CONSTRUCCIÓN DE UN DEPURADOR PORTABLE DE CÓDIGO. UN ENTORNO GRÁFICO PARA EL DEPURADOR.

José L. Arjona Fernández, José M. Prieto Pérez, R. Corchuelo Gil
Dpto. de Lenguajes y Sistemas Informáticos
Facultad de Informática y Estadítica
Universidad de Sevilla
Web: http://www.lsi.us.es

El presente artículo corresponde con el último de una serie de artículos donde se ha venido mostrando las técnicas utilizadas para la implementación de un depurador de código.

Concretamente, en este artículo nos centraremos en la implementación de un entorno gráfico para el depurador que hemos construido en los anteriores que amenice la relación entre la persona que pretende depurar su programa y la herramienta que se lo permite. Por ello, aquí no se mostrarán ya técnicas tan específicas de implementación de depuradores como en los anteriores, pero consideramos que es igualmente importante pues la presentación de este entorno precisamente porque potencia el uso del depurador, convirtiendo en fácil de usar e intuitivo.

El desarrollo de un entorno gráfico para el depurador se puede considerar casi obligado. Podemos decir que actualmente una gran cantidad de aplicaciones software entra dentro de lo que denominamos como *aplicaciones gráficas* donde todo se basa en iconos gráficos, botones y todo tipo de componentes gráficos a los cuales se acceden, en la gran mayoría de los casos, a través del ratón o mediante teclas de acceso rápido. Esta filosofía se ve apoyada por la aparición en los últimos tiempos de sistemas operativos gráficos como los mundialmente extendidos “Windows” de Microsoft (Windows 3.1, ya en vías de extinción, Windows 95, Windows 98 y Windows NT), MacOS de Apple o X-Windows en los sistemas operativos UNIX. Por todo ello, no hay excusas ante la posibilidad de construcción de un entorno gráfico que facilite la depuración de los programas autodepurables que se generen a partir de la librería que implementa el depurador.

En este caso, el entorno que presentamos está desarrollado para la plataforma Windows (95/NT), aunque la idea básica que queremos dejar clara es que está implementación sirva de modelo para la construcción de otro entorno en cualquier otra plataforma.

Ventajas obtenidas con el uso de un entorno gráfico.

Los compiladores que generan código adecuado para el uso de nuestra librería de depuración obtendrán un programa que es capaz de autodepurarse. No obstante, la depuración se basa en un ejecutar una serie de comandos (junto con sus parámetros) dirigidos a la librería que nos permitan en todo momento obtener información relevante acerca de los objetos que forman el programa para poder conocer su buen o mal comportamiento a partir de las especificaciones iniciales.

Con la elaboración de un entorno gráfico (en nuestro caso bajo Windows 95/NT) nos permitirá obviar los detalles para el uso de esta línea de comandos y centrar nuestro tiempo y esfuerzo en la depuración del programa. Mediante el manejo de los objetos gráficos de este entorno se debe ser capaz de llegar a depurar el programa del mismo modo que se consigue con la línea de comando, pero añadiendo una característica: un entorno gráfico es siempre mucho más agradable e intuitivo para el usuario que una triste y monótona línea de comandos.

He aquí un simple ejemplo. Supongamos que hemos compilado el fichero fuente *Media.pas*, correspondiente a cierto lenguaje de programación, con la opción de depuración activada,

generándose dentro del programa la información de depuración necesaria por nuestra librería de depuración. Si ejecutásemos este programa sin más, obtendríamos en pantalla la línea de comandos que presenta la librería al usuario para la introducción de comandos, donde por ejemplo podremos inspeccionar el valor de una variable:

```
(Media.pas:18)? watch media
(Real) 0
(Media.pas:18)? _
```

Con el entorno gráfico ejecutándose "encima" de este programa esto mismo se reduce a añadir el identificador `media` dentro de una lista de comandos `watch` que se presenta al usuario en una ventana gráfica como se observa en la figura 1. En esta ventana el usuario tendría siempre a vista el valor de la variable, mientras que en la línea de comandos debe ejecutar el comando `watch` cada vez que desee inspeccionar el valor de la variable.

FIGURA 1. VENTANA QUE MUESTRA LA LISTA DE "WATCHES".

El ejemplo de arriba sólo constituye uno de las ventajas. Existen más ventajas sobre su equivante en la línea de comandos, como pueden ser: la colocación y edición de puntos de ruptura (en terminología inglesa son conocidos como *breakpoints*), modificación y evaluación de expresiones, etc.

Propiedades del entorno gráfico.

Unas de las principales características de nuestra librería de depuración es precisamente su portabilidad. Está escrita en código ANSI C siendo compilable desde cualquier plataforma que soporte este estándar. Esta característica no se debe perder con la incorporación del entorno. Esto obliga a no poder modificar el código de la librería que implique la pérdida de dicha portabilidad. Por ello, debe existir un medio de comunicación entre dos procesos diferentes (el programa autodepurable y el entorno gráfico): por un lado el entorno en sí, y por otro lado, el programa autodepurable (véase la figura 2).

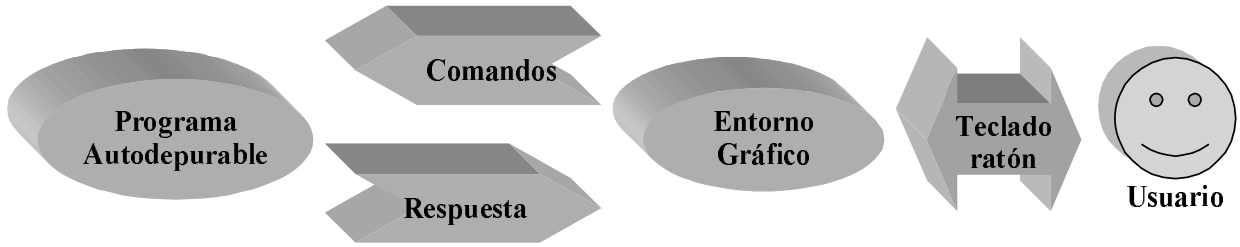

FIGURA 2. ESQUEMA GRÁFICO DE INTERACCIÓN "LIBRERÍA DEPURACIÓN-ENTORNO-USUARIO".

Por otra parte, interesa integrar el depurador en un entorno de trabajo configurable (también conocido como *Workbench*) que nos permita crear, editar, compilar y depurar los programas. Un workbench es un entorno de programa que nos abstrae el proceso de compilación en una línea de comandos. Para escribir aplicaciones en estos entornos se crea un proyecto que contendrá todo los archivos fuentes que constituyan la aplicación final. Estos archivos fuentes pueden ser de distinta índole (fuentes C, la especificación de un analizador léxico en `lex`, la especificación de un analizador sintáctico en YACC, etc.). Por esta razón, debemos configurar una serie de herramientas para el tratamiento de cada tipo de archivo fuente y obtener los ficheros objetos, que serán linkados con otra herramienta (linker) para la obtención del fichero final.

Por otro lado, el trabajo que presentamos en este artículo establece un modelo de implementación de una interfaz gráfica para el depurador. A partir de esta versión podremos construir o integrar el depurador en otros entornos y/o en otras plataformas siguiendo como modelo esta implementación.

Esquema de diseño

Como ya remarcábamos anteriormente en el documento, uno de las premisas principales que debe cumplir el entorno es que el depurador de código actualmente implementado no perdiera su portabilidad. Por ello, se ha optado por el diseño de la clase TAutoDebug encargada de abstraer desde el punto de vista del entorno los detalles de la interfaz entre lo que es el programa autodepurable y el entorno gráfico en sí (véase figura 3).

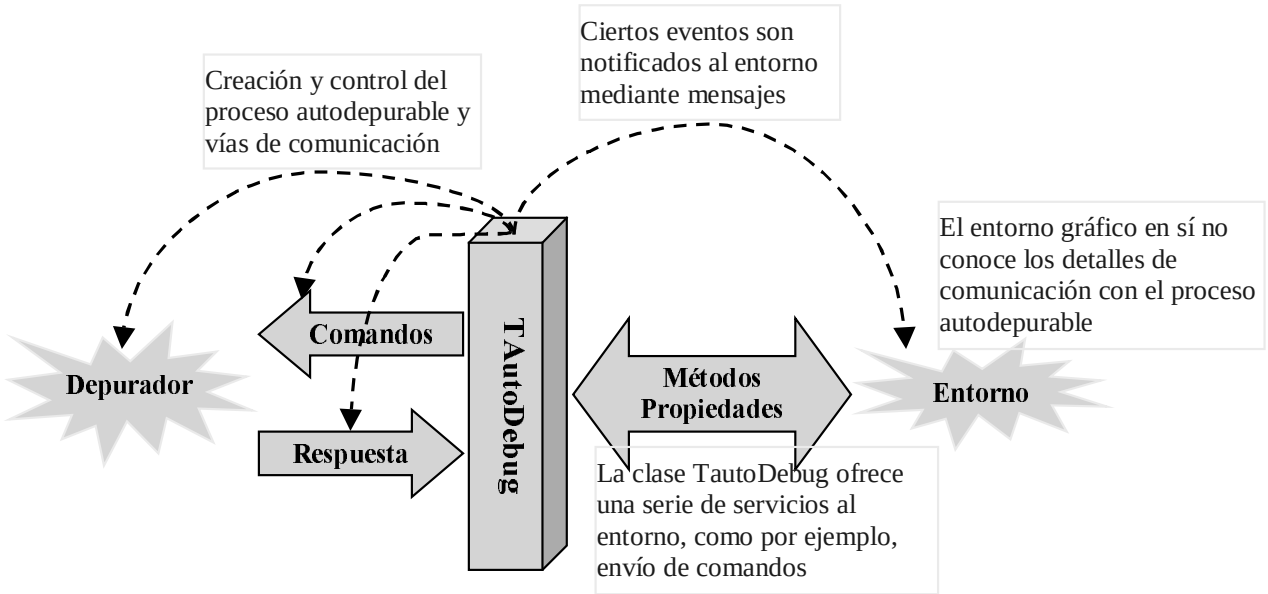

FIGURA 3. PAPEL DE LA CLASE *TAUTODEBUG* EN LA IMPLEMENTACIÓN DEL ENTORNO.

Cuando se crea una instancia de esta clase se le asocia el programa autodepurable con el que deberá comunicarse cuando comience su depuración. Es precisamente en ese momento cuando esta

clase lanza en *background* un proceso independiente correspondiente a una instancia de este programa, pero antes se crean y configuran las vías de comunicación entre el proceso y la instancia TAutoDebug. A partir de que este proceso está en ejecución comienza el control sobre su ejecución. Por otro lado, esta clase ofrece como servicio al entorno gráfico el envío de comandos y recepción de sus respuestas, y le notifica, mediante mensajes, los eventos asíncronos que se produzcan.

Otro tipo de servicios que ofrece esta clase al entorno es el manejo y control sobre los *watches* (inspecciones de variables y/o valores de expresiones) y de los *breakpoints* (puntos de ruptura en el código). El entorno sólo tiene que preocuparse de saber qué es lo que quiere inspeccionar (o que expresión ya no interesa inspeccionar) o donde quiere colocar el breakpoint (o qué breakpoint quiere quitar) y presentarlo gráficamente al usuario.

En cuanto al comportamiento dinámico de esta clase, se resume en la figura 4. Básicamente una vez que hemos creado las vías de comunicación y lanzado el proceso autodepurable (INICIO), éste contesta con una respuesta de bienvenida (WELCOME) y espera el primer comando. A partir de este momento se entra en el ciclo "envía comando-obtiene respuesta" (COMANDO-RESPUESTA) hasta que finaliza la ejecución del proceso autodepurable. Podemos distinguir también ciertos eventos que deben ser notificados al entorno gráfico a lo largo del proceso de depuración, como pueden: "comienzo de la depuración", "se ha producido el error...", "recepción de respuesta", "finalizó la depuración", etc.

Las ventajas de este diseño son varias:

1. Las ventajas de la propia programación orientada a objetos.

2. Es más modulable. Podemos distinguir claramente lo que son aspecto gráficos en sí del propio entorno (como por ejemplo, qué es lo que se hace cuando se pulsa cierto botón) y la necesidad de comunicar procesos independientes (el programa autodepurable y el entorno gráfico). Está última es la parte puramente técnica del problema y es lo que se resuelve con el diseño de la clase TAutoDebug (los aspectos puramente gráficos dependen del leguaje y plataforma en el que se esté desarrollando).

3. En relación con el punto anterior, la clase TAutoDebug abstrae todos los detalles relacionados puramente con el control de la depuración y los detalles puramente gráficos del entorno.

4. A partir de la especificación de la clase TAutoDebug tenemos establecido un modelo de implementación de la comunicación con la librería de depuración, y por tanto podremos construir nuevos entornos o integrar la posibilidad de depuración en otros entornos ya existentes a partir de este esquema de diseño independientemente de la plataforma y lenguaje utilizados (recordar que esto era uno de los objetivos que se pretendían al comienzo).

Implementación Delphi 3.0

Este entorno está actualmente implementado en Delphi 3.0. La plataforma de desarrollo ha sido Windows 95/NT, haciendo uso de las *Win32 API*.

La comunicación entre el depurador y esta clase está basada en *anonymous pipes* (tuberías sin nombre), creadas con la llamada `CreatePipe` de las *Win32 API*. Todo el flujo de información entre ambas partes atraviesa estas estructuras de datos. Se crean tres pipes correspondientes al flujo de entrada al depurador (envío de comandos), al flujo de salida (respuesta a comandos enviados) y al flujo de error (mensajes de error que el depurador notifica ante situaciones erróneas).

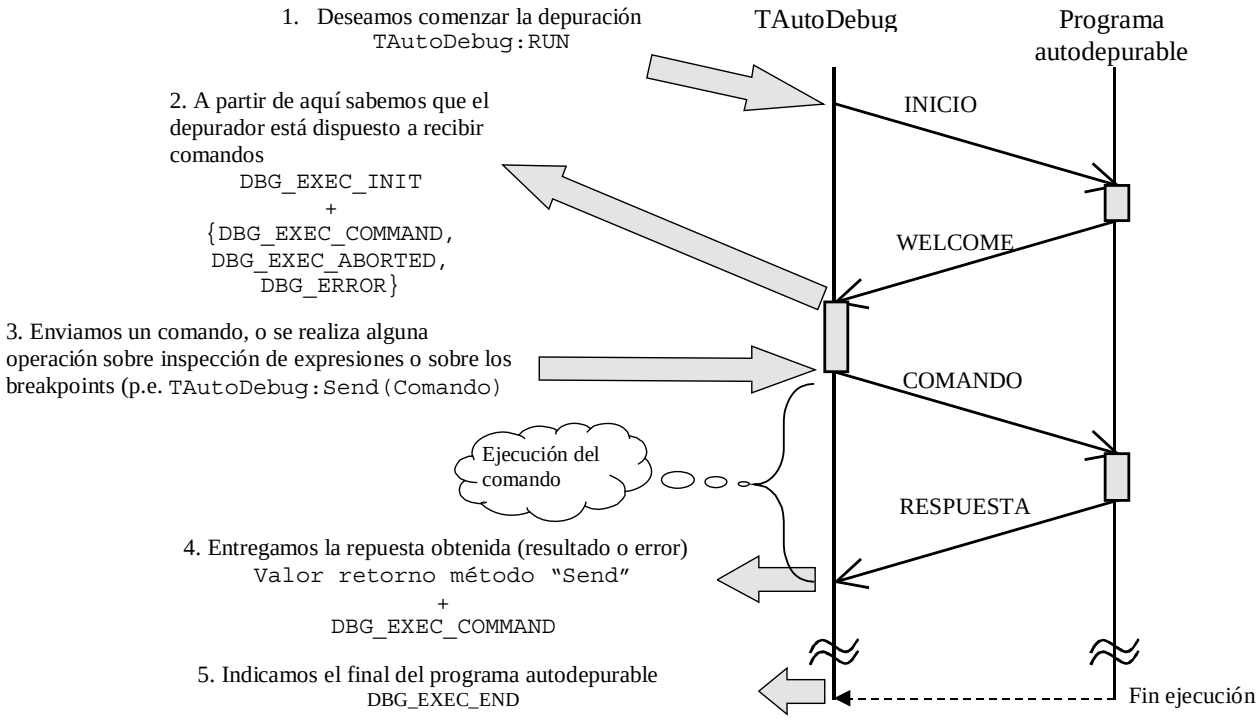

FIGURA 4. COMPORTAMIENTO DINÁMICO DE LA CLASE *TAUTODEBUG*.

Cuando el entorno desea lanzar la ejecución del proceso autodepurable llama al método `Run` de clase TAutoDebug. En este método se crean las ya mencionadas pipes y se lanza el proceso autodepurable con la llamada `CreateProcess` de la *Win32 API*, pero previamente se redirecciona la entrada, salida y salida de error estándar del proceso a las pipes correspondientes. La librería de depuración (insertada en el código que deseamos depurar) se encarga entonces de tomar éstas y redireccionarlas de nuevo para no estorbar la entrada, salida y salida de error estándar de este programa. Por tanto, una vez ejecutándose el proceso se tienen las vías de comunicación establecidas (véase figura 5).

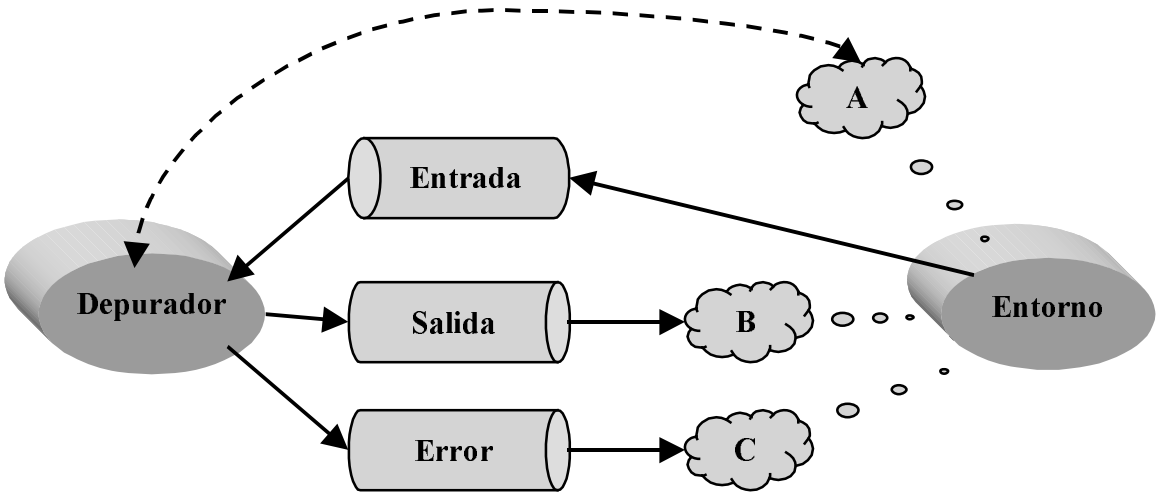

FIGURA 5. ESQUEMA DE LA COMUNICACIÓN ENTRE EL DEPURADOR Y EL ENTORNO.

Cuando el entorno desea ejecutar un comando del depurador solo tiene que llamar al método `Send(Comando)` de la clase TAutoDebug (`Comando` hace referencia al comando concreto que se quiere ejecutar y sus parámetros). Ella se encarga de enviar el comando a través de la pipe al depurador. Cuando éste ejecuta el comando, le responde, bien con el resultado o bien con una respuesta de error, llegados desde cualquiera de las dos pipes por donde la clase TAutoDebug lee

("Salida" o "Error" respectivamente). Esta respuesta es la que retorna este método a su llamador en foma de la estructura `TCommand`:

```
Tcommand = record
   IdCmd: Integer;      { Id. de comando }
   eType: TECommand;    { Tipo de comando: watch, next, etc. }
   sParams: string;     { Parametros del comando }
   sResponse: string;   { Respuesta al comando }
   sError: string;      { Este campo solo se utiliza si hay error. En
                          este caso, contendrá la cadena con la que
                          responde el programa por culpa del error }
   bError:boolean;      { True si se ha producido algun error durante
                          la ejecución del comando }
   atPos: Tposition;    { Posicion dentro del programa a donde se va
                          una vez ejecutado el comando }
end;
```

Además de los métodos `Run` y `Send(Comando)`, la clase TAutoDebug ofrece métodos que nos permiten añadir y quitar expresiones que deseamos inspeccionar (*watches*) y obtener el valor actualizado de la evaluación de estas expresiones: `AddWatch(Expression)`, `RemoveWatch(Index)`, `WatchAt(Index)`, etc. Normalmente, la llamada a estos métodos generan envíos de comandos internamente en la clase TAutoDebug, hecho totalmente transparente para el entorno gráfico (usuario de esta clase). Lo mismo podemos decir respecto a la manipulación de los *breakpoints*: `AddBreak(CodePoint)`, `RemoveBreak(Index)`, `BreakAt(Index)`, etc.

Hilos de control y escucha

La clase TAutoDebug debe tener controlado en todo momento el estado de ejecución del proceso, es decir, controlar si se está ejecutando, si se aborta su ejecución, si ha acabado su ejecución, etc. A la vez, debe estar leyendo por las pipes por donde el proceso autodepurable escribe las respuestas y errores para generar la información de respuesta que ha de devolver cuando se le ordena que ejecute un comando (valor de retorno del método `Send`).

Por ello, esta clase crea tres hilos o subprocesos (*threads*) realizando llamadas a `CreateThread` de las *Win32 API* (A, B y C en la figura 5). Estos hilos se encargan:

1. Hilo A. Control del estado de ejecución del proceso. Espera a que la ejecución del proceso autodepurable termine y recoge el código de error devuelto. Los hilos B y C están permanentemente escuchando de las pipes de salida y de error del depurador respectivamente.
2. Hilo B. "Escucha" por la pipe de la salida del depurador.
3. Hilo C: "Escucha" por la pipe de la salida de error del mismo.

Para la sincronización de estos hilos y el acceso a zonas de memoria compartidas se utilizan semáforos creados a partir de la llamada `CreateSemaphore` de las *Win32 API*.

Notificación de eventos: Mensaje Windows

La notificación de eventos durante la depuración se realiza a partir de mensajes Windows con la llamada `PostMessage` de las *Win32 API*. Concretamente se utilizan cinco mensajes de usuario (WM_USER), uno por cada tipo de evento. Ya que la clase TAutoDebug es utilizada por el entorno gráfico, esta clase manda estos mensajes al procedimiento de ventana principal del entorno (su *handle* se le pasa como parámetro al constructor de esta clase), que deberá contar con sus rutinas de manejos (*message handlers*) si quiere enterarse de estos envíos. La tabla 1 muestra estos mensajes.

Nombre mensaje	Descripción	Parámetros	
		WPARAM	LPARAM
DBG_EXEC_INIT	Indica que el proceso correspondiente al programa autodepurable acaba de empezar su ejecución	0	0
DBG_EXEC_END	Indica que el proceso autodepurable ha finalizado su ejecución normalmente, sin producir ningún error anormal	0	0
DBG_EXEC_ABORTED	Indica que el proceso autodepurable ha finalizado anormalmente (se ha dado una situación de error que lo ha hecho abortar). Este evento se puede producir en cualquier momento durante la ejecución de un comando	Puntero al comando que produjo el error	Código de salida del proceso
DBG_EXEC_COMMAND	Indica que se acaba de recibir la respuesta al último comando enviado	Puntero al comando correspondiente a la repuesta recibida	0
DBG_ERROR	Indica una situación de error que impidió crear el proceso autodepurable	Causa del error (constante indicando el tipo de error)	0

TABLA 1. MENSAJES WINDOWS UTILIZADOS PARA LA NOTIFICACIÓN DE EVENTOS.

En la figura 4 podemos observar cuando se pueden generar estos mensajes durante la dinámica de ejecución de la clase TAutoDebug.

El entorno gráfico que ve el usuario

Como ya hemos mencionado arriba esta primera versión del entorno se ha implementado bajo la plaforma Windows 95/NT. La figura 6 muestra la ventana principal del entorno. Podemos ver en ella el aspecto típico de cualquier aplicación Windows: barra de menús, barra de herramientas con botones de acceso rápido y una barra de estado. Al ser una aplicación MDI (*Multiple Document Interface*) tendremos una ventana de documento por cada archivo fuente que vayamos trazando.

En el menú principal del entorno podremos encontrar todas las opciones que la aplicación nos permite: seleccionar el programa que queremos depurar, añadir puntos de ruptura, añadir un nuevo "watch", etc.

En la barra de herramientas tenemos accesibles rápidamente las opciones más comunes del menú principal. Entre estas opciones están: seleccionar el programa autodepurable, lanzar su ejecución, poner puntos de ruptura, avanzar en la traza de ejecución, etc.

En la barra de estado podemos observar que existe un pequeño panel en el lado derecho de la misma que nos da información del estado actual del programa seleccionado. En este caso, se nos indica que está ejecutándose.

Por último, el resto de la ventana principal del entorno se reserva para las ventanas de documentos que se tengan abiertas.

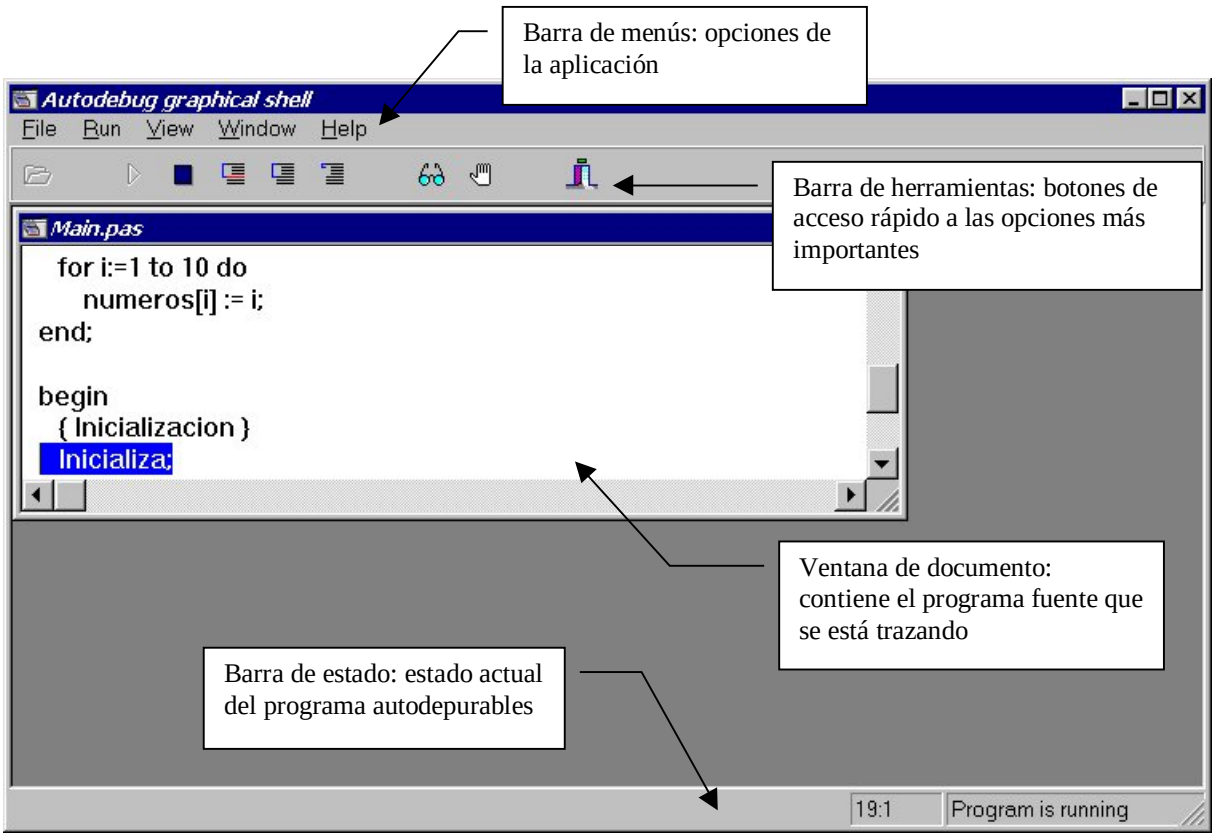

FIGURA 6. VENTANA PRINCIPAL DEL ENTORNO.