2º Depuración nodo listsum → Válido
3º Depuración nodo computs3 → No válido
4º Depuración nodo partialsums → Marcamos subtermino (inicio algoritmo SDT)
5º Depuración nodo sum2 → No válido.
6º Depuración nodo incr → Válido.



Error encontrado en el nodo sum2.

6 Conclusiones

La depuración declarativa es una técnica muy útil, que permite la depuración de programas, sin necesidad de consultar el código fuente. Sólo se necesita conocer la lógica pretendida por el programador en cada computación, y el depurador nos ofrece el estado del sistema en cada momento. DDJ emplea una estructura de datos, el Árbol de Ejecución, que representa éstas computaciones, y que se puede explorar mediante su Interfaz gráfica de usuario.

Existen diversas estrategias de depuración del Árbol de Ejecución implementadas en DDJ, todas ellas realizan una búsqueda del nodo erróneo mediante las respuestas dadas por el programador a una serie de preguntas acerca de si la ejecución de ciertos nodos ha ido correctamente o no. Pero solamente con esas respuestas, dejamos aparte cierta información que sería beneficiosa para la búsqueda. Si un usuario marca un nodo como incorrecto, conoce implícitamente el conjunto de soluciones correctas y observa que éstas difieren de las que hay representadas en el nodo.

En la mayoría de los casos, la salida de la computación es casi correcta, y sólo un conjunto de valores de retorno (u objetos modificados) es incorrecto. Ante nodos grandes, con un subárbol extenso, resultaría eficiente podar aquellas partes correctas y centrar la búsqueda en las partes incorrectas. Por ello se ha implementado en DDJ la estrategia Subterm Dependency Tracking, que permite al usuario especificar qué argumentos y valores de retorno de las computaciones son incorrectos.

Mediante la información sobre la identidad del subtérmino erróneo, DDJ guía la búsqueda del error. Concretamente, empieza preguntando por los nodos que generaron el subtérmino marcado, ya que probablemente sean éstos quienes tienen errores en su árbol de llamadas.

De este modo, centrar la búsqueda resulta muy beneficioso, ya que no se exploran las partes que generaron las partes correctas, y se evitan preguntas innecesarias. Y por otra parte, aporta un comportamiento más comprensible, ya que es el que el usuario intuitivamente esperaría.

Bibliografía:

1. MacLarty, I. 2005. Practical Declarative Debugging of Mercury Programs. Ph.D. thesis, Department of Computer Science and Software Engineering, The University of Melbourne.
2. *DDJ: A Declarative Debugger for Java.* [http://users.dsic.upv.es/~jsilva/DDJ/]
3. Shapiro, E. 1982. *Algorithmic Program Debugging*. MIT Press.
4. 1998, IEEE Std 830-1998, *IEEE Recommended Practice for Software Requirements*.
5. Josep Silva. 2006. *A comparative study of algorithmic debugging strategies.* In Proceedings of the 16th international conference on Logic-based program synthesis and transformation (LOPSTR'06), (Ed.). Springer-Verlag, Berlin, Heidelberg, 143-159.
6. *Av-Ron, E. 1984. Top-down diagnosis of prolog programs. Ph.D. thesis, Weizmanm Institute.*
7. *SWT: The Standard Widget Toolkit.* Disponible en la URL: *http://www.eclipse.org/swt/*
8. *Java Developement Kit (JDK).* Disponible en la URL: *http://download.oracle.com/javase/6/docs/*
9. Grupo de investigación de la UPV, MIST (Multi-paradigm Software Technology) *http://users.dsic.upv.es/~gvidal/german/mist/*
10. *D.Insa and J. Silva. An Algorithmic Debugger for Java. In Proc. Of the 2010 International Conference on Software Maintenance (ICSM 2010) Timisoara, Romania, September 12-18 2010.*
11. *D.Insa. Proyecto final de carrera: Diseño e implementación de un depurador declarativo para Java.* Universidad Politécnica de Valencia. Septiembre 2010.
12. Mark Allen Weiss. *Estructuras de datos en JAVA : compatible con JAVA 2.* Madrid [etc.] : Addison-Wesley, D.L. 2000, 2006
13. Luis Joyanes Aguilar, Matilde Fernández Azuela. *Java 2 : manual de programación.* Madrid [etc.] : McGraw-Hill/Interamericana de España, D.L. 2001