Tesina de Máster

Diseño y Desarrollo de un Depurador Híbrido para Java

Autor

Juan González Martínez

Director

Josep Silva Galiana

16 deSeptiembre de 2013

Abstract

During many years, Print Debugging has been the most used method for debugging. Nowadays, however, industrial languages come with a trace debugger that allows programmers to trace computations step by step. Almost all modern programming environments include debugging utilities that allows us to place breakpoints and to inspect the state of a computation in any given point. Nevertheless, this debugging method has been criticized for being completely manual and time-consuming. Other debugging techniques have appeared to solve some of the problems of Trace Debugging, but they suffer from other problems such as scalability. In this work we present a new hybrid debugging technique. It is based on a combination of Trace Debugging, Algorithmic Debugging and Omniscient Debugging to produce a synergy that exploits the best properties and strong points of each technique. We describe the architecture of our hybrid debugger and our implementation that has been integrated into Eclipse as a plugin.

Resumen

Durante muchos años “*Print Debugging*” ha sido el método más usado para la depuración de programas. Hoy en día, sin embargo, los lenguajes industriales incorporan un depurador de trazas que permite al programador realizar trazas del programa paso a paso. La mayoría de los entornos modernos de programación incluyen herramientas de depuración que nos permiten insertar breakpoints e inspeccionar el estado de la computación en cualquier punto dado. Sin embargo este método ha sido criticado debido a su alto coste temporal ya que se trata de un método completamente manual. Para resolver los problemas de esta técnica han surgido otras técnicas de depuración, aunque estas sufren de otro tipo de problemas como la escalabilidad. En este trabajo se presenta una nueva técnica de depuración híbrida. Esta técnica está basada en la combinación de la depuración por trazas, la depuración algorítmica y la depuración omnisciente para producir una unión que explote las mejores propiedades y los puntos fuertes de cada técnica. Se describirá una arquitectura de nuestro depurador híbrido y la implementación realizada la cual ha sido integrada en el IDE Eclipse como un plugin.

Contenido

Tabla de figuras

1. Introducción

La depuración es una de las tareas que más tiempo consume en la ingeniería del software. Sin embargo, la automatización de esta tarea está aún lejos de ser una realidad. De hecho, durante muchos años, *Print Debugging* (también conocida como *Echo Debugging*) ha sido la técnica más usada para la depuración.

Afortunadamente todos los entornos de depuración modernos, como por ejemplo, BorlandJBuilder [**1**], NetBeans [**2**] o Eclipse [**3**] incluyen un depurador por trazas, el cual permite a los programadores realizar trazas del programa paso a paso. Sin embargo, la depuración por trazas es una tarea completamente manual y el programador es el encargado de inspeccionar la computación de un programa en un nivel bajo de abstracción. Por esta razón, han sido propuestas otras técnicas de depuración que tratan de resolver estos problemas, aunque éstas presentan otros problemas. Por ejemplo, la Depuración Algorítmica [**4**] [**5**] (también conocida como Depuración Declarativa) es semi-automática puesto quela búsqueda del error está dirigida por el depurador y no por el programador; y debido a su alto nivel de abstracción el programa puede ser depurado sin ver una sola línea de código. Sin embargo, dicha técnica sufre de problemas de escalabilidad.

En este trabajo se introduce una técnica de depuración híbrida (*Hybrid Debugging* HD*)* que combina tres técnicas de depuración distintas, llamadas depuración por trazas (*Trace Debugging* TD), depuración omnisciente (*Omniscient Debugging* OD) y depuración algorítmica (*Algorithmic Debugging* AD). La combinación de estas técnicas está realizada explotando los puntos fuertes de cada técnica y contrarrestando o eliminando los puntos débiles mediante su composición. La implementación elegida es para el lenguaje de programación Java (la implementación realizada es un plugin para Eclipse) aunque la técnica y la arquitectura desarrolladas para nuestro depurador pueden ser usadas para cualquier lenguaje de programación.

A continuación, en este trabajo se procede a la explicación de las distintas partes con la intención de proveer al lector de una visión global de las técnicas de depuración usadas (TD, OD, AD, así como la realizada HD) con la siguiente estructura: en una primera sección se describen las técnicas de depuración usadas para crear la técnica de depuración propuesta en este trabajo, seguida de la sección de explicación de dicha técnica. Acto seguido se realiza una breve introducción y descripción del entorno elegido para la implementación de esta técnica (Eclipse). La sección siguiente describe una especificación de requisitos la cual describe y detalla los requisitos posee la herramienta desarrollada.

En las dos secciones posteriores se realiza una visión global de la arquitectura usada en el desarrollo así como la implementación del mismo. Además después de estas se ha realizado un manual de usuario el cual describe los componentes de la herramienta desarrollada así como la utilidad de cada uno

de ellos. Además en este manual de usuario se ha introducido un caso de estudio en el que se realizará de una manera práctica, mediante la herramienta desarrollada, una sesión de depuración híbrida. Por último, en las dos últimas secciones se describen los trabajos relacionados así como las conclusiones de este trabajo.

2. Técnicas de depuración

En esta sección se explican las diversas técnicas usadas por la técnica de depuración híbrida propuesta en este trabajo: TD, OD, y AD. Para cada técnica, además, se analizan sus puntos fuertes así como los débiles y su aplicabilidad al lenguaje de programación escogido en este trabajo, Java.

2.1. Depuración por trazas

El método más usado para la depuración de programas es la depuración por trazas (*Trace Debugging* TD). Ésta permite al programador moverse paso a paso a través de la computación de un programa. El programador sitúa un breakpoint en una determinada línea del código fuente y el depurador detiene la ejecución del programa cuando dicha línea es alcanzada. Una vez en ese punto, paso a paso, el programador puede inspeccionar el estado de la ejecución (variables, estado de las mismas, excepciones ocurridas, etc.). Mientras el programador atraviesa la traza, cuando un método es llamado, el depurador puede entrar en dicho método (*stepinto*) o saltarlo (*stepover*). Los breakpoint más avanzados pueden ser condicionales, por ejemplo, incluyendo clausulas de condición sobre variables y su valor o sobre la acción realizada donde se definen. Por ejemplo, es posible definir un breakpoint que sólo detendrá la ejecución cuando ocurra una excepción o cuando una clase sea cargada. TD tiene una ventaja importante sobre el resto de técnicas: su escalabilidad. El depurador sólo necesita tomar el control sobre el intérprete y ejecutar normalmente el programa. Por lo tanto la escalabilidad de esta técnica es la mima que la del intérprete usado. Por otro lado esta técnica posee cuatro inconvenientes principales:

1. Todo el proceso de depuración se realiza a un nivel muy bajo de abstracción. El programador únicamente sigue los pasos del intérprete, y necesita comprender como varían los valores de las variables para identificar el error.
2. El depurador puede generar una cantidad abrumadora de información.
3. El proceso de depuración es completamente manual. El programador usa su intuición para situar los breakboints. Si el breakpoint se sitúa después del error, es necesario recolocarlo y reiniciar el programa. Si el breakpoint se sitúa mucho antes del error, entonces es necesario analizar manualmente gran parte de la ejecución.
4. La inspección de la ejecución se realiza hacia delante, cuando la manera natural de descubrir el error sería ir hacia atrás desde la detección del síntoma del error.

2.2. Depuración omnisciente

La depuración omnisciente [**6**] resuelve el cuarto problema de la depuración mediante trazas con un coste de escalabilidad significativo. Básicamente las dos técnicas se basan en el uso de breakpoints y son similares desde un punto de vista funcional. La diferencia reside en que la depuración omnisciente permite al programador navegar hacia delante y atrás (cronológicamente) por la ejecución del programa. Esto es muy útil, ya que, permite al programador navegar hacia atrás por la ejecución del programa desde cualquier punto (p.e., donde se detecta el error). Uno de los esquemas más escalables para realizar esta tarea está representado en la FIG. 1.En dicha figura tenemos una línea horizontal, que representa la ejecución del programa como una sucesión de eventos. Algunos de estos eventos son invocaciones de métodos (representadas con un círculo blanco) y salidas de métodos (representadas con un círculo negro). Cada evento es identificado por un instante de tiempo. Para la ejecución del programa la depuración omnisciente almacena un histórico de las variables, el cual contiene los valores de las variables junto con el instante de tiempo en el que dicho valor ha sido actualizado. El depurador omnisciente también almacena información sobre el ámbito de las variables, la cual omitiremos en esta sección para mayor claridad. Con toda esta información el depurador puede recomponer cualquier estado de la ejecución. Por ejemplo en el estado de la ejecución en el instante de tiempo 42, el valor de *M.N.* y no existe, y el último valor para las variables *O.x* y *O.v* son 23 y 3 respectivamente.

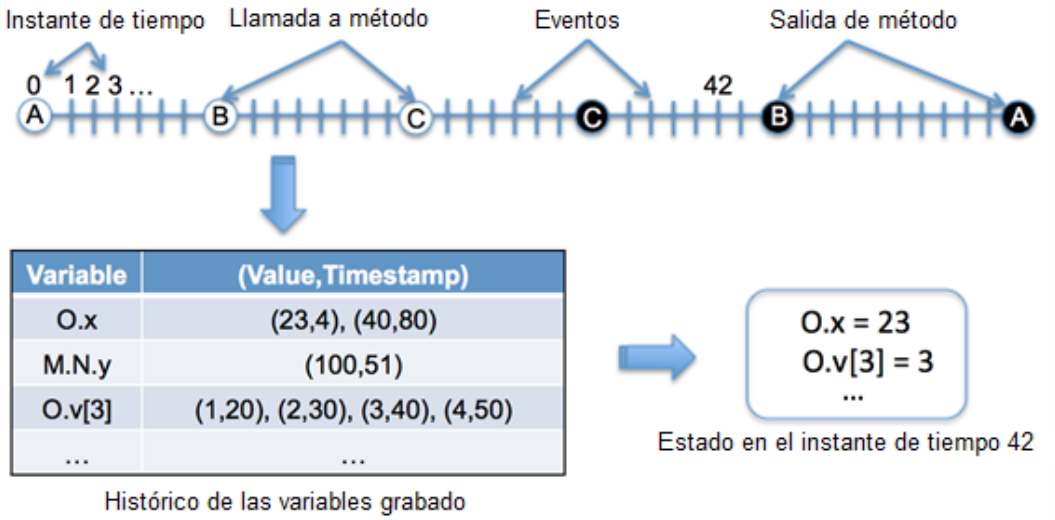


Fig. 1 Esquema basado en instantes de tiempo para almacenar trazas en depuración omnisciente

Siendo capaz de reconstruir la traza completa, se le permite al programador comenzar la ejecución en cualquier instrucción. Sin embargo, almacenar todos los valores tomados por todas las variables en ejecución es normalmente imposible en programas industriales reales (programas grandes), e incluso para programas de tamaño medio. Por esta razón, la escalabilidad está muy limitada para esta técnica.

2.3. Depuración algorítmica

La depuración algorítmica [**4**] [**5**] es una técnica de depuración semi-automática que está basada en las respuestas del programador a preguntas generadas automáticamente por el depurador. Estas preguntas cuestionan si la computación de un determinado método, es decir, la ejecución de un método con unos valores de entrada están establecidos, devuelve un resultado correcto o no. Las preguntas proveen de información al depurador sobre la corrección de algunas computaciones(o subcomputaciones) de un programa dado; y el depurador usa dicha información para guiar la búsqueda del error hasta que una porción errónea del código es aislada.

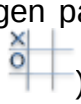
```
public class Replay {
        public static void main(String[] args) throws IOException {
                TicTacToe game = new TicTacToe();
                FileReader file = new FileReader("./game.rec");
                play(game, file);
        }

        private static void play(TicTacToe game, FileReader file) throws IOException {
                BufferedReaderbr = new BufferedReader(file);
                String linea = br.readLine();
                while ((linea = br.readLine()) != null) {
                        char player = linea.charAt(0);
                        int row = Integer.parseInt(linea.charAt(2) + "");
                        int col = Integer.parseInt(linea.charAt(4) + "");
                        game.mark(player, row, col);
                }
        }
}

public class TicTacToe {
        private static boolean equals(char c1, char c2, char c3) {
                return c1 == c2 && c2 == c3;
        }

        private char turn = 'X';
        private char[][] board = new char[3][3];
        public void mark(char player, int row, int col) {
                if (turn == 'nu0000' || turn != player
                        || row< 0 || row > 2 || col < 0 || row > 2
                        || board[row][col] != '\u0000')
                        return;
                board[col][row] = player; // ¡¡Bug!! Correct: board[row][col] = player;
                turn = turn == 'X' ? 'O' : 'X';
                if (win(row, col))
                        turn = '\u0000';
        }
        privateboolean win(int row, int col) {
                if (board[row][col] == '\u0000')
```

```
                return false;
            if (equals(board[row][0], board[row][1], board[row][2]))
                return true;
            if (equals(board[0][col], board[1][col], board[2][col]))
                return true;
            if (col == row && equals(board[0][0], board[1][1], board[2][2]))
                return true;
            if (col + row == 2 && equals(board[0][2], board[1][1], board[2][0]))
                return true;
            return false;
        }
    }
```

Fig. 2 Programa de ejemplo

Ejemplo1. Consideremos el programa Java dado en la Fig. 2.

Éste simula un juego de tres en raya—se recomienda el lector no ver el código fuente en este momento e intentar depurar el programa sin ver el código. Esto es posible con AD como veremos en la próxima sesión de depuración—. Este programa es erróneo, lo que hace que no produzca las marcas esperadas en el tablero. La clase *Replay* lee desde un fichero un juego nuevo y reproduce el juego utilizando un objeto de la clase *TicTacToe*. El carácter nulo en Java está representado por '\u0000'.

A continuación se muestra una sesión de depuración para este programa donde los tableros serán representados con una imagen para mayor claridad (por ejemplo, {{X,",}{O,",}{",",}} está representado por )

```
Starting Debugging Session...
Node  Initial context          Method call          Final context       Answer
(2)  [turn='X',board=  ]  game.mark('X',0,0)  [turn='O',board=  ] ?  YES

(7)  [turn='O',board=  ]  game.mark('O',0,1)  [turn='X',board=  ] ?  NO

(8)  [turn='X',board=  ]  game.win(0,1)=false [turn='X',board=  ] ?  YES

Bug found in method:
TicTacToe.mark(char, int, int)
```

Fig. 3 Sesión de depuración algorítmica

Como podemos apreciar en la Fig. 3 el depurador genera preguntas, y el programador sólo ha de responder a estas preguntas con un SÍ o un NO. Por lo que no es necesario ver el código. Cada pregunta hace referencia a la llamada a un método, y el programador responde SÍ cuando la ejecución sea correcta (por ejemplo, si el contexto final de la computación es el esperado con respecto a las variables de entrada y el contexto inicial dados, entendiendo como contexto el estado de la máquina: variables globales, variables locales y todo lo

visible dentro del ámbito del método); por el contrario, si la ejecución no es correcta el programador responderá NO.

Al final, el depurador señala como salida un método que contiene un error. En este caso, el método *TicTacToe.mark* es erróneo. Este método primero comprueba que el movimiento sea correcto (por ejemplo, que es el turno del jugador, la posición seleccionada está dentro del tablero, etc.). Si el movimiento es correcto, entonces coloca la marca en la posición indicada del tablero, cambia cual es el jugador que realizará el siguiente movimiento, y finalmente comprueba si el movimiento realizado termina la partida debido a que es el movimiento ganador. Desafortunadamente el programador ha intercambiado la fila y la columna produciendo un error. Este error puede ser subsanado fácilmente reemplazando *board[col][row] = player* por *board[row][col] = player.*

Normalmente, los depuradores algorítmicos poseen un *front-end* que produce una estructura de datos que representa la ejecución de un programa —también llamada árbol de ejecución (ET) [**7**]— y un *back-end* que usa el ET para realizar preguntas al programador y procesar las respuestas de éste para localizar el error. Cada nodo del ET contiene una ecuación que consiste en un texto que representa la ejecución de un método con los argumentos y el resultado totalmente evaluados. El nodo, además, contiene información adicional sobre el contexto antes y después de su ejecución (valores de los atributos o variables globales visibles desde el ámbito del método).

Esencialmente, AD es un proceso de dos fases: Durante la primera fase se construye el ET, mientras que en la segunda fase el ET es explorado. El ET se construye de la siguiente manera: El nodo principal (o raíz) es (normalmente) la función principal (o *main*) del programa. Para cada nodo n asociado con un método m y para cada llamada hecha en la definición de m, se añade recursivamente un nuevo nodo al ET como hijo de n.

Ejemplo2. Consideremos de nuevo el programa de la FIG. 2.

La FIG. 4 representa la porción del ET asociada con la ejecución del método *play(game, file)* utilizando game.rec como fichero de entrada. Cada nodo contiene:

- Un string representando la llamada del método (incluyendo la entrada y la salida) en lo alto de cada nodo.
- Las variables (y sus valores) en el ámbito al principio y al final de la ejecución del método. Cuando el valor de una variable es modificado durante la ejecución de un método, el nodo contiene ambos valores a la izquierda y la derecha del nodo respectivamente. Cuando las variables no son modificadas, se muestra una única vez en el centro del nodo.

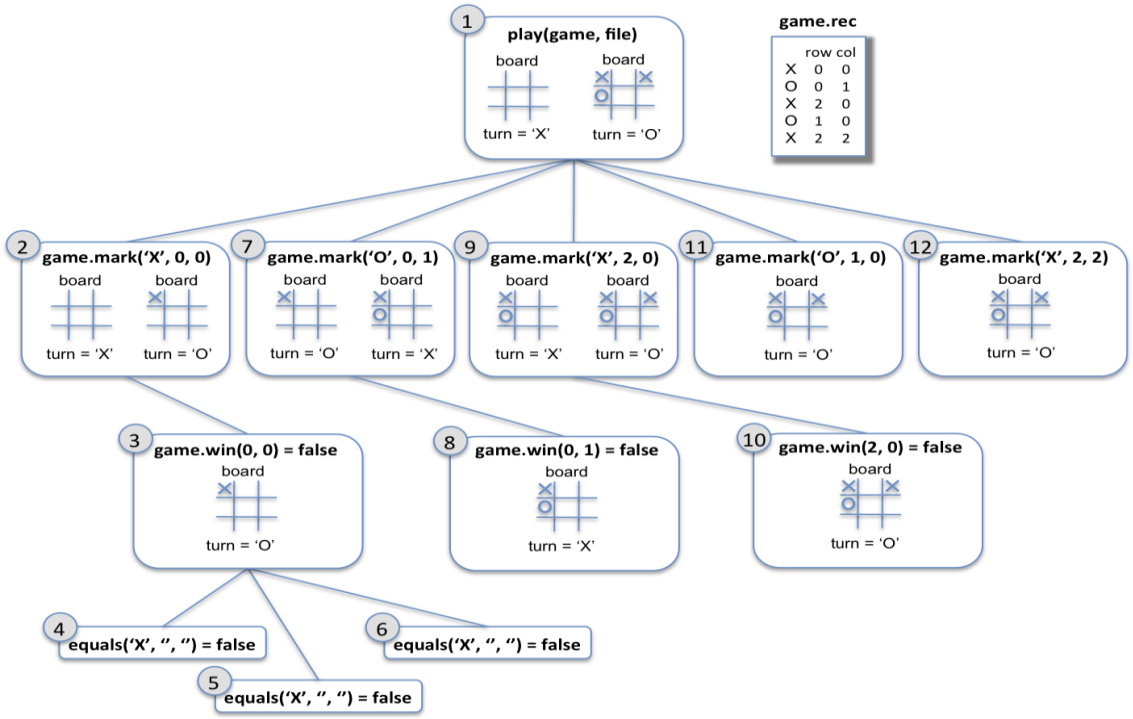


Fig. 4 ET asociado a la llamada del método *play (game, file)* del programa de la Fig. 2

Una vez está construido el ET, en la segunda fase, el depurador usa una estrategia para atravesar el ET preguntando a un oráculo la respuesta para cada pregunta. Por ejemplo, cada pregunta en la sesión de depuración del Ejemplo1 corresponde a un nodo en el ET de la FIG. 4. Estos nodos han sido seleccionados mediante la estrategia *Divide & Query* [**4**]. Después de cada pregunta, algunos nodos del ET son marcados como correctos o incorrectos. Cuando todos los hijos (si tiene) de un nodo incorrecto son correctos, el nodo incorrecto pasa a ser erróneo y el depurador localiza el error en la parte del programa asociada a dicho nodo [**8**].

Teorema 1 (*corrección de AD* [8]). Dado un ET con un nodo n erróneo, el método asociado con n contiene un error.

Teorema 2 (*completitud de AD* [4]). Dado un ET con un error detectado (por ejemplo, el nodo principal es un método con un contexto final erróneo), si todas las preguntas realizadas por el depurador son contestadas, entonces, un error será eventualmente localizado.

La ventaja más importante de AD es su alto nivel de abstracción y su naturaleza semi-automática. Los principales puntos débiles de esta técnica son:

1. Baja escalabilidad. Cada nodo del ET necesita grabar una parte del estado de la ejecución (por ejemplo, el contexto antes y después de la ejecución del método). Almacenar el ET de toda la ejecución puede ser impracticable.
2. La estrategia elegida puede hacer preguntas innecesarias antes de alcanzar la parte de la ejecución que contiene el error.
3. Baja granularidad del error encontrado. Esta técnica devuelve un método como error en vez de una línea o expresión.

3. Depuración híbrida

En esta sección se explica el depurador creado en este trabajo, un depurador híbrido para Java (*Hybrid Debugger for Java* HDJ), basado en las técnicas de depuración explicadas en la sección anterior. Dicho depurador combina TD, AD y OD para producir una combinación que explota las mejores propiedades y los puntos fuertes de cada técnica.

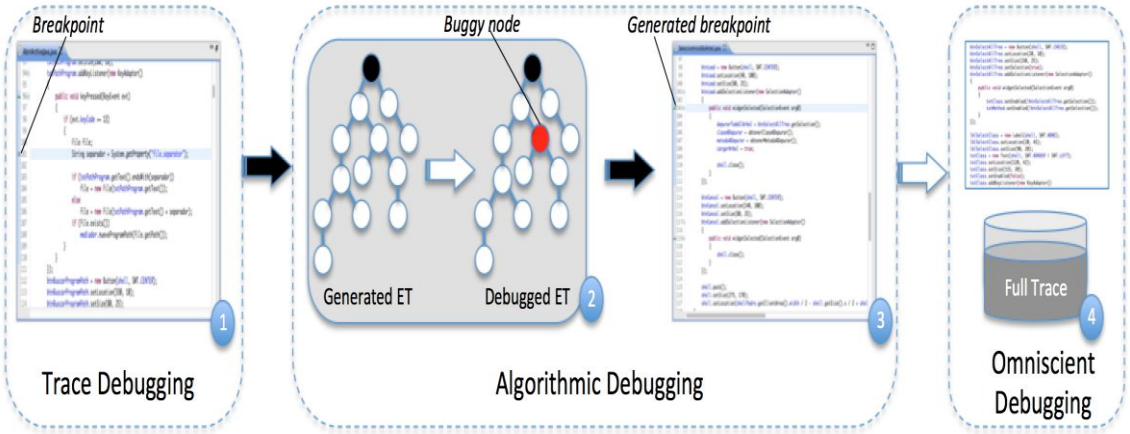


Fig. 5 Depuración híbrida con HDJ

Empezamos describiendo los pasos a seguir en una sesión de depuración híbrida. Considerando el diagrama de la FIG. 5, esta figura resume el método de depuración desarrollado en este trabajo. Podemos apreciar tres bloques principales correspondientes a TD, AD y OD. Estos bloques contienen cuatro elementos que han sido numerados y conectados entre sí mediante flechas. Las flechas negras representan procesos automáticos (realizados por el depurador); por el contrario las fechas blancas representan procesos manuales (realizados por el programador):

- **Depuración mediante trazas.** Primero, después de observar un error en el programa, el usuario explora el código como de costumbre con el depurador por trazas y añade un breakpoint*b1* en una línea sospechosa (probablemente, dentro de la última parte del código modificada).
- **Depuración algorítmica.** En segundo lugar, el depurador identifica el método *m1* que contiene el breakpoint*b1* y genera un ET cuyo método raíz es *m1*. Después de esto, el usuario explora el ET usando AD hasta encontrar el nodo *n* erróneo. Observamos que, de acuerdo con el Teorema 2, si un método *m1* es incorrecto, entonces se garantiza que AD encontrará un nodo erróneo (y por consiguiente un método erróneo). En el nodo marcado como erróneo *n* AD genera un nuevo breakpoint*b2*. Éste es colocado en la primera línea del método *m2* asociado con *n*. Y, además, *b2* es un breakpoint condicional que detendrá la ejecución del programa solamente cuando haya garantías de que se producirá el error. Las condiciones aseguran que todos los valores de los parámetros de *m2* serán exactamente iguales a aquellos de la llamada al método *m2* asociado a *n*.

- **Depuración omnisciente.** En tercer lugar, el depurador actúa como un depurador omnisciente que explora el método *m2* reproduciendo la ejecución concreta donde apareció el error durante la depuración mediante AD. El usuario puede explorar el método hacia atrás desde el resultado incorrecto del método. Obsérvese que la fase de depuración OD es escalable, ya que, sólo necesita grabar la información de un método. Nótese también que la fase de depuración AD asegura que los métodos ejecutados por el método *m2*funcionancorrectamente.

Estas tres fases descritas producen una técnica de depuración que reúne las ventajas y las mejores propiedades de cada técnica. Además, uno de los objetivos más importantes de nuestro depurador es no poseer una metodología rígida. Nuestro objetivo es dar al programador la libertad de poder cambiar de una técnica a otra en cualquier momento. Por ejemplo, si un programador está usando TD y decide usar OD en un método, debe de tener la posibilidad de hacerlo. La arquitectura de la herramienta desarrollada provee de esta flexibilidad que incrementa notablemente la usabilidad de la misma, y pienso que ésta es la aproximación más realística para la depuración de programas.

4. Eclipse

Eclipse [**3**] es un entorno de desarrollo integrado de código abierto multiplataforma para desarrollar lo que Eclipse denomina "Aplicaciones de Cliente Enriquecido". Eclipse es, en el fondo, un armazón (*workbench*) sobre el que se pueden montar herramientas de desarrollo para cualquier lenguaje, mediante la implementación de los plugins adecuados. La arquitectura de plugins de Eclipse permite, además de integrar diversos lenguajes sobre un mismo IDE, introducir otras aplicaciones accesorias que pueden resultar útiles durante el proceso de desarrollo como: herramientas UML, depuradores, editores visuales de interfaces, ayuda en línea para librerías, etc.

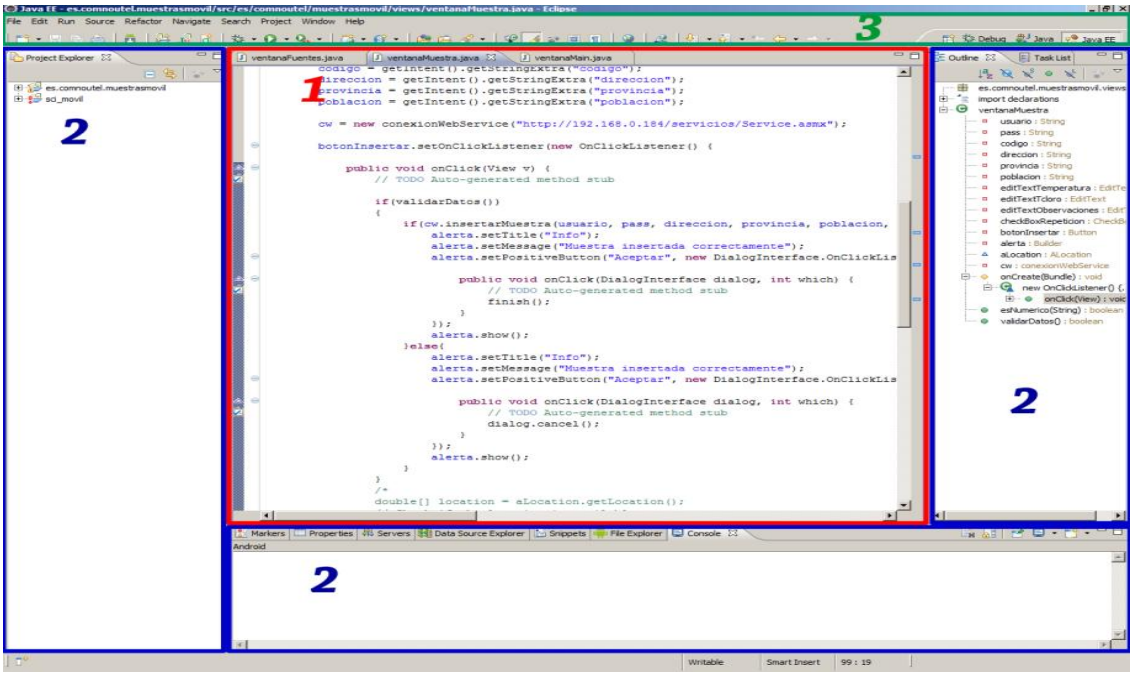


Fig. 6 IDE Eclipse

En la FIG. 6se muestra una imagen del entorno dividida en 3 bloques los cuales pasaremos a describir a continuación:

1.- Ventana principal: se llama “Editor”. Los editores son el lugar donde se escribirán los programas. Es posible tener varios editores abiertos a la vez, apilados uno encima de otro, en la parte superior de la ventana de editores, mostrando solapas que permiten acceder a cada uno de los editores abiertos (o bien cerrarlos directamente).

2.- Vistas: un segundo tipo de ventanas “secundarias” que sirven para cualquier cosa, desde navegar por un árbol de directorios, hasta mostrar el contenido de una consulta SQL. Se puede decir que las Vistas son ventanas auxiliares para mostrar información, requerir datos, etc. Para seleccionar qué Vistas se deben mostrar, se utiliza la opción “*Show View*” en el menú “*Window*”. Cada plugin puede definir Editores propios y todas las Vistas que sean necesarias.

3.- Barras de Herramientas: existen dos barras de herramientas: la barra de herramientas principal y la barra de perspectivas. La barra de herramientas principal contiene accesos directos a las operaciones más usuales (guardar, abrir, etc.), botones que permiten lanzar la ejecución de herramientas externas y tareas relacionadas con el editor activo (ejecutar un programa, depurar, etc.). La barra de perspectivas contiene accesos directos a las perspectivas que se están utilizando en el entorno de desarrollo actualmente.

Eclipse emplea plugins, los cuales son los módulos usados para proporcionar toda su funcionalidad al frente de la plataforma de cliente enriquecido, a diferencia de otros entornos monolíticos donde las funcionalidades están todas incluidas, las necesite el usuario o no. Este sistema modular permite a Eclipse trabajar con diversos lenguajes de programación (C/C++ y Python), así como lenguajes de procesado de texto como LaTex, además de otras herramientas auxiliares para la programación, como la integración con servidores de aplicaciones (Tomcat, Jboss, etc.).

4.1. Arquitectura del IDE

La Plataforma de Eclipse es un marco de trabajo con un conjunto poderoso de servicios que soporta complementos, como JDT y el Entorno de Desarrollo de Complementos. Consiste de varios componentes principales: un tiempo de ejecución de la plataforma, un Espacio de Trabajo, un Entorno de Trabajo, Equipo de Soporte y Ayuda. Todo esto ilustrado en la FIG. 7.

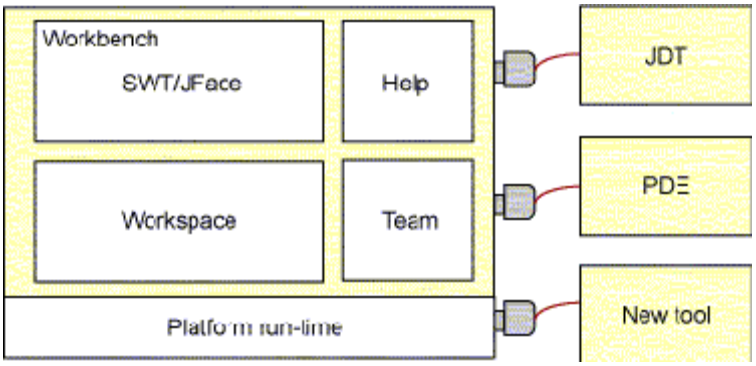


Fig. 7 Arquitectura plataforma Eclipse

A continuación procederemos a describir de manera breve dichos servicios.

4.1.1. Plataforma

El tiempo de ejecución de la plataforma es el núcleo que descubre en el inicio qué complementos están instalados y crea un registro de información sobre ellos. Para reducir el tiempo de inicio y el uso de los recursos, no carga ningún

complemento hasta que realmente se necesita. Excepto el núcleo, todo lo demás se implementa como un complemento.

4.1.2. Espacio de trabajo

El espacio de trabajo es el complemento responsable de la administración de los recursos del usuario. Éste incluye todos los proyectos que crea el usuario, los archivos de esos proyectos y los cambios en los archivos y otros recursos. El espacio de trabajo también es responsable de notificar a otros complementos interesados sobre los cambios en los recursos, como archivos que se crearon, eliminaron o cambiaron.

4.1.3. Entorno de trabajo

El entorno de trabajo proporciona a Eclipse una interfaz de usuario (IU). Se crea con la utilización de un Kit de Herramientas Estándar (SWT)—una alternativa no estándar para el Swing/AWT GUI API de Java—y un API, JFace con un nivel superior, construido sobre SWT que proporciona componentes de la IU.

Se ha comprobado que el SWT es la parte más polémica de Eclipse. SWT se grafica de forma más parecida a las capacidades nativas de los gráficos del sistema operativo subyacente que Swing o AWT, que no sólo hace que SWT sea más rápido, sino también permite que los programas JAVA se parezcan y se sientan más como aplicaciones nativas. El uso de esta nueva API GUI podría limitar la portabilidad del entorno de trabajo de Eclipse, pero los puertos del SWT para los sistemas operativos más populares ya están disponibles.

4.1.4. Soporte del equipo

El componente de soporte del equipo es responsable de proporcionar soporte para el control de la versión y la gestión de la configuración. Agrega vistas según sea necesario para permitir al usuario interactuar con cualquier sistema de control de versiones (si hay) que se esté utilizando. La mayoría de los componentes no necesitan interactuar con el componente de soporte del equipo a menos que proporcionen servicios de control de versiones.

4.1.5. Ayuda

El componente de ayuda pone en paralelo la expansión de la propia Plataforma de Eclipse. De la misma forma que los complementos agregan funcionalidad a Eclipse, la ayuda proporciona una estructura de navegación de complementos que permite a las herramientas agregar documentación en la forma de archivos HTML.

5. Especificación de requisitos

En ésta sección describimos la especificación de requisitos software para el desarrollo del plugin HDJ. Todo su contenido irá dirigido a elaborar una especificación que demarque claramente los requisitos y funcionalidades que han de estar presentes en el plugin HDJ. Esta sección se ha estructurado en concordancia con las directrices dadas por el estándar "*IEEE Recomended Practice for Software Requirement Specifications ANSI/IEEE 830 1998".* Aunque se cuenta de inicio con un depurador algorítmico para programas Java, Declarative Debugger for Java (DDJ) [**9**], se han incluido sus requisitos en este análisis ya que dicho programa y toda su funcionalidad ha de ser adaptada a la plataforma Eclipse [**3**], por lo que se deberán implementar los cambios necesarios, así como, comprobar los mismos.

5.1. Propósito

El objetivo de esta especificación es definir de manera clara y presita todas las funcionalidades y restricciones del sistema que se desea construir. Esta especificación ha sido dirigida por el grupo que ha elaborado esta tesina final de máster, el cual está compuesto por: Josep Silva Galiana y Juan González Martínez. En esta sección se reflejan las funcionalidades y restricciones acordadas por los miembros del grupo a lo largo de diversas reuniones mantenidas durante la especificación y desarrollo de la herramienta expuesta en este trabajo.

5.2. Ámbito del sistema

El objetivo de este sistema es realizar una implementación de la técnica de depuración desarrollada en este trabajo HD, por lo que se creará un programa llamado HDJ (*Hybrid Debugger for Java).*

Se trata de un plugin para la plataforma de desarrollo Eclipse. Éste es un sistema complejo el cual debe realizar las funciones descritas anteriormente para la nueva técnica de depuración desarrollada en este trabajo llamada HD. Como hemos visto anteriormente, esta técnica combina otras técnicas para formar una nueva, por lo que el sistema deberá realizar las funciones necesarias para el correcto funcionamiento de estas técnicas (las cuales son TD, AD y OD), así como implementar la funcionalidad necesaria para el correcto ensamblaje y cohesión de todas ellas. Culminando todo esto con una implementación correcta y válida de HD.

5.3. Definiciones y acrónimos

En esta sección, al no tratarse de una documento de ERS formal y aislado, será obviada ya que las definiciones y acrónimos usados en esta sección ya han sido introducidos y descritos a lo largo de todo el documento.

5.4. Referencias

Al igual que en el punto anterior, este punto será obviado en beneficio de referencias globales de este documento.

5.5. Descripción general

En este punto describimos el sistema a desarrollar con un nivel de abstracción alto. Introducimos cuales son las principales características a las que el sistema debe dar soporte, las funcionalidades que el sistema ha de realizar (de una manera general), la información que el sistema ha de usar para lograr el correcto funcionamiento de estas funcionalidades, las restricciones con las que ha de cumplir el programa y otros factores que puedan afectar al desarrollo de dicho sistema.

5.5.1. Perspectiva del producto

El sistema a desarrollar es un sistema novedoso, el cual implementa funcionalidades para una nueva técnica de depuración descrita en este trabajo. Por este motivo, el programa estará únicamente relacionado con la plataforma eclipse, ya que se desarrollará como un plugin para dicha plataforma. Por lo que tendrá que cumplir con los requisitos y restricciones impuestos por esta plataforma para los plugins que forman parte de ella.

5.5.2. Funciones del producto

De una forma abstracta, el sistema ha de permitir la realización de todas la tareas asociadas a las técnicas de depuración antes mencionadas, tanto a las técnicas que componen la técnica propuesta en este trabajo como a la integración de las mismas, la cual es, en sí misma, la nueva técnica propuesta.

Por lo que el sistema debe ser capaz de dar soporte a todas las tareas necesarias para el correcto funcionamiento de las siguientes técnicas:

- Depuración por trazas
- Depuración algorítmica (o declarativa)

- Depuración omnisciente
- Depuración híbrida

Las funcionalidades necesarias para la correcta implementación de estas técnicas son descritas con mayor detalle a continuación.

- Depuración por trazas

El sistema ha de ser capaz de permitir al usuario inspeccionar la ejecución de un programa en un momento dado. Para esta tarea el sistema ha de hacer uso de breakpoints para detener la ejecución de un programa y permitir inspeccionar el estado de la ejecución en dicho momento. Además, el programa debe permitir al usuario continuar con la ejecución de dicho programa, bien hasta el siguiente breakpoint (si se encuentra algún breakpoint más antes del fin de la ejecución o la suspensión de la misma) o siguiendo la ejecución del mismo si no es así.

El sistema debe de proveer de mecanismos que permitan al usuario introducir por si mismo dichos breakpoints, así como controlar la ejecución del programa. Además, deberá proveer al usuario de interfaces gráficas amigables que le permitan inspeccionar el estado de la máquina, cuando ésta esté detenida (ya sea por un breakpoint o por causas ajenas al sistema).

- Depuración algorítmica

Cuando la ejecución del programa esté detenida en un breakpoint, el sistema debe permitir continuar la depuración utilizando la depuración algorítmica.

Para realizar esta tarea deberá ser capaz en primer lugar de crear el ET de la ejecución del programa desde el breakpoint hasta el final del método donde éste se encuentre, siendo capaz de almacenar la información necesaria para cada nodo del ET.

Una vez creado el ET, el sistema ha de ser capaz de realizar automáticamente las preguntas al usuario con el fin de encontrar el nodo erróneo. Por lo que también ha de ser capaz de procesar las respuestas proporcionadas por el usuario para hallar dicho nodo.

Además, ha de proveer de todas las interfaces gráficas al usuario, tanto como para realizar las dos tareas anteriormente escritas, como para la inspección del ET creado y de la información asociada a cada nodo de dicho ET.

- Depuración omnisciente

Al igual que en el punto anterior el sistema ha de tomar como punto de partida la ejecución del programa cuando ésta esté detenida en un breakpoint. A partir de este punto el sistema debe ser capaz de guardar la ejecución del programa hasta el final del método en el que se encuentre dicho breakpoint.

El sistema ha de ser capaz de almacenar toda la información referida a la modificación de las variables visibles para el ámbito de dicho breakpoint, tanto de las variables locales, como del contexto en el que se encuentre.

Además, el sistema debe de proveer al sistema de las herramientas necesarias para la visualización de la información almacenada en dicha traza. Permitiendo moverse a través de ella, visualizando el valor de las variables en cada momento de la traza, así como el instante en el que se han modificado dichas variables.

- Depuración híbrida

Para esta sección el sistema ha de ser capaz de integrar las funcionalidades anteriormente descritas con el fin de encontrar un error. Para esto, el sistema usará diversos mecanismos para la integración de estas técnicas.

Para ello, el primer requisito, ya descrito anteriormente, requiere que tanto la depuración algorítmica como la depuración omnisciente han de poder tomar como punto de inicio un breakpoint, por lo que dichos breakpoints han de ser comunes para todas las técnicas.

Cuando un error sea detectado mediante la técnica de la depuración algorítmica, éste ha de generar un brekpoint condicional para garantizar que, cuando se detenga el programa en la próxima ejecución del programa, sea en las mismas condiciones en las que se produjo el error. Después, mediante este breakpoint y si el usuario lo considera necesario, podrá ejecutar la técnica de depuración omnisciente.

Con esta funcionalidad se integran las tres técnicas anteriores lo que da lugar a la nueva técnica de depuración descrita en este trabajo.

5.5.3. Características de los usuarios

Los usuarios de este sistema han de ser desarrolladores familiarizados con la depuración de programas. No necesariamente han de conocer todas las técnicas descritas anteriormente, pero han de estar acostumbrados a la depuración de programas para encontrar errores. El sistema debe poseer una interfaz sencilla para que los desarrolladores que no conozcan alguna de estas técnicas o simplemente no estén acostumbrados a la depuración de programas sean capaces de usar el sistema de forma intuitiva.

5.5.4. Restricciones

El sistema ejecutará el programa del que se desea encontrar el error por lo que una restricción esencial es que éste pueda ser ejecutado. Además, las técnicas de depuración omnisciente y algorítmica podrán ser ejecutadas únicamente si el sistema se encuentra depurando un programa y parado en un breakpoint.

5.5.5. Suposiciones y dependencias

El sistema ha sido desarrollado para el entorno de desarrollo Eclipse, aunque este entorno es multiplataforma, la herramienta sólo ha sido testeada para versiones de 32 y 64 bits de Eclipse funcionando en una máquina Windows, por lo que no es posible asegurar su funcionamiento en el resto de sistemas operativos en los que puede ser instalado Eclipse.

5.5.6. Requisitos futuros

La principal mejora seria adaptar el plugin para más entornos de desarrollo (Visual Studio, NetBeans, etc.) ya que cuantos más entornos sean abarcados más popularidad alcanzará la nueva técnica de depuración propuesta con lo que su uso se extenderá.

5.6. Requisitos específicos

En esta sección se describen en detalle los requisitos del sistema, con un nivel de detalle suficiente como para permitir a los diseñadores diseñar un sistema que satisfaga todos estos requisitos, así como poder planificar y realizar pruebas que corroboren si el sistema cumple con los requisitos establecidos o no. Todos estos requisitos especifican comportamientos del sistema perceptibles por los usuarios y otros sistemas. Este punto es el más importante de esta sección por lo que se aplicarán los siguientes principios.

- Este punto debe de ser perfectamente legible por personas de muy distintas formaciones e intereses.
- Deben referenciarse aquellos documentos relevantes que poseen alguna influencia sobre los requisitos.
- Todo requisito debe ser unívocamente identificable mediante algún código o sistema de numeración adecuado.
- Lo ideal, aunque en la práctica no siempre realizable, es que los requisitos posean las siguientes características:
    - ***Corrección***: La ERS es correcta si y sólo si todo requisito que figura aquí (y que será implementado en el sistema) refleja alguna necesidad real. La corrección de la ERS implica que el sistema implementado será el sistema deseado.
    - ***No ambiguos***: Cada requisito tiene una sola interpretación. Para eliminar la ambigüedad inherente a los requisitos expresados en

lenguaje natural, se deberán utilizar gráficos o notaciones formales. En el caso de utilizar términos que, habitualmente, poseen más de una interpretación, se definirán con precisión en el glosario.

- ***Completos***: Todos los requisitos relevantes han sido incluidos en la ERS. Conviene incluir todas las posibles respuestas del sistema, a los datos de entrada, tanto válidos como no válidos.
- ***Consistentes***: Los requisitos no pueden ser contradictorios. Un conjunto de requisitos contradictorio no es implementable.
- ***Clasificados***: Normalmente, no todos los requisitos son igual de importantes. Los requisitos pueden clasificarse por importancia (esenciales, condicionales u opcionales) o por estabilidad (cambios que se espera que afecten al requisito). Esto sirve, ante todo, para no emplear excesivos recursos en implementar requisitos no esenciales.
- ***Verificables***: La ERS es verificable si y sólo si todos sus requisitosson verificables. Un requisito es verificable (comprobable) si existe unproceso finito y no costoso para demostrar que el sistema cumplecon el requisito. Un requisito ambiguo no es, en general, verificable. Expresiones como a veces, bien, adecuado, etc. Introducen ambigüedad en los requisitos. Requisitos como “en caso de accidente la nube tóxica no se extenderá más allá de 25Km" no es verificable por el alto costo que conlleva.
- ***Modificables***: La ERS es modificable si y sólo si se encuentra estructurada de forma que los cambios a los requisitos pueden realizarse de forma fácil, completa y consistente. La utilización de herramientas automáticas de gestión de requisitos (por ejemplo *RequisitePro* o *Doors*) facilitan enormemente esta tarea.
- ***Trazables***: La ERS es trazable si se conoce el origen de cada requisito y se facilita la referencia de cada requisito a los componentes del diseño y de la implementación. La trazabilidad hacia atrás indica el origen (documento, persona, etc.) de cada requisito. La trazabilidad hacia delante de un requisito R indica qué componentes del sistema son los que realizan el requisito R.

5.6.1. Depuración por trazas

RF1. El programa ha de ser capaz de ejecutar un programa en modo de depuración.

RF2. Durante la ejecución de un programa en modo de depuración el programa deberá detenerse en los breakpoints.

RF3. Debe permitir la inserción de un breakpoint en una línea de código determinada.

RF4. Debe permitir la creación de breakpoints asociados a métodos, además debe permitir indicar si este breakpoint detendrá la ejecución del programa cuando este método sea llamado o cuando finalice.

RF5. Ha de ser posible insertar breakpoints de exepciones, los cuales sólo suspenderán la ejecución del programa cuando ocurra la excepción indicada en el mismo.

RF6. Ha de ser posible insertar condiciones a los breakpoints, indicando además si la ejecución del programa se suspenderá cuando ésta se cumpla o por el contrario si la ejecución del programa no se suspenderá cuando se cumpla la condición.

RF7. El sistema debe permitir la detención de la ejecución en modo depuración cuando el usuario lo crea conveniente.

RF8. Cuando la ejecución del programa este suspendida en un breakpoint, debe permitir al usuario continuar con la misma.

RF9. Cuando la ejecución del programa esté suspendida, el programa deberá permitir la inspección del contexto de la ejecución en el momento en el que el programa ha quedado suspendido.

RF10. El programa debe permitir realizar un paso hacia delante (*stepover*).

RF11. El programa debe permitir realizar un paso hacia el interior de un método (*stepinto*).

5.6.2. Depuración algorítmica

RF1. Cuando el programa esté detenido en un breakpoint, debe permitir iniciar la depuración algorítmica iniciando así la creación del árbol de ejecución. Cuando finalice la ejecución del método, el programa ha de finalizar la

creación del árbol por lo que el árbol sólo debe abarcar la ejecución del programa hasta la salida de dicho método.

RF2. El programa debe generar un árbol de ejecución, el cual ha de ser correcta acorde con la ejecución del programa realizada.

RF3. El programa debe permitir indicar si un nodo es correcto, incorrecto o si el usuario desconoce este dato pero confía en que el método sea correcto.

RF4. El programa ha de permitir iniciar la depuración guiada de un árbol basada en la técnica de depuración algorítmica seleccionada por el usuario con el fin de encontrar el nodo erróneo.

RF5. Una vez encontrado el nodo erróneo, el programa ha de posicionar un breakpoint condicional, el cual debe incluir los argumentos de la llamada al método asociado al nodo erróneo.

5.6.3. Depuración omnisciente

RF1. Cuando la ejecución de un programa esté detenida en un breakpoint, deberá ser posible iniciar la depuración omnisciente, almacenando toda la información de la traza para el método en el que se encuentre dicho breakpoint.

RF2. El programa debe permitir al usuario realizar pasos hacia delante sobre la traza guardada (stepover).

RF3. El programa deberá permitir al usuario realizar pasos hasta el siguiente breakpoint o la finalización del método (longstepover)

RF4. El programa debe permitir al usuario realizar pasos hacia detrás sobre la ejecución del programa (step back)

RF5. El programa debe permitir al usuario realizar pasos hacia detrás hasta el breakpoint anterior o hasta el inicio de la traza generada por la depuración omnisciente (longstep back).

6. Arquitectura

Esta sección describe la arquitectura interna de HDJ, y describe sus características principales. HDJ es un plugin de Eclipse que toma las ventajas de las capacidades de depuración ya implementadas en Eclipse (por ejemplo, usamos su depurador por trazas) y adapta el existente DDJ [**9**] al entorno de trabajo de Eclipse. La integración de HDJ en Eclipse está descrita en la FIG. 8.

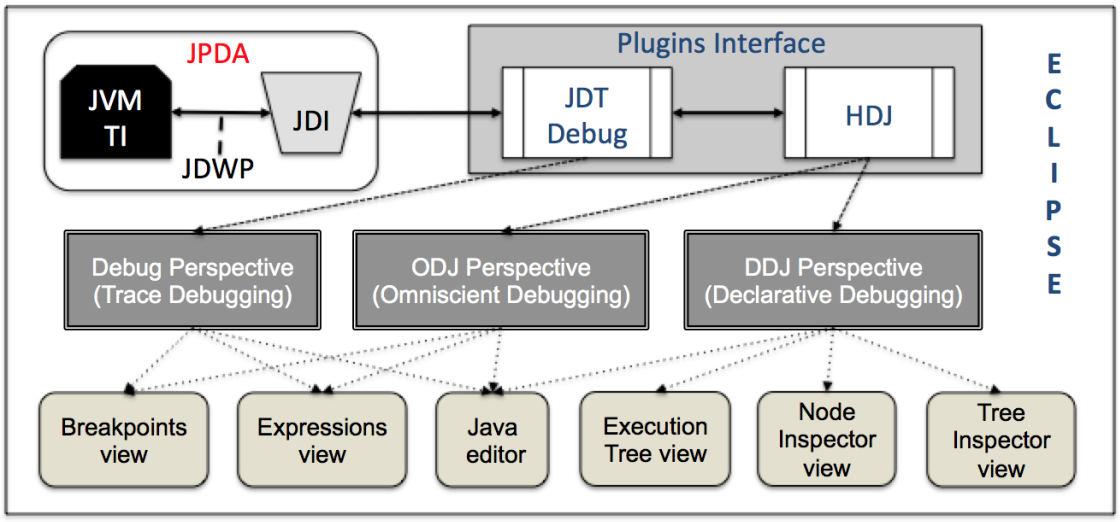


Fig. 8 Integración de HDJ en Eclipse

Uno de los retos más importantes cuando se han integrado dos nuevos depuradores en Eclipse ha sido permitir a cualquiera de ellos depurar el mismo programa (por ejemplo, dotar al programador de libertad para cambiar de un depurador a otro en la misma sesión). Para esto, todos ellos han de tener acceso al mismo código fuente objetivo (por ejemplo, un breakpoint en el código fuente objetivo debe estar compartido por todos los depuradores), y usar la misma máquina virtual de Java objetivo y el mismo control sobre la ejecución de esta máquina virtual de Java. En la FIG. 8 la máquina virtual objetivo, común para todos los depuradores, está representada con una caja negra. La interfaz de herramientas de la máquina virtual de Java (JVM TI) provee tanto de una forma de inspeccionar el estado de la máquina como de tomar el control de la ejecución de la máquina virtual de Java objetivo. Los depuradores acceden a ésta mediante la Java Debug Interface (JDI) cuyas comunicaciones están regidas por el Java Debug Wire Protocol (JDWP). Esta pequeña arquitectura para controlar la máquina virtual se llama Java Platform Debugger Architecture (JPDA) [**10**].

Uno de los depuradores usados, en concreto el depurador por trazas, ya está implementado por Eclipse en un plugin llamado JDT Debug. Los otros dos depuradores han sido implementados para este trabajo en el plugin HDJ. La herramienta permite al programador moverse entre tres perspectivas:

- **Debug.** Esta perspectiva nos permite manejar la depuración por trazas. Ésta es la perspectiva estándar de Eclipse para la depuración. Está

compuesta por un gran número de vistas y editores y nos ofrece una gran funcionalidad que incluye el uso de diversos tipos de breakpoints así como control sobre la ejecución del programa.

- **ODJ.** Esta perspectiva nos permite manejar la OD. Contiene las mismas vistas y editores que la perspectiva estándar de depuración. Además, incluso si el programador está usando un depurador distinto con un mecanismo de depuración completamente distinto, la IGU es exactamente la misma, e incluso, las diferencias internas son transparentes para él. La única diferencia es que ODJ permite explorar la ejecución del programa hacia atrás. Internamente, éste utiliza una traza de la ejecución como la descrita en Fig. 1 para generar los estados que serán explorados por el programador.
- **DDJ.** Esta perspectiva nos permite controlar la AD. Un ejemplo de uso de esta perspectiva está representado en la FIG. 9. En la figura podemos ver dos de sus tres vistas y un editor. Primero, situada a la izquierda podemos ver la *ET view*, que contiene el ET y las cuestiones generadas por el depurador. En segundo lugar, en la parte derecha, podemos encontrar el *Node inspector*, el cual nos muestra toda la información asociada al nodo seleccionado del ET. Esto incluye el contexto inicial, la llamada al método y el contexto final, donde los cambios están resaltados en rojo. Por último, abajo podemos ver el editor de Java, que contiene el código fuente y los breakpoints. Este editor está compartido entre los tres depuradores, y además, todos ellos manipulan el mismo código fuente y ofrecen los mismos breakpoints al programador.

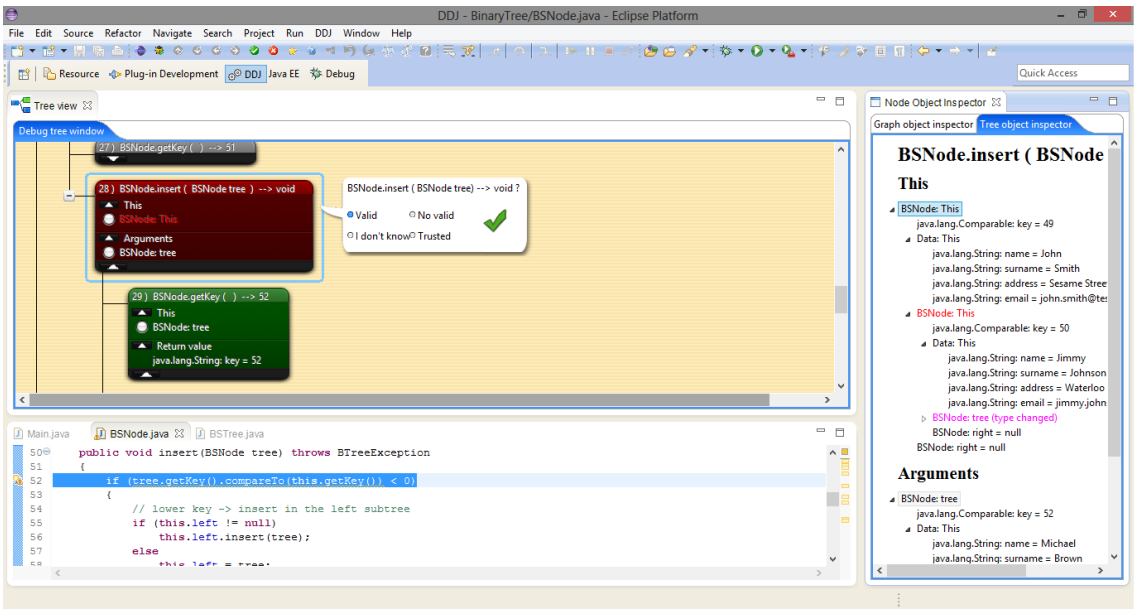


Fig. 9 Captura de pantalla del plugin HDJ (perseptiva DDJ)

7. Implementación

HDJ ha sido implementado completamente en Java. Contiene sobre 30000 líneas de código, 19000 de éstas corresponden a la implementación de la depuración algorítmica (el funcionamiento interno del depurador algorítmico ha sido adaptado del depurador DDJ [**9**] con algunas extensiones que incluyen la comunicación con JPDA mediante el depurador JDT, y la perspectiva IGU), 7500 líneas de código corresponden a la implementación del depurador omnisciente el cual ha sido implementado desde cero, y 3500 líneas de código corresponden con la implementación del propio plugin y su integración y comunicación con Eclipse. El depurador puede hacer uso de una base de datos para almacenar la información del ET y de la traza usada en OD (si la base de datos no está activada, el ET y la traza son guardados en la memoria principal). Gracias a JDBC, HDJ puede interactuar con distintas bases de datos. La distribución actual incluye bases de datos MySQL y Access. Todas las funcionalidades descritas en este trabajo están completamente implementadas en la última versión estable. Esta versión es abierta y públicamente accesible en:

http://www.dsic.upv.es/~jsilva/HDJ/

En este sitio web el usuario puede encontrar información sobre los pasos de instalación, ejemplos, videos demostrativos y otro material útil.

8. Trabajos relacionados

Mientras que un depurador por trazas está siempre presente en los entornos de depuración modernos, los depuradores algorítmicos y omniscientes son muy inusuales debido a sus problemas de escalabilidad ya discutidos en este trabajo. No obstante, existen algunos intentos de implementación de depuradores algorítmicos como JDD [**11**] y su más reciente re-implementación DDJ [**9**]. Existen otros depuradores que incorporan aspectos de la depuración como el plugin de Eclipse JavaDD [**12**] o el depurador de Oracle JDeveloper's declarative debugger [**13**]. Sin embargo, éstos no son capaces de producir cuestiones automáticamente y controlar la búsqueda automática para encontrar el error. Esto significa que no usan las estrategias comunes de AD implementadas en depuradores algorítmicos estándar de lenguajes declarativos como Haskell (Hat delta [**14**]) o Toy (DDT [**15**]). Ninguno de estos depuradores puede trabajar con breakpoints como el depurador descrito en este trabajo.

La situación es similar en el caso de los depuradores omniscientes. Por lo conocido por mi hasta el momento, OmniCoreCodeGuide [**16**]es el único entorno de desarrollo para Java que incluye un depurador omnisciente. Sin embargo, para abordar el problema de escalabilidad, este depurador utiliza una traza limitada a algunos miles de los últimos eventos. Existen algunas implementaciones ad-hoc que pueden trabajar individualmente o ser integradas en entornos comerciales [**17**] [**18**] [**6**] [**19**] [**20**]. Además, todos estos trabajos están orientados a hacer OD más escalable [**20**] [**18**]. Por ejemplo, reduciendo el coste de la captura de la traza, así como la cantidad de información a almacenar sobre ésta, usando trazas parciales que excluyen ciertas clases verificadas en el proceso de generación de la traza [**6**]. Otros trabajos intentan mejorar OD, por ejemplo, con enlaces de causalidad [**19**] que proveen la capacidad de saltar desde el punto donde un valor es observado en una variable hasta el punto en el pasado cuando el valor fue asignado a dicha variable. Esto puede ser ciertamente valorable para resolver el conjunto de causas y efectos que están relacionados con un error.

Ha habido numerosos intentos de producir un depurador híbrido que combine diferentes técnicas. El depurador ODB [**6**] combina TD con OD. Éste permite al usuario depurar un programa usando TD y empezar a grabar la ejecución para OD cuando sea requerido por el usuario. El depurador de Kouth et al [**21**] combina AD con TD. Una vez el depurador algorítmico ha encontrado el método erróneo, continúa la búsqueda con TD para explorar el método (hacia delante) paso a paso. Esta idea sin embargo está presente en el depurador implementado en este trabajo, sin embargo en este trabajo se usa OD en vez de TD, y por lo tanto nosotros permitimos además ir hacia atrás en la ejecución del método. El depurador JIVE [**22**] combina TD, OD y dynamic slicing. Éste no usa AD, pero permite al programador realizar preguntas a la traza.

JHyde [**23**] es la única técnica previa que combina TD, OD y AD. Desafortunadamente, no he sido capaz de probar esta herramienta (no está accesible públicamente); pero considerando su arquitectura, es altamente

probable que sufra los mismos problemas de escalabilidad que el resto de los depuradores omniscientes (los autores no especifican ninguna medida del tiempo o el espacio necesario para almacenar la ejecución del programa). Al contrario que la solución propuesta en este trabajo, su arquitectura está basada en transformación de programas, esto almacena la ejecución del código en un archivo como efecto secundario. Primero, esta instrumentalización y ejecución de la traza normalmente toma mucho tiempo con programas industriales, por lo que el programador ha de esperar a la instrumentalización antes de depurar. En segundo lugar, el depurador almacena la traza de todo el programa, mientras que el depurador propuesto aquí sólo almacena la traza de un único método. El punto común que ambos están implementados como un plugin para Eclipse, y ambos usan la misma estructura de datos para OD y AD. Esto es importante para reutilizar la información de rastreo recopilada por el depurador. Otra importante característica implementada por ambas técnicas es el uso de un vocabulario de colores usado en las vistas. Esto es muy útil para permitir al programador detectar rápidamente los cambios en el estado de la ejecución.

9. Conclusiones

La depuración por trazas, la depuración algorítmica y la depuración omnisciente son tres de las técnicas de depuración más importantes. Algunas de ellas se ajustan mejor a ciertos tipos específicos de problemas, mientras que para otros tipos otra técnica puede ser mejor. Además, es posible que una técnica sea deseable para depurar una parte del programa, mientras que otra técnica sea más adecuada para depurar otra parte del mismo programa. Por esta razón, en cualquier entorno de desarrollo las tres técnicas han de ser accesibles.

En este trabajo se introduce un nuevo depurador llamado HDJ que implementa e integra tres técnicas. La implementación utiliza una nueva arquitectura de depuración que permite a las tres técnicas compartir la misma máquina virtual objetivo y el mismo código objetivo. Esto permite cambiar de una técnica a otra en una misma sesión de depuración. Además, se presenta un nuevo modelo de depuración que combina las tres técnicas. La nueva arquitectura de depuración expuesta en este trabajo es particularmente interesante ya que explota las mejores propiedades de cada técnica (por ejemplo, alta precisión, nivel alto de abstracción, etc.) y esto minimiza problemas como la escalabilidad. HDJ es abierto y libremente distribuido como un plugin de Eclipse.

10. Manual de usuario del depurador

En esta sección describiremos el funcionamiento de los distintos componentes desarrollados para el uso del depurador híbrido. Comenzaremos con una guía de instalación del plugin para Eclipse, seguida de una explicación de la composición del mismo dividida en dos partes: depuración algorítmica y depuración omnisciente. Además, concluiremos con una sesión de depuración de ejemplo. En esta sección se ha omitido el manual sobre la depuración por trazas, ya que en este trabajo se ha utilizado el depurador por trazas de la plataforma Eclipse sobre el cual se puede encontrar toda la documentación en el propio sitio web de Eclipse [**24**].

10.1. Instalación del plugin

Para instalar el plugin desarrollado en este trabajo deberemos dirigirnos a la opción “Install new software” situada dentro del menú “Help” de la plataforma Eclipse como se indica en laFIG. 10. Una vez allí, deberemos introducir el repositorio donde se encuentra disponible la herramienta como se indica en laFIG. 11, el cual es: http://www.dsic.upv.es/~jsilva/HDJ/repository. Una vez hecho esto, deberemos asegurarnos que la opción para agrupar los elementos por categoría está deshabilitada tal y como se indica en la FIG. 11. Por último, sólo debemos seguir los pasos indicados para finalizar la instalación de la herramienta.

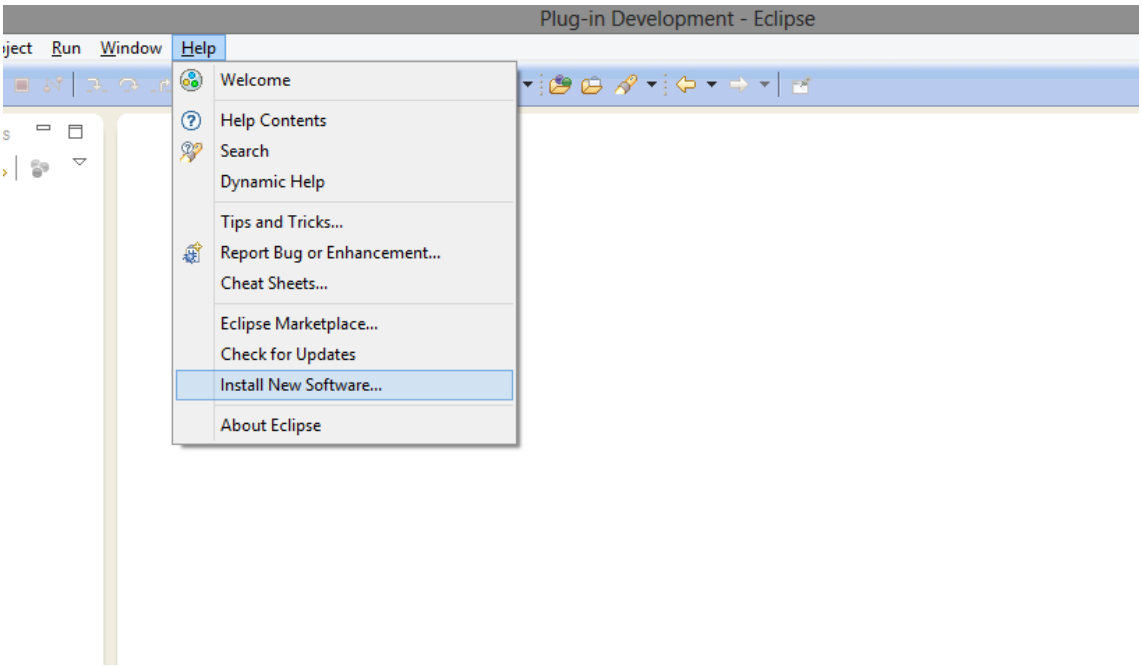


Fig. 10 "Install new software" Eclipse

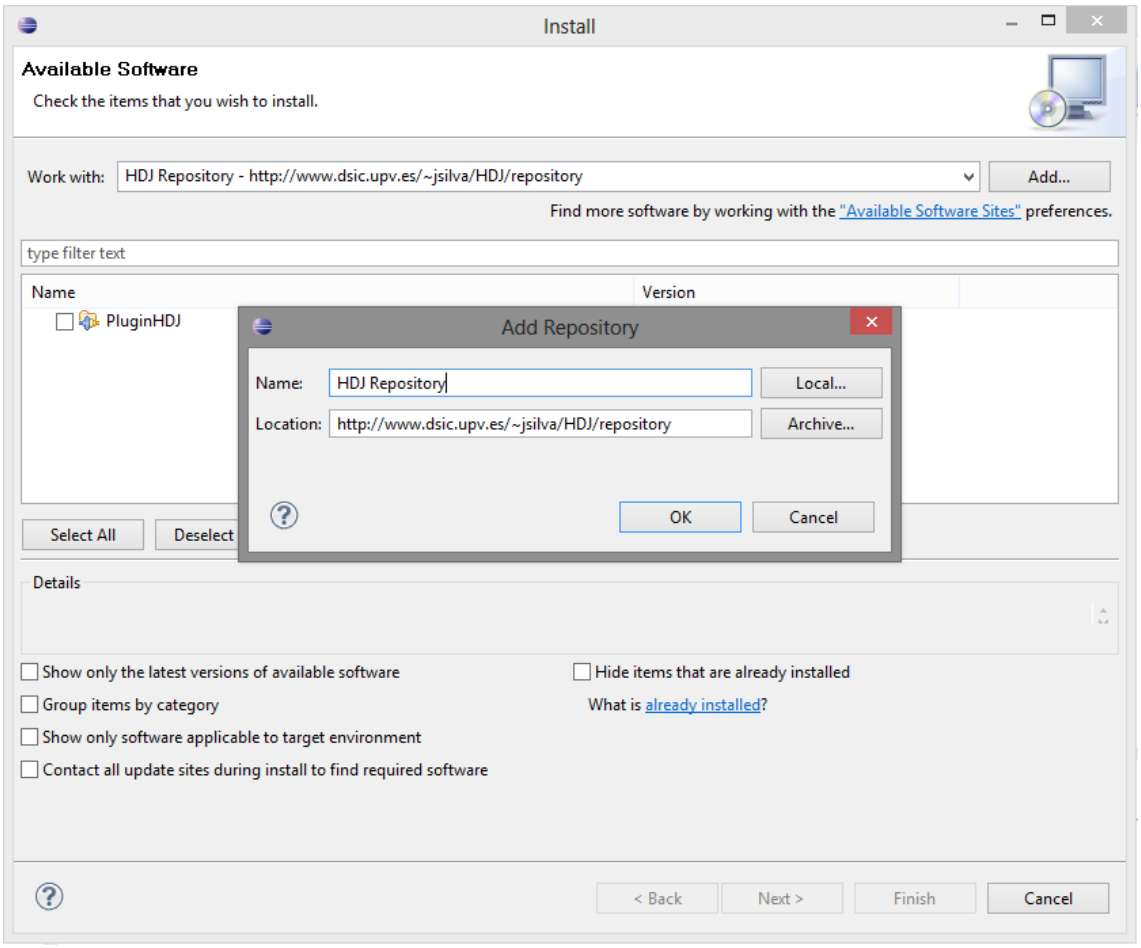


Fig. 11 Añadir repositorio

10.2. Componentes del plugin

Como se ha descrito anteriormente en la sección 10, esta sección está dividida en dos partes. Para cada una de ellas introducimos al usuario en la perspectiva creada para cada entorno de depuración (Algorítmico y Omnisciente), continuaremos con una explicación más detallada de cada uno de los componentes que conforman dichas perspectivas, tanto las vistas asociadas a cada una de ellas como los menús y las acciones que realizan cada una de las opciones asociadas a estos.

10.2.1. Depuración algorítmica

10.2.1.1. Perspectiva

En la Fig. 12 podemos observar la perspectiva creada, para dar soporte al usuario cuando éste decida realizar una depuración algorítmica. En esta perspectiva encontramos agrupados los componentes para realizar la depuración algorítmica, en esta sección describimos la situación de los

mismos. En una posterior sección estos componentes serán detallados en profundidad, con la salvedad del editor de Java ya que es propio de eclipse y su único objetivo es mostrar el código de los métodos asociados a los nodos.

Como podemos ver en la FIG. 12, en esta perspectiva encontramos los siguientes componentes:

- **El editor de Java:** está situado en la parte inferior izquierda, este editor es propio de la plataforma Eclipse y en esta perspectiva es usado para visualizar el código asociado a los nodos del árbol de ejecución.
- **La vista del árbol de ejecución:** se trata de la vista situada en la parte superior izquierda. Como podemos observar, es la vista que mayor tamaño ocupa en esta perspectiva, ya que es la encargada de mostrar al usuario el árbol de depuración generado y mostrar sobre él las preguntas.
- **La barra de herramientas de la perspectiva:** está situado en la barra de menús de las perspectivas, situada ésta a su vez en la parte superior de todas las vistas. En ella encontraremos los botones necesarios tanto como para la activación de la depuración algorítmica, así como para la depuración del ET mediante preguntas al usuario una vez generado éste.
- **Menú de la perspectiva:** está situado en la parte superior del entorno de desarrollo y se trata del menú existente en la gran mayoría de las aplicaciones (por ejemplo MS Word). Mediante este menú tendremos acceso a las distintas estrategias de depuración algorítmica para que usuario pueda elegir cualquiera de ellas.

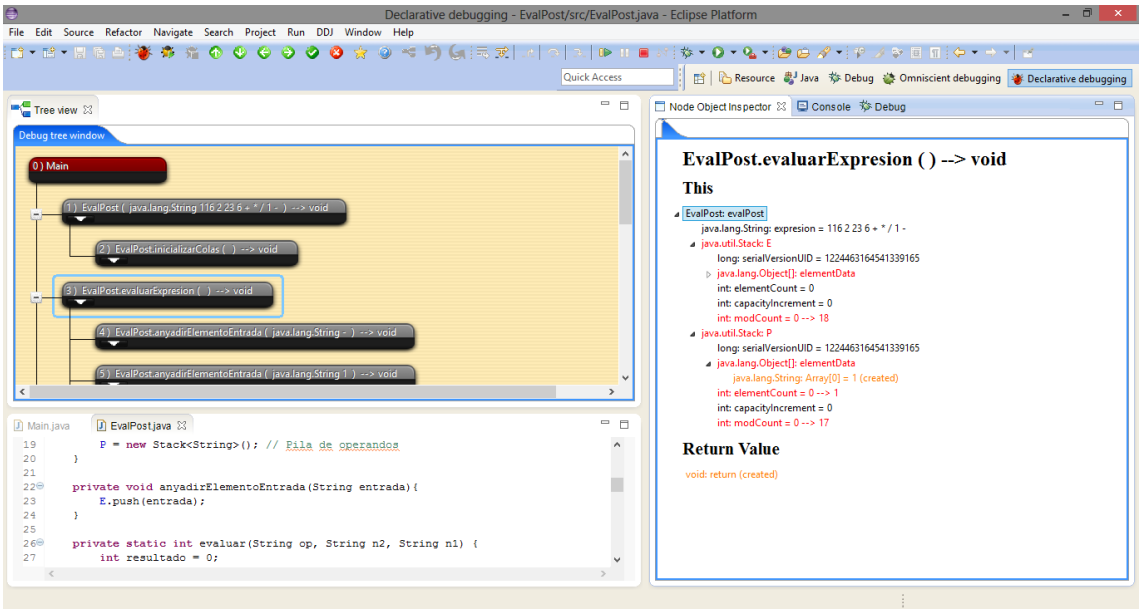


Fig. 12 Perspectiva de depuración algorítmica

10.2.1.2. Vista del árbol de depuración

Ésta es la vista más importante de la perspectiva de depuración algorítmica, ya que es la encargada de mostrar el árbol de ejecución como podemos comprobar en la FIG. 13.

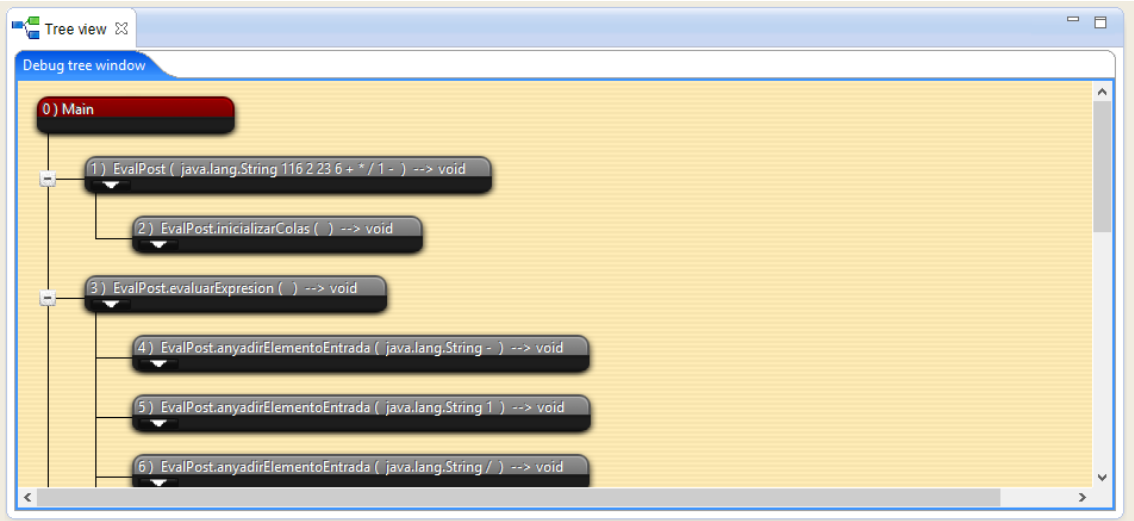


Fig. 13 Vista del árbol de ejecución

Como podemos ver en la FIG. 14, esta vista permite también la posibilidad de desplegar los nodos mostrando al usuario información referente al estado de la máquina al inicio y al final de la ejecución del método asociado con dicho nodo. Para ello el usuario solo ha de hacer clic sobre la flecha situada debajo del texto que indica el método asociado. Además nos permite colapsar y expandir los nodos del árbol mostrando u ocultando los hijos de dicho nodo, pulsando los botones con un signo menos o más dependiendo del caso.

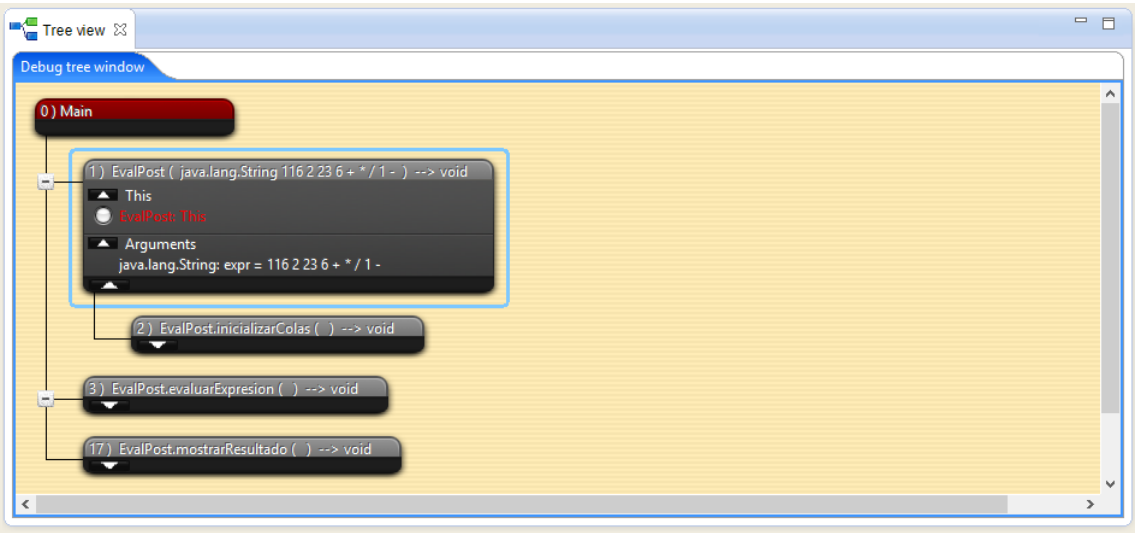


Fig. 14 Operaciones sobre el árbol de ejecución

Además esta vista es la encargada de guiar al usuario por la depuración algorítmica del árbol, mostrando sobre el propio árbol las preguntas sobre los nodos y permitiendo además que el usuario conteste a las mismas. Para tener una idea de cómo se realizan dichas preguntas podemos observar la FIG. 15.

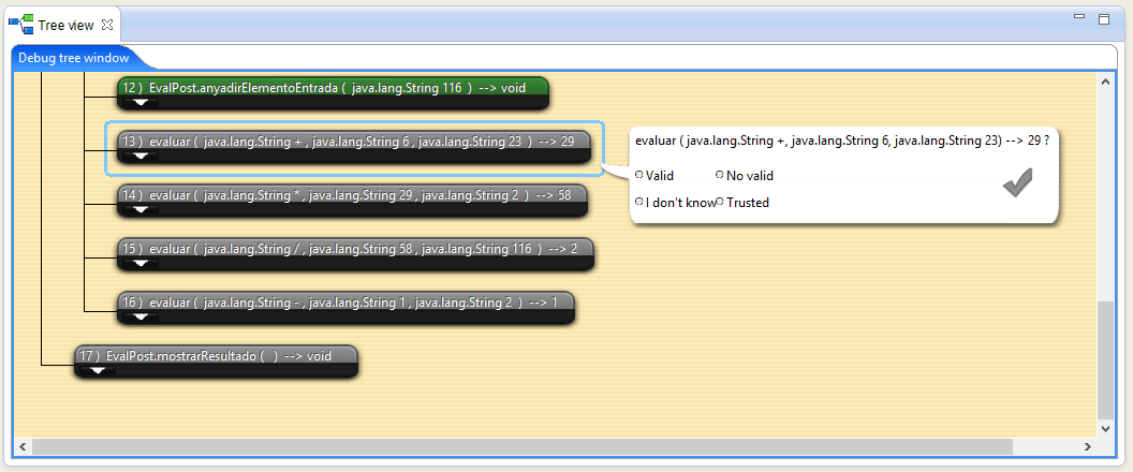


Fig. 15 Depuración del árbol de ejecución

10.2.1.3. Vista de inspección del nodo

En esta vista el usuario puede visualizar toda la información referente al estado de la máquina en el nodo seleccionado, tanto en la invocación de dicho método como en el retorno del mismo. Como podemos observar en la FIG. 16, en esta vista se muestra la cabecera del nodo seleccionado, y las variables asociadas a éste. Como podemos ver, las variables están divididas en: variables del objeto (siempre que el método no sea estático), argumentos y la variable de retorno. Para representar dichas variables esta vista sigue un código de colores para representar la variación del estado de éstas desde el inicio hasta el fin de la ejecución del método: rosa para las variables que han cambiado de tipo, naranja para las variables que han sido creadas, rojo para las variables que han sido modificadas y negro para las variables que no han sufrido ninguna alteración. Además, cuando una variable ha sido modificada se muestra el nuevo valor a continuación del inicial precedido por una flecha.

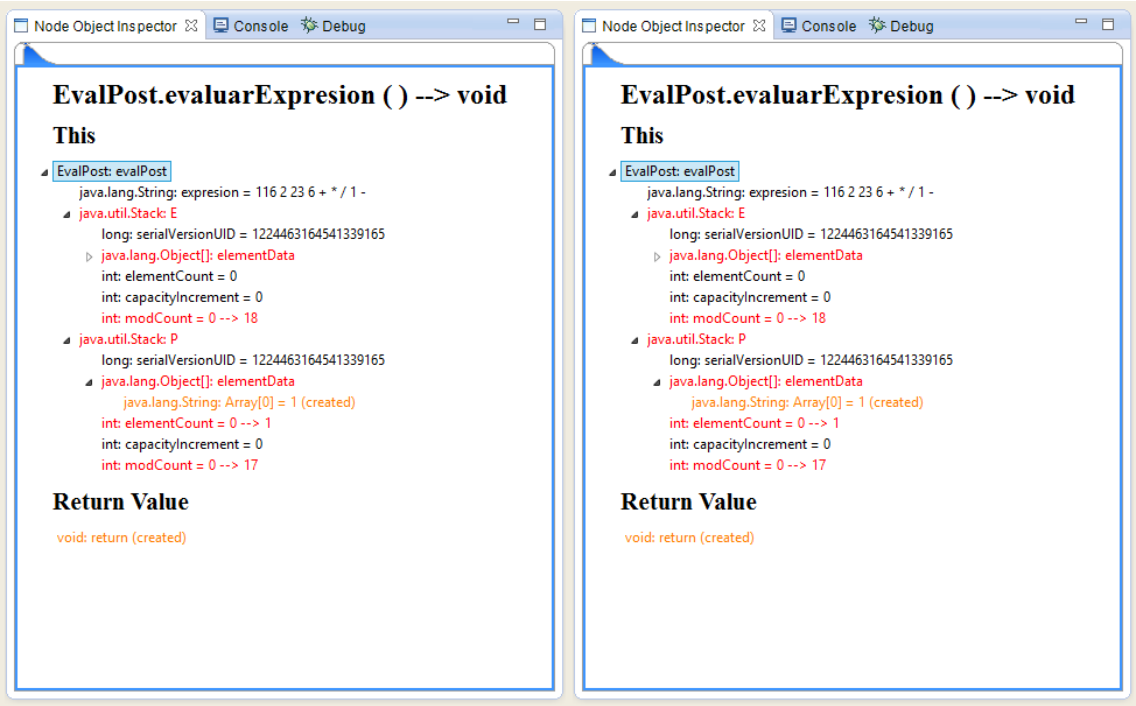


Fig. 16 Vista de inspección del nodo

10.2.1.4. Barra de herramientas de la perspectiva

Se trata de la barra de herramientas indicada en la FIG. 17, la cual permite al usuario controlar la depuración algorítmica del árbol de ejecución.

Fig. 17 Barra de herramientas depuración algorítmica

A continuación describiremos la funcionalidad de los botones mostrados en la barra de herramientas:


Inicia la depuración algorítmica, para ello es necesario que la ejecución del programa se encuentre detenida en un método. Una vez pulsado el depurador continúa con la ejecución del programa y genera el árbol de ejecución asociado a la ejecución del resto del método.


Inicia la depuración guiada del árbol de ejecución, realizando preguntas al usuario sobre los nodos elegidos mediante una estrategia de depuración algorítmica.


Detiene la depuración guiada del árbol de ejecución.


Selecciona el nodo anterior, si el nodo actual tiene hermanos y no es el primer nodo hijo, selecciona el anterior hermano, de lo contrario selecciona el nodo padre.

 Selecciona el siguiente nodo, si el nodo actual tiene hijos selecciona el primero de ellos, de lo contrario selecciona el siguiente nodo hermano.

 Selecciona el nodo del nivel anterior, es decir, el padre del nodo seleccionado actualmente.

 Selecciona el nodo del siguiente nivel, es decir, el primer hijodel nodo seleccionado actualmente.

 Marca el nodo seleccionado como valido.

 Marca el nodo seleccionado como incorrecto.

 Marca el nodo seleccionado como de confianza, por lo que todos los nodos que contengan ejecuciones del método asociado a este nodo serán marcados como válidos (incluido el nodo seleccionado).

 Marca el nodo seleccionado como desconocido.

 Reinicia el estado de todos los nodos del árbol de ejecución.

 Deshace el último paso realizado.

 Rehace el último paso deshecho.

10.2.1.5. Menú de depuración algorítmica

Como podemos observar en la FIG. 18, este menú permite al usuario seleccionar una de las estrategias de depuración algorítmica desarrolladas. Esta estrategia determinará el orden de las preguntas hechas al usuario sobre el árbol de depuración.

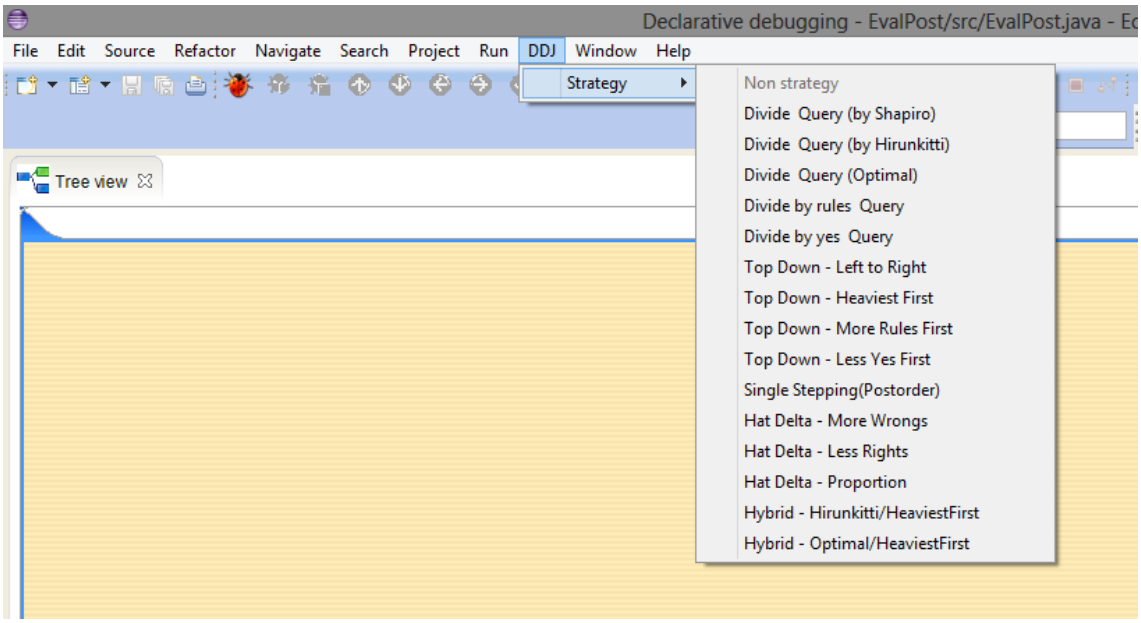


Fig. 18 Menú de depuración algorítmica

10.2.2. Depuración omnisciente

10.2.2.1. Perspectiva de depuración omnisciente

En la FIG. 19 se muestra la perspectiva creada para la depuración omnisciente. Como podemos observar, consta de dos partes de igual tamaño que dividen en área de depuración, ya que ambas son igual de importantes para realizar la depuración omnisciente. Se tratan del editor de Java y de la vista de inspección omnisciente que serán detallados a continuación.

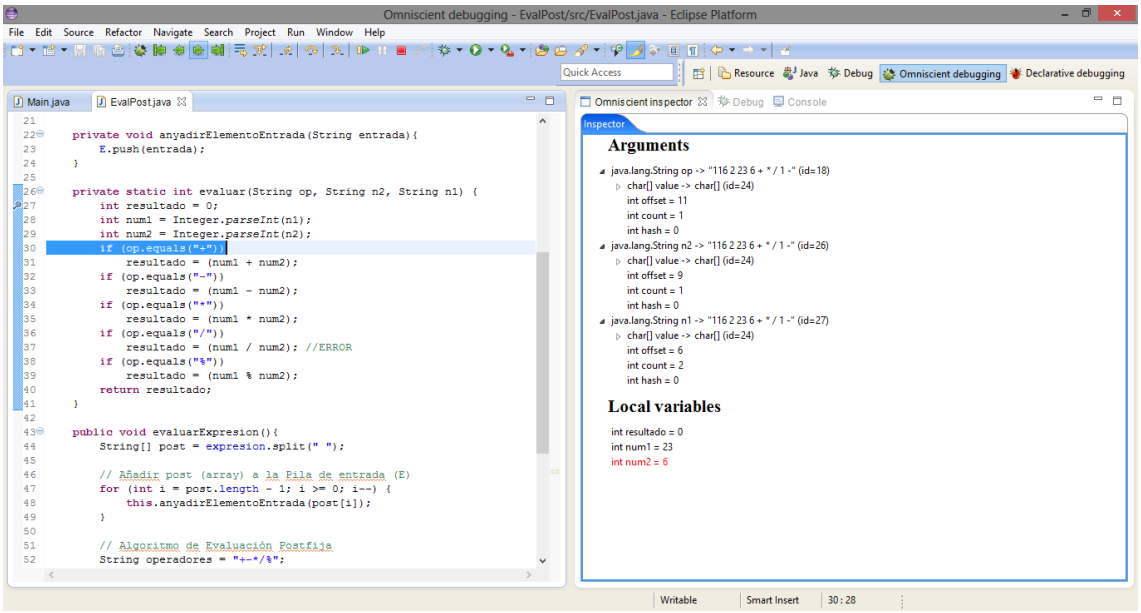


Fig. 19 Perspectiva de depuración omnisciente

10.2.2.2. El editor de Java

Como se ha mencionado con anterioridad, este editor es propio de la plataforma Eclipse. En el caso de la depuración omnisciente, este editor es usado para indicar al usuario en que línea de código se encuentra en cada instante de tiempo. Cuando el usuario selecciona un instante de tiempo, la línea de código asociada a éste se colorea (como podemos ver en la FIG. 20) para que el usuario sepa de cual se trata.

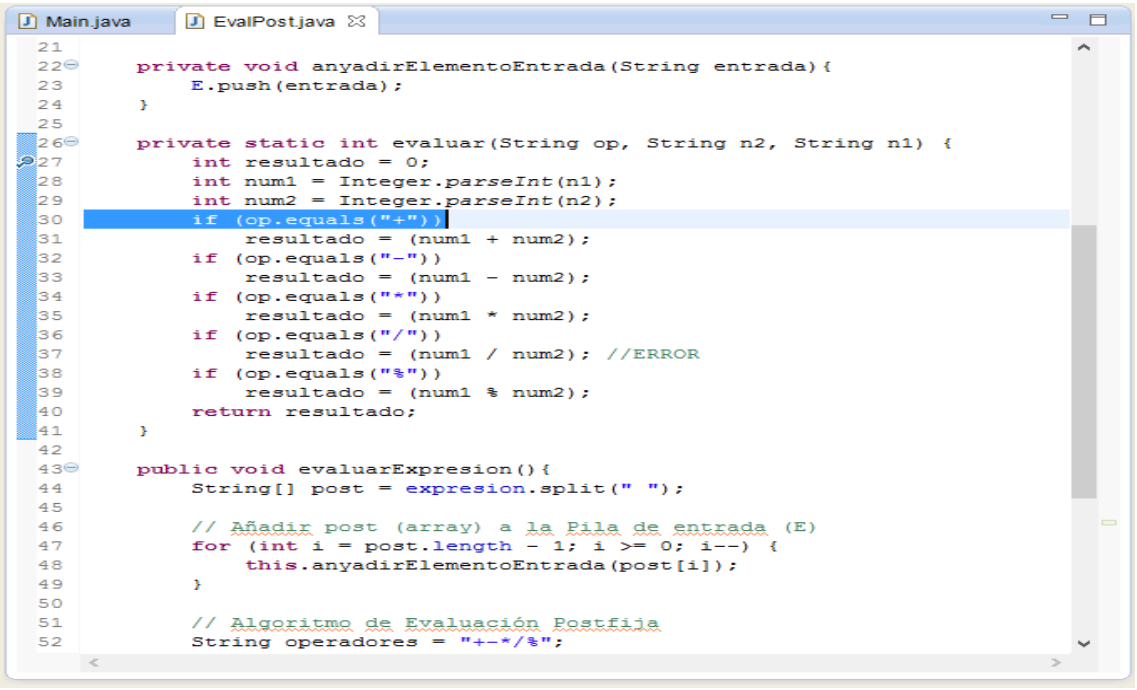


Fig. 20 El editor de Java para la depuración omnisciente

10.2.2.3. La vista del inspector omnisciente

Ésta es la vista que permite al usuario inspeccionar el estado de la máquina para cada instante de tiempo grabado en la traza. Como se puede observar en la FIG. 21,esta vista también sigue un código de colores, mostrando de color negro las variables que no han sido modificadas desde el instante de tiempo donde se encontraba la ejecución hasta el instante de tiempo actual, y en rojo las que sí han sido modificadas.

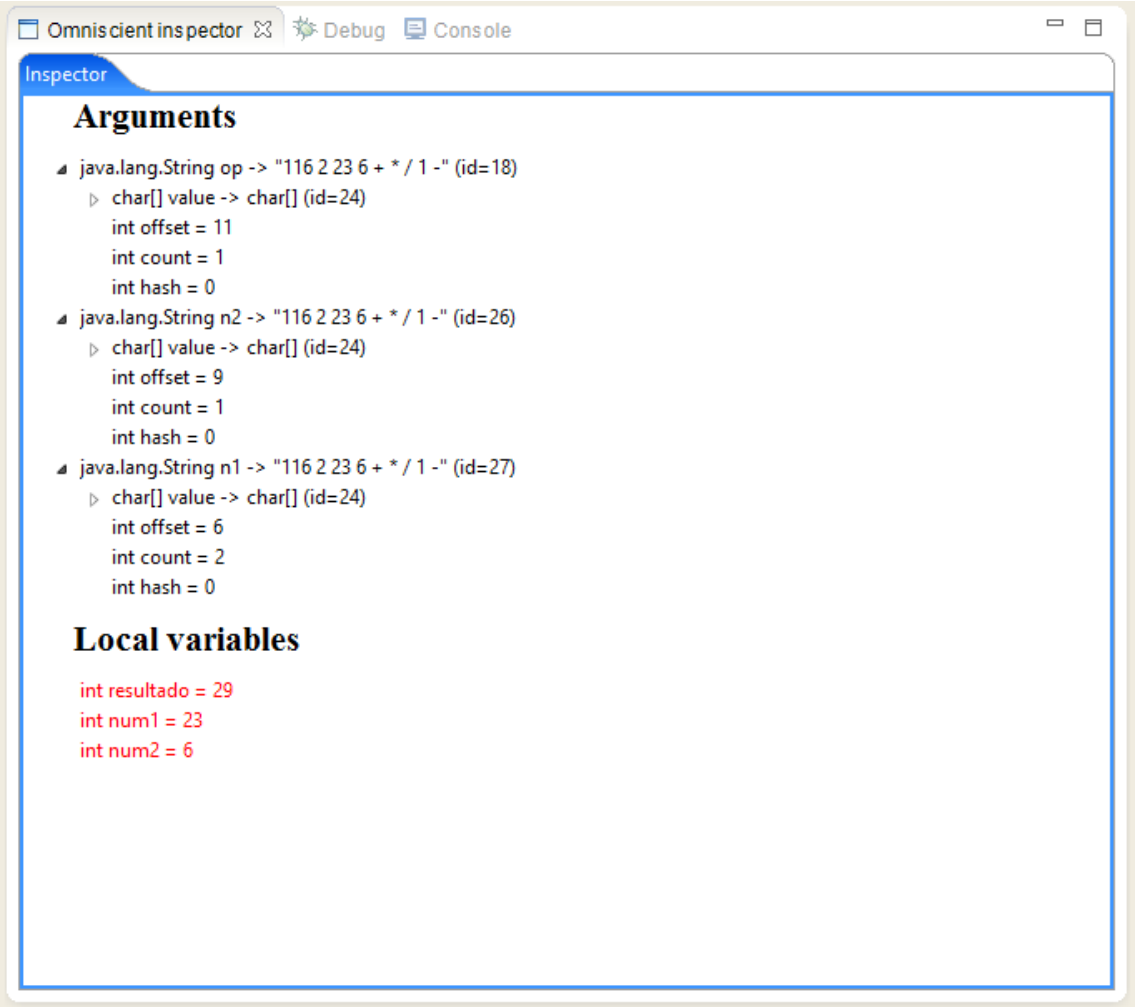


Fig. 21 Inspector omnisciente

10.2.2.4. Barra de herramientas de la perspectiva

Se trata de la barra de herramientas mostrada en la FIG. 22, la cual permite al usuario tanto iniciar la captura de la traza del método, como moverse hacia atrás y hacia delante a través de dicha traza.

Fig. 22 Barra de herramientas de la depuración omnisciente

A continuación describiremos la funcionalidad de los botones mostrados en la FIG. 22:


Inicia la depuración omnisciente de un método cuando la ejecución se encuentra parada dentro del mismo.


Navega hacia atrás por la traza almacenada hasta que encuentra el breakpoint anterior, si no existen breakpoints se sitúa al inicio de la traza.

 Realiza un paso hacia atrás sobre la traza almacenada.

 Realiza un paso hacia delante sobre la traza almacenada.

 Navega hacia delante a través de la traza almacenada hasta el siguiente brekpoint, de no existir ningún breakpoint se sitúa al final de la traza.

10.2.3. Caso de uso: sesión de depuración algorítmica

En esta sección mostramos un caso de uso de la herramienta desarrollada a través de una sesión de depuración hibrida. En esta sección se usa el código de la FIG. 2, el cual ya se usó en la sección 3 de este trabajo, para explicar de una forma más teórica una sesión de depuración hibrida. A continuación realizamos de nuevo esta sesión de una manera más práctica a través de la herramienta desarrollada.

En primer lugar indicar que el error en este programa se encuentra en el método *mark(char player, int row, int col)* en la línea de código *board[col][row] = player.* Inicialmente el usuario coloca un breakpoint, gracias a su conocimiento acerca del error, en el método cuya ejecución provoca el error. Después el usuario activa la depuración por trazas y la ejecución continua hasta que se alcanza el breakpoint.

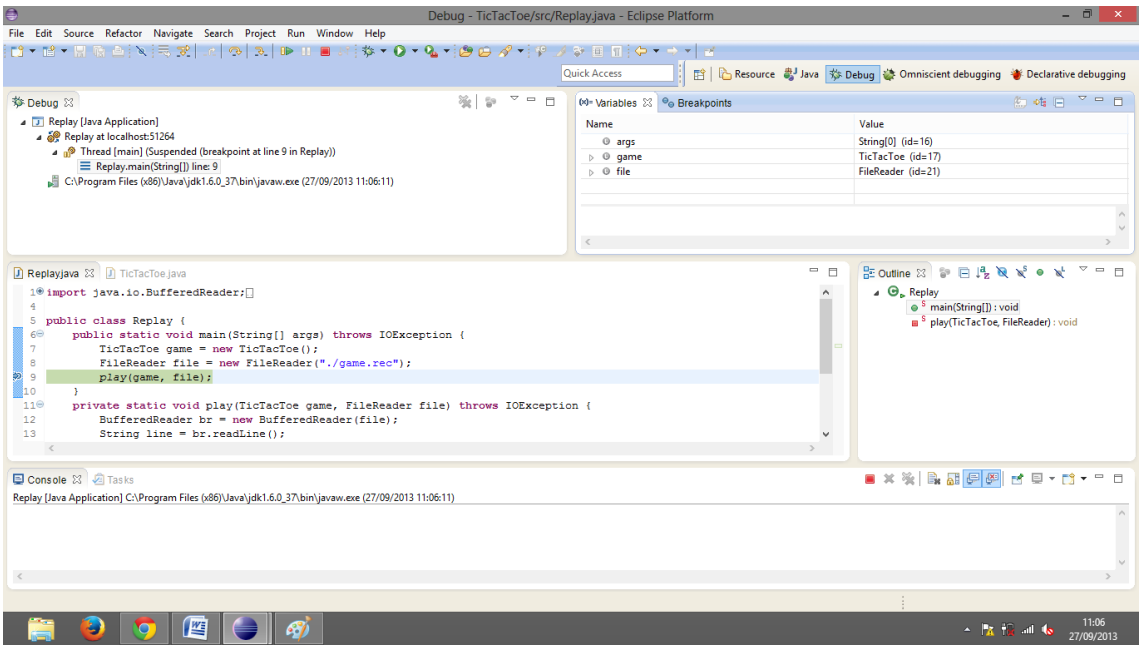


Fig. 23 Breakpointen la depuración por trazas

A continuación el usuario activa la depuración algorítmica del programa lo que genera el árbol de ejecución como se muestra en la FIG. 24.

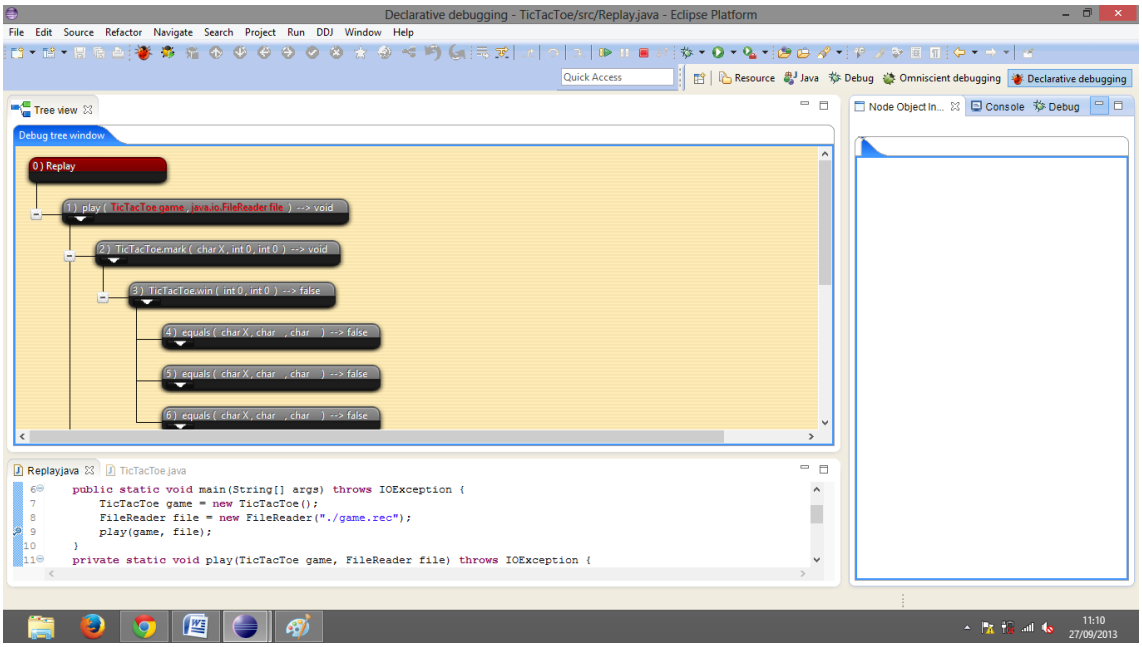


Fig. 24 Árbol de ejecución creado

A continuación el usuario inicia la depuración guiada del árbol de ejecución, por lo que el depurador realiza preguntas al usuario hasta encontrar el nodo erróneo, y por lo tanto el método que ha producido el error. Todo este proceso queda ilustrado en la FIG. 25.

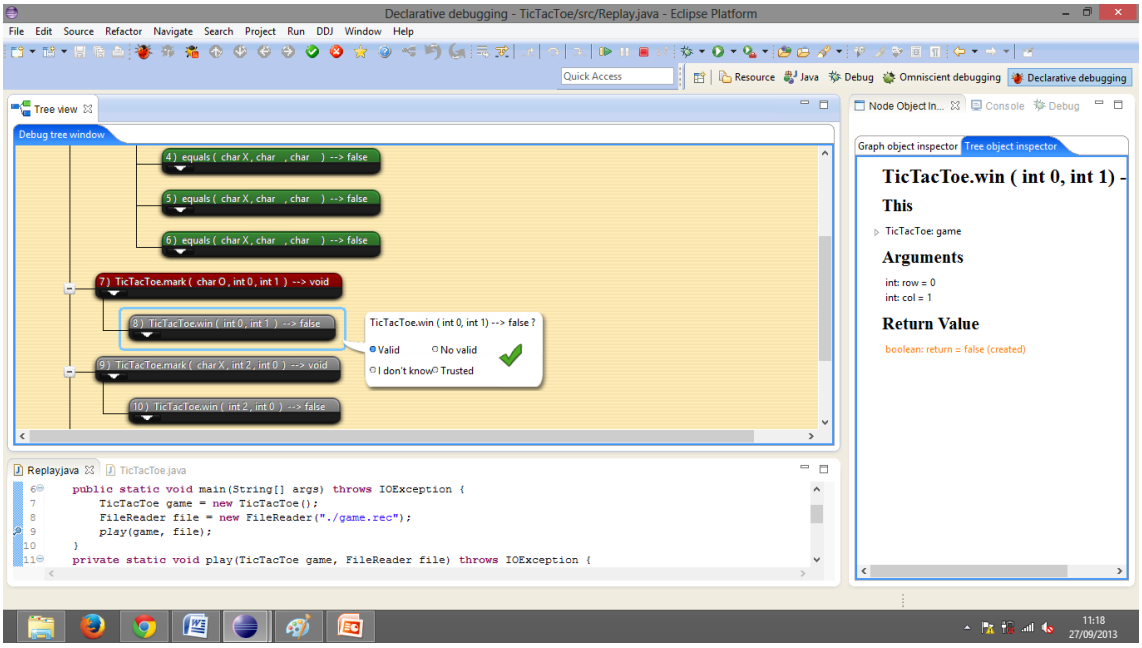


Fig. 25 Depuración del árbol de ejecución

Una vez la depuración algorítmica localiza el nodo erróneo, el depurador sitúa un breakpoint condicional al inicio del método asociado a este nodo. El usuario puede volver a ejecutar el programa, y éste sólo se detendrá cuando se dan las condiciones que han producido el error. Cuando el usuario se encuentra detenido en este método como se muestra en la FIG. 26, el usuario inicia la depuración omnisciente de dicho método erróneo.

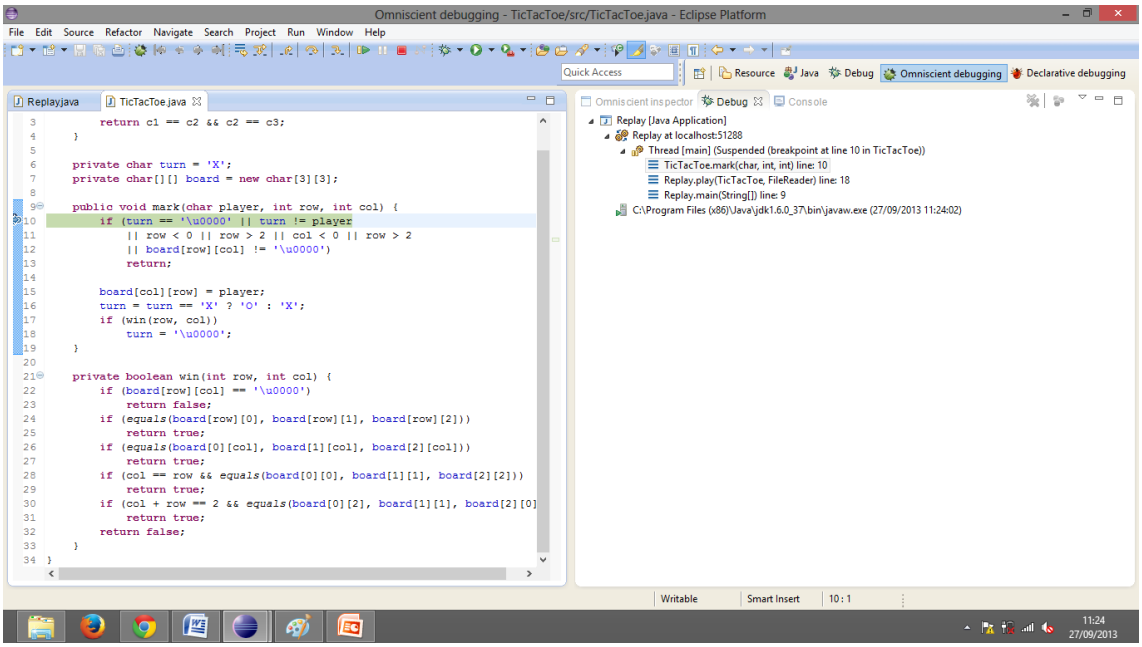


Fig. 26 Inicio de la depuración omnisciente

Una vez generada la traza, el usuario puede usar la barra de herramientas del depurador omnisciente para moverse a través de la misma. Mostrando el estado de la máquina en cada instante como se muestra en la FIG. 27.

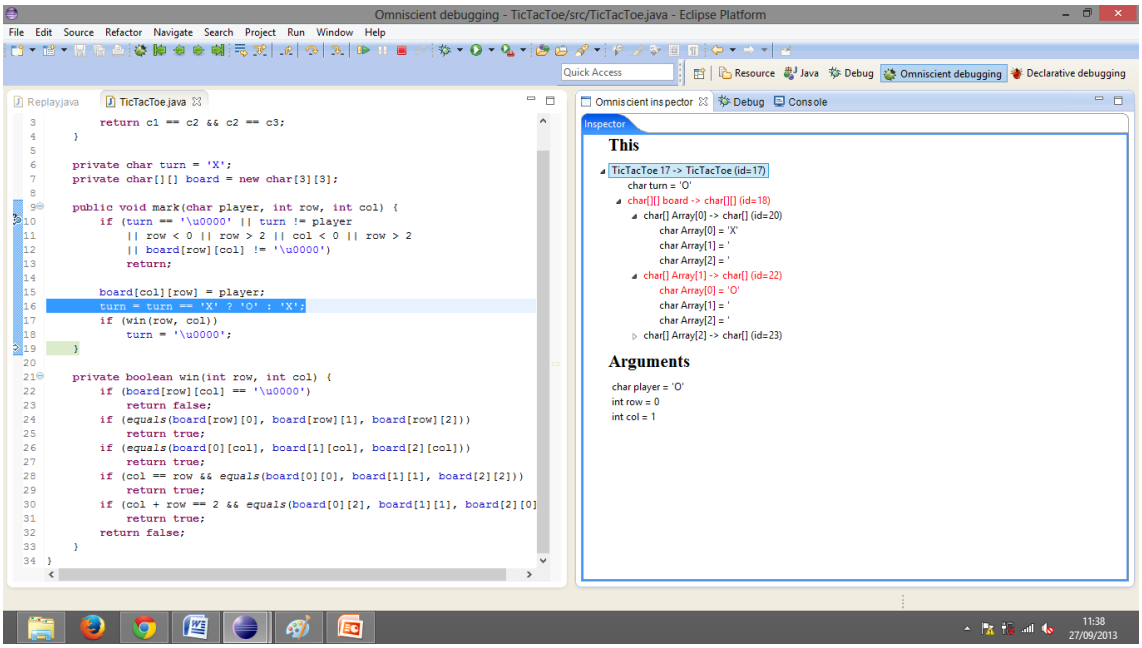


Fig. 27 Exploración de la traza con la depuración omnisciente

11. Referencias

[1] (2008) Borland JBuilder. [Online]. http://www.embarcadero.com/products/

[2] (1999) Netbeans. [Online]. http://www.netbeans.org/

[3] (2003) Eclipse. [Online]. http://www.eclipse.org/

[4] E. Shapiro, "Algorithmic Program Debugging," , 1982.

[5] J. Silva, "A Survey on Algorithmic Debugging Strategies," , 2011.

[6] B. Lewis. (2008) Debugging Backwards in Time. [Online]. http://arxiv.org/abs/cs.SE/0310016

[7] H. Nilsson, "Declarative Debugging for Lazy Functional Languages," Sweden, 1998.

[8] H. Nilsson and P. Fritzson, "Algorithmic Debugging for Lazy Functional Languages," , 1994.

[9] D. Insa and J. Silva, "An Algorithmic Debugger for Java," , 2010.

[10] S. Microsystems. (2010) Java Platform Debugger Architecture - JPDA. [Online]. http://java.sun.com/javase/technologies/core/toolsapis/jpda/

[11] R. De Miguel, and S. Serrano. F. González. Technical report, Universidad Complutense de Madrid. [Online]. http://eprints.ucm.es/9114/

[12] H. Girgis and B. Jayaraman, "JavaDD: a Declarative Debugger for Java," 2006,.

[13] D. Giammona., "ORACLE ADF - Putting It Together," 2009.

[14] T. Davie and O. Chitil. Hat-delta, "One Right Does Make a Wrong," , 2006.

[15] R. Caballero, "A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic," , New York, USA, 2005.

[16] (2007) Omnicore codeguide. [Online]. http://www.omnicore.com/en/codeguide

[17] M. Denker, and S. Ducasse C. Hofer, "Design and Implementation of a Backward-In-Time Debugger," , Erfurt, Germany, 2006.

[18] G. Pothier, "Towards Practical Omniscient Debugging," , 2011.

[19] J. Barton, and C. Petitpierre S. Mirghasemi. (2011) Debugging by lastChange. [Online]. http://people.epfl.ch/salman.mirghasemi

[20] T. Girba, and O. Nierstrasz A. Lienhard, "Practical object-oriented back-in-time debugging," , 2008.

[21] H.-J. Kouh and W.-H. Yoo, "The Efficient Debugging System for Locating Logical Errors in Java Programs," , Montreal, Canada, 2003.

[22] P. Gestwicki and B. Jayaraman, "JIVE: Java Interactive Visualization Environment," , New York, NY, USA, 2004.

[23] C. Hermanns and H. Kuchen, "Hybrid Debugging of Java Programs," , Springer Berlin Heidelberg, 2013.

[24] Documentación Eclipse. [Online]. http://help.eclipse.org/