ESCUELA SUPERIOR POLITECNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

“Análisis, Diseño E Implementación De Un Ambiente De Desarrollo Para C/C++ Bajo El Entorno Windows Orientado A Estudiantes De Cursos Básicos De Programación”

TESIS DE GRADO

Previo la obtención del Titulo de:

INGENIERO EN COMPUTACION ESPECIALIZACION EN SISTEMAS DE INFORMACION

Presentada por:

Andrey David Del Pozo Tkatchenko

GUAYAQUIL – ECUADOR

Año: 2004

AGRADECIMIENTO

A todas las personas que de una u otra manera colaboraron con la realización de este trabajo.

DEDICATORIA

A MIS PADRES

TRIBUNAL DE GRADUACION

Ing. Miguel Yapur
SUBDECANO DE LA FIEC
PRESIDENTE

Ing. Guido Caicedo
DIRECTOR DE TESIS

Ing. Ana Tapia
VOCAL

Ing. Marisol Villacres
VOCAL

DECLARACION EXPRESA

"La responsabilidad del contenido de esta Tesis de Grado, me corresponden exclusivamente; y el patrimonio intelectual de la misma a la ESCUELA SUPERIOR POLITECNICA DEL LITORAL"

(Reglamento de Graduación de la ESPOL)

____

Andrey Del Pozo T.

RESUMEN:

En los programas de estudio de ingeniería en computación los primeros cursos tratan sobre programación estructurada y orientada a objetos, estructuras de datos y lenguajes de programación y establecen bases necesarias para los cursos siguientes sobre tópicos más especializados.

Es en este nivel en el que los estudiantes deben asimilar nuevos conocimientos y fortalecer sus habilidades de abstracción y de resolver problemas utilizando herramientas de programación.

Sin embargo, las herramientas de programación disponibles generalmente no enfatizan el aspecto del aprendizaje y la capacidad de personalizar el nivel de complejidad para así permitir en los primeros cursos el manejo de pequeños programas y a medida que crezca el nivel de experiencia, extenderse para dar soporte a proyectos más avanzados. Además sería conveniente que sin importar el nivel complejidad, la herramienta provea facilidades de manipulación y visualización que sean naturales y fáciles de usar para el usuario.

Las herramientas actuales de programación pueden ser divididas en dos grandes grupos: Aquellas que están orientadas hacia los desarrolladores profesionales y asumen cierto nivel de conocimiento y experiencia, y las que se desarrollan con propósitos de instrucción y son simples de usar pero al mismo tiempo muy limitadas en cuanto a recursos de desarrollo para el programador.

Las primeras se enfocan en mejorar la productividad del programador experimentado, la eficiencia del código, el soporte a librerías y otros aspectos tecnológicos. Las segundas, que generalmente son producidas por pequeñas compañías de software y/o universidades, están orientadas a simplificar el interfaz e incluso a proveer interactividad para facilitar el aprendizaje pagando el precio de imponer ciertas limitaciones como por ejemplo el uso de lenguajes no actuales o propietarios o el uso de un interfaz que no aprovecha los GUI modernos de la mejor manera y que generalmente no proveen facilidades de visualización o depuración similares a las ofrecidas por herramientas profesionales.

Este vacío en la oferta de herramientas de desarrollo afecta al aprendizaje de los estudiantes novatos debido a que pueden presentarse los siguientes escenarios.

1. Si se opta por una herramienta orientada hacia los desarrolladores profesionales entonces se corre el riesgo de que el estudiante pueda enfocarse demasiado en el aprendizaje de la herramienta y no en aprender a resolver problemas usando la misma.

2. Por otro lado, el uso de una herramienta de instrucción puede producir en el estudiante una experiencia incompleta dado que este tipo de herramientas de programación que no proveen mecanismos para manejar eficientemente el proceso de programación o no explotan los beneficios que proveen los GUI modernos debido a la falta de disponibilidad de mecanismos sencillos para hacerlo.

Con el sistema ha desarrollarse se pretende reducir la brecha que existe entre las herramientas orientadas a desarrolladores profesionales y las herramientas de instrucción. Al crear una herramienta que sea más intuitiva y con más potencial de desarrollo que las herramientas de instrucción mientras se reduce la carga cognitiva y la base de conocimientos que demanda una herramienta profesional.

ÍNDICE GENERAL

Pág.

RESUMEN………….....………..………………………………………….……..…...... VI

ÍNDICE GENERAL……………………………………………………………….... IX

ÍNDICE DE FIGURAS……….....…………………………………………………..... XII

ÍNDICE DE CUADROS……………………………………………………………..... XV

ÍNDICE DE TABLAS………………………………………………………………… XVII

CAPÍTULO 1

1. INTRODUCCIÓN Y JUSTIFICACIÓN………………………………………….... 1

1.1 Entornos Integrados de Desarrollo………………………………................... 1

1.2 Objetivos…………………………………………………………..................... 4

1.3 Justificación……………………………………………………………………. 6

CAPÍTULO 2

2. ENTORNOS INTEGRADOS DE DESARROLLO…………………………..…. 9

2.1 Componentes De Un Entorno Integrado De Desarrollo………………….. 9

2.2. Interacción De Los Entornos Integrados De Desarrollo Con El Compilador……………………………………………………………………… 12

2.3 Interacción De Los Entornos Integrados De Desarrollo Con El Depurador………………………………………………………………………… 26

2.4 Gramáticas, Lexing Y Parsing……………………………………………... 32

2.5 Presentación Visual De Las Estructuras………………………………… 43

CAPÍTULO 3

3. ANÁLISIS…………………………………………………………………..... 47

3.1 Requerimientos Funcionales Del Sistema………………….................. 47

3.2 Factores De Interacción Hombre Maquina……………………………… 50

3.2.1 Asunciones Hechas Sobre Los Usuarios………………………….. 51

3.2.2 Requerimientos De Usabilidad……………………………………… 53

3.3 Análisis De Estrategias De Desarrollo Del Proyecto…………………… 58

3.4 Los Componentes Del Sistema Y El Código Abierto…….................... 63

3.5 Análisis Y Determinación De La Plataforma De Ejecución…………….. 67

3.6 Análisis Y Determinación De Las Herramientas De Desarrollo……… 75

CAPÍTULO 4

4. DISEÑO……………………………………………………………………… 78

4.1 Componentes De Terceros………………………………………………. 79

4.2 Layout De Ventanas……………………………………………………… 92

4.3 Arquitectura Del Sistema………………………………………………… 105

4.3.1 Modelo Estático ………………………………………………... 105

4.3.2 Modelo Dinámico………………………………………………………….. 137

4.4 Arquitectura De La Interfase Con El Depurador Externo………………. 146

4.4.1 Modelo Estático……..……………………………………………….... 146

4.4.2 Modelo Dinámico………………………………………………………. 154

4.4.3 Protocolo De Comunicación Entre El IDE Y El Depurador...…… 172

CAPÍTULO 5

5. IMPLEMENTACIÓN……………………………………………………………….. 193

5.1 Componentes De Terceros……………………………………………… 193

5.2 Arquitectura Del Sistema………………………………………………… 207

5.3 Arquitectura De La Interfase Con El Depurador Externo………………. 253

6. CONCLUSIONES Y RECOMENDACIONES……………………………… 274

APÉNDICES

BIBLIOGRAFÍA

ÍNDICE DE FIGURAS

Pág.

Figura 2.1. Modelo de funcionamiento de un compilador……………………… 13

Figura 2.2. Diagrama de redirección de I/O del compilador…………...……. 25

Figura 2.3. Modelo de funcionamiento de un depurador……………….……. 26

Figura 2.4. Diagrama de redirección de I/O del depurador…………………….. 31

Figura 2.5. AEF de la gramática de ejemplo…………………………………….. 40

Figura 2.6. Captura de pantalla de DDD…………………………………………. 45

Figura 3.1. Diagrama de bloques de los componentes del IDE……………….. 61

Figura 4.1. Ejemplo de las características de Scintilla…………………………. 83

Figura 4.2. Demo de ProfUI……………………………………………………….. 87

Figura 4.3. Edición en el modo avanzado……………………………………….. 93

Figura 4.4. Edición en el modo novato…………………………………………… 96

Figura 4.5. Paneles flotantes……………………………………………………… 98

Figura 4.6. Sesión de depuración………………………………………………… 99

Figura 4.7. Exploración de la pila de llamadas………………………………….. 103

Figura 4.8. Jerarquía de clases de la arquitectura documento/vista………… 109

Figura 4.9. Arquitectura del IDE…………………………………………………... 113

Figura 4.10. Arquitectura de paneles flotantes………………………………….. 116

Figura 4.11. Arquitectura del panel de vista de archivos……………………… 118

Figura 4.12. Arquitectura de paneles de depuración…………………………… 121

Figura 4.13. Arquitectura del panel de visualización de variables…………….. 125

Figura 4.14. Arquitectura de paneles de salida del compilador……………….. 127

Figura 4.15. Arquitectura del componente de datos de aplicación/proyectos.. 130

Figura 4.16. Arquitectura del componente del editor de código……………….. 133

Figura 4.17. Arquitectura de ayudas a la edición de código…………………… 134

Figura 4.18. Arquitectura del componente de la interfase con el compilador... 136

Figura 4.19. Diagrama de secuencia de la inicialización del IDE……………... 141

Figura 4.20. Diagrama de secuencia de la compilación de un aplicación……. 145

Figura 4.21. Componente de la interfase con el depurador……………………. 150

Figura 4.22. Interacción de la interfase del depurador con la interfase de

Usuario……………………………………………………………………………… 153

Figura 4.23. Diagrama de secuencia del Inicio de una sesión depuración…... 158

Figura 4.24. Diagrama de secuencia de la inserción de una variable………… 162

Figura 4.25. Diagrama de secuencia de la inserción de una variable para

visualizarla como estructura……………………………………………………….. 165

Figura 4.26. Diagrama de Secuencia del Encuentro de un Punto de Ruptura 168

Figura 4.27. Diagrama de secuencia de la actualización del valor de una

variable……………………………………………………………………………. 171

Figura 4.28. Diagrama de bloques de la gramática del depurador……………. 177

Figura 4.29. Diagrama de la comunicación con GDB…………………………... 184

Figura 5.1. Dialogo de creación de nuevo proyecto…………………………... 211

Figura 5.2. Dialogo de creación de nuevo archivo……………………………… 212

Figura 5.3. Identificadores de menús y barras de herramientas………………. 213

Figura 5.4. Diagrama de clases de los paneles flotantes………………………. 214

Figura 5.5. Elementos de visualización y enlaces………………………………. 218

Figura 5.6. Diagrama de clases de elementos de visualización y enlaces…... 221

Figura 5.7. Diagrama de secuencia del cargado de un proyecto……………… 229

Figura 5.8. Identificación de un archivo modificado…………………………….. 241

Figura 5.9. Paréntesis con pareja………………………………………………… 243

Figura 5.10. Paréntesis sin pareja………………………………………………… 243

Figura 5.11. Lista de autocompletar……………………………………………… 245

Figura 5.12. Tip de los parámetros de una función…………………………... 250

Figura 5.13. Diagrama de estados del FSA del depurador……………………. 266

Figura 5.14. Detalle del diagrama de estados de la recuperación de valores del depurador………………………………………………………………….. 267

ÍNDICE DE CUADROS

Pág.

Cuadro 1. Compilación con errores……………………………………………… 14

Cuadro 2. Ejemplo de un Archivo Make……………………………………… 15

Cuadro 1. Compilación usando nmake……………………………………….. 22

Cuadro 2. Sesión de Depuración en una Consola…………………………… 29

Cuadro 3. Gramática de Ejemplo……………………………………………… 37

Cuadro 4. Notación empleada………………………………………………….. 174

Cuadro 5. Sintaxis de Comandos de GDB/mi………………………………… 175

Cuadro 6. Sintaxis de la Salida de GDB/mi…………………………………… 176

Cuadro 7. Suspensión del Programa Depurado……………………………… 178

Cuadro 8. Respuesta del Depurador a un Comando………………………… 178

Cuadro 9. Respuesta del Depurador a un Comando (continuación)………… 179

Cuadro 10. Inicio de la sesión de depuración………………………………… 185

Cuadro 11. Fin de la sesión de depuración………………………………….... 185

Cuadro 12. Arranque del programa depurado………………………………... 186

Cuadro 13. Terminación del programa depurado…………………………….. 186

Cuadro 14. Inserción de un punto de ruptura…………………………………. 187

Cuadro 15. Eliminación de un punto de ruptura……………………………... 187

Cuadro 16. Inserción de una variable………………………………………….. 188

Cuadro 17. Eliminación de una variable………………………………………… 189

Cuadro 18. Consulta del tipo y valor de una variable…………………………... 189

Cuadro 19. Consulta de la dirección de memoria de una variable……………. 190

Cuadro 20. Consulta del tamaño en memoria de una variable………………... 190

Cuadro 21. Consulta de la estructura de una variable………………………… 191

Cuadro 22. Consulta de la pila de llamadas…………………………………….. 191

Cuadro 23. Creación de una Ventana Scintilla…………………………………. 195

Cuadro 24. Envío de Mensajes a Scintilla……………………………………….. 196

Cuadro 25. Comunicación Directa con Scintilla…………………………………. 197

Cuadro 26. Manejo de Notificaciones de Scintilla……………………………… 198

Cuadro 27. Uso de GCC…………………………………………………………… 201

Cuadro 28. Archivo de Proyecto………………………………………………….. 225

Cuadro 29. Creación de un Proceso……………………………………………… 232

Cuadro 30. Redirección de la E/S en Windows…………………………………. 235

Cuadro 31. Archivo de los Prototipos de Funciones……………………………. 252

Cuadro 32. Definiciones de Estructuras………………………………………….. 260

Cuadro 33. Salida de un Comando Complejo de GDB………………………… 261

Cuadro 34. Seudo código de un Autómata de Estado Finito………………….. 265

ÍNDICE DE TABLAS

Pág.

Tabla 1 Comparación entre una solución de escritorio y una solución Web ... 70

Tabla 2. Iconos de los Marcadores……………………………………………….. 239

Tabla 3. Tabla de Resultados del Depurador……………………………………. 259

Tabla 4. Tabla de Resultados con Claves Duplicadas…………………………. 262

Tabla 5.Tabla de Resultados Mejorada………………………………………….. 263

Tabla 6. Detalle de las Condiciones del FSA……………………………………. 268

CAPÍTULO 1

1. INTRODUCCIÓN Y JUSTIFICACIÓN

1.1 Entornos Integrados de Desarrollo

Se conocen como Entornos Integrados de Desarrollo o IDE por sus siglas en ingles (Integrated Development Environment) a aquellas herramientas de software utilizadas para el desarrollo de aplicaciones de manera alternativa a la construcción basada en archivos Makefile.

Los IDEs agrupan un conjunto de aplicaciones relacionadas a la construcción de aplicaciones, entre las que se podría encontrar un editor, un compilador y un depurador. Cabe recalcar que un IDE agrupa otras herramientas además de las mencionadas

anteriormente como por ejemplo editores de imágenes, exploradores de dependencias, medidores de rendimiento (profilers) entre otras. La característica principal de un IDE es el uso uniforme de las herramientas dentro del mismo entorno.

Entre las características que un IDE moderno brinda tenemos:

Capacidades de edición sofisticadas: El código fuente se puede presentar en diferentes tipos fuentes, tamaños y colores. Esto da la posibilidad de agregar información a la presentación de texto. También se esperan capacidades de sangrado (indentación) automático, presentación de ayuda contextual y terminación automática de las expresiones.

Manipulación de múltiples archivos: Permite la edición de más de un archivo sin tener que iniciar otra instancia de la aplicación u otra aplicación diferente. De esta manera se puede tomar segmentos de código para incorporarlos en el proyecto actual, para comparar diferentes versiones o trabajar en varios archivos que tiene relación para el proyecto actual.

Automatización de la compilación y configuración: Los compiladores son en su mayoría herramientas con una interfase de texto controladas por un gran numero argumentos. Escribir y recordarlos es una tarea difícil y propensa a errores. Por configuración no referimos a los archivos de código fuente y las opciones del compilador necesarias para construir una aplicación. El entrono de presenta al usuario las opciones del compilador de manera descriptiva. Las opciones son reconfiguradas para proyectos nuevos y recordadas por el IDE para aligerar la carga cognitiva del usuario.

Depuración a nivel de código fuente: De la misma manera que los compiladores son herramientas de línea de comando también lo son los depuradores. La interacción con los depuradores es compleja y la presentación de la salida, tanto la información de la sesión de depuración, así como del programa, se realiza sobre la misma ventana. El entorno debería conectar el código fuente de la aplicación con la salida de depurador. Además de hacer más sencillo el ingreso de puntos de ruptura y la observación del valor de las variables.

Capacidades de edición de diferentes recursos: En ciertas ocasiones las aplicaciones necesitan de archivos diferentes a los de texto, tales como mapas de bits, guiones que definen interfaces de usuario o html. Con el fin de mantener cierta consistencia en el desarrollo el entorno debería proveer capacidades de edición/visualización archivos de formatos diferentes.

1.2 Objetivos

El objetivo principal de esta tesis es proveer un ambiente de desarrollo en C/C++ para cursos básicos que fomente buenas prácticas de programación como base para la producción de código de calidad. Al mismo tiempo se busca que el entorno provea una alta usabilidad para programadores novatos integrando múltiples facilidades para el ingreso, la revisión y la ejecución del código utilizando en lo posible principios de IHM y técnicas para la visualización de código y estructuras de datos. Se busca con esto compensar la complejidad inherente del lenguaje C como un primer lenguaje de programación.

Entre las submetas a lograr están:

- Establecer guías de diseño basadas en la IHM y la Ingeniería de software tomando en cuenta la facilidad de aprendizaje.
- Permitir la observación de la ejecución de las aplicaciones paso a paso relacionándolas con el respectivo código.
- Proveer mecanismos para poder realizar una depuración visual de las estructuras de datos dinámicas.
- Proveer una librería para GUI sencilla que simplifique la complejidad del API de Windows en el manejo de ventanas.
- Proveer una librería sencilla de primitivas gráficas.
- Proveer información de ayuda durante el proceso de programación referente a los tipos de datos, variables, funciones disponibles, etc.
- Permitir la ocultación de código dentro del editor para reducir la carga visual y fomentar diseños modulares.
- Asistir e incentivar buenas practicas de programación.

1.3 Justificación

En los programas de estudio de ingeniería en computación los primeros cursos tratan sobre programación estructurada y orientada a objetos, estructuras de datos y lenguajes de programación y establecen bases necesarias para los cursos siguientes sobre tópicos más especializados.

Es en este nivel en el que los estudiantes deben asimilar nuevos conocimientos y fortalecer sus habilidades de abstracción y de resolver problemas utilizando herramientas de programación.

Sin embargo, las herramientas de programación disponibles generalmente no enfatizan el aspecto del aprendizaje y la capacidad de personalizar el nivel de complejidad para así permitir en los primeros cursos el manejo de pequeños programas y a medida que crezca el nivel de experiencia, extenderse para dar soporte a proyectos más avanzados. Además sería conveniente que sin importar el nivel complejidad, la herramienta provea facilidades de manipulación y visualización que sean naturales y fáciles de usar para el usuario.

Las herramientas actuales de programación pueden ser divididas en dos grandes grupos: Aquellas que están orientadas hacia los desarrolladores profesionales y asumen cierto nivel de conocimiento y experiencia, y las que se desarrollan con propósitos de instrucción y son simples de usar pero al mismo tiempo muy limitadas en cuanto a recursos de desarrollo para el programador.

Las primeras se enfocan en mejorar la productividad del programador experimentado, la eficiencia del código, el soporte a librerías y otros aspectos tecnológicos. Las segundas, que generalmente son producidas por pequeñas compañías de software y/o universidades, están orientadas a simplificar el interfaz e incluso a proveer interactividad para facilitar el aprendizaje pagando el precio de imponer ciertas limitaciones como por ejemplo el uso de lenguajes no actuales o propietarios o el uso de un interfaz que no aprovecha los GUI modernos de la mejor manera y que generalmente no proveen facilidades de visualización o depuración similares a las ofrecidas por herramientas profesionales.

Este vacío en la oferta de herramientas de desarrollo afecta al aprendizaje de los estudiantes novatos debido a que pueden presentarse los siguientes escenarios.

1) Si se opta por una herramienta orientada hacia los desarrolladores profesionales entonces se corre el riesgo de que el estudiante pueda enfocarse demasiado en el aprendizaje de la herramienta y no en aprender a resolver problemas usando la misma.

2) Por otro lado, el uso de una herramienta de instrucción puede producir en el estudiante una experiencia incompleta dado que este tipo de herramientas de programación que no proveen mecanismos para manejar eficientemente el proceso de programación o no explotan los beneficios que proveen los GUI modernos debido a la falta de disponibilidad de mecanismos sencillos para hacerlo.

CAPÍTULO 2

2. ENTORNOS INTEGRADOS DE DESARROLLO

2.1 Componentes de un Entorno Integrado de Desarrollo

Un Entorno Integrado de Desarrollo es un conjunto de herramientas orientadas hacia la programación que se integran bajo la misma interfase de usuario. Normalmente y no de manera exclusiva un IDE esta compuesto por un editor de texto o código, un compilador, un depurador, herramientas para diseñar interfaces de usuario y herramientas para editar y visualizar gráficos. La forma más simple de un IDE consiste en un editor y un compilador o interprete.

Un editor de código en un IDE es en principio un editor de texto ASCII sin formato que además brinda soporte a la tarea de

desarrollo manejando de manera automática tareas como el sangrado (indentación) del código, agregando información visual por medio de colores en palabras claves, subrayando errores de sintaxis o prestando ayuda contextual basada en la lógica de la aplicación en desarrollo.

El compilador es un programa traductor que convierte un programa fuente escrito en algún lenguaje de programación a otro lenguaje destino. Se suele llamar lenguaje objeto al leguaje destino. Como ejemplos tenemos los compiladores de C++, que convierte código fuente en C++ a lenguaje de maquina o los compiladores de JAVA convierte código fuente en escrito en JAVA a bytecodes.

Un depurador es una herramienta de software que permite ejecutar, examinar y modificar el flujo de control de otra aplicación en un ambiente controlado con el objetivo de encontrar defectos y errores que no pudieron ser detectados durante la compilación. Un depurador puede interrumpir un programa durante su ejecución (por medio de un punto de ruptura) y observar el estado del mismo. Por estado nos referimos a la instrucción actual, el valor de las variables, y la pila de llamadas. A más de observar el estado actual del programa un depurador permite ejecutar paso a paso las

instrucciones y observar los cambios en el estado del programa. Los depuradores modernos tienen la capacidad de poner puntos de ruptura programables (interrumpen la ejecución del programa después de cierto numero de iteraciones o cuando alguna variable cambie de valor) y controlan aplicaciones con más de un hilo de ejecución.

Para que sea posible la controlar el depurador, el IDE debe ser capaz de interpretar la salida del depurador. Esta tarea resulta compleja dado que en su mayoría los depuradores de línea de comandos no poseen una interfase programable, solamente presentan cadenas de texto para ser leídas por un usuario.

La mejor manera de resolver el problema del procesamiento de la salida del depurador consiste en encontrar un conjunto de reglas que describa de manera general y abstracta todas las posibles salidas del depurador. Una vez determinado el conjunto de reglas es posible implementar componentes que validen estas reglas. El proceso mencionado anteriormente se denomina parsing y se describe más adelante en detalle en la sección 2.4.

Un ejemplo de un IDE es KDEvelop (http://www.kdevelop.org). KDEvelop se ejecuta sobre la plataforma Linux, utiliza a GCC (http://www.gcc.org) como compilador de C/C++ y GDB (http://www.gdb.org) como depurador.

En el contexto de este proyecto de tesis es necesario indicar un componente adicional, el mismo que sirve como ayuda a la depuración y compresión de la lógica de la aplicación. El componente en cuestión esta en encargado de presentar de manera visual la estructura y relaciones de las diferentes variables en tiempo de depuración. Permitiendo así su exploración y manipulación.

2.2. Interacción de los Entornos Integrados de Desarrollo con el Compilador

Los compiladores son en su mayoría herramientas con una interfase basada en texto. La figura 2.1 muestra el proceso de compilación de un programa escrito en C utilizando gcc o de cualquier compilador de manera general. En este caso el compilador toma como argumentos el nombre del archivo a compilarse el nombre del archivo que debe producir y un conjunto de argumentos indican como se desea compilar.

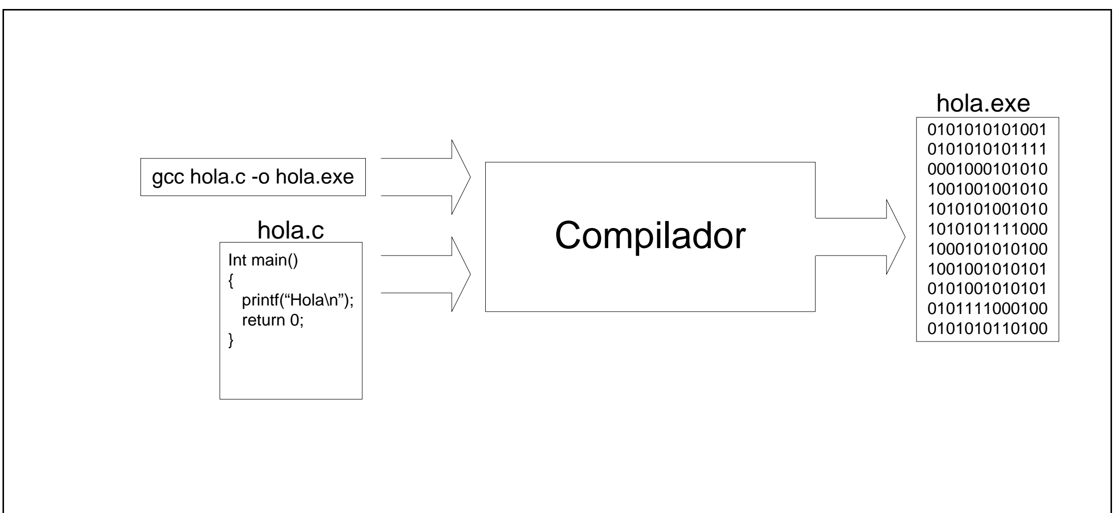


Figura 2.1. Modelo de funcionamiento de un compilador

Cualquier error de compilación se presentara en forma de texto en la misma consola donde se invoco al compilador. Se presenta un error por línea, conteniendo información tal como código de error, nombre del archivo y numero de línea donde se encontró el error.

El cuadro 1 muestra la captura de pantalla de la compilación de una aplicación con errores en su programación.

```
C:\MinGW\bin\g++.exe "C:\MyProj\error.c" -o
↳ "C:\MyProj\error.exe"  -g3  -Wall  -O0

C:/MyProj/error.c: In function `int main(int, char**)':
C:/MyProj/error.c:8: warning: comparison between signed and
↳unsigned integer expressions

C:/MinGW/include/stdio.h:245: too few arguments to function
↳ `char* gets(char*)'
C:/MyProj/error.c:9: at this point in file
C:/MyProj/error.c:13: parse error before `return'
```

Cuadro 1. Compilación con errores

En caso de que se necesiten varios archivos para generar la aplicación, los nombres de tales archivos deben ser pasados como argumentos al compilador. Este número puede llegar a ser considerable. Cabe recalcar que el caso presentado en la captura de pantalla anterior es el caso más simple de compilación, de manera habitual se especifican un mayor número de argumentos y estos pueden variar de archivo a archivo.

Para manejar este problema se diseñaron métodos para controlar la construcción del código (build en inglés). El método más antiguo comprende el uso de archivos makefile y la utilería make. Si

embargo estas siguen siendo herramientas con una interfase basada en texto y presentan todos los problemas de interacción de una herramienta de esta clase.

Tomemos como ejemplo un guión de make, en este caso uno guión específico para la herramienta make de Microsoft:

```
 1 # Microsoft Developer Studio Generated NMAKE File, Based on
    GetDirSize.dsp
 2 !IF "$(CFG)" == ""
 3 CFG=GetDirSize - Win32 Debug
 4 !MESSAGE No configuration specified. Defaulting to GetDirSize
    ↳- Win32 Debug.
 5 !ENDIF

 6 !IF "$(CFG)" != "GetDirSize - Win32 Release" && "$(CFG)" !=
    ↳"GetDirSize - Win32 Debug"
 7 !MESSAGE Invalid configuration "$(CFG)" specified.
 8 !MESSAGE You can specify a configuration when running NMAKE
 9 !MESSAGE by defining the macro CFG on the command line. For
    ↳example:
10 !MESSAGE
11 !MESSAGE NMAKE /f "GetDirSize.mak" CFG="GetDirSize - Win32
    ↳Debug"
12 !MESSAGE
13 !MESSAGE Possible choices for configuration are:
14 !MESSAGE
15 !MESSAGE "GetDirSize - Win32 Release" (based on "Win32 (x86)
    ↳Console Application")
16 !MESSAGE "GetDirSize - Win32 Debug" (based on "Win32 (x86)
    ↳Console Application")
17 !MESSAGE
18 !ERROR An invalid configuration is specified.
19 !ENDIF

20 !IF "$(OS)" == "Windows_NT"
21 NULL=
22 !ELSE
23 NULL=null
24 !ENDIF

25 CPP=cl.exe
```

```
26  RSC=rc.exe

27  !IF  "$(CFG)" == "GetDirSize - Win32 Release"

28  OUTDIR=.\Release
29  INTDIR=.\Release
30  # Begin Custom Macros
31  OutDir=.\Release
32  # End Custom Macros

33  ALL : "$(OUTDIR)\GetDirSize.exe"


34  CLEAN :
35  -@erase "$(INTDIR)\GetDirSize.obj"
36  -@erase "$(INTDIR)\GetDirSize.pch"
37  -@erase "$(INTDIR)\StdAfx.obj"
38  -@erase "$(INTDIR)\vc60.idb"
39  -@erase "$(OUTDIR)\GetDirSize.exe"

40  "$(OUTDIR)" :
41  if not exist "$(OUTDIR)/$(NULL)" mkdir "$(OUTDIR)"

42  CPP_PROJ=/nologo /ML /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D
      ↳"_CONSOLE"    /D    "_MBCS"    /Fp"$(INTDIR)\GetDirSize.pch"
      ↳/Yu"stdafx.h" /Fo"$(INTDIR)\\" /Fd"$(INTDIR)\\" /FD /c
43  BSC32=bscmake.exe
44  BSC32_FLAGS=/nologo /o"$(OUTDIR)\GetDirSize.bsc"
45  BSC32_SBRS= \

46  LINK32=link.exe
47  LINK32_FLAGS=kernel32.lib  user32.lib  gdi32.lib  winspool.lib
      ↳comdlg32.lib     advapi32.lib     shell32.lib     ole32.lib
      oleaut32.lib      ↳uuid.lib      odbc32.lib      odbccp32.lib
      kernel32.lib      user32.lib      ↳gdi32.lib      winspool.lib
      comdlg32.lib     advapi32.lib     shell32.lib      ↳ole32.lib
      oleaut32.lib  uuid.lib  odbc32.lib  odbccp32.lib  ↳/nologo
      /subsystem:console                          /incremental:no
      ↳/pdb:"$(OUTDIR)\GetDirSize.pdb"              /machine:I386
      ↳/out:"$(OUTDIR)\GetDirSize.exe"
48  LINK32_OBJS= \
49  "$(INTDIR)\StdAfx.obj" \
50  "$(INTDIR)\GetDirSize.obj"

51  "$(OUTDIR)\GetDirSize.exe"    :    "$(OUTDIR)"    $(DEF_FILE)
      ↳$(LINK32_OBJS)
52  $(LINK32) @<<
53  $(LINK32_FLAGS) $(LINK32_OBJS)
54  <<

55  !ELSEIF  "$(CFG)" == "GetDirSize - Win32 Debug"

56  OUTDIR=.\Debug
```

```
57 INTDIR=.\Debug
58 # Begin Custom Macros
59 OutDir=.\Debug
60 # End Custom Macros

61 ALL : "$(OUTDIR)\GetDirSize.exe"


62 CLEAN :
63 -@erase "$(INTDIR)\GetDirSize.obj"
64 -@erase "$(INTDIR)\GetDirSize.pch"
65 -@erase "$(INTDIR)\StdAfx.obj"
66 -@erase "$(INTDIR)\vc60.idb"
67 -@erase "$(INTDIR)\vc60.pdb"
68 -@erase "$(OUTDIR)\GetDirSize.exe"
69 -@erase "$(OUTDIR)\GetDirSize.ilk"
70 -@erase "$(OUTDIR)\GetDirSize.pdb"

71 "$(OUTDIR)" :
72 if not exist "$(OUTDIR)/$(NULL)" mkdir "$(OUTDIR)"

73 CPP_PROJ=/nologo /MLd /W3 /Gm /GX /ZI /Od /D "WIN32" /D
  ↳"_DEBUG" /D "_CONSOLE" /D "_MBCS"
  ↳/Fp"$(INTDIR)\GetDirSize.pch" /Yu"stdafx.h"
  /Fo"$(INTDIR)\\" ↳/Fd"$(INTDIR)\\" /FD /GZ /c
74 BSC32=bscmake.exe
75 BSC32_FLAGS=/nologo /o"$(OUTDIR)\GetDirSize.bsc"
76 BSC32_SBRS= \

77 LINK32=link.exe
78 LINK32_FLAGS=kernel32.lib user32.lib gdi32.lib winspool.lib
  ↳comdlg32.lib advapi32.lib shell32.lib ole32.lib
  oleaut32.lib ↳uuid.lib odbc32.lib odbccp32.lib
  kernel32.lib user32.lib ↳gdi32.lib winspool.lib
  comdlg32.lib advapi32.lib shell32.lib ↳ole32.lib
  oleaut32.lib uuid.lib odbc32.lib odbccp32.lib ↳/nologo
  /subsystem:console /incremental:yes
  ↳/pdb:"$(OUTDIR)\GetDirSize.pdb" /debug /machine:I386
  ↳/out:"$(OUTDIR)\GetDirSize.exe" /pdbtype:sept
79 LINK32_OBJS= \
80 "$(INTDIR)\StdAfx.obj" \
81 "$(INTDIR)\GetDirSize.obj"

82 "$(OUTDIR)\GetDirSize.exe" : "$(OUTDIR)" $(DEF_FILE)
  $(LINK32_OBJS)
83 $(LINK32) @<<
84 $(LINK32_FLAGS) $(LINK32_OBJS)
85 <<

86 !ENDIF

87 .c{$(INTDIR)}.obj::
88 $(CPP) @<<
```

```
 89  $(CPP_PROJ) $<
 90  <<

 91  .cpp{$(INTDIR)}.obj::
 92  $(CPP) @<<
 93  $(CPP_PROJ) $<
 94  <<

 95  .cxx{$(INTDIR)}.obj::
 96  $(CPP) @<<
 97  $(CPP_PROJ) $<
 98  <<

 99  .c{$(INTDIR)}.sbr::
100  $(CPP) @<<
101  $(CPP_PROJ) $<
102  <<

103  .cpp{$(INTDIR)}.sbr::
104  $(CPP) @<<
105  $(CPP_PROJ) $<
106  <<

107  .cxx{$(INTDIR)}.sbr::
108  $(CPP) @<<
109  $(CPP_PROJ) $<
110  <<


111  !IF "$(NO_EXTERNAL_DEPS)" != "1"
112  !IF EXISTS("GetDirSize.dep")
113  !INCLUDE "GetDirSize.dep"
114  !ELSE
115  !MESSAGE Warning: cannot find "GetDirSize.dep"
116  !ENDIF
117  !ENDIF


118  !IF "$(CFG)" == "GetDirSize - Win32 Release" || "$(CFG)" ==
       ↳"GetDirSize - Win32 Debug"
119  SOURCE=.\GetDirSize.cpp

120  "$(INTDIR)\GetDirSize.obj"     :     $(SOURCE)     "$(INTDIR)"
       ↳"$(INTDIR)\GetDirSize.pch"


121  SOURCE=.\StdAfx.cpp

122  !IF  "$(CFG)" == "GetDirSize - Win32 Release"

123  CPP_SWITCHES=/nologo /ML /W3 /GX /O2 /D "WIN32" /D "NDEBUG"
       ↳/D  "_CONSOLE"  /D  "_MBCS"  /Fp"$(INTDIR)\GetDirSize.pch"
       ↳/Yc"stdafx.h" /Fo"$(INTDIR)\\" /Fd"$(INTDIR)\\" /FD /c
```

```
124 "$(INTDIR)\StdAfx.obj"  "$(INTDIR)\GetDirSize.pch"        :
     ↳$(SOURCE) "$(INTDIR)"
125 $(CPP) @<<
126 $(CPP_SWITCHES) $(SOURCE)
127 <<


128 !ELSEIF  "$(CFG)" == "GetDirSize - Win32 Debug"

129 CPP_SWITCHES=/nologo /MLd /W3 /Gm /GX /ZI /Od /D "WIN32" /D
     ↳"_DEBUG"       /D       "_CONSOLE"       /D       "_MBCS"
     ↳/Fp"$(INTDIR)\GetDirSize.pch"                /Yc"stdafx.h"
     /Fo"$(INTDIR)\\" ↳/Fd"$(INTDIR)\\" /FD /GZ  /c

130 "$(INTDIR)\StdAfx.obj"  "$(INTDIR)\GetDirSize.pch"        :
     ↳$(SOURCE) "$(INTDIR)"
131 $(CPP) @<<
132 $(CPP_SWITCHES) $(SOURCE)
133 <<


134 !ENDIF
135 !ENDIF
```

Cuadro 2. Ejemplo de un Archivo Make

Un guión de make contiene un conjunto de reglas sobre como construir una aplicación. Para darle mayor flexibilidad y hacer más simple el mantenimiento de un guión se emplean variables. Existen dos tipos de variables, las primeras son las definidas por el programador y las predefinidas por make. El lenguaje de guiones make soporta instrucciones condicionales para poder construir diferentes versiones de la aplicación.

La interacción con el usuario se efectúa por medio de instrucciones !MESSAGE, las cuales presentan texto en la consola, se pueden observar a lo largo de todo el guión.

En la línea 20 tenemos una instrucción condicional que pregunta si una variable predefinida esta definida como "Windows NT".

La línea 40 la declaración de una variable definida por el programador. Las declaraciones de variables simplifican el mantenimiento un guión debido a que una vez definidas se pueden usar en lo que resta del guión. En el caso de la línea 40 se definen los parámetros generales que se usaran ara compilar los archivos fuentes.

Otro componente importante de los guiones make son los objetivos (targets en ingles). Por target entendemos un conjunto de acciones adicionales o diferentes a las que se ejecutan por defecto en un guión. La línea 60 define un target, denominado CLEAN. Este target especifica que se debe borrar ciertos archivos producidos durante una compilación anterior. Más adelante se presenta un ejemplo de invocación un guión make donde se especifica el target

Las fuentes que se van a compilar están especificadas como reglas. En el guión de ejemplo encontramos un caso de regla entre las líneas 87 y 90. Esta regla indica que todos los archivos terminados en .c que se encuentran en la misma directorio que el archivo make se deben compilar a archivos .obj que deberán estar en la carpeta intermedia usando el compilador especificado en la variable $(CPP) y con los argumentos especificados en la variable $(CPP_PROJ).

El guión anterior esta guardado bajo el nombre de GetDirSize.mak. A continuación se muestra la manera como se invocaría el guión:

```
nmake –f GetDirSize.mak
```

O si deseamos otro target:

```
nmake –f GetDirSize.mak CLEAN
```

La siguiente captura de pantalla muestra la ejecución exitosa del guión make de ejemplo:

```
F:\GetDirSize>nmake /f GetDirSize.mak

Microsoft (R) Program Maintenance Utility Version 7.10.3077
Copyright (C) Microsoft Corporation.  All rights reserved.

No configuration specified. Defaulting to GetDirSize - Win32 Debug.
        if not exist ".\Debug/" mkdir ".\Debug"
        cl.exe @C:\DOCUME~1\andrey\LOCALS~1\Temp\nmAD.tmp
StdAfx.cpp
        cl.exe @C:\DOCUME~1\andrey\LOCALS~1\Temp\nmAE.tmp
Skipping... (no relevant changes detected)
StdAfx.cpp
        cl.exe @C:\DOCUME~1\andrey\LOCALS~1\Temp\nmAF.tmp
GetDirSize.cpp
        link.exe @C:\DOCUME~1\andrey\LOCALS~1\Temp\nmB0.tmp
LINK : warning LNK4224: /PDBTYPE is no longer supported;  ignored

F:\GetDirSize>
```

Cuadro 3. Compilación usando nmake

Los guiones make, además de agrupar loas archivos de código fuente y los argumentos necesarios par compilar una aplicación, simplifican tareas tales como la construcción automática de las aplicaciones por la noche (night builds en ingles) o cuando se esta

compilado una aplicación cuyas fuentes están en otra maquina y estamos conectados remotamente por medio de una terminal.

Un IDE provee una alternativa a las aplicaciones basadas en línea de comando al trabajar como front-end del compilador o de make. Haciendo uso de un compilador ya existente, el IDE pasa los argumentos al compilador por la línea de comandos y redirige la salida estándar del compilador para que sea capturada por el IDE y, proveer retroalimentación al usuario en la interfase del IDE.

Redirección de Entrada y Salida:

En sistemas operativos como DOS, Windows, UNIX y sus variantes todos los procesos poseen tres canales de entrada/salida disponibles. Estos canales son entrada estándar, salida estándar y error estándar. En aplicaciones con interfase de texto estos tres canales se mapean de la siguiente manera: la salida y error estándar se conectan a la pantalla y la entrada estándar al teclado. Estos canales tienen un comportamiento de FIFO orientado a caracteres.

Para redirigir la entrada/salida, en primer lugar se necesita una referencia al proceso, como por ejemplo el identificador del proceso.

Posteriormente usando cualquiera de las interfaces provistas por el sistema operativo se redirigen las entradas y las salidas hacia un archivo o alguna otra abstracción (pipes por ejemplo).

Es importante mencionar que a pesar de que el concepto de redirección de la entrada y salida es común para los sistemas operativos antes mencionados la mecánica del proceso difiere significativamente entre ellas.

En el caso de un IDE, el compilador es ejecutado por el IDE y se envía la línea de comando necesaria para compilar la aplicación. Luego, utilizando la redirección de entrada/salida, la salida es redirigida para presentar en el IDE el resultado de la compilación.

En la figura 2.2 se muestran los pasos anteriormente descritos.

Figura 2.2. Diagrama de Redirección de I/O del Compilador

Como un valor agregado, el IDE puede procesar la salida del compilador para darle estructura (filtrar los errores) o para permitir saltar directamente al archivo fuente que se presente en la salida del compilador.

2.3 Interacción de los Entornos Integrados de Desarrollo con el Depurador

Un depurador es al igual que un compilador una aplicación con una interfase de texto, pero a diferencia del compilador, un depurador opera de manera interactiva con el usuario.

La figura 2.3 muestra las interacciones de un depurador.

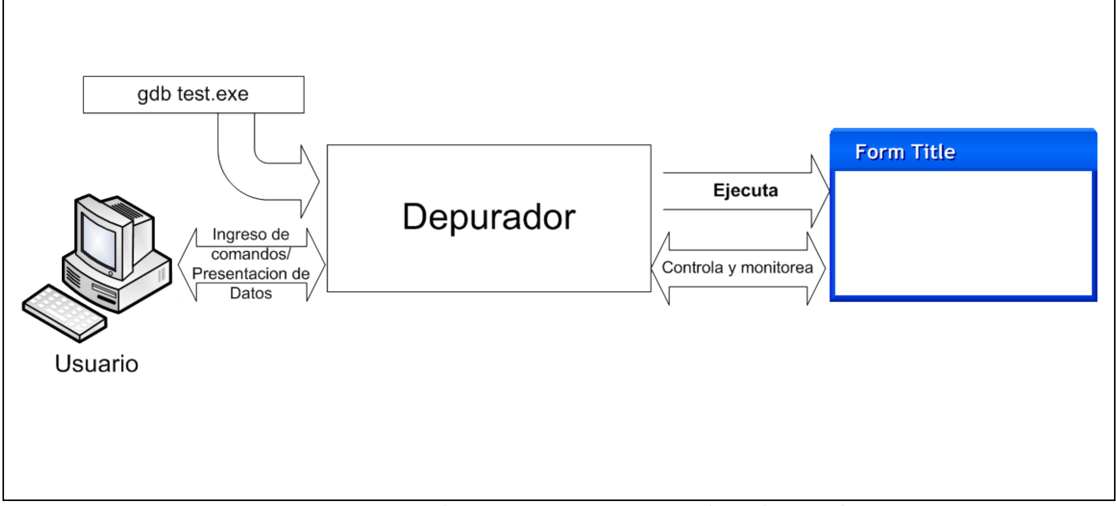


Figura 2.3. Modelo de funcionamiento de un depurador

Un depurador soporta instrucciones para interrumpir la ejecución, inspeccionar las variables y para controlar el flujo de control de la aplicación en depuración.

La siguiente captura de pantalla muestra una sesión de depuración utilizando gdb.

a) En al línea 1 se invoca el depurador pasando como argumento la aplicación que se desea depurar.

b) De la línea 2 a la 8 tenemos los mensajes de licencia, garantía y configuración del depurador

c) En la línea 9 insertamos un punto de ruptura en la línea 17 del archivo main.cpp. Es necesario conocer el nombre de los archivos fuente para insertar un punto de ruptura, debido a que el depurador no muestra cuales son los archivos usados para compilar esta aplicación.

d) La línea 10 muestra la confirmación de haber insertado exitosamente el punto de ruptura.

e) En la línea 11 pedimos que se comience a ejecutarse la aplicación que queremos depurar.

f) La línea 13, 14 y 15 indican que el programa depurado alcanzó un punto de ruptura. Se muestra automáticamente la línea del archivo fuente donde se detuvo la aplicación.

g) En las líneas 16 y 17 presentamos el valor de una variable de la aplicación.

h) En la línea 18 pedimos ejecutar la instrucción actual del programa depurado y luego continuar detenidos.

i) En la línea 22 se pide la pila de llamadas actual. El resultado se muestra en la línea 23 y presenta la función llamada, el archivo fuente donde esta definida y los argumentos pasados.

j) La instrucción list 17, en la línea 24, presenta el código fuente alrededor de la línea 17 del archivo fuente donde nos encontramos detenidos.

k) En la línea 35 pedimos que la aplicación depurada continúe ejecutándose hasta que termine o alcance otro punto de ruptura.

l) La línea 37 indica que la aplicación termino exitosamente.

m) Terminamos la sesión de depuración saliendo del depurador con el comando ‘quit’.

```
 2   C:\MyProj\test>gdb test.exe
 3   GNU gdb 5.1.1 (mingw experimental)
 4   Copyright 2002 Free Software Foundation, Inc.
 5   GDB is free software, covered by the GNU General Public License, and
     you are
 6   welcome to change it and/or distribute copies of it under certain
     conditions.
 7   Type "show copying" to see the conditions.
 8   There is absolutely no warranty for GDB.  Type "show warranty" for
     details.
 9   This GDB was configured as "mingw32"...
10   (gdb) break main.cpp:17
11   Breakpoint 1 at 0x4012cf: file C:/MyProj/test/main.cpp, line 17.
12   (gdb) run
13   Starting program: C:\MyProj\test/test.exe
14   Breakpoint 1, main (argc=1, argv=0x3d2cf0)
15   at C:/MyProj/test/main.cpp:17
16   buff[0] = 50;
17   (gdb) p buff[0]
18   $1 = 0
19   (gdb) step
20   buff[0] += 70;
21   (gdb) p buff[0]
22   $2 = 50
23   (gdb) backtrace
24   #0  main (argc=1, argv=0x3d2cf0) at C:/MyProj/test/main.cpp:18
25   (gdb) list 17
26   {
27   int* buff = new int[50];
28   14
29   memset(buff,0,sizeof(int)*50);
30   16
31   buff[0] = 50;
32   buff[0] += 70;
33   19
34   int otroBuff[100];
35   memset(otroBuff,20,sizeof(int)*100);
36   (gdb) continue
37   Continuing.

38   Program exited normally.
39   (gdb) quit
```

Cuadro 4. Sesión de Depuración en una Consola

Como se puede observar en el ejemplo anterior, la salida de un depurador de línea de comandos, como lo es GDB, es apta para la

lectura por un ser humano, mas la depuración en línea de comandos dista de ser intuitiva.

Otro inconveniente de los depuradores de línea de comandos es que los mismos no muestran las fuentes del programa de manera automática. Para ver las fuentes es necesario invocar un comando que presenta cierto número de líneas por encima y debajo del actual punto de ejecución de la aplicación. Un IDE compensa esta deficiencia al recordar el punto de ejecución de la aplicación, al mismo tiempo presentando el archivo y la línea donde se encuentra detenida la aplicación. El punto de ejecución de la aplicación es actualizado por la salida del depurador.

Redirección de la entrada y la salida

De forma similar a la redirección de la salida del compilador, el IDE redirige la entrada y la salida del depurador para presentar una interfase amigable al usuario.

La figura 2.4 muestra los la redirección de la entrada y salida del depurador que ejecuta un IDE.

El IDE genera las cadenas de texto de los comandos como si hubieran sido escritos por un usuario y las dirige a la entrada estándar del depurador para que sean ejecutadas. La salida estándar redirigida del depurador es leída constantemente en espera datos para ser procesados.

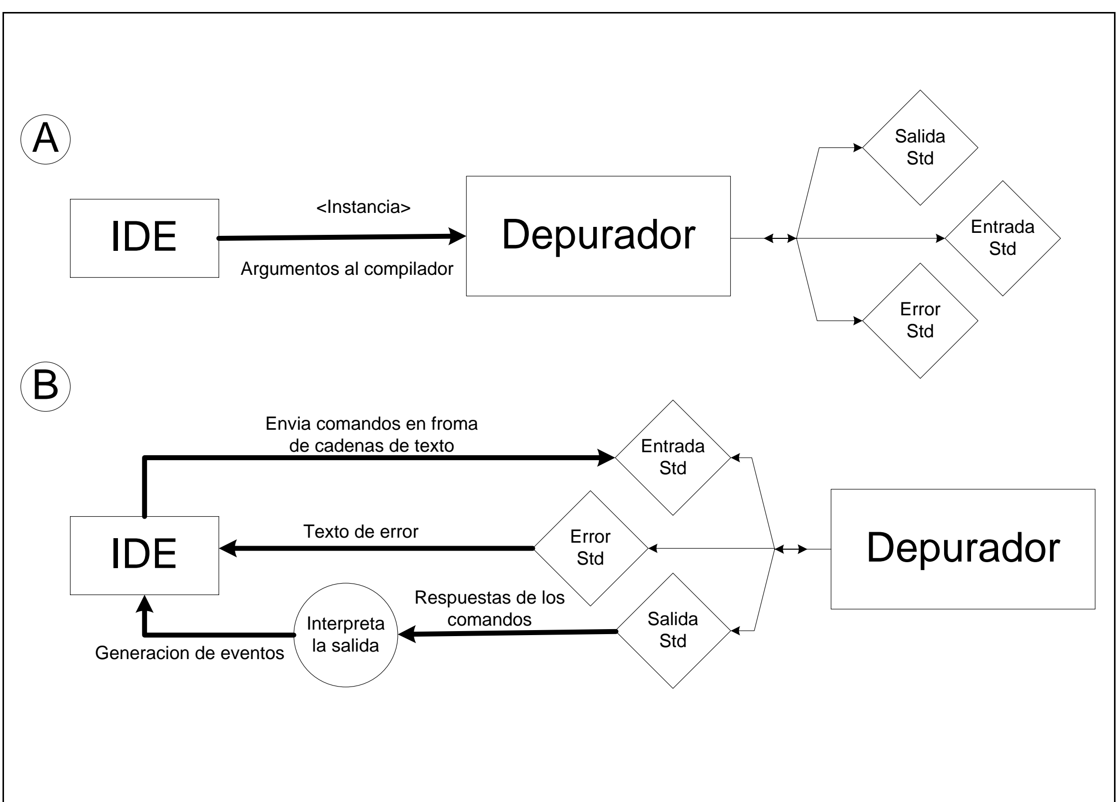


Figura 2.4. Diagrama de Redirección de I/O del Depurador

Como se menciono cerca del final de la sección 2.2 “Componentes de un Entorno Integrado de Desarrollo” el IDE ejecuta un proceso denominado parsing sobre esta salida para generar eventos, obtener datos y actualizar su interfase para presentar cambios sobre el estado de la sesión de depuración.

La complejidad del parser de la salida del depurador esta en función de la calidad de la documentación de la salida y su estructura. Afortunadamente, hoy por hoy, los depuradores de consola son diseñados pensando en la posibilidad de ser usados como back-ends de otras aplicaciones y pueden darle un formato más apropiado a la presentación del texto de salida para que sea descompuesto por otra aplicación.

2.4 Gramáticas, Lexing y Parsing

El parsing es una técnica útil para la construcción de procesadores de datos que se encuentren en formato de texto ASCII. El parsing permite interpretar cadenas de texto y validar todos los casos posibles, dando estabilidad al componente encargado de procesar el texto.

El lexing o análisis léxico consiste en el conocimiento de patrones de caracteres dentro de una cadena de caracteres. El parsing se apoya en esta técnica para trabajar con elementos definidos de la cadena.

Para comprender de mejor manera el funcionamiento del lexing y del parsing es necesario presentar la teoría sobre la que se basa el parsing. El parsing esta íntimamente relacionado con la teoría de lenguajes de programación y gramáticas de los lenguajes de programación.

Una gramática es una forma de describir un lenguaje. Una gramática esta formada por reglas que especifican como están formadas las “oraciones” del lenguaje. Es a su vez una forma de determinar que es valido o no en una construcción del lenguaje.

Para describir lenguajes se utiliza un metalenguaje, la notación o metalenguaje que usualmente se emplea es la forma extendida Backus-Naur o EBNF (por sus siglas en ingles extended Backus-Naur Form).

La EBNF esta compuesta por símbolos y un conjunto de reglas. Las reglas son también conocidas como productores. Existen dos tipos de símbolos o elementos, los terminales y noterminales.

Los símbolos terminales aparecen en las oraciones del lenguaje en cuestión, por ejemplo en Español se consideran como símbolos terminales las palabras: perro, mover, el, abajo. Se los considera simplemente como cadenas de caracteres y no se le atribuyen ninguna otra característica, tal como el tipo o significado.

Los símbolos noterminales no aparecen explícitamente en las oraciones del lenguaje y son usados para agrupar clases de elementos en un lenguaje. Tomando una vez más el Español como ejemplo, los noterminales podrían ser: los verbos, las conjunciones, los sustantivos, los predicados, etc. Los símbolos noterminales son compuestos por símbolos terminales, noterminales o ambos.

Las reglas de producción en una gramática EBNF se dividen en tres partes: el lado izquierdo, el operador de sobre escritura y el lado derecho. Se utiliza el símbolo de la flecha derecha ($\rightarrow$) para representar el operador de sobre escritura. De manera general las reglas de producción en EBNF se escriben de la siguiente forma:

Lado-Izquierdo $\rightarrow$ Lado-Derecho

La regla se lee: “El Lado-Izquierdo puede ser reescrito como Lado-Derecho”.

Los símbolos noterminales se pueden encontrar tanto en el lado derecho como en el lado izquierdo. Los símbolos terminales se pueden encontrar solo en el lado derecho.

Existen otros elementos de la notación EBNF que simplifica la escritura de gramáticas. Supongamos que tenemos una regla cuyo el lado izquierdo se puede reescribir de más de una manera:

A $\rightarrow$ R

A $\rightarrow$ S

A $\rightarrow$ T

Esto se abrevia de la siguiente forma:

A $\rightarrow$ R | S | T

Se lee: A puede reescribirse como R, S o T.

Cuando se desea escribir una elección entre algunos elementos del leguaje se escribe:

A $\rightarrow$ k(R | S)

Esto se lee: A se puede reescribir como k seguido por R o S.

A $\rightarrow$ kR

A $\rightarrow$ kS

Usualmente en los lenguajes existe repetición de elementos, esta situación es escribe en EBNF como:

Lista_de_Argumentos $\rightarrow$ Argumento

Lista_de_Argumentos $\rightarrow$ Argumento ‘,’ Lista_de_Argumentos

O de forma abreviada:

Lista_de_Argumentos $\rightarrow$ Argumento ‘,’ { Argumento }

Esto se lee: Una lista de argumentos puede tener uno o más argumentos.

Esta capacidad de describir de manera recursiva las gramáticas es una herramienta poderosa y nos da grandes posibilidades para expresar muchos tipos diferentes de gramáticas.

Para ilustrar de mejor manera consideremos la siguiente gramática de una expresión de asignación que pertenece a un lenguaje de manipulación de imágenes:

(P1) ASSIGNMENT_statement → ImageVariableName ‘ =’ ImageExpression

(P2) ImageVariableName → Identifier

(P3) ImageExpression → ImageExpressionElement { ‘+’ ImageExpression } | ImageExpressionElement { ‘-’ ImageExpression } | ‘(’ ImageExpression ’)’

(P4) ImageExpressionElement → ImageFunction | ImageVariableName

(P5) ImageFunction → ImageFunctionName ’(’ ArgumentList ‘)’

(P6) ImageFunctionName → Identifier

(P7) ArgumentList → Argument { ‘,’ Argument }

(P8) ArgumentList → ε

(P9) Argument → Integer | Real | String | ImageVariableName

Cuadro 5. Gramática de Ejemplo

Los símbolos no terminales en esta gramática son:

ASSIGNMENT_statement

ImageVariableName

ImageExpression

ImageExpressionElement

ImageFunction

ImageFunctionName

ArgumentList

Argument

Y los terminales son:

Integer

Real

String

Identifier

Para mantener sencilla la explicación no vamos a definir formalmente los símbolos terminales. Consideramos que Integer es un entero, Real es un número de punto flotante, String es una cadena de caracteres delimitado por comillas dobles e Identifier es una cadena de texto arbitraria.

Basados en la gramática anterior tenemos que la siguiente expresión es valida:

NuevaImagen = AGRANDAR(img1, 30, 30, 50, 50, 1.2) + SECCION(img2, 30, 30, 50, 50)

Mientras que la siguiente expresión no es valida, ya que la gramática no permite que después del carácter '+' continúe un número entero:

OtraImagen = AGRANDAR(img1, 30, 30, 50, 50, 1.2) + 50

Implementación de gramáticas

La implementación de las gramáticas se lleva acabo utilizando maquinas computacionales conocidas como Autómatas de Estado Finito (AEF). Por procesamiento entendemos la verificación de que una entrada de caracteres satisfaga la gramática y la ejecución de acciones predefinidas cuando se encuentran diferentes símbolos de la gramática.

Los AEF se representan usando grafos dirigidos. En el grafo los nodos son los diferentes estados en que se encuentra el autómata.

Los bordes en el grafo están dados por los símbolos terminales y caracteres encontrados en la gramática. Las reglas de producción determinan las transiciones posibles.

Empleando la gramática anterior como base se puede construir el siguiente AEF:

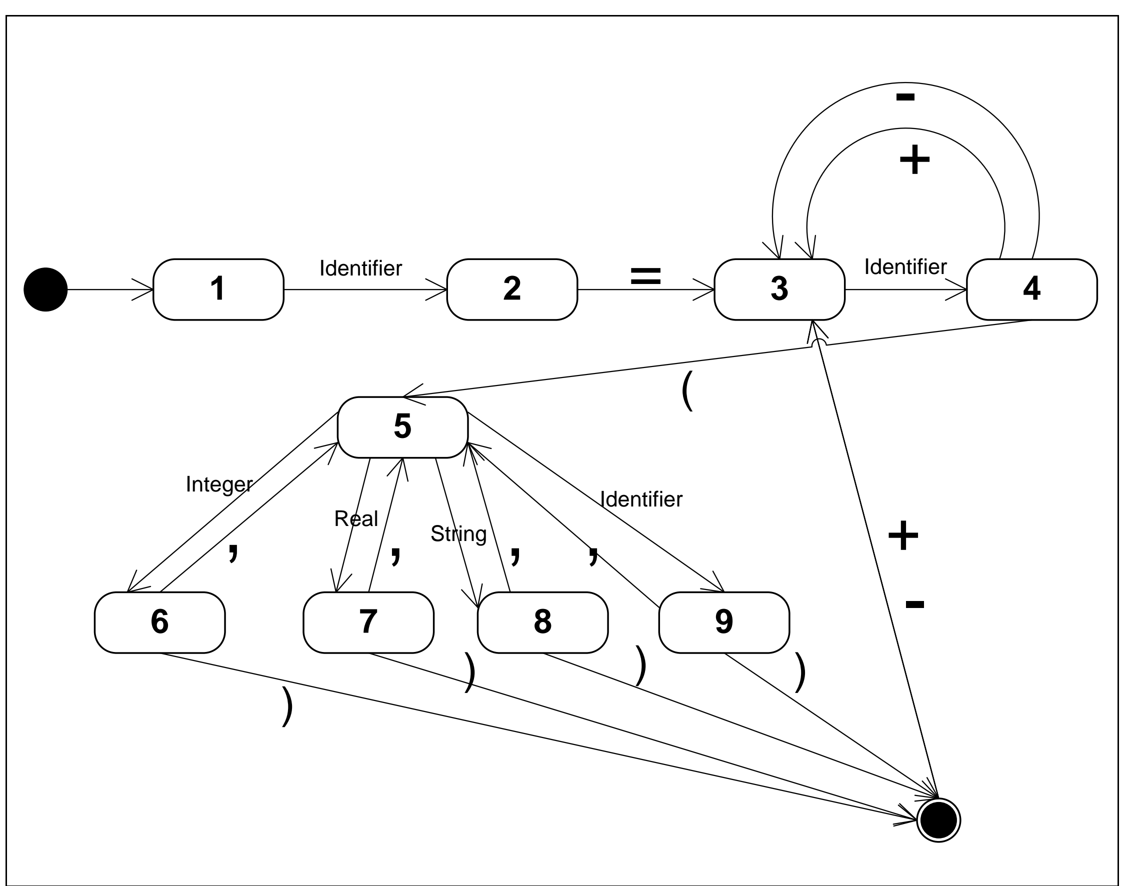


Figura 2.5. AEF de la gramática de ejemplo

El procesamiento de una gramática puede descomponerse en dos etapas. La primera consiste en descomponer la cadena de texto en

tokens, a continuación se procesan estos tokens para darles un valor Sintáctico, es decir determinar si son terminales o noterminales y que regla de producción se puede aplicarles.

El primer proceso se conoce como Análisis Léxico (Lexing en ingles) y el segundo como Análisis Sintáctico (Parsing en ingles).

El análisis léxico consiste en escanear las cadenas de texto y separar subcadenas de texto que representan diferentes elementos terminales de la gramática. Se denomina lexer a un componente de software que realiza el análisis léxico. Un lexer debe retornar la subcadena y el tipo carácter terminal que encontró. Generalmente los lexer operan subordinados al control del parser y son componentes que recuerdan su estado, es decir en que posición de la cadena de texto se encontró el ultimo token, entre una invocación y la siguiente.

El Parsing puede ser considerado como el proceso de seguir las reglas de la gramática. El parser toma los token producidos por el lexer y busca alguna regla de producción para aplicar. Las reglas que se hayan procesado antes o en otras palabras el camino que

haya seguido el parser también se emplean en la tarea de buscar una regla aplicable.

Dependiendo de la gramática del lenguaje, el parser podría necesitar obtener el siguiente o más tokens para poder encontrar correctamente la regla de producción. El último caso produce parser muy complejos. En el caso de C, el parser solo necesita observar el siguiente token para poder encontrar la regla apropiada. Mientras que en C++ el parser puede necesitar observar varios tokens adelante del token actual, haciendo a C++ un lenguaje difícil de procesar y en ocasiones de entender.

Para simplificar la tarea de implementación de un procesador de gramáticas se puede utilizar Lex y YACC, ambas son herramientas de software incluidas en las distribuciones originales de UNIX de AT&T y que ahora se pueden encontrar de manera libre. La abreviación YACC significa Yet Another Compiler Compiler y fue la primera de estas dos herramientas en ser desarrollada. Es una herramienta de línea de comando que toma como entrada un archivo de texto con una gramática EBNF, y produce un parser escrito en C de dicha gramática. Lex es un generador de analizadores léxicos, toma como entrada una lista de expresiones regulares que

representan los tokens que se desean encontrar, produce un Lexer escrito en C que identifica estos token.

El uso de estas herramientas ahorra tiempo en el desarrollo y depuración de un procesador de gramáticas. El mayor problema de estas herramientas es su pronunciada curva de aprendizaje y falta de interfaces intuitivas. Existen alternativas comerciales, sin embargo estas no se encuentran tan difundidas como Lex y YACC, o cualquiera de los clones de las mismas (por ejemplo Flex y Bison).

2.5 Presentación visual de las estructuras

Para comprender de mejor manera la estructura de las aplicaciones y los algoritmos empleados es necesario algo más que una representación tabular de las diferentes variables durante una sesión de depuración. Es deseable obtener una representación grafica de las mismas ya que provee mayor retroalimentación al desarrollador. Un ejemplo de aplicación que permite visualizar las estructuras de datos es el Data Display Debugger o simplemente DDD. El DDD es la aplicación que dio origen a la idea de incorporar características similares en el IDE con el objetivo de mejorar la comprensión de las estructuras de datos y sus algoritmos.

DDD (http://www.ddd.org) es un front-end para varios depuradores que logra la visualización de estructuras de datos y sus relaciones. DDD es producto del trabajo de tesis de Dorothea Krabiell. En 1995. El depurador inicial que operada bajo la cubierta de DDD era GDB aunque actualmente se puede utilizar otros tales como dbx y xdb. DDD se ejecuta sobre Linux utilizando Motif para la interfase de usuario. El componente de visualización de estructuras esta implementado utilizando VSL. VSL es un lenguaje estructurado basado en cajas diseñado por Andreas Zelle en 1990.

La figura 2.6 muestra una captura de pantalla de la ventana de visualización durante una sesión de depuración empleando DDD. Es posible manipular las estructuras cambiando su posición, tamaño y datos mostrados. Las relaciones entre las estructuras se dan a través de apuntadores; los cuales deben ser desreferenciados por el usuario para observar a que apuntan. Para desreferenciar un puntero solo es necesario invocar el menú contextual de algún elemento de la estructura cuyo tipo de dato sea puntero.

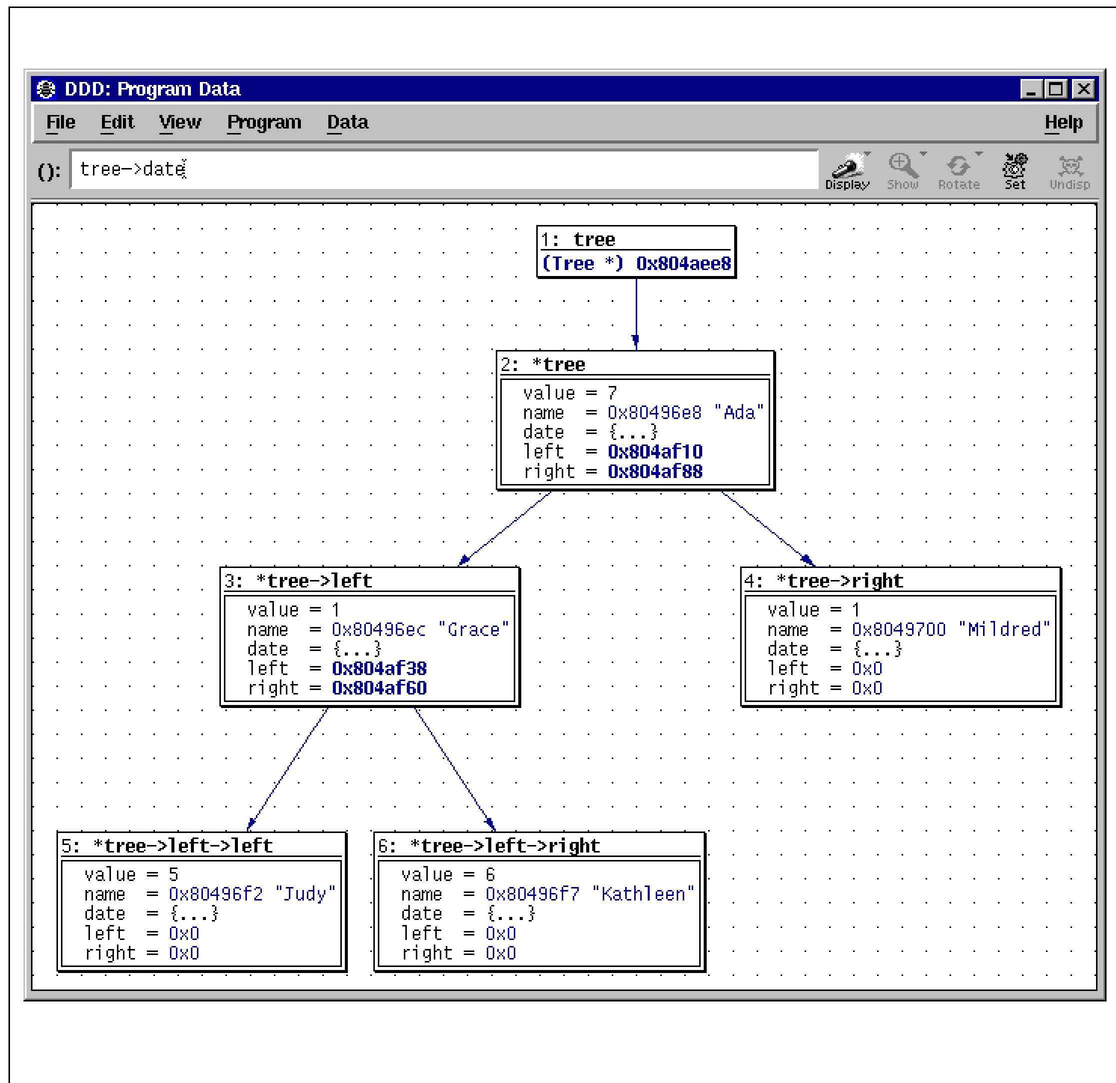


Figura 2.6. Captura de Pantalla de DDD

Es posible, demás de manipular los datos de manera grafica, escribir una expresión que se desea visualizar en una área de texto sobre la ventana de visualización. Es valida cualquier expresión escrita correctamente en C/C++ y que este de acuerdo al contexto actual de la aplicación que se esta depurando (por ejemplo es necesario que

la variable a la que se hace referencia en el expresión escrita aun exista).

A pesar de que este tipo de visualización es bastante mejor en comparación a una representación tabular, aun presenta ciertos inconvenientes de interacción hombre - maquina. El más significativo de ellos es el hecho de que las estructuras no se expanden automáticamente cuando se realizan asignaciones de memoria y es necesario que el usuario seleccione el puntero que desea desreferenciar.

CAPÍTULO 3

3. ANÁLISIS

3.1 Requerimientos Funcionales Del Sistema

El sistema que se desea implementar deberá efectuar lo siguiente:

1) Compilar aplicaciones escritas en C/C++ estándar y depurar las aplicaciones desarrolladas.

Como objetivo esencial el sistema debe permitir compilar aplicaciones escritas en C/C++ usando algún compilador disponible como back-end. El compilador deberá soportar todas las librerías estándares de C/C++. Deberá poder producir tanto aplicaciones de consola como aplicaciones con interfase grafica.

Las aplicaciones de consola se podrán emplear cuando se desee demostrar conceptos usando solamente un interfaz de texto y las herramientas de depuración y no se desee tener que manejar la complejidad añadida de una librería grafica, por más simple que esta sea.

Las aplicaciones con interfase grafica se podrán emplear cuando se desee motivar al estudiante a través de aplicaciones con mayor interactividad y retroalientación visual. Además, es una buena alternativa para proyectos finales por cuanto se dispone de más tiempo para su implementación.

El sistema podrá abrir, modificar y guardar archivos de texto ASCII para ser editados. Deberá ser un editor de texto funcional (incluyendo operaciones que se encuentran en editores estándar como permitir buscar y reemplazar cadenas de texto), ya que no se desea que el usuario se sienta frustrado con el proceso de escritura de código dado que este es el componente con el cual el usuario tiene mayor contacto.

A más de lo descrito anteriormente, el sistema dará la posibilidad de depurar las aplicaciones desarrolladas en el mismo. Las

capacidades de depuración incluyen la inserción de puntos de ruptura, inspección de variables, pila de llamadas de la aplicación y control del flujo de ejecución de la aplicación depurada.

2) Proveer un ambiente integrado de desarrollo

El sistema deberá poseer de un entorno integrado de desarrollo o IDE que permita llevar acabo las tareas descritas en el punto anterior. El IDE deberá integrar las diferentes tareas posibles del sistema bajo una interfase común.

3) Proveer librerías para el manejo del GUI y de primitivas graficas

Otra fuente de frustración para los estudiantes se encuentra en la complejidad de crear interfaces de usuario modernas, y en la manipulación de gráficos 2D. Generalmente las librerías de manejo de GUI y de primitivas graficas de herramientas profesionales tienen como objetivo la flexibilidad, personalización y rendimiento en detrimento de la claridad y la simplicidad. Esto es lamentable ya que este tipo de recursos de programación tienen un efecto motivador en los estudiantes dado que estos podrían crear aplicaciones interactivas y vistosas. Para tener este elemento motivador dentro del entrono de programación se debería construir o encontrar una librería de GUI y de primitivas graficas con una interfase lo

suficientemente clara y sencilla para ser usada por los estudiantes de primeros niveles.

3.2 Factores de Interacción Hombre Maquina

Un elemento muy importante que se busca con el desarrollo del proyecto es brindar una experiencia positiva y agradable a los estudiantes al usar su entorno de desarrollo que facilite el proceso de aprender a programar. Con esta idea en mente debemos determinar características particulares de los usuarios del sistema y definir algunas características que el sistema deberá poseer para lograr nuestro objetivo.

En esta sección se describen características que persiguen el objetivo de proveer al usuario de una experiencia agradable y facilitar el aprendizaje de una herramienta de programación y el uso del IDE. Cabe recalcar que además de los criterios expuestos en esta sección, se busca que el IDE a desarrollar esté construido de manera consistente con elementos conocidos de las aplicaciones modernas sobre el entorno Windows. Características tales como barras de herramientas, menús que recuerdan las selecciones mas frecuentes del usuario, tips informativos entre otras.

3.2.1 Asunciones hechas sobre los usuarios

Además de los aspectos funcionales del sistema debemos considerar que tipo de usuario tendrá el sistema. Se considera muy importante brindar al usuario una experiencia agradable y libre de frustraciones producidas por la herramienta, ya que, por la naturaleza misma de la tarea, el desarrollo de aplicaciones es una actividad que puede ser psicológica y cognitivamente dura.

Los usuarios del sistema serán los estudiantes que se encuentran en los primeros dos años de la carrera de Ingeniería en Computación. Para los mismos el IDE será, probablemente, el primer acercamiento a una herramienta de desarrollo.

Se espera utilizar el sistema como la herramienta de desarrollo por lo menos en los cursos de Fundamentos de Programación y Estructuras de Datos. En lo posible, se desea evitar frustraciones en las primeras experiencias de desarrollo para reducir el rechazo hacia la programación.

Se considera que estos estudiantes han tenido un contacto moderado con la informática. Si bien no podemos asumir que la mayoría de estudiantes posee un computador en su hogar, si podemos asumir debido a la popularidad del sistema operativo Windows que muchos de ellos han tenido alguna exposición a dicho sistema operativo.

Otro supuesto, sobre los estudiantes, es el tipo de aplicaciones empleadas por los mismos. Se considera que muchos estudiantes, han utilizado suites de aplicaciones de oficina (tales como MS Office) y han empleado algún navegador de Internet.

Los estudiantes aun no poseen conocimientos sobre la arquitectura y funcionamiento de las computadoras personales, ni de sus aplicaciones. Este último punto resulta interesante debido a que existen herramientas de desarrollo que asumen cierto tipo de conocimiento previo y no son adecuadas para un novato. Un ejemplo de conocimiento previo es la opción de enlazar con una librería de manera dinámica o estática, para comprender la implicación de esta decisión se debe conocer como el sistema operativo ejecuta las aplicación. Por lo tanto, inicialmente se debe mantener a los estudiantes lejos de este

tipo de decisiones, pasando esta responsabilidad al sistema, pero permitiendo que, más adelante el estudiante tome el control de las opciones de compilación y enlace.

3.2.2 Requerimientos de Usabilidad

Con el objetivo de que el sistema sirva de manera adecuada a usuarios con diferente tipo de experiencia y necesidades se debe permitir alguna capacidad de personalización en la manera de operar el entorno.

El IDE deberá manejar dos niveles de uso de la herramienta, uno para novatos y otro para usuarios con más experiencia

La mayoría de las herramientas en C/C++ manejan el concepto de proyecto que implica la administración de múltiples archivos fuente y recursos para generar un ejecutable. Se consideró que un entorno de desarrollo basado en proyectos es demasiado complejo para un estudiante completamente novato pues implica manejar al inicio muchos elementos que aumentan el conocimiento declarativo. Es más fácil al inicio construir un modelo mental mapeando directamente un archivo a un

programa donde el estudiante pueda inmediatamente probar la teoría de programación aprendida. Con esa idea en mente se planteo un entorno donde el concepto de proyecto no sea obligatorio.

Pero un entorno sin el concepto de proyecto no puede manejar un desarrollo un poco más avanzado que requiera una arquitectura modular. Debido a esto se plantea que la herramienta sea "bimodal". Por bimodal queremos decir una herramienta que trabaje o con el concepto de proyectos para estudiantes avanzados, o con archivos simples para estudiantes novatos.

La idea de tener dos modos de trabajo se refiere también a la cantidad de información que se le presenta al estudiante. Por ejemplo durante una sesión de depuración para un usuario novato, las ventanas que presentan información algo compleja (pila de llamadas, memoria, registro del procesador) están ocultas y la ventana que presenta el valor de las variables de la aplicación no muestra información de tipos de datos o posiciones de memoria. De esta manera también se permite administrar la carga cognitiva del interfaz para permitir que el

conocimiento declarativo presente en el interfaz sea limitado hasta que sea asimilado como conocimiento procedimental.

El sistema además de manejar dos niveles de experiencia deberá brindar algún mecanismo de visualización de estructuras de datos y de algoritmos con el objetivo de mejorar el entendimiento de las mismas. En un entorno de programación orientado a facilitar a los estudiantes la resolución de problemas, es importante incorporar mecanismos para la visualización de las estructuras de datos del programa/problema. Es más fácil entender un tipo de estructura de datos y los algoritmos que trabajan sobre ellas cuando se presentan explícitamente. Además, la visualización de las estructuras de datos debe presentar las estructuras y sus relaciones en tiempo de depuración y actualizarse dinámicamente para reflejar un cambio en el estado de la aplicación. Por este motivo se plantea que la herramienta tenga un mecanismo de visualización de las estructuras de datos en tiempo de depuración.

Otro punto importante de interacción con el usuario es la edición de código. En lo que se refiere a la edición de código existe gran

variedad de criterios de interacción que se pueden observar en productos comerciales similares:

- A medida que los programas crecen en tamaño, su lectura se dificulta y en ciertos momentos puede ser difícil interpretar el ámbito y la lógica de los bloques de código. Una técnica importante de las herramientas modernas de programación es el "doblado de código". Esto tiene dos objetivos, el primero, reducir la carga visual cuando se revisa el código. El segundo objetivo es enseñar a pensar modularmente a manera de caja negra, para que los estudiantes se concentren más en las interfaces de clases, funciones y estructuras de control.

- Otra técnica importante que mejora la claridad de código es el sangrado. El sangrado permite ver el ámbito (scope) en el código de las variables y de las estructuras de control. El sistema deberá manejar la tarea del sangrado de manera automática para evitar que los estudiantes desvíen su atención de lo que se desea que aprendan o

que por falta de costumbre escriban código ilegible sin sangrado alguno.

- El sistema también debería apoyar la edición de código presentando información de ayuda dependiente del contexto. El tipo de información que se debe presentar durante la edición de código consiste en el nombre correcto de variables/funciones y los argumentos de una función. Esta información generalmente es buscada dentro de los archivos de ayuda o de código fuente, tal búsqueda puede interrumpir el tren de pensamiento del estudiante, cosa que se desea evitar.

Finalmente, si se observa los IDEs modernos se puede notar que existe gran cantidad de información que compite por espacio en la pantalla. El sistema que se desea construir podría adolecer del mismo problema. De la mano a este problema esta el hecho de que cierta información es relevante mientras se realiza una tarea y una vez terminada la tarea deja de serlo; por ejemplo las advertencias de compilación no son relevantes durante la depuración. Teniendo en mente la maximización del uso del espacio de pantalla y la relevancia

de la información se deberá diseñar algún mecanismo que permita ocultar información no importante para la tarea en cuestión y que además permita configurarse de acuerdo a las necesidades de la tarea.

3.3 Análisis de estrategias de desarrollo del proyecto

El sistema que se desea construir es un proyecto que puede requerir una cantidad significativa de tiempo debido al número y complejidad de las funciones del mismo. Dado esto es necesario determinar una estrategia que nos permita desarrollar el proyecto en un marco de tiempo aceptable para una tesis de grado.

Para desarrollar el proyecto tenemos dos alternativas: desarrollarlo todo desde cero o encontrar la manera de sacar provecho de componentes escritos por terceros.

La primera estrategia de desarrollo se la considera mala debido a que consumirá una cantidad significativa de tiempo de desarrollo y depuración. Esto se debe a que estimamos que el sistema que se desea construir es de un tamaño y complejidad alta. Y de acuerdo a estudios realizados sobre métricas de software se calcula que un

proyecto con estas características tomaría alrededor de 120 personas/mes en terminarse.

En lo referente a la alternativa de utilizar componentes de terceros, podemos decir que existe una gran oferta de los mismos. Es posible encontrarlos de manera gratuita o como productos comerciales, incluyendo o no su código fuente. Sin embargo es importante mencionar que los componentes de terceros pueden presentar problemas en áreas como la interoperabilidad, dependencias de otras componentes, estabilidad o escalabilidad.

Entre nuestras dos opciones, desarrollo desde cero o uso de componentes de terceros, se opto por la última. Nuestra decisión se basa la imposibilidad de contratar a personal calificado para reducir el tiempo de desarrollo si se seleccionara la primera opción. Asimismo contar con más gente en el proyecto tampoco garantiza que se reducirá el proyecto hasta un margen de tiempo aceptable para una tesis. Además los problemas que se pueden presentar con el uso de componentes de terceros pueden ser controlados, ya sea escribiendo código o simplemente seleccionando otro componente.

Para poder determinar que componentes se van ha emplear necesitamos describir a un nivel conceptual los componentes generales del IDE.

La figura 3.1 tenemos un diagrama de bloques resume de manera general los componentes del IDE y las interacciones de los mismos. En la figura la dirección de las flechas indica el flujo de información y control.

El editor es el componente que permite la escritura y visualización del código fuente. El componente de Datos de Aplicación contiene la información del proyecto de programación en si, es decir que archivos se necesitan para compilar la aplicación y las opciones del compilador. El componente del Entorno es el encargado de integrar bajo la misma interfase de usuario y proveer de soporte a los demás componentes del proyecto. El componente de Visualización de Estructuras será el encargado de crear, administrar y presentar las estructuras de datos del programa cuando se este depurando. La librería grafica es el componente de código que simplifica el uso de ventanas y el dibujo en las aplicaciones desarrolladas en el IDE.

En la figura 3.1, el compilador y el depurador son programas externos. La interfase del compilador y la interfase del depurador están encargadas de interactuar con los programas del compilador y del depurador al generar los comandos en el formato apropiado y procesar las salidas de estos programas.

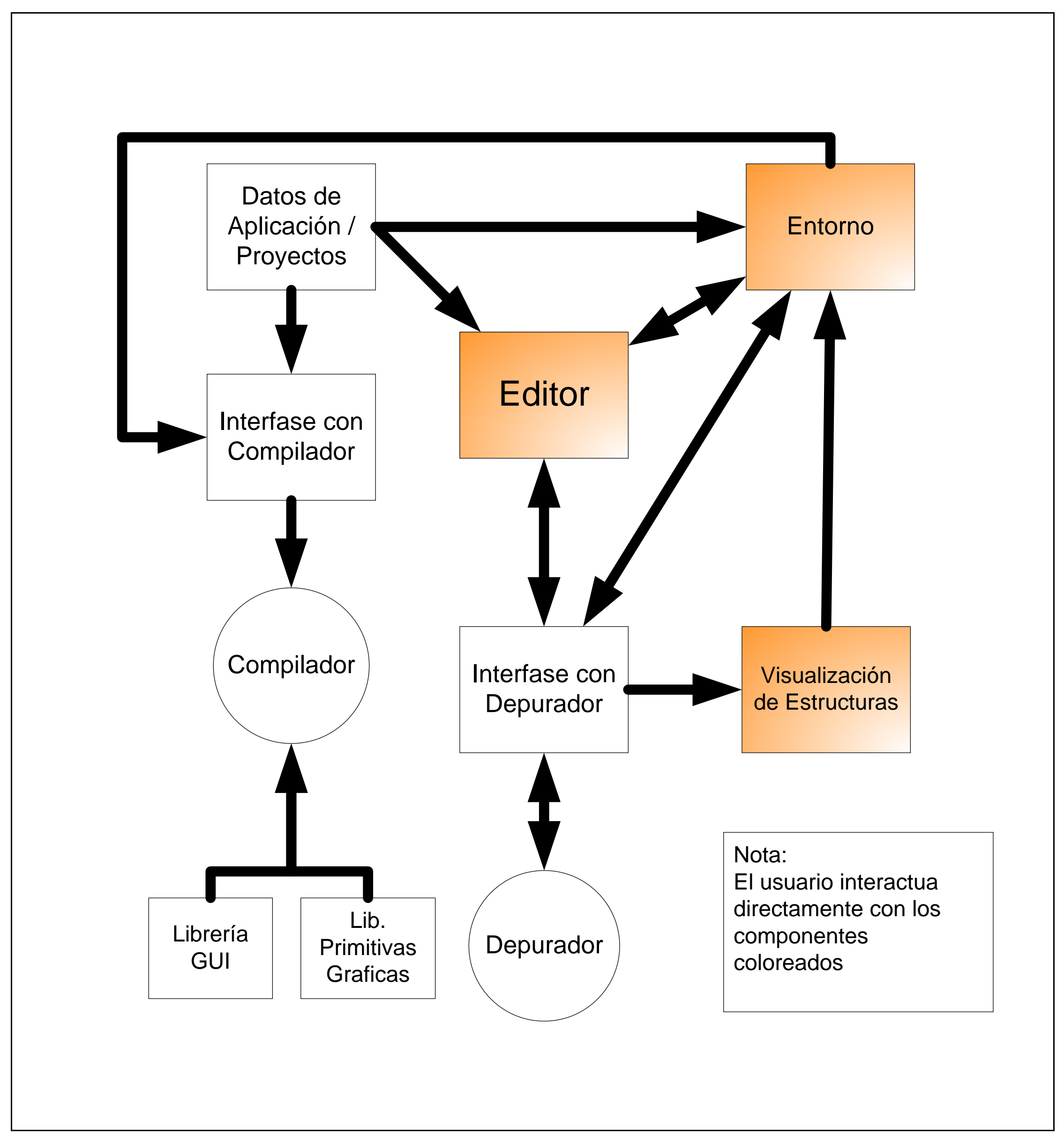


Figura 3.1. Diagrama de Bloques de los Componentes del IDE

El componente de datos de Aplicación, la visualización de estructuras y las interfaces con el compilador y el depurador son demasiado especializados como para que exista algún componente ya desarrollado que nos pueda ser útil. Los componentes restantes del diagrama si pueden ser encontrados ya que son lo suficientemente generales.

El compilador y el depurador son componentes del sistema dado que son necesarios para que el IDE satisfaga los objetivos de permitir compilar y depurar las aplicaciones. Más no se desarrollaran durante el proyecto, en lugar de esto se utilizaran como programas externos. Estos componentes no deberían ser difíciles de encontrar ya que existe una gran oferta de compiladores, comerciales y gratuitos.

El componente del editor, la librería de ventanas y la librería grafica son componentes bastante generales como para que exista una buena probabilidad de encontrarlos ya desarrollados y probados.

3.4 Los componentes del sistema y el Código Abierto

Una vez identificadas las áreas/partes del proyecto y determinados cuales son los componentes lo suficientemente generales como para poder encontrarlos ya desarrollados, tenemos que determinar donde podrían ser obtenidos.

Para nuestro a análisis clasificamos a los componentes de terceros en dos grandes grupos: los componentes desarrollados con fines comerciales y los componentes con origen en el mundo del Código Abierto (mejor conocido como Opensource).

Para este proyecto se decidió emplear componentes encontrados en el mundo del OpenSource debido a que los componentes comerciales poseen varias restricciones tales como costos, esquemas rígidos de licenciamiento, entre otros que dificultan la realización de esta tesis. Por otro lado, al parecer no existe una gran diferencia en factores como calidad o soporte cuando comparamos productos comerciales con sus contrapartes en código abierto.

Existen varios factores negativos que evitan que usemos componentes comerciales. El primer factor negativo que poseen los

componentes comerciales es el costo de licenciamiento. Dado que son desarrollados con fines de lucro no es extraño que las licencias de uso exijan el pago de valores elevados por proyecto o por el número de instalaciones.

El segundo factor es más técnico y consiste en la poca disponibilidad de código fuente. Cabe indicar que por lo general si es posible obtener el código de los componentes comerciales, pero a cambio de un pago extra adicionalmente al valor de la licencia de uso. Asimismo el código fuente de los componentes usualmente se distribuye con fines de depuración, es decir lo podemos leer y compilar mas no lo podemos adaptar, o tomar porciones del mismo para su redistribución.

Previo a la exposición de las razones por las que se tomo la decisión de utilizar componentes de OpenSource es necesario definir que entendemos por OpenSource.

La definición de OpenSource utilizada en este documento es bastante general. Existen diferentes tipos de licencias OpenSource, tales como GNU, GPL, la licencia de freeBSD o del proyecto Apache, pero en este documento se interpreta OpenSOurce como una licencia donde se permita el uso y la modificación de código fuente y no sea

necesario el pago de alguna comisión. Son estas dos características del licenciamiento OpenSource que resultan particularmente útiles en el desarrollo de este proyecto.

Existen otras condiciones en las licencias de tipo OpenSource; sin embargo estas condiciones no crean problemas para el desarrollo del proyecto. Entre tales condiciones encontramos la obligación, dado que el proyecto es un trabajo derivado de un proyecto OpenSource, de distribuir nuestro código fuente junto con los archivos binarios del proyecto.

En general los componentes OpenSource brindan lo opuesto a los componentes comerciales en aspectos como el valor del licenciamiento y el acceso, modificación y distribución de su código fuente. Cabe recalcar que las licencias varían ligeramente de un componente a otro. Por ejemplo, hay tipos de licencias que no permiten la modificación del código, sin embargo mantienen el espíritu de un libre intercambio de conocimiento.

Los cuestionamientos más importantes que se hacen normalmente sobre los componentes OpenSource giran en torno a su estabilidad y

rendimiento. En otras palabras la calidad del componente, y el soporte que existente del mismo.

Generalmente la calidad de un componente de software con origen en un proyecto de OpenSource esta en función de la madurez del proyecto y el interés que exista sobre el componente. Los desarrolladores usualmente tienen un buen nivel de conocimientos y mantienen el proyecto como una actividad en paralelo a un trabajo regular.

La madurez del proyecto asegura que el componente ha sido usado y probado por un tiempo lo suficientemente largo para encontrar errores en el componente o en sus interfases.

El interés de la comunidad sobre el proyecto ayuda a atraer usuarios y desarrolladores que implementan nuevas características y corrigen errores en cortos intervalos de tiempo. El interés del proyecto permite también que los componentes sean escritos de manera que funcionen en varios sistemas operativos, lo cual da bastante flexibilidad a los proyectos que emplean estos componentes.

La calidad de los componentes y el soporte son aspectos que se desean evaluar por medio del desarrollo de este proyecto.

3.5 Análisis y determinación de la plataforma de ejecución

Una vez terminado el análisis del proyecto como tal, queda por determinar la plataforma de ejecución y las herramientas de desarrollo con las que se llevara a cabo el proyecto. Se considera necesario definir estos dos aspectos por que se piensa que tienen un efecto directo sobre la experiencia que el usuario tiene sobre el IDE a desarrollar.

Usualmente, los IDE de programación son implementados como herramientas de escritorio o clientes gruesos (fat client en ingles); sin embargo, dada la popularidad de soluciones implementadas sobre la Web, seria interesante considerar la posibilidad de implementar el IDE sobre la WEB.

Para su análisis, se han diferenciado dos tipos de soluciones Web: Primero, aquellas implementadas sobre algún lenguaje de guión o código ejecutado en el servidor, tal como PHP, Java Servlets o

ASP.Net. Segundo, las soluciones que incorporan código que se ejecuta en al maquina del cliente, tal como los Java Applets o ActiveX.

En la tabla 1 comparamos una solución de escritorio y una solución Web en base a los siguientes parámetros:

Riqueza de Controles de Interfase de Usuario: La capacidad de la plataforma de presentar controles de interfase de usuario ricos en características. Controles que tengan ayudas contextuales, capacidades de personalización, tiempos de respuesta cortos, entre otras características.

Comunicación entre Aplicaciones: El nivel de interacción de una solución desarrollada en esta plataforma con el programa compilado con fines de depuración y ejecución.

Funcionamiento fuera de línea (off line): Indica si una aplicación desarrollada en esta plataforma necesita estar conectada a otra aplicación o a una red de computadoras.

Facilidad de Actualización y mantenimiento: Este parámetro mide la complejidad de la tarea de distribuir actualizaciones y correcciones de la aplicación.

Facilidad de Distribución: Indica si es necesario entregar una copia de la aplicación a los estudiantes, el tamaño del instalador (si existiese) y el tiempo que una actualización tomaría en llegar a los usuarios.

Consumo de Recursos Computacionales: Indica cual es el consumo de memoria, almacenamiento en disco y uso del procesador en la maquina del usuario.

Simplicidad de Instalación: Este parámetro mide el nivel de conocimiento que debería poseer un usuario para poder instalar de manera exitosa la aplicación.

Tipos de aplicaciones generadas: Este parámetro es específico para el IDE. Mide la gama de aplicaciones que se podrían crear con el sistema si el sistema fuera desarrollado sobre esta plataforma. Por ejemplo, si se pueden hacer aplicaciones de consola, con interfase grafica, que manipulen archivos gráficos, reproduzcan sonidos, etc.

	Solución de escritorio	Solución WEB	Solución WEB (Java Applet)
Riqueza de Controles de Interfase de Usuario	Alta	Baja	Alta
Comunicación entre Aplicaciones	Alta	Baja	Baja
Funcionamiento fuera de línea (off line)	Si	No	no
Facilidad de Actualización y mantenimiento	compleja	Simple	simple
Facilidad de Distribución	compleja	Simple	simple
Consumo de Recursos Computacionales	Alto / medio	Bajo	bajo
Simplicidad de Instalación	media	Alta	alta
Tipos de aplicaciones generadas	Variadas, extensibles con el uso de librerías	Limitadas por la interactividad de los productos	Limitadas por la interactividad de los productos

Tabla 1 Comparación entre una solución de escritorio y una solución Web

Basándonos en la comparación hecha sobre las soluciones desarrolladas sobre las diferentes plataformas y nuestros requerimientos decidimos que la mejor opción es una solución basada

en escritorio. Un factor determinante es la necesidad de que el IDE depure de manera interactiva la aplicación compilada. Este factor y la necesidad de funcionar “en línea” eliminaron a las soluciones implementadas con Java Applets como una opción.

Otro factor de decisión fue la posibilidad de crear aplicaciones más ricas en características debido a que se quiere que los estudiantes tengan una buena retroalimentación de su trabajo. Esto se conseguiría con el uso de una solución basada en escritorio. Esto elimina como opción a las soluciones Web implementadas con guiones.

Como ejemplo de una solución basada en Web que no emplea Java applets tenemos a el Web-based Compiler System (http://webcpp.csci.unt.edu/). Esta solución fue desarrollada por la Facultad de Ingeniería y Ciencias de la Computación de la Universidad de North Texas.

A pesar de considerar que una solución basada en escritorio es la más adecuada para nosotros, es necesario mencionar las dificultades que nos presenta. Se presentan problemas en la configuración, instalación y distribución de la aplicación.

La configuración se vuelve compleja dado que existen diferencias en la programación de las diferentes versiones de los sistemas operativos. Como se desea que la aplicación se ejecute en diferentes versiones de la misma plataforma es necesario realizar un esfuerzo extra de programación para asegurar compatibilidad.

El proceso de instalación también se ejecutara en diferente versiones de un sistema operativo, debido a esta circunstancia el instalador deberá resolver conflictos sin intervención del usuario durante la instalación. Conflictos tales como la existencia de las librerías necesarias para el funcionamiento del sistema o la existencia de versiones más antiguas o nueves de las librerías necesarias. Además el proceso de desinstalación deberá remover correctamente las librerías no necesarias siempre y cuando no existan otras aplicaciones instaladas que aun las necesiten.

Debido a los dos factores mencionados anteriormente (la compatibilidad con varias versiones de la misma plataforma de ejecución y los posibles conflictos de versiones de las librerías) es probable que el instalador tenga un tamaño que haga complicada su distribución. El mayor tamaño del instalador afecta asimismo el tiempo

que toma en entregar correcciones y actualizaciones a todos los usuarios

Una vez determinado que una solución basada en escritorio es la más conveniente, es necesario seleccionar el sistema operativo sobre el cual se ejecutara. Se consideraron dos alternativas de sistemas operativos usados comúnmente en el ambiente académico: Windows o Linux.

Para determinar el sistema operativo a emplearse se considero la reducción de la frustración del usuario al inicio del uso de la herramienta como el parámetro más importante.

Las capacidades de interfase de usuario y programación de Linux y Windows son similares. Sin embargo, desde el punto de vista de este proyecto, la diferencia radica en la facilidad y familiaridad de la instalación y configuración de las dependencias de las aplicaciones cuando se instalan. Por ejemplo tomemos el caso de las aplicaciones sobre Linux. En Linux es posible desarrollar interfaces graficas utilizando una gran variedad de librerías (KDE, Qt, GTK, Motif) las cuales no se encuentran estandarizadas en todas las distribuciones de Linux. Luego cuando la aplicación se desea ejecutar en una

computadora que no sea la del desarrollador se deberían encontrar las versiones correctas de estas librerías para que la aplicación se ejecute de manera correcta. La falta de estandarización en las diferentes distribuciones de Linux junto con la posible complejidad de la instalación y actualización de estos componentes de Linux representan una diferencia entre las dos plataformas para nosotros. Cabe recalcar que Windows sufre de problemas similares pero en menor escala ya que existe una interfase común sobre la que nos podemos apoyar y aun obtener buenos resultados de programación.

Además, los usuarios hacia los que está dirigido este sistema están más familiarizados con el uso Windows y no se desea imponer un sistema operativo no familiar que puede causar frustración a los usuarios novatos.

En base a la familiaridad que tienen los usuarios con las aplicaciones en Windows y su instalación se seleccionó a Windows como la plataforma de ejecución de la aplicación.

3.6 Análisis y determinación de las herramientas de desarrollo

Existe un sinnúmero de plataformas a escoger para desarrollar para Windows. Para seleccionar la plataforma de desarrollo consideramos los siguientes parámetros:

- Flexibilidad en la creación de nuevos controles de interfase de usuario
- Poder determinar correctamente cuales son las dependencias que se crean
- Posibilidad de controlar el uso de recursos computacionales
- Cantidad de código fuente y componentes ya escritos que pudiera ser reutilizados
- Madurez de la herramienta

El primer parámetro, flexibilidad en la creación de nuevos controles de interfase de usuario, elimina el uso de herramientas RAD (Rapid Application Development) tales como Delphi o Visual Basic. Las herramientas RAD encapsulan los controles de interfase de usuario provistos por el sistema operativo en componentes fáciles de usar pero complejos o imposibles de modificar. Este problema normalmente es solucionado con componentes de terceros, que usualmente se

desarrollan empleando herramientas que proveen mayor control y flexibilidad y presentan una interfase de programación compatible con la herramienta RAD. Los problemas que presentan los componentes de terceros es la posible falta de código fuente, necesario para realizar modificaciones, la dificultad de determinar dependencias que el componente tiene con otras librerías y posiblemente el valor de una licencia de uso y distribución.

Las aplicaciones desarrolladas en Java no presentan dificultades en la extensión de los componentes de interfase de usuario. Sin embargo, la complicación que trae Java es el control del consumo de recursos y la librería de tiempo de ejecución. Generalmente en un entrono de ejecución donde existe un recolector de basura, tal como en Java, no es posible controlar la destrucción de los objetos empleados y por lo tanto tiende a aumentar el consumo de memoria. Este hecho tiene un efecto adverso en la ejecución de aplicaciones escritas en java sobre computadores con poca memoria instalada.

La alternativa restante es C++ como la plataforma de desarrollo. C++ permite el acceso directo a los controles del sistema operativo, manejo manual de la memoria y un control fino de los múltiples hilos de

ejecución. Cabe señalar que consideramos mejor un entorno donde el manejo de la memoria se haga de manera automática, a pesar de eso uno de nuestros objetivos es construir una aplicación que maneje de manera eficiente los recursos del ordenador, para que así no se demande de un ordenador poderoso para ejecutar el IDE.

Entre los compiladores de C++ para Windows se selecciono a Visual C++ como plataforma de desarrollo por la madurez de la herramienta, la disponibilidad de la misma y la penetración que tiene en al comunidad de desarrolladores en C++ para Windows. Si bien Visual C++ es un producto comercial, su uso en el proyecto no requiere del pago de una licencia de uso ya que existe un convenio entre la compañía que desarrolladora de la herramienta y la universidad donde se esta desarrollando el sistema.

CAPÍTULO 4

4. DISEÑO

Dado que en el proyecto se planea emplear librerías de terceros con el objetivo de reducir el tiempo de desarrollo y mejorar la calidad, es necesario como primer paso en el diseño del proyecto, seleccionar de manera especifica los componentes que serán empleados en el desarrollo. La razón de discutir primero estos componentes yace en el hecho de que las interfaces y el funcionamiento de los mismos dictaran criterios de diseño del proyecto ya que adaptar las interfaces de estos a unas prediseñadas requeriría de tiempo y de esta manera anulando parcialmente el beneficio del ahorro de tiempo que se da con el uso de componentes de terceros.

4.1 Componentes de Terceros

En el capitulo 3 se determinaron de manera general los componentes en que puede descomponerse el sistema. De manera más específica las características que se esperan de los componentes deseados son:

Entorno

Este componente es el pegamento que uno todos los demás componentes y es sobre este que se piensa implementar la lógica del IDE. Debe interactuar de la mejor manera posible con el sistema operativo donde se piensa ejecutar la aplicación (en este caso Windows). Deberá facilitar tareas repetitivas y monótonas del desarrollo, tales como la liberación de recursos del sistema operativo. Seria deseable que este componente tenga soporte por parte de la herramienta de desarrollo (que en este caso es el Microsoft Visual Studio).

En base a lo descrito en el párrafo anterior se seleccionaron las clases de fundación de Microsoft o MFC (Microsoft Foundation Classes). Otra razón para emplear esta librería es el hecho de que ha convertido en un estándar de facto para el desarrollo de aplicaciones

de escritorio para Windows. Esto nos brinda abundantes ejemplos de código, información y comunidades de desarrolladores.

MFC es la librería para desarrollar aplicaciones en C++ para Windows de Microsoft. MFC y su código fuente se distribuye como parte Visual C++, el ultimo es la herramienta para desarrollo en C/C++ de Microsoft.

MFC no es más que una delgada capa sobre la interfase de programación de Windows, MFC provee un modelo orientado objetos para el desarrollo para Windows. Por otro lado, se encarga de tareas rutinarias como la implementación del lazo de mensajes de la aplicación (el lazo de mensajes es un patrón para implementar aplicaciones en Windows) y el manejo de los mensajes de las ventanas del sistema operativo.

Existe además, como parte de la librería, un juego de clases que definen una arquitectura de las aplicaciones. Estas clases no son la versión de MFC de ningún control del sistema operativo, en su lugar definen abstracciones para separar la presentación y el almacenamiento de los datos de la aplicación. Esta arquitectura se

conoce como documento/vista. Esta arquitectura podría ser de utilidad para el ahorro de tiempo durante el desarrollo.

Componente de editor:

Este componente estará encargado de presentar el texto escrito por el usuario, permitir darle el formato apropiado y ser lo suficientemente extensible como para poder satisfacer nuestros requerimientos de interacción. Este es un componente crítico en el proyecto debido a que es la parte de la interfase de usuario con la que el usuario interactuará la mayor parte de tiempo.

En base a lo descrito en el párrafo anterior se seleccionó como componente de editor a Scintilla. Scintilla (http://www.scintilla.org/) es un componente de edición de código fuente para Win32 (sobre Windows) y GTK (sobre Linux).

Scintilla presenta muchas ventajas para el IDE. La primera de ellas es su especialización, debido a que esta orientado para la creación de aplicaciones que editen código fuente. Scintilla posee características que se encuentran en editores modernos de código tales como el formateo del texto (color, sangrando), cuadros de auto completar, tips

informativos, números de línea, marcadores en el margen, doblado de código, y búsqueda de texto, además de no comprometerse con la edición de algún lenguaje en particular y ser sumamente adaptable a necesidades especificas.

La elección del scintilla es significativa ya que nos permite cumplir con varios requerimientos de usabilidad importantes como: abstracción de código, presentación de información contextual y sangrado automático. El doblado de código es una característica que ya se encuentra implementada en Scintilla, al igual que el sangrado automático y la presentación de información contextual. Las tres características anteriormente mencionadas tienen que ser configuradas por el programador ya que el control provee los mecanismos para realizarlas mas no toma todas las decisiones por su cuenta. Esto es una ventaja ya que nos da bastante flexibilidad para personalizar el comportamiento del control.

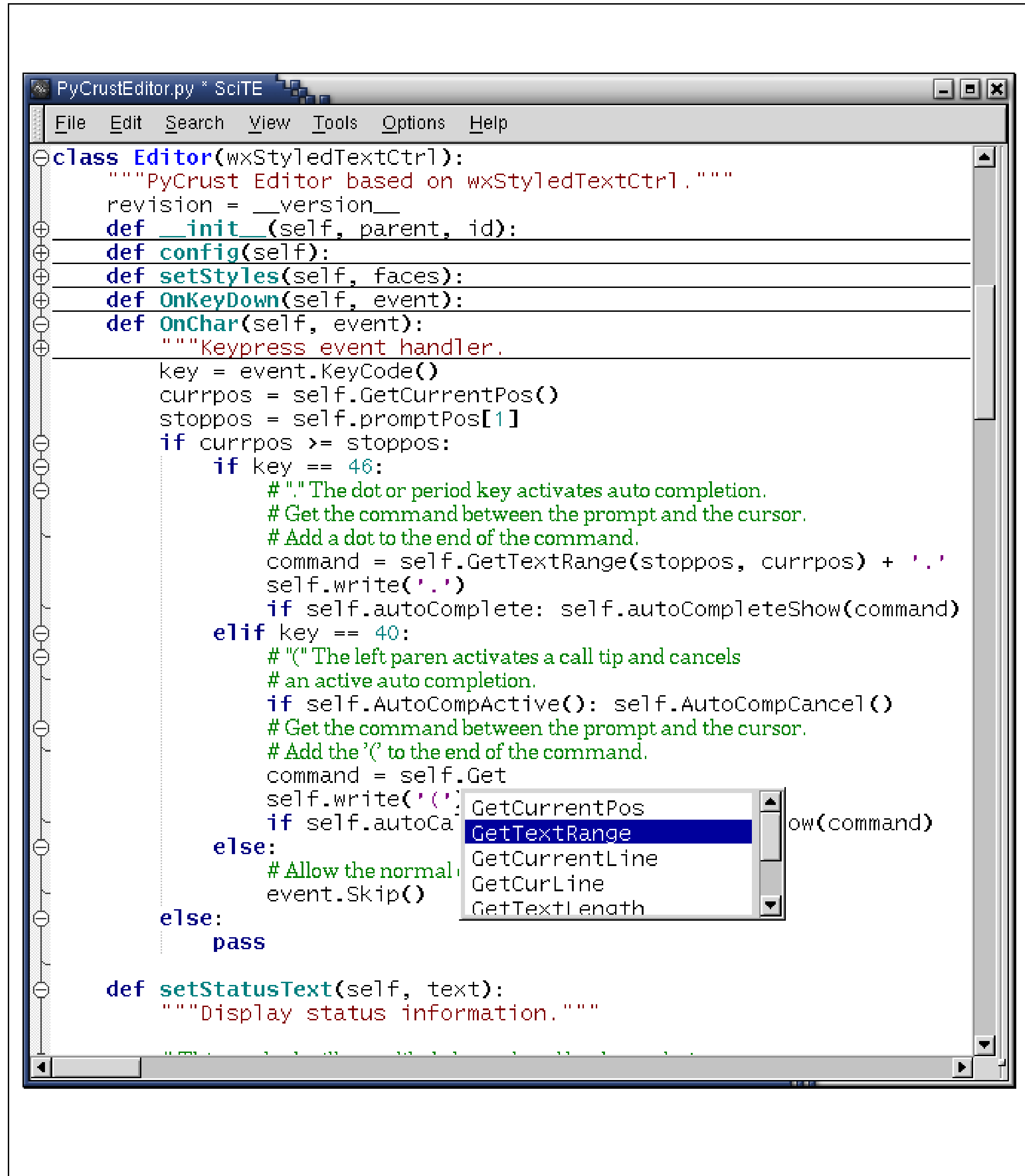


Figura 4.1. Ejemplo de las características de Scintilla

Scintilla se distribuye de manera gratuita y la licencia bajo la cual se distribuye permite su uso en tanto aplicaciones gratuitas como

comerciales, además de permitir la modificación del código fuente, el cual se distribuye junto con Scintilla.

Scintilla es un proyecto activo del que se pueden encontrar actualizaciones y correcciones regularmente. Al mismo tiempo es un proyecto donde se experimenta con diferentes técnicas para mejorar la legibilidad y comprensión del código fuente.

Controles de interfase de usuario:

Este componente deberá poseer controles de GUI que mejorarán a los provistos por el sistema operativo. El componente de controles de interfase no fue mencionado en el análisis del sistema pero lo creemos necesario para enriquecer la interfase de usuario y así obtener una aplicación atractiva a la vista en un tiempo corto. Consideramos que no agregamos ningún valor extra al proyecto al desarrollar por nuestra cuenta estos controles en lugar de utilizar controles ya implementados y probados. Por otro lado se hace necesario que estos controles permitan su extensión, ya sea por medio de su código fuente o alguna interfase de programación. Esto no implica que en el proyecto no se deberá desarrollar controles de

interfase, sino que se desarrollarán los controles más especializados tratando a la vez de emplear algún control existente como base.

En base a lo expuesto en el párrafo anterior y en el capitulo de análisis se opto por usar ProfUI como librería de controles de interfase de usuario. ProfUI es una librería de controles de interfase de usuario implementada en MFC que extiende los provistos por el sistema operativo.

A pesar de que el sistema operativo provee un gran numero de controles con amplia funcionalidad, estos controles resultan insuficientes cuando se desea crear una interfase de usuario avanzada. Además existen tareas de programación comunes y repetitivas que podrían ser manejadas de mejor manera por una librería que se encargue de ellas en lugar del programador.

ProfUI implementa menús y barras de herramientas con iconos con cualquier profundidad de color, ventanas anclables que pueden contener cualquier tipo de ventana o control, manejo automático del estado de habilitado/deshabilitado de los ítems de los menús y las barras de herramientas, serialización automática de la posición de las ventana.

Las ventanas anclables son una buena alternativa para satisfacer el requerimiento del un buen uso del espacio de pantalla. Las ventanas anclables pueden flotar para liberar espacio en la ventana principal de la aplicación. También pueden ocultarse o cambiar de tamaño de acuerdo a la relevancia de la información que contienen en función de la tarea que se este llevando acabo. Se puede observar este tipo de diseño de aplicación en la mayoría de las aplicaciones modernas que manejan gran cantidad de información o con una gran cantidad de funciones.

Los paneles flotantes no son provistos por el sistema operativo y en caso de no utilizar una librería, seria necesario implementarlos desde cero. En nuestro caso, ProfUI seria la librería que nos permite ahorrar este tiempo de desarrollo. La figura 4.2 muestra la captura de pantalla de una aplicación de demostración que emplea ProfUI.

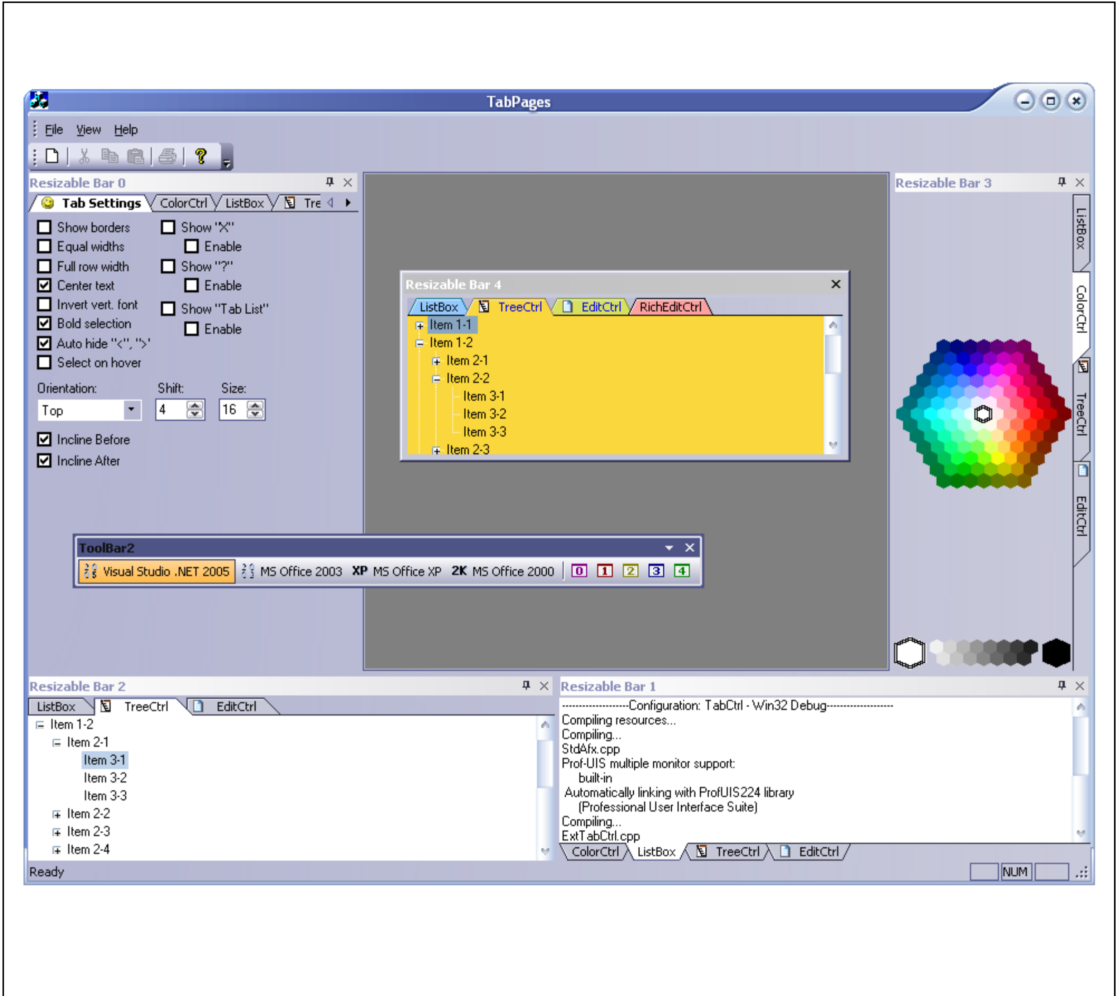


Figura 4.2. Demo de ProfUI

ProfUI es un producto comercial y se debe pagar un valor por su uso en aplicaciones. Afortunadamente la compañía desarrolladora permite, a manera de técnica de mercadeo, el uso gratuito no comercial de la penúltima versión de la librería. Se distribuye el código fuente de la librería con propósitos de depuración ya que la licencia no permite la modificación del código. Se espera que esta última

restricción no cree problemas en el proyecto por cuanto en el proyecto sólo se va a usar la librería sin modificar el código fuente.

Compilador:

Este componente es el encargado de realizar la tarea de compilación de la aplicación y producir los ejecutables. Ya que se desea que con el IDE se puedan producir una amplia gama de aplicaciones, con el objetivo de que el estudiante se motive, el compilador deberá tener librerías incluídas que sean suficientes para poder generar estas aplicaciones. En caso de que la distribución original del compilador no tenga las librerías, se desea que sea lo suficientemente usado como para que exista una comunidad de usuarios del compilador que se encuentren implementando, manteniendo o migrando librerías.

Depurador:

Este componente deberá ser capaz de depurar las aplicaciones que se hayan generado con el compilador escogido. Deberá ser capaz de realizar las operaciones más comunes en los depuradores, tales como insertar puntos de ruptura, evaluar variables, controlar la ejecución de la aplicación y explorar la pila de llamadas de funciones.

Por otro lado, ya que se piensa interactuar con el depurador, seria muy recomendable que la salida del depurador tenga un formato consistente y apropiado para ser procesado por el IDE.

La disponibilidad del código fuente del compilador o de sus librerías no es un hecho estrictamente obligatorio sin embargo es recomendable, ya que el código fuente de las librerías ayudaría de la depuración. Decimos que no es obligatorio debido a que aun sin el código fuente es posible depurar las aplicaciones generadas por el compilador.

Otro factor que limita la búsqueda de un compilador adecuado es la plataforma de ejecución (el sistema operativo) donde se va ha ejecutar el IDE. En este caso se trata de Windows, por lo tanto necesitamos un compilador que se ejecute sobre Windows y genere aplicaciones que se ejecuten sobre Windows.

El compilador que posee la mayoría de características buscadas es GCC. GCC es el compilador del proyecto GNU (http://gcc.gnu.org/). GCC es en realidad la colección de compiladores de GNU. Existen compiladores para C, C++, Objective-C, Fortran, Java, y Ada, así

como librerías para estos lenguajes y herramientas de depuración. Es una herramienta madura, moderna y con una gran comunidad de usuarios.

El único inconveniente de GCC es que no se ejecuta sobre la plataforma de ejecución seleccionada, en este caso Windows. Por fortuna para el desarrollo de este proyecto existe una versión migrada de GCC para Windows. Minimalist GNU for Windows (http://www.mingw.org/), abreviado MinGW, es una colección librerías, archivos de definiciones específicos para Windows y herramientas GNU (compilador, enlazador, depurador, etc.) que se ejecutan de manera nativa en Windows y permiten producir programas nativos de Windows sin necesidad de librerías de terceros (como el caso del proyecto CygWin).

MinGW en parte esta cubierto por la licencia de GNU. Las partes no cubiertas son absolutamente libres para su uso, distribución, modificación.

Librería de GUI y de primitivas graficas para C:

En la distribución del compilador se desea incluir librerías que permitan de manera clara emplear primitivas graficas y de un sistema de ventanas. Esta librería deberá ser compatible con el compilador seleccionado.

GraphApp se seleccionó como la librería de GUI y de primitivas graficas. GraphApp es una librería para programar interfaces graficas y gráficos simples en Windows implementada en C. El objetivo de esta librería es simplificar la interfase y el modelo de programación de Windows a los desarrolladores novatos. Se distribuye en forma de código fuente lo que permite que sea compilada para cualquier compilador en la que se desee emplear.

Actualmente es la librería utilizada por los profesores de Fundamentos de Programación de la FIEC en la ESPOL. Este hecho también es un factor determinante en la selección de esta librería, ya que tanto los estudiantes y los profesores ya esta familiarizados con la misma. Emplear esta librería facilitaría la transición de la antigua herramienta de desarrollo.

4.2 Layout de ventanas

Se plantea que la aplicación presente un layout de ventanas predeterminado, sin embargo deberá ser posible que el usuario configure la posición y tamaño de las ventanas en la interfase según su gusto. El entrono deberá guardar el nuevo layout de ventanas para mantener las preferencias del usuario. El entorno no guarda diferentes configuraciones de ventanas para diferentes usuarios. El layout de ventanas se aplica a todos los usuarios del computador.

La interfase desde un punto de vista macro esta compuesta por:

- Un área de trabajo donde se presentan las ventanas de edición de código
- La(s) ventana(s) de edición de código.
- Una área en la parte superior para el menú de la aplicación
- Una área para las barras de herramientas
- Una barra de estado que indica el estado de las teclas de bloque de mayúsculas, teclado numérico, modo de inserción, línea y columna actual del cursor.
- Varias ventanas flotantes que puede desprenderse de la interfase y flotar o agruparse para mejorar la visibilidad. Estas ventanas agrupan la interfase de diferentes funciones de la

aplicación, tales como el manejo de los archivos de un proyecto, la lista de variables durante la depuración o la visualización de estructuras.

A continuación se detallan el layout de ventanas en diferentes estados del sistema:

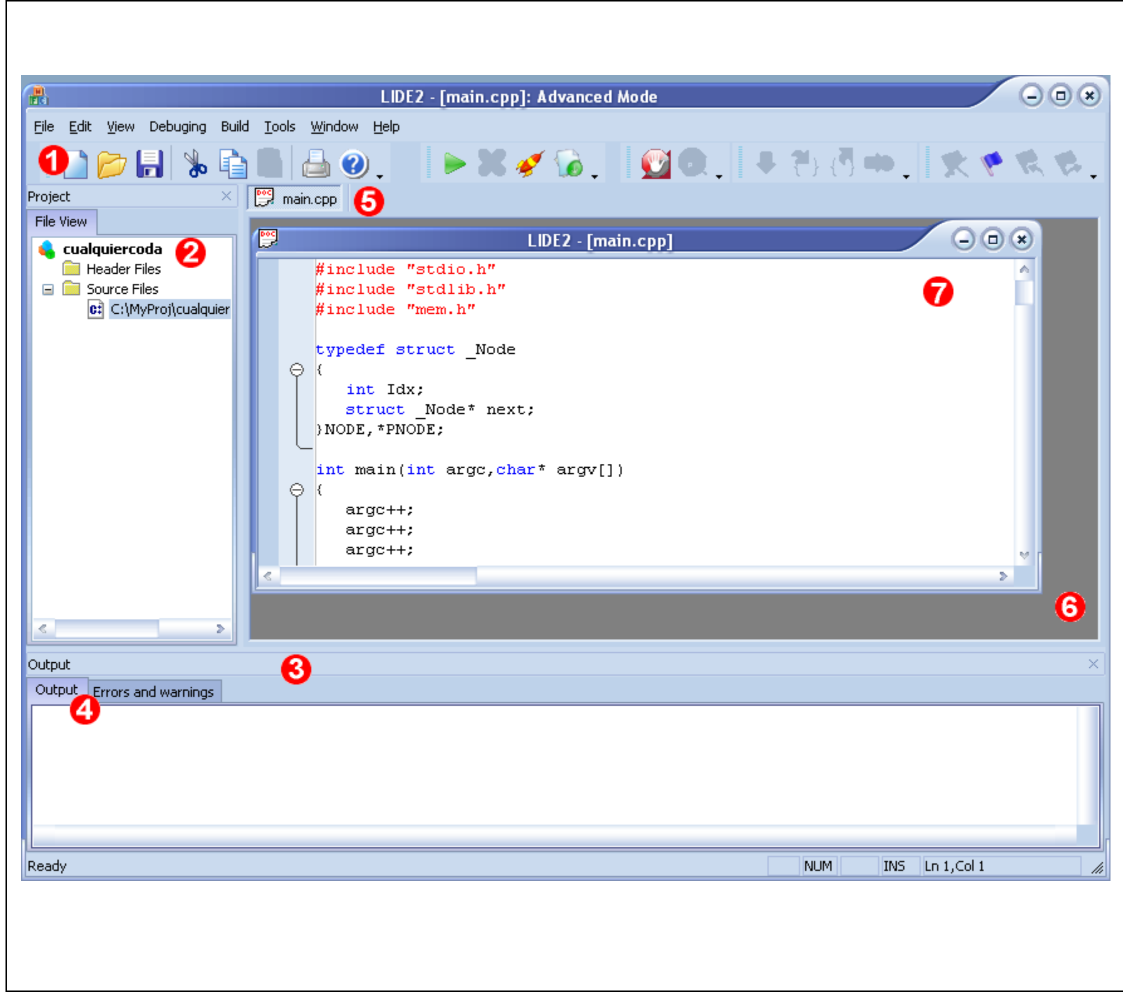


Figura 4.3. Edición en el Modo Avanzado

Titulo: Edición en el Modo Avanzado

Descripción: Esta es la configuración típica del modo avanzado cuando se edita código fuente.

Elementos:

1 – Un área para colocar barras de herramientas. Los botones de las barras están agrupados por el tipo de función que realizan. Por ejemplo los comandos que se necesitan para la depuración (insertar punto de ruptura o ejecutar la siguiente instrucción) están agrupados en la misma barra. Los botones se deshabilitan para indicar que el comando no es valido en ese momento.

2 – Panel de vista de archivos. Contiene los archivos que van ha ser compilados para generar el ejecutable. Por medio de este panel se tiene un acceso rápido a los archivos con los que el usuario esta trabajando. Es la representación grafica del proyecto cuando se trabaja en modo avanzado.

3 – Panel de salida de la compilación. Se agrupan las funciones relacionadas dentro de un panel por medio de pestañas.

4 – En el caso del panel de salida de la compilación tenemos en la primera pestaña la salida textual del compilador (tal cual se vería si se estuviera compilando en una línea de comandos) y en la segunda pestaña tenemos una presentación procesada de la salida del compilador en caso de errores o advertencias. Lo ultimo se emplea para una rápida navegación a la línea y archivo que presenta el error o la advertencia y para resumir los puntos importantes de la salida del compilador en caso de que se haya producido un error o una advertencia.

5 – Pestañas de los archivos. Permite una rápida navegación entre los archivos abiertos.

6 – Área de trabajo. Sirve para contener las ventanas del editor y los paneles de controles.

7 – Ventana del editor. Es en esta ventana donde se presentan el código fuente en edición. Se emplea el margen de esta ventana para presentar números de línea, doblado de código e iconos para indicar puntos de ruptura y marcadores de posición dentro del código.

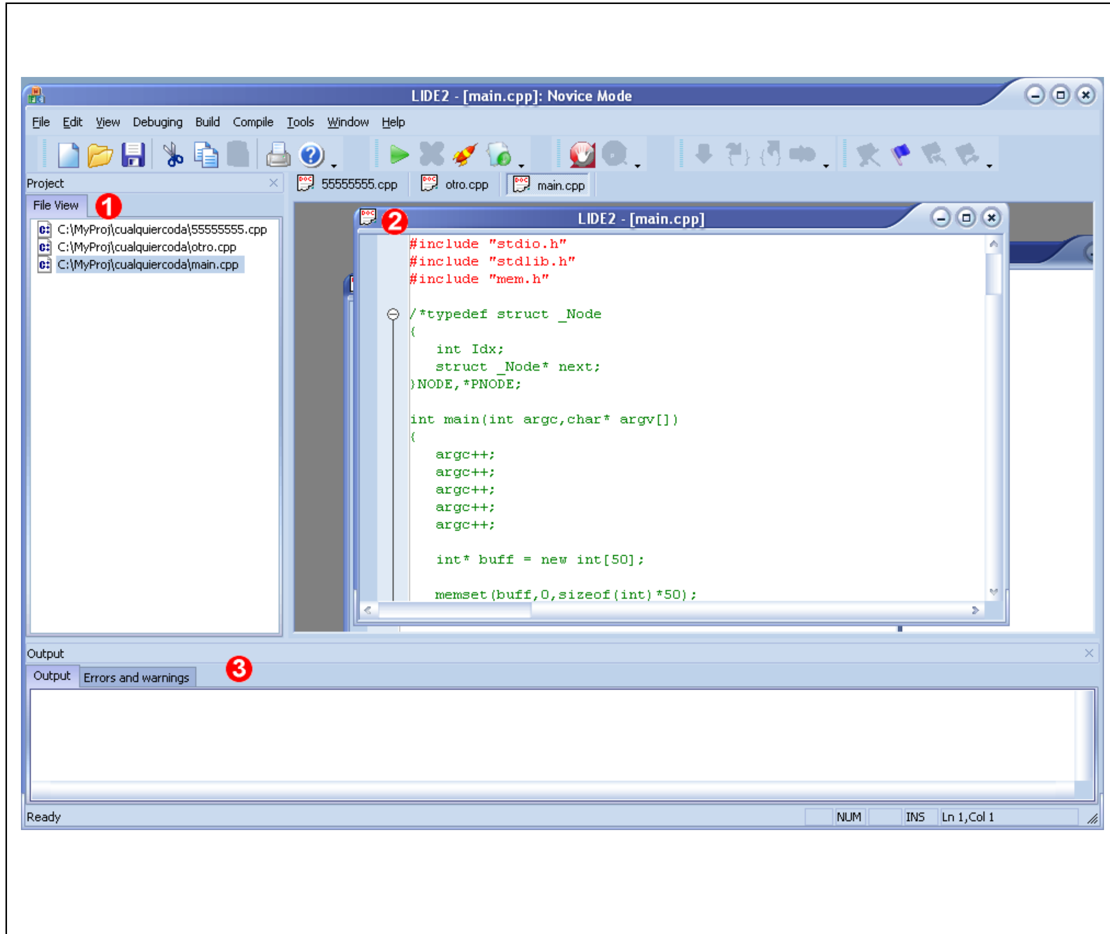


Figura 4.4. Edición en el Modo Novato

Titulo: Edición en el Modo Novato

Descripción: El modo novato tiene un layout similar al modo avanzado. La diferencia en la interfase radica en la interpretación del panel de vista de archivos.

Elementos:

1 – Panel de vista de archivos. Cuando se ejecuta en modo novato este panel contiene los archivos con los que se ha trabajado. Cada archivo representa un ejecutable, por lo tanto no se emplea la presentación de árbol para agrupar los diferentes tipos de archivos del proyecto (archivos de cabecera y archivos de código fuente).

2 – Ventana de edición de código. Desde el punto de vista del usuario esta ventana no cambia su comportamiento y funciones entre los modos novato y avanzado.

3 – Paneles de controles. Estos paneles retienen el mismo comportamiento tanto en el modo novato como en el modo avanzado.

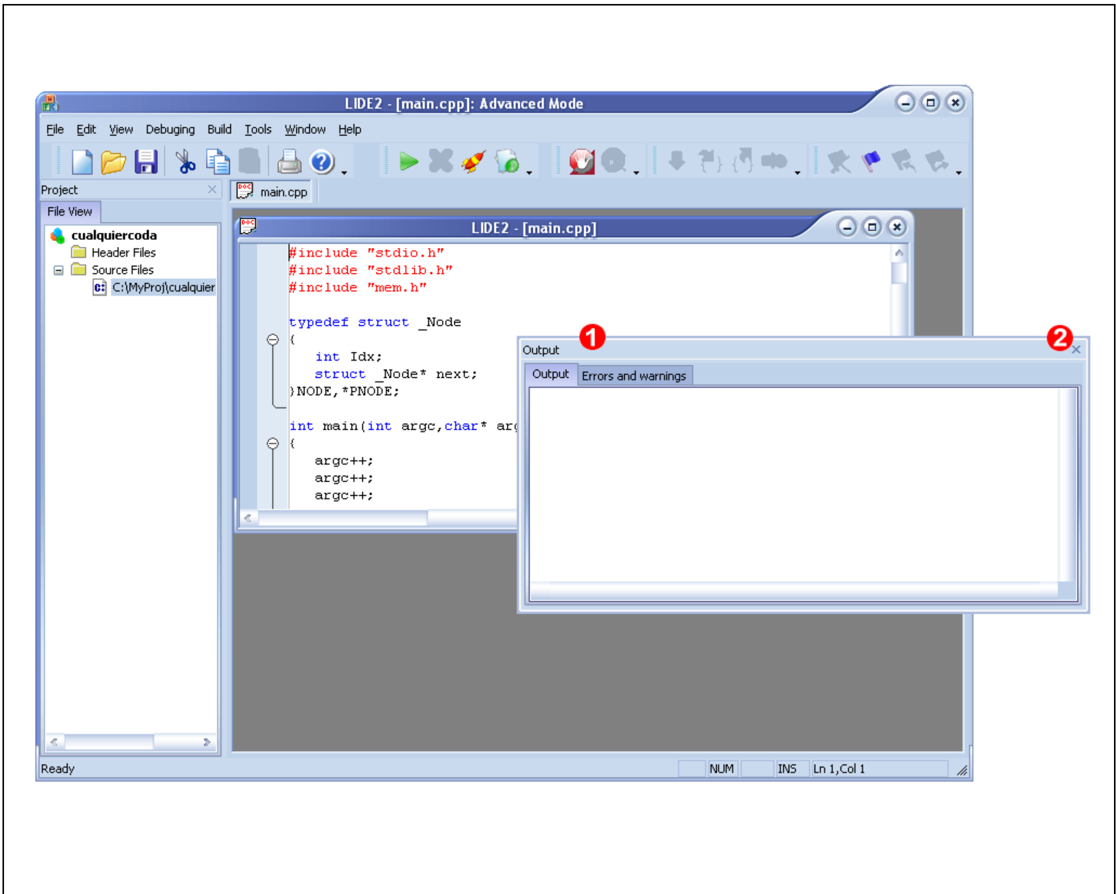


Figura 4.5. Paneles flotantes

Titulo: Paneles flotantes

Descripción: Muestra el comportamiento de los paneles que se necesitan en el proyecto. En caso de que se necesite más espacio en el área de trabajo los paneles pueden desprenderse. Además es posible cambiar el tamaño de los paneles, ya sea flotando o no.

Elementos:

1 – Panel flotante de salida del compilador.

2 – Control de visibilidad del panel. Permite cerrar el panel. Si se desea volver a verlo se debe seleccionar desde el menú de la aplicación. El panel recordara la última posición y tamaño que tuvo.

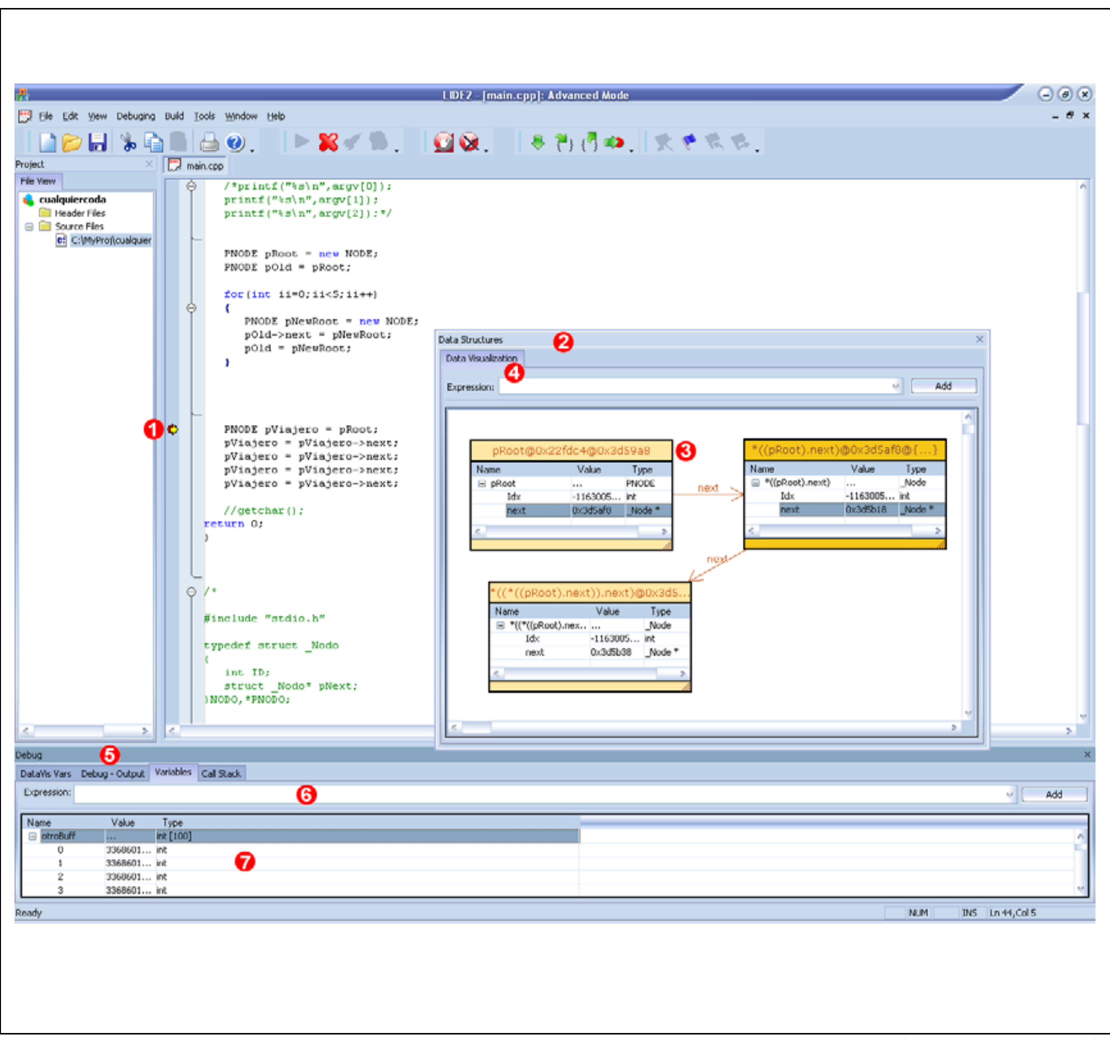


Figura 4.6. Sesión de depuración

Titulo: Sesión de depuración

Descripción: Este es el layout cuando se lleva acabo una sesión de depuración con el IDE. Durante la sesión de depuración se oculta el panel de salida del compilador y se muestra el panel de salida del depurador. El comportamiento y las funciones de todos los paneles, con la excepción del panel de vista de archivos, es similar en el modo novato y el modo avanzado.

Elementos:

1 – Punto actual de ejecución. El icono de una flecha de color amarillo indica el punto actual donde se ha detenido la aplicación que se depura. Con esto el IDE recuerda por el usuario y muestra donde se detuvo la aplicación.

2 – Panel de visualización de estructuras. Contienen la ventana que muestra las estructuras seleccionadas por el usuario para ser exploradas.

3 – Área de visualización. Esta área permite la creación de elementos gráficos que representan las estructuras de datos y las relaciones entre ellas de la aplicación en depuración. En esta área se puede manipular las estructuras, pudiendo:

compactar y expandir, ocultar y presentar, mover y cambiar de tamaño y explorar los punteros.

4 – Ingreso manual de expresiones. Normalmente se agregan las estructuras al panel cuando arrastra el nombre de la variable desde la ventana del editor hasta el panel, sin embargo es posible que el usuario prefiera ingresar manualmente la expresión que desea visualizar. Esto es útil para el usuario cuando desea modificar la expresión (cambiar el tipo de dato, o incrementar un valor son un par de acciones comunes).

5 – Panel del Depurador. Contiene ventanas que muestran información relevante solo durante las sesiones de depuración. Las ventanas están agrupadas en cuatro pestañas: Visualización de Variables, Variables, Salida del depurador y Pila de Llamadas. En la pestaña de Visualización de Variables vemos una representación tabular del contenido de la ventana de visualización de datos, con el objetivo de facilitar la manipulación. En la pestaña de Variables vemos el valor de las variables arrastradas por el usuario desde la ventana del editor. La pestaña de Salida del Depurador muestra todo el texto enviado por el depurador al usuario en caso de errores o condiciones excepcionales. La pestaña de la Llamada de Funciones contiene la lista ordenada de las llamadas sucesivas

de las funciones del programa en depuración hasta el punto actual donde se detuvo la aplicación.

6 – Ingreso manual de expresiones. De igual manera, como se permite el ingreso manual de una expresión para ser depurada en la ventana de Visualización de Estructuras, se permite el ingreso manual en la Ventana de Variables.

7 – Lista de Variables Observadas. Esta lista contiene las variables de las que se desea conocer el valor. Se muestra en una lista el nombre, el valor y el tipo de dato de la variable. En caso de que sea una variable con estructura, no un tipo de dato primitivo, se usara un árbol para mostrar la relación de los miembros.

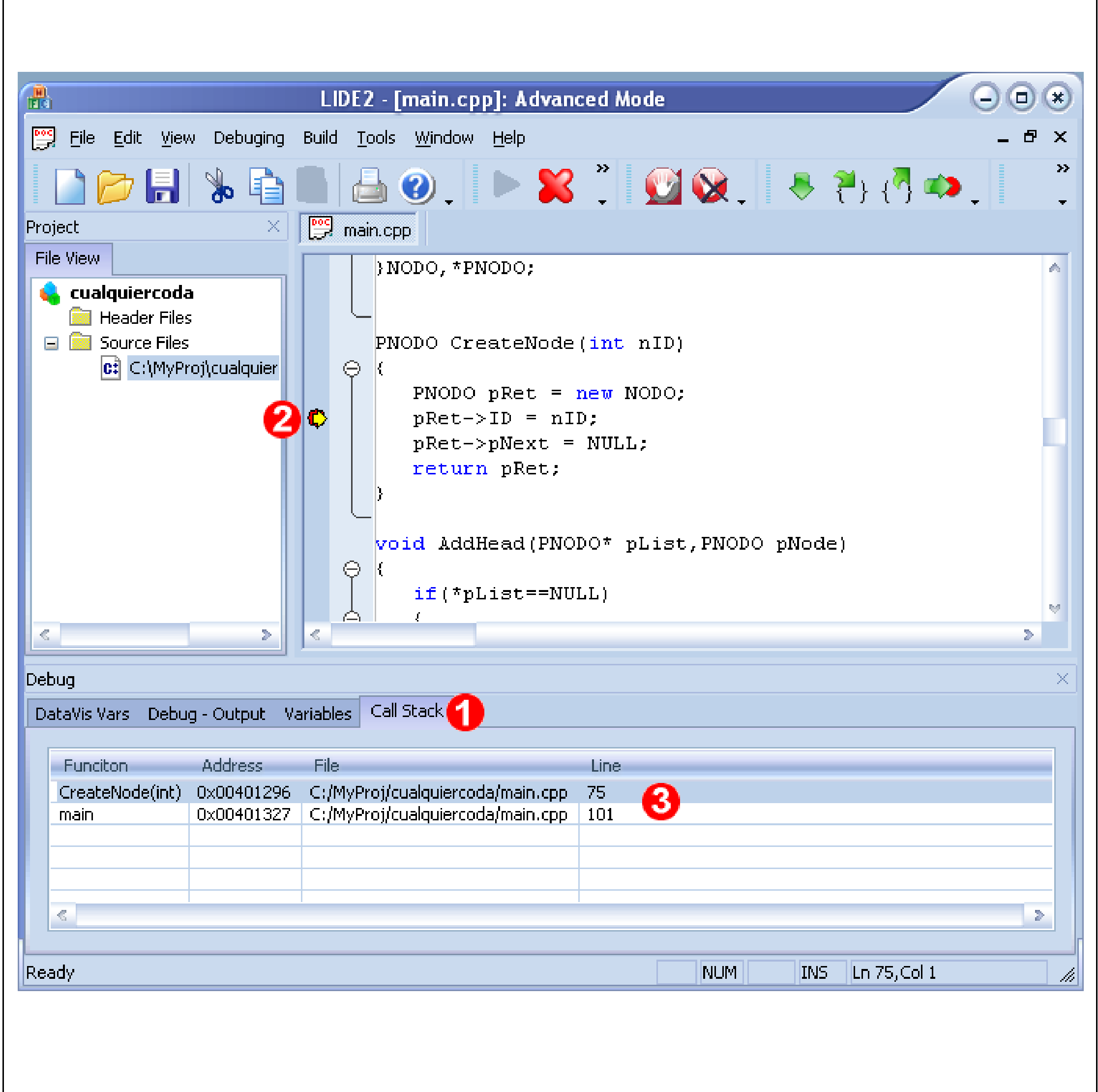


Figura 4.7. Exploración de la pila de llamadas

Titulo: Exploración de la pila de llamadas

Descripción: Esta es la presentación de la aplicación cuando se esta explorando la pila de llamadas. La lista de las llamadas de funciones permite navegar directamente al archivo y línea donde se

produjo la llamada a la función en cuestión, permitiendo observar el valor de las variables en ese contexto de la aplicación.

Elementos:

1 – La pestaña de la pila de llamadas a funciones.

2 – Indicador del punto de llamada. El icono de una flecha amarrilla se ubicara en el archivo y la línea que corresponda a la posición donde se realizo la llamada a la función o al punto de ejecución actual.

3 – Lista de llamadas. Presenta de manera ascendente las llamadas de funciones. Por cada entrada de la lista se presenta el nombre de la función, la dirección en memoria donde se encuentra, el archivo y la línea donde se hizo la llamada.

4.3 Arquitectura del sistema

4.3.1 Modelo Estático

Aun vez que hemos definido como nuestra herramienta de desarrollo a Microsoft Visual C++ (VC++) nos enfocamos en la arquitectura de la aplicación y las alternativas que VC++ nos provee.

Es posible desarrollar una aplicación para Windows empleando únicamente llamadas a la Interfase de Programación de Aplicaciones (API) de Windows. Sin embargo esta es una tarea laboriosa y que requiere gran cantidad de tiempo de implementación y depuración. Además programar directamente la API de Windows implica desarrollar la aplicación con un lenguaje no orientado a objetos como C y perder todos los beneficios de la orientación a objetos o implementar nuestra propia jerarquía de clases sobre la API, consumiendo más tiempo del necesario.

Por lo mencionado en el párrafo anterior se empleo las Clases de Fundación de Microsoft (MFC). MFC es una librería de

clases implementadas en C++ que provee una capa de orientación a objetos sobre la API de Windows y define patrones de diseño que podrían seguir los desarrolladores.

Uno de tales diseños es el modelo Documento/Vista (Doc/View). El modelo Documento/Vista implementa un patrón de diseño Model/View/Controller, este patrón ayuda a separar los datos de una aplicación de su presentación.

MFC soporta dos tipos de aplicaciones Documento/Vista: Interfaz de Documento Sencilla o SDI por sus siglas en ingles (Single Document Interface) y Interfaz de Documento Múltiple o MDI por sus siglas en ingles (Multiple Document Interface).
Una aplicación SDI puede tener solamente un documento abierto a la vez, mientras que una aplicación MDI puede tener dos o más documentos abiertos simultáneamente. Por ejemplo notepad es una aplicación SDI y Microsoft Word es una aplicación MDI.

Una aplicación que sigue la arquitectura Documento/Vista MDI emplea de manera obligatoria la siguiente jerarquía de clases:

En esta jerarquía cada clase tiene bien definidas sus responsabilidades:

- **CWinApp:** Esta clase es representa la aplicación. Implementa los métodos llamados en la inicialización y finalización de la aplicación. Contiene la implementación del lazo de mensajes. El lazo de mensajes es un patrón de diseño usado en las aplicaciones en Windows, extrae los mensajes dirigidos hacia la ventana principal de la aplicación.

- **CView:** Es la encargada de presentar los datos de la aplicación. Es derivada de CWnd (la clase que abstrae el concepto de ventana del sistema operativo). Generalmente esta contenida dentro de otra ventana marco, esta ventana marco provee el soporte para el cambio de posición y tamaño.

- **CDocument:** Se utiliza para contener los datos de la aplicación. No se presenta en la interfase del usuario. Da soporte a la serialización de los datos de la aplicación.

- **CMultiDocTemplate:** esta es una clase interna de MFC. Es relevante en las aplicaciones MDI debido a que contienen los documentos abiertos y es con esta clase que las demás clases de la arquitectura interactúan de manera automática cuando se desea recuperar un documento.

- **CMDIChildWnd:** Es la ventana marco mencionada anteriormente en CView. Contiene a las vistas de una paliación documento Vista y a su vez esta contenida dentro de la ventana principal de la aplicación. CMDIChildWnd redimensiona y mueve automáticamente la ventana de la clase CView de la aplicación.

- **CMDIFrameWnd:** Representa la ventana principal de la aplicación. Contiene los menús, las barras de herramientas, ventanas anclables y en especial las ventanas hijas (CMDIChildWnd) que contiene las vistas de la aplicación (CView).

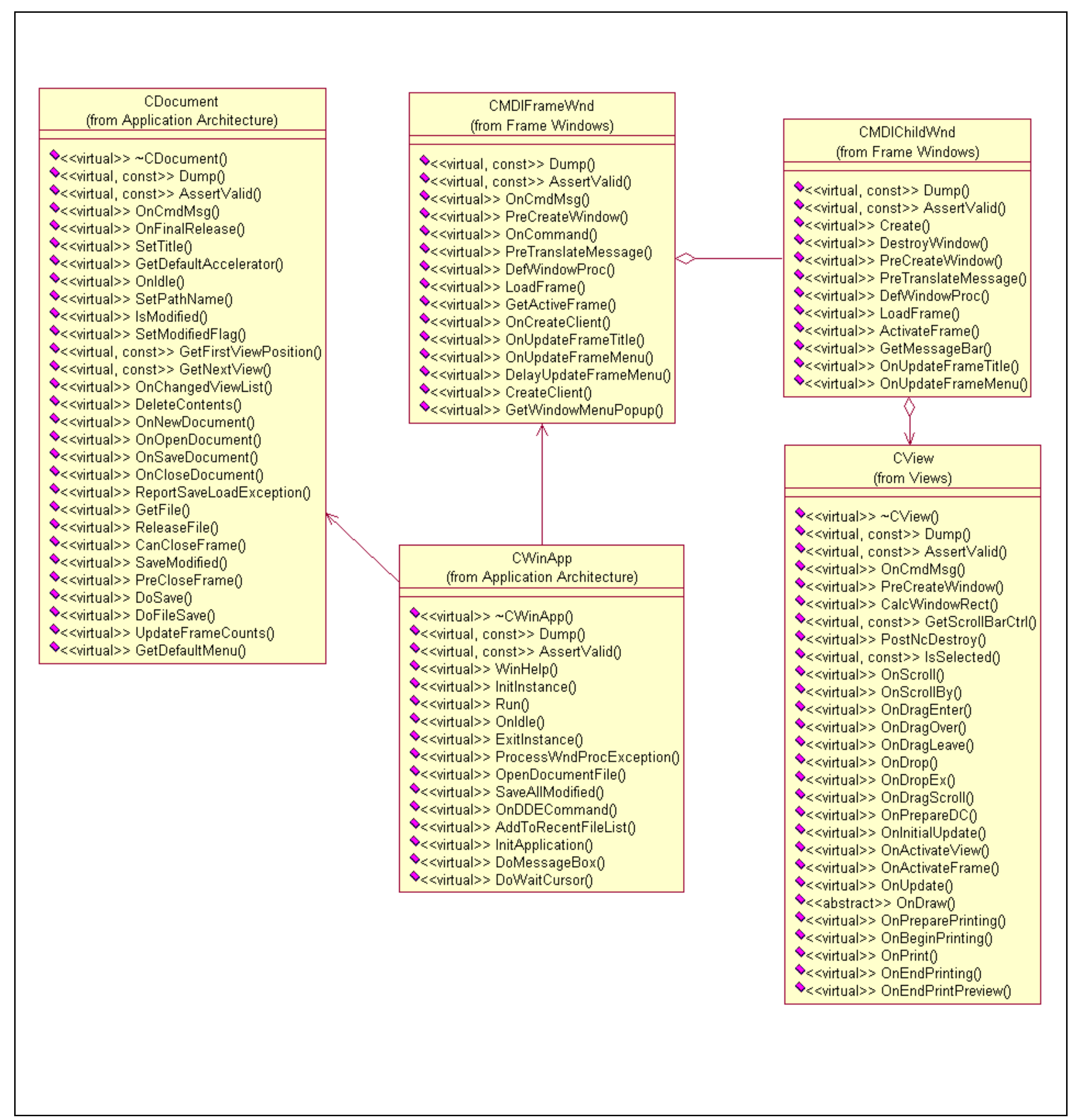


Figura 4.8. Jerarquía de Clases de la Arquitectura Documento/Vista

Se opto por una arquitectura Documento/Vista MDI por las siguientes razones:

En primer lugar se quiere permitir editar más un archivo a la vez. En segundo lugar se desea que la manera en que se almacena el texto de los archivos de código fuente este separado de su presentación. Y tercero la arquitectura seleccionada maneja automáticamente ciertas tareas, tales como la desactivación de los menús o la presentación de tips emergentes (tooltips).

Para diseñar una aplicación alrededor de arquitectura Documento/Vista se deberá derivar de todas las clases mencionadas anteriormente y se sobrescribirán los métodos necesarios para obtener el comportamiento deseado.

En la figura 4.9 se muestra el diagrama de clases del IDE. Las clases mostradas en el diagrama se describen a continuación:

<u>CLIDE2Doc:</u> Deriva de CDocument y contiene información especifica de los archivos de texto que es editan. Esta clase implementa las rutinas de guardado del texto y cambio de

nombre de los archivos de código. Existe una instancia de esta clase por cada archivo abierto.

CLIDE2App: Deriva de CWinApp, existe únicamente una instancia de esta clase. Esta clase coordina la compilación y la depuración, mantiene una referencia al proyecto activo y es la ruta de acceso desde otras partes del sistema hacia la ventana de edición activa.

CMainFrame: Deriva de CMDIFrameWnd, existe únicamente una instancia de esta clase. Esta clase es sumamente importante en el diseño por dos razones: Primero, es a través de esta clase que se reciben las notificaciones de eventos sobre de la interfase de usuario por parte del sistema operativo y segundo, esta clase es la responsable de crear, contener, administrar y brindar acceso a los paneles flotantes de la interfase.

CChildFrame: Deriva de CMDIChildWnd, existe una instancia de esta clase por cada documento abierto. Es una clase utilitaria pero importante para la arquitectura Documento/Vista. El propósito de esta clase es interactuar con la ventana

contenedora, en este caso CMainFrame, y albergar la clase derivada de CView de la aplicación y de esta manera separar el procesamiento de eventos de cambio de tamaño y posición de la ventana de la clase derivada de CView.

CLIDE2View: Deriva de CView, existe una instancia por cada documento abierto. Esta clase se emplea para mostrar al usuario los datos de la aplicación, en el caso especifico de este sistema presenta el texto de los archivo de código fuente. De acuerdo a la arquitectura Documento/Vista esta contenida dentro de la clase derivada de CMDIChildWnd. Esta clase contiene el control del editor de código fuente.

CSintillaProxy: No deriva de ninguna clase y existe una instancia por cada documento abierto. Es un patrón de adaptador para crear una interfase mas sencilla y directa para el control del editor de código fuente. Esta clase hace de interfase entre lo que se desea hacer en el sistema y lo que provee el control del editor.

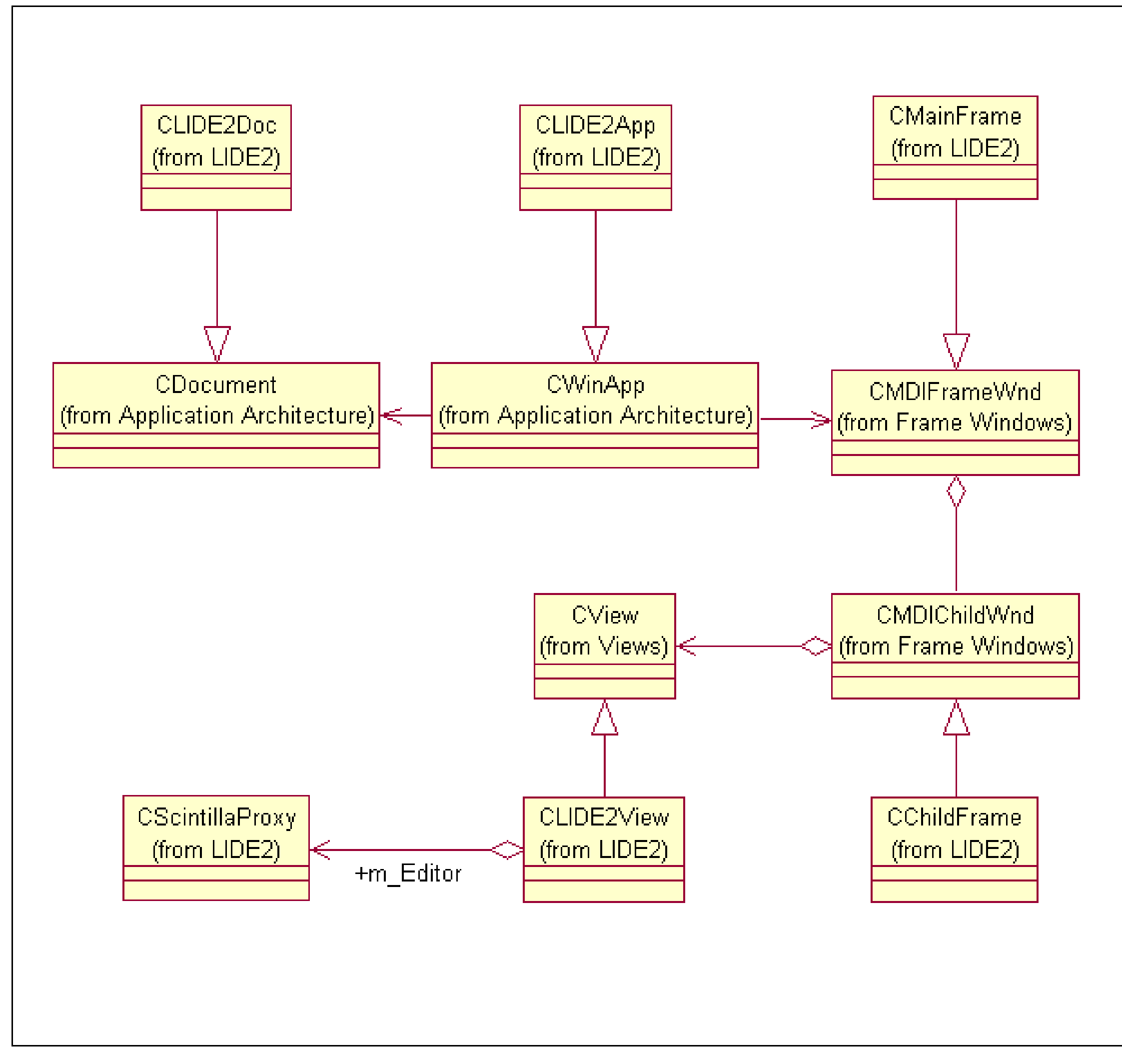


Figura 4.9. Arquitectura del IDE

Paneles flotantes de la interfase de usuario:

Un elemento importante en el diseño de la interfase de usuario son los paneles que agrupan las funciones del IDE. Se pensó en utilizar el modelo de ventanas anclables que se pueden encontrar en algunas aplicaciones de Windows. El inconveniente que se presenta debido ha este diseño de interfase es la falta de soporte por parte de Windows. Normalmente es responsabilidad del programador implementar estos paneles. Afortunadamente la librería de controles de interfase de usuario escogida, ProfUI, posee implementados estos paneles flotantes.

De esta manera nuestro diseño tiene que emplear parte del diseño de los paneles flotantes encontrados en ProfUI. En la figura 4.10 se puede observar el diagrama de clases de la arquitectura empleada en el diseño de los paneles flotantes del IDE.

Se puede observar que cualquier clase que desee ser panel flotante debe derivar de CExtResizablePropertyPage. CExtResizablePropertyPage es parte de ProfUI, esta clase

implementa los métodos necesarios para poder anclarse, flotar, cambiar de tamaño y contener cualquier otra ventana y/o control. Las clases derivadas de CExtResizablePropertyPage contendrán los controles relacionados con una función específica de la aplicación, tal como la vista de archivos, la ventana de visualización de estructuras o la salida en crudo del compilador. Las instancias de estas clases son creadas por y están contenidas en CMainFrame.

Con el objetivo de mantener la interfase de usuario visualmente despejada se agrupan estos paneles de acuerdo a su función. Es decir los paneles que tienen relación a la compilación de aplicaciones se agrupan entre si y así también los paneles que tiene relación a con la depuración. De esta manera podemos ocultar estos grupos cuando no se este realizando la actividad relacionada con el grupo de paneles. Dentro de la arquitectura se emplea la clase CExtChildResizablePropertySheet para agrupar los paneles.

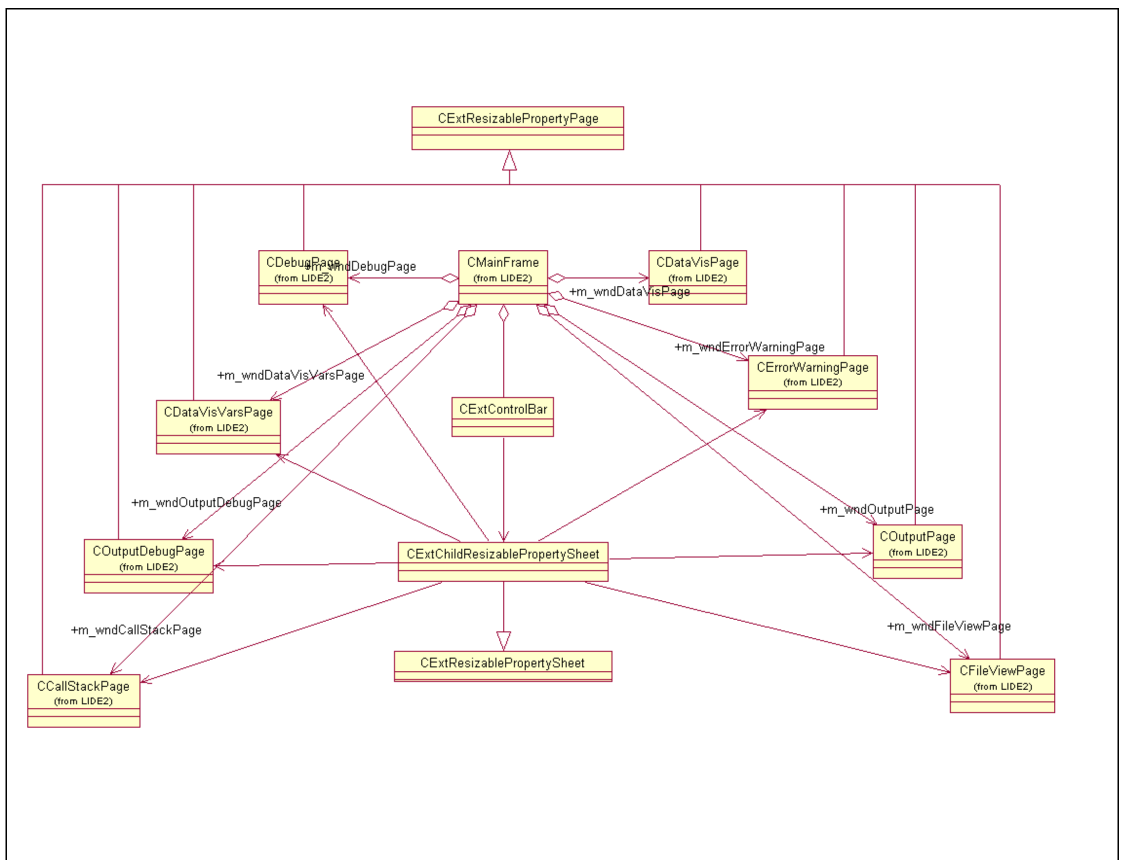


Figura 4.10. Arquitectura de Paneles Flotantes

Vale la pena describir la arquitectura de los paneles ya que los mismos representan funciones específicas del sistema, además de interactuar con otras partes del sistema y ser otro punto de interacción con el usuario.

A continuación se describe la arquitectura de los paneles en grupos de acuerdo a las funciones que realizan, es decir los paneles relacionados con la compilación se muestran juntos.

Panel de Vista de Archivos:

Este panel permite manejar el proyecto que se este desarrollando en el IDE, muestra los archivos que conforman le proyecto actual. Es inicializado por el objeto CLIDE2App al momento de cargar un proyecto. El panel de vista de archivos contiene dos componentes principales, un control de vista de árbol y una tabla de dispersión.

El control de vista de árbol, CDragDropTreeCtrl, deriva de CTreeCtrl, el cual es provisto por MFC y es un envoltorio del control provisto por el sistema operativo. CDragDropTreeCtrl implementa la visualización de los archivos que componen un proyecto dentro del IDE.

La tabla de dispersión mantiene la relación de los archivos del proyecto y los ítems del árbol. Esto permite determinar cual es el archivo que se debe abrir cuando se seleccione desde la vista de árbol y se eviten ambigüedades cuando se tienen archivos con nombre iguales.

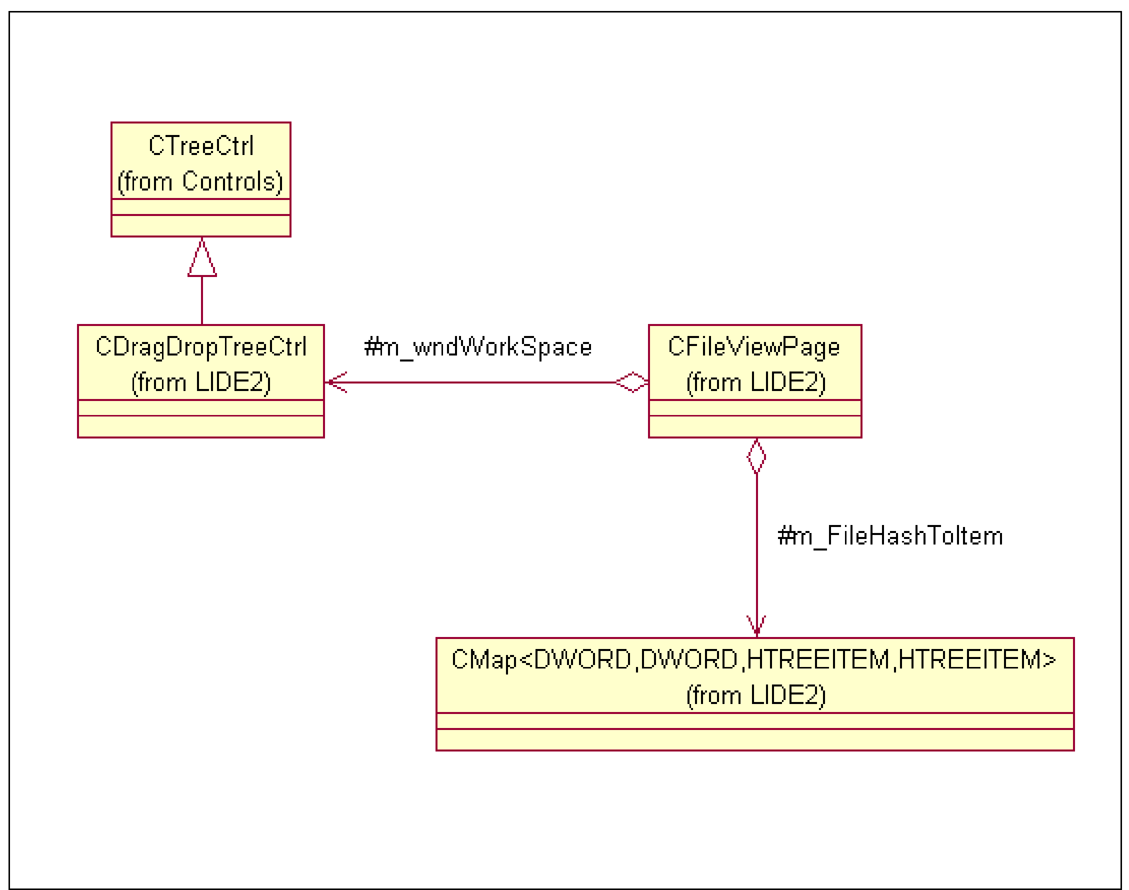


Figura 4.11. Arquitectura del Panel de Vista de Archivos

Paneles de Depuración:

Los paneles de depuración presentan información y permiten al usuario interactuar con el depurador durante una sesión de depuración. El diagrama de clases se muestra en la figura 4.12.

Se tienen tres paneles de depuración:

1. Un panel para presentar los mensajes de notificación del depurador en formato texto. Modelado en la COutputDebugPage de la figura 4.10.
2. Un panel que presenta de forma tabular las variables que seleccionadas por el usuario para explorar su valor, tipo y estructura. Es modelado con la clase CDebugPage.
3. Un panel que muestra una lista ordenada de llamadas de funciones y el punto de actual ejecución. Representada en la figura 4.12 con la clase CCallStackPage.

El panel que presenta los mensajes de notificación del depurador es bastante simple, contiene un control de edición modificado. El control derivado del control de edición (CEdit en este caso) mantiene su estado de solo-lectura y simplifica la inserción de texto.

El panel que contiene la lista de variables contiene esencialmente dos elementos. Un control de para el ingreso de las variables de manera manual por parte del usuario y un control hibrido de lista-árbol para representar las estructuras.

Se decidió utilizar un control hibrido de lista-árbol por ser un punto medio entre el tipo de información a presentarse y la familiaridad de uso de la interfase por parte del usuario. Por un lado se tiene que las variables que se van a presentar en este panel poseen algún tipo estructura, por lo tanto una representación jerárquica (como la de un árbol) seria la mas apropiada, además se requiere poder presentar mas de una variable a la vez, así que una representación de lista minimiza el espacio requerido. Y por otro lado se tiene que la construcción de un control completamente nuevo que represente correctamente la información y utilice eficientemente el área de la pantalla podría confundir al usuario, además de requerir un tiempo de aprendizaje. Por lo tanto se decidió aprovechar la familiaridad de los usuarios con los controles de árboles y de lista provistos por Windows e integrarlos para poder representar adecuadamente las variables observadas.

En el diagrama de clases de la figura 4.12 la clase CDebugVarWnd implementa el control hibrido árbol/lista. Se observa herencia múltiple sobre la clase esta clase, esto se debe a que se deseaba que fuera una ventana (para que pudiera tener interacción con el usuario y estar integrara al

panel) y además fuera posible hacer Arrastrar-Soltar sobre la ventana para simplificar el ingreso de variables.

En lo referente al panel que muestra la lista de llamadas a funciones podemos decir que la clase que modela este comportamiento es CCallStackPage. Como se puede observar en el diagrama de la figura 4.12, esta clase contiene un control de lista para presentar en orden las funciones que fueron llamadas hasta llegar el punto donde se encuentra detenida la aplicación actualmente.

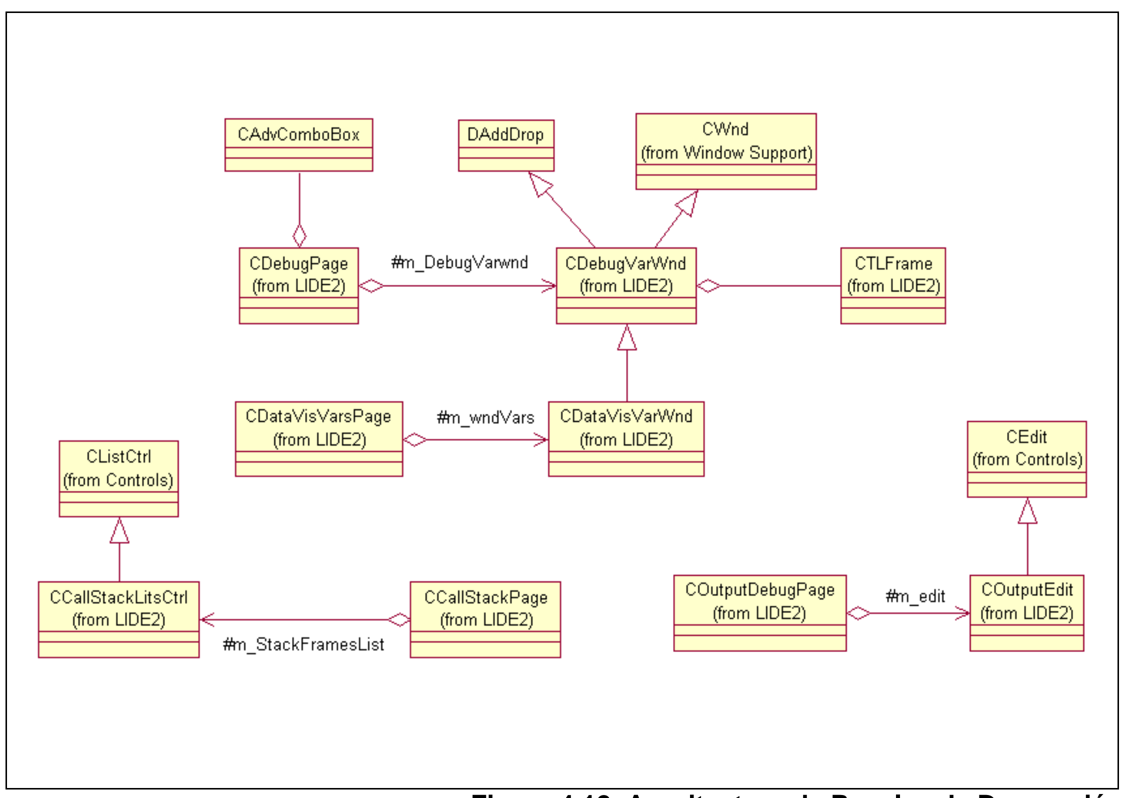


Figura 4.12. Arquitectura de Paneles de Depuración

Panel de Visualización de Variables:

El panel de visualización de estructuras implementa las funciones de funciones de presentar y manipular las variables y sus relaciones de manera grafica. En este panel se pretende cumplir con el objetivo de permitir la visualización de estructuras durante una sesión de depuración.

Desde un punto de vista macro este panel contiene dos elementos: el primero es un control para el ingreso manual de las variables, igual a la lista de variables del panel de variables y una ventana que contiene la representación visual de las estructuras de datos. Este ultimo esta modelado por la clase CDataVisCanvas.

CDatavisCanvas deriva de DAddDrop y de CExtScrollWnd. De la primera deriva para soportar capacidades de Arrastrar-Soltar y simplificar el ingreso de variables para ser exploradas. De la segunda deriva para heredar la capacidad de desplazar sus contenidos (scrolling) de manera automática y sin parpadeo.

CDatavisCanvas emplea dos principalmente componentes para realizar su trabajo:

CDataVisWidget: Esta clase modela la visualización de estructuras. Era posible implementar esta clase de dos maneras: no derivar de ninguna clase ya definida y encargarnos completamente del dibujo, cambio de posición y tamaño o derivar de una clase existente que heredar el comportamiento de dibujo, cambio de posición y tamaño.

Se decidió emplear la última opción debido a que esta clase deberá presentar la estructura de las variables que el usuario seleccionó. Ya que se diseño un control que presenta la estructura y valores de variables (el hibrido árbol/lista) se pensó que seria mejor para mantener la consistencia en la interfase y evitar que el usuario se confunda reutilizar este control en la visualización de variables. Entonces si deseamos agregar este control a la clase CDataVisWidget, la cual representa de forma grafica la estructura de una variable, entonces obligatoriamente deberemos hacer que herede de una ventana (CWnd). No esta por demás decir que la primera alternativa, dibujar todo por nuestra cuenta, implica implementar un control de árbol y de

lista desde cero, lo cual consumiría una cantidad significativa de tiempo de desarrollo y de depuración.

Esta clase responde por si misma a eventos del usuario, tales como exploración de punteros o compactación para reducir el área ocupada, con el objetivo de hacerla interactiva.

CWidgetLink: Esta clase representa los enlaces que existen entre las variables mostradas en la visualización, lo ultimo modelado por CDataVisWidget. Con la intención de que el enlace sea interactivo con el usuario e independiente de otras partes de la arquitectura, esta clase esta encargada de dibujarse de acuerdo a la posición de los elementos que enlaza, posee un nombre y responde a eventos tales como los menús contextuales.

El par de clases antes mencionadas, CDataVisWidget y CWidgetLink, tienen sus respectivas clases de listas, para facilitar su manejo por parte del programador. Estas listas están contenidas dentro de la clase CDatavisCanvas.

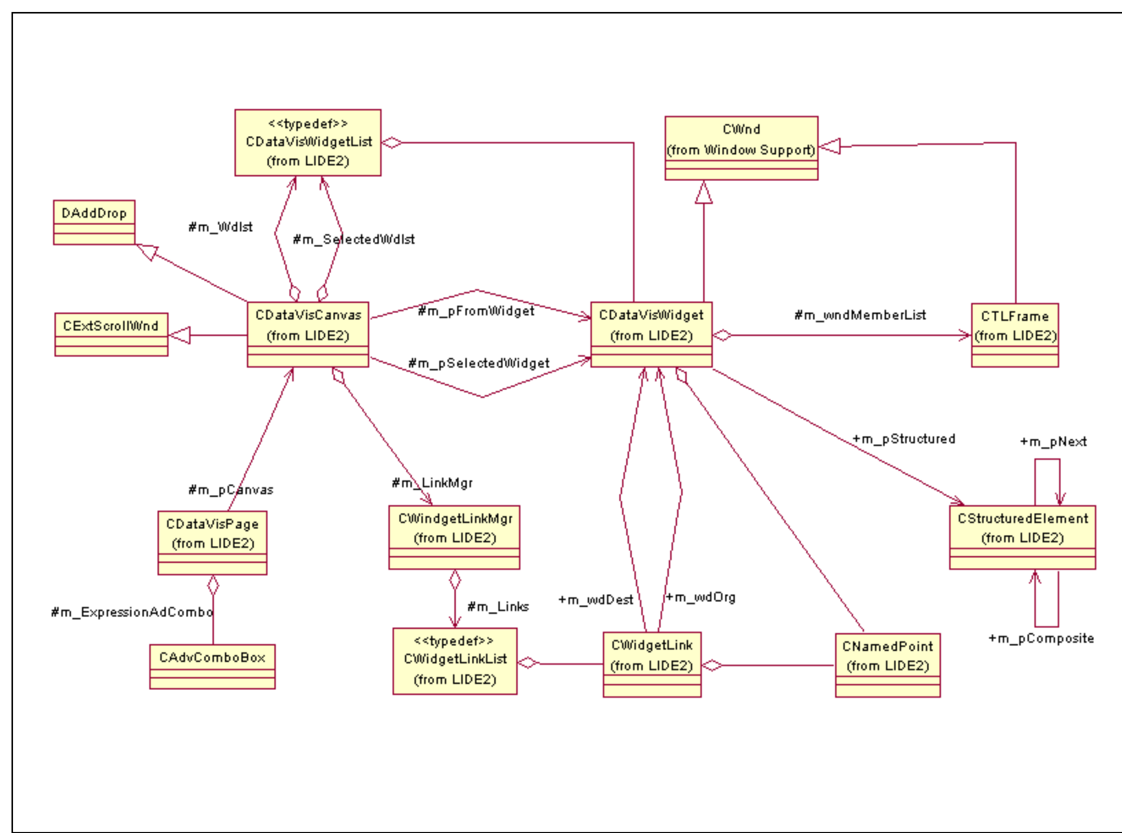


Figura 4.13. Arquitectura del Panel de Visualización de Variables

Paneles de Salida del Compilador:

Estos paneles cumplen el objetivo de dar retroalimentación que se da al usuario durante el proceso de compilación. El primero de los paneles, COutputPage, muestra en crudo la salida del compilador. El segundo de estos paneles, CErrorWarningPage, muestra la misma información que el panel de salida en crudo con la diferencia que lo hace en formato tabular presentando

únicamente los errores y advertencias del compilador y filtrando la información de menor importancia para el usuario, tal como la versión del compilador.

En el panel de salida en crudo del compilador (COutputPage) tenemos una clase derivada de CEdit mantiene su estado de solo-lectura y simplifica la inserción de texto.

El panel que implementa el filtrado de los mensajes de error contiene una clase que presenta la información, en este caso el control de lista del sistema operativo, y una clase que mantiene los datos de la pila de llamadas de funciones, en este caso CErrorWarningList. Esta clase CErrorWarningList deberá permitir que el usuario seleccione una línea que contiene la llamada a una función y salte al archivo de código fuente y la línea correspondiente a ese error.

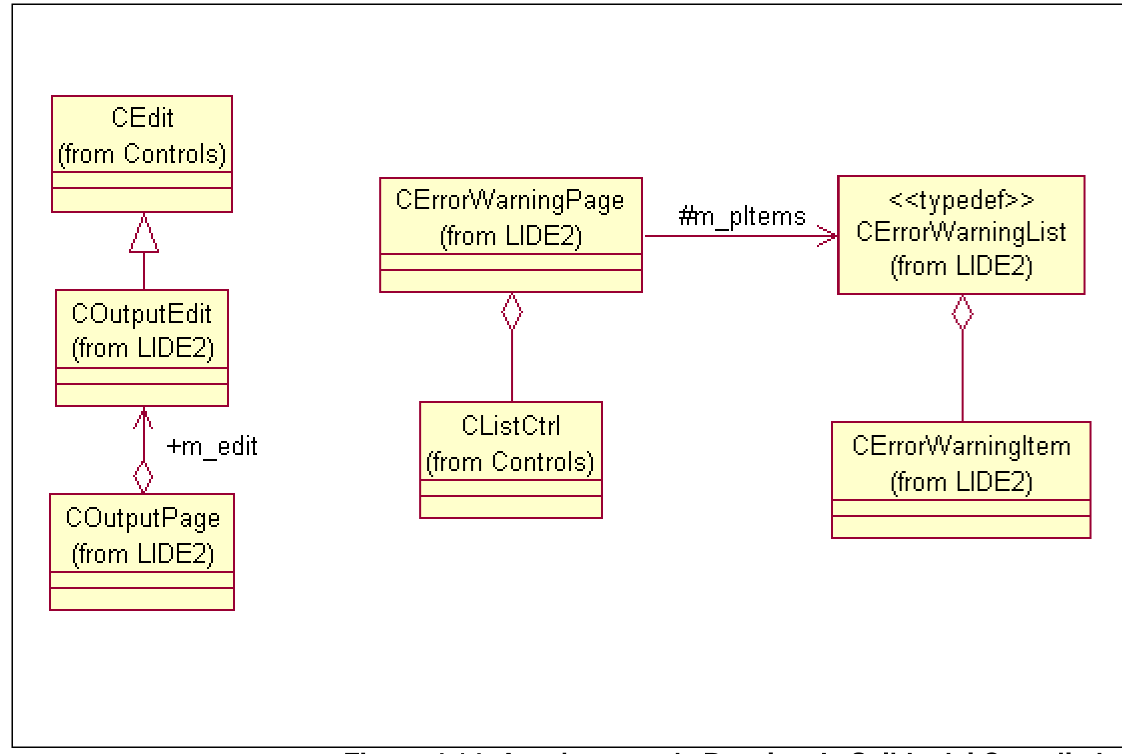


Figura 4.14. Arquitectura de Paneles de Salida del Compilador

Componente de Datos de Aplicación/Proyectos:

Otro componente importante de los ya descritos en la sección 3.3 de análisis de estrategias de desarrollo del proyecto es el componente de Datos de Aplicación/Proyectos. Este componente contiene la información relevante a la configuración del proyecto sobre el que actualmente el IDE esta trabajado. Por configuración entendemos toda la información de

estado de la aplicación como por ejemplo los archivos empleados y las diferentes opciones que se le pasaran al compilador para generar el proyecto.

Ya que se espera soportar diferentes tipos de proyectos, se modelaron los proyectos que serán soportados por el IDE en la jerarquía de clases. Debido a esto, para crear otro tipo de proyecto es necesario derivar de una clase base y realizar la inicialización de los parámetros y librerías que se necesiten. Para la versión inicial de la aplicación se planea soportar proyectos de consola (interfase basada en texto), proyectos para usuarios novatos, proyectos de Windows (con una interfase de usuario basado en ventanas pero sin ninguna librería de soporte) y proyectos de Windows empleando MFC.

Para modelar los datos del proyecto empleamos cuatro clases, algunas de las cuales sirven como base para clases más especializadas:

CWorkSpace: Esta clase representa todo el espacio de trabajo y contiene los proyectos sobre los que se esta trabajando. Esta clase esta diseñada para que a futuro sea posible soportar el

trabajo sobre múltiples proyectos simultáneamente. Esta clase forma parte de CLIDE2App y es el punto de entrada para obtener los objetos proyecto, los archivos que conforman el proyecto y las opciones del mismo.

CProject: Esta clase representa un proyecto dentro del IDE. Contienen grupos de archivos. En esta clase se mantienen los grupos de configuración específicos del proyecto.

CFileGroup: Esta clase modela los grupos de archivos. El propósito es permitir agrupar los archivos dentro del proyecto con fines de minimizar la carga visual al usuario en la vista de archivos.

CLideFile: Modela un archivo del proyecto. La información más importante que mantiene un objeto CLideFile es la ruta del archivo que representa.

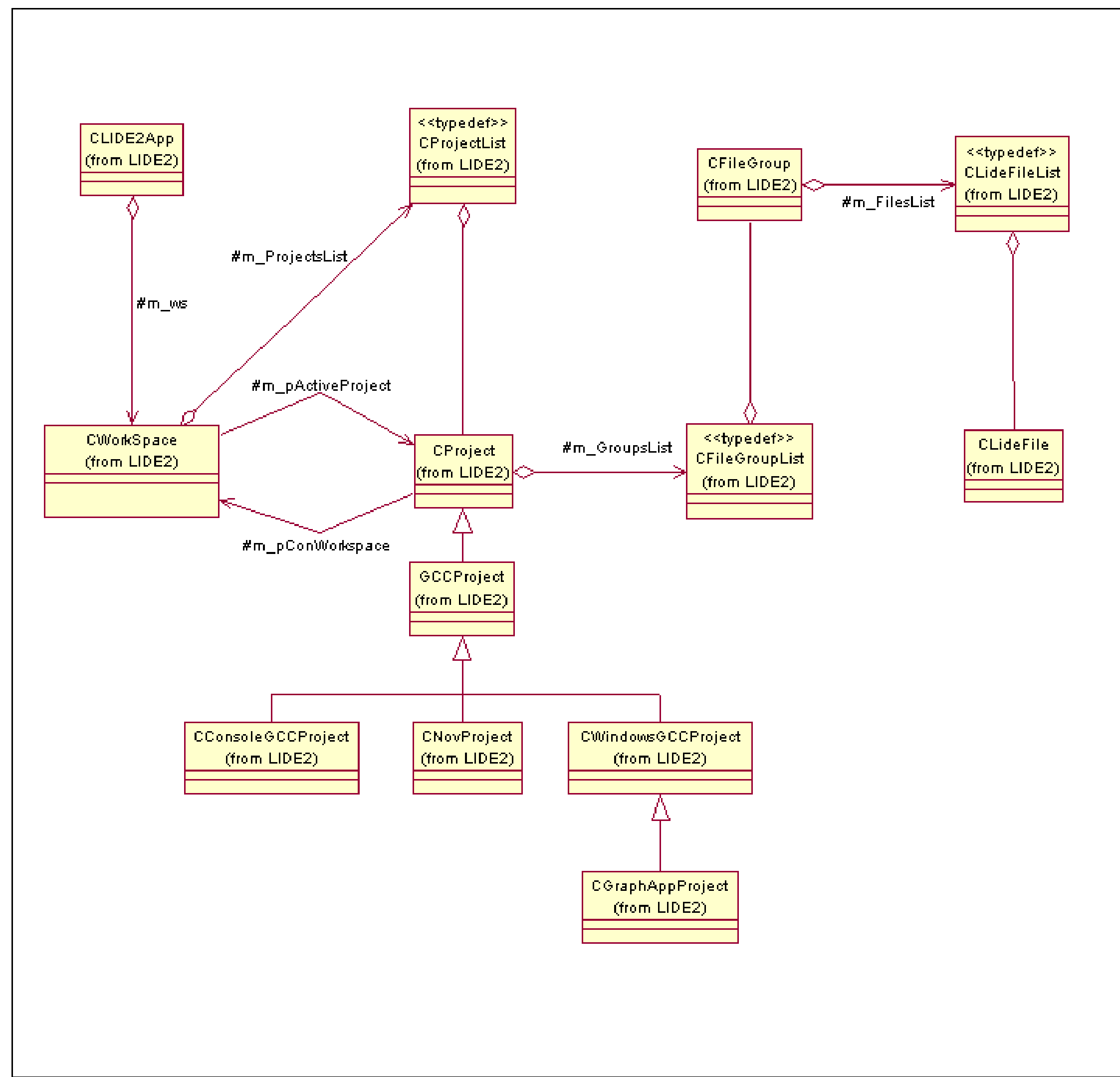


Figura 4.15. Arquitectura del Componente de Datos de Aplicación/Proyectos

Componente del Editor de Código:

El componente del Editor de Código realiza las tareas de un editor completo de código. Este componente implementa algunas características que intentan cumplir con el objetivo de reducir la carga visual en la edición y facilitar la navegación dentro del código fuente.

Como se menciono en la sección 4.1.3 se planea utilizar el componente editor ya desarrollado conocido como Scintilla. Sin embargo, decidimos modelar la interfase con este componente empleando un patrón de adaptador con el objetivo de poder cambiar más fácilmente, si se diera la necesidad, el componente de edición utilizado.

En el caso del IDE, la clase que hace de adaptador es la clase CSintillaProxy. Para diseñar de mejor manera el adaptador se definieron las funciones que el IDE (de forma más específica la clase CLIDE2View) deberá encontrar en el adaptador. Estas funciones son:

- Cargar y obtener el texto.
- Capacidad de copiar, cortar y pegar.
- Insertar marcas en el margen del texto.
- Presentar números de línea en el margen.
- Permitir que el código se oculte para simplificar su visualización.
- Obtención de información sobre el último carácter ingresado, recuperación de palabras o secciones de texto.
- Obtener y modificar la actual posición de inserción.
- Búsqueda y reemplazo de texto
- Modos de inserción y sobre escritura.
- Impresión del texto.

En caso de que el control de edición no soporte alguna de las funciones especificadas anteriormente, el adaptador podrá implementarlas utilizando alguna otra clase (no necesariamente relacionada al Scintilla) y así evitar la modificación de las clases clientes del mismo.. Específicamente en este proyecto el control de edición seleccionado (Scintilla) implementa estas funciones o provee primitivas con las cuales se pueden implementar estas funciones.

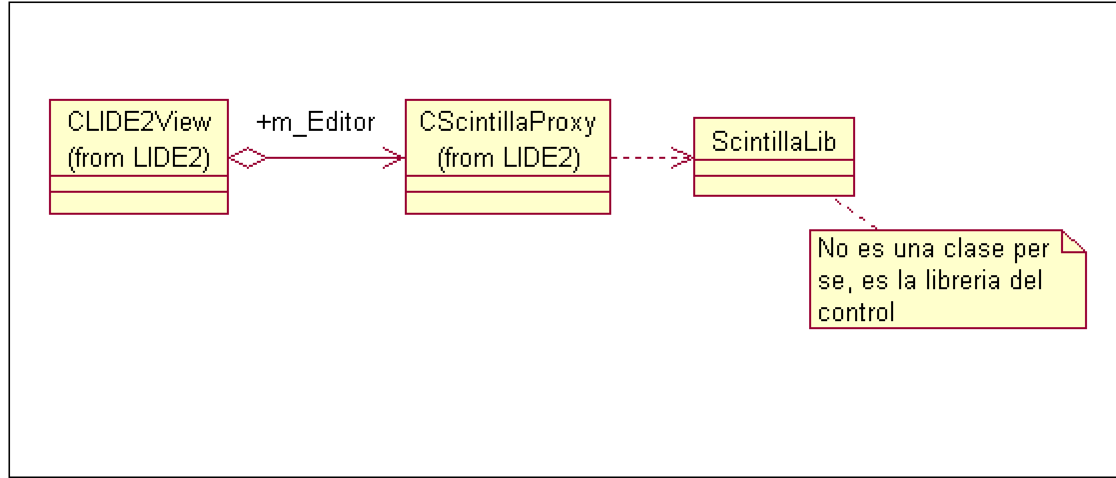


Figura 4.16. Arquitectura del Componente del Editor de Código

Ayudas a la edición de Código:

Para lograr el objetivo de dar ayudas al estudiante durante la edición de código, se diseño un componente responsable de recorrer el archivo de texto que se esta editando y obtener nombres de variables y funciones que encuentre.

El componente en cuestión es CWordListCollection. Este componente debe, cada vez que se pida la lista de auto-completar, recorrer que de la manera mas rápida posible el archivo de texto y formar una lista de palabras que encuentre en el archivo. En el diseño se empleó la clase auxiliar CWordListBuilder para realiza el trabajo real de generar la lista.

Se tomo la decisión de generar la lista de palabras sobre demanda en lugar de tener una tarea que este continuamente analizando el archivo, porque además de simplificar el diseño nos asegura que no existirán palabras faltantes por no haberlas encontrado en el archivo. Un diseño alterno hubiera podido ser la inserción del generador de palabras en el componente de edición. De esta forma, se detectaría el final del ingreso de una palabra en base a un conjunto de reglas (por ejemplo la inserción de un salto de línea). Sin embargo, esta opción se descartó ya que la eliminación de las palabras de la lista hubiera sido mas complicado de lo necesario y en caso de cambiar de editor o de generador de palabras los cambios podrían repercutir demasiado en el resto del sistema.

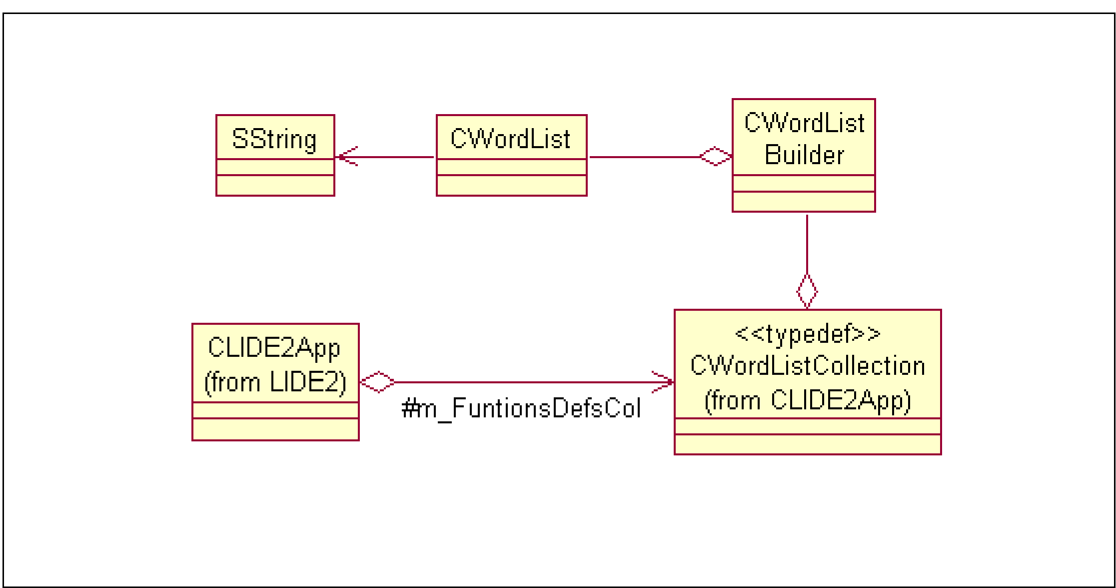


Figura 4.17. Arquitectura de Ayudas a la edición de Código

Componente de la Interfase con el Compilador:

Este componente es otro de los expuestos en la sección 3.3 donde se trata el análisis de estrategias de desarrollo del proyecto. Como se menciono en dicha sección, este componente interactúa con el compilador para producir un ejecutable.

En la figura 4.18 podemos observar el diagrama de clases de dicho componente. La clase que modela la interfase del resto del sistema con el compilador es CCompiler. Esta es una clase que define la interfase para emplear el compilador por lo que se requiere derivar de la misma e implementar los métodos respectivos de acuerdo al compilador que se desee utilizar. Aunque anteriormente se estableció que el compilador a utilizar será el MinGW, el objetivo de este diseño es tener la capacidad de soportar más de un compilador en el IDE.

Por otro lado, la clase CCompiler y sus derivadas están íntimamente relacionadas con las clases que modelan el

concepto de proyecto en el sistema. Esta es una relación obvia ya que el proyecto sabe si un compilador es capaz de generar el ejecutable que el proyecto representa.

Es importante señalar que la redirección de la entrada y salida, tema tratado en la sección 2.2, esta encapsulada en la clase CSpawn y CSpawnConsumer (usadas por CCompiler), siendo la ultima una clase de soporte a CSpawn. El objetivo de la clase CSpawnConsumer (y sus derivadas) es recibir la salida de la aplicación que se esta controlando, en este caso el compilador.

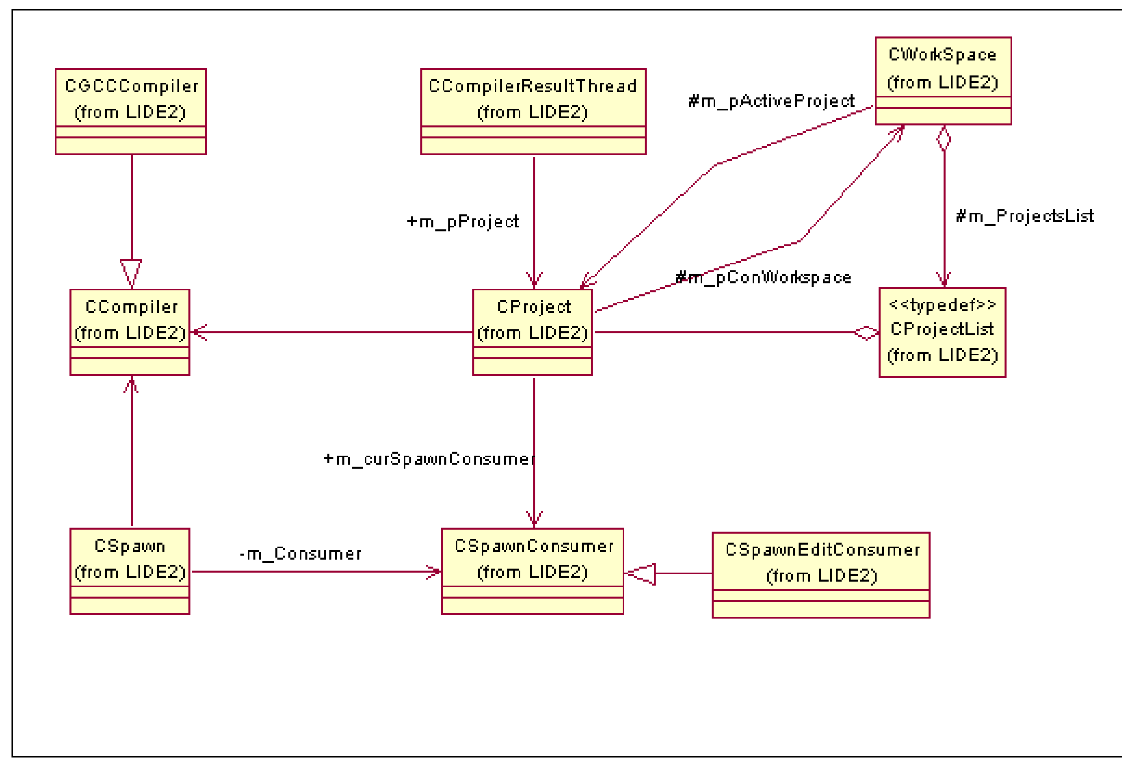


Figura 4.18. Arquitectura del Componente de la Interfase con el Compilador

Finalmente, se debe crear algún mecanismo de serialización para las clases mostradas en esta sección. Es necesario que la información que guardan estas clases exista entre varias sesiones de programación, ya que contienen información de configuración específica de cada proyecto. Se considera que la manera más simple de guardar la información es que cada clase retorne la representación de sus datos en forma de una cadena de texto. De esta manera mantenemos separado el mecanismo de serialización de la representación de los datos. La forma como se guarda la información es simplemente un detalle de implementación.

4.3.2 Modelo Dinámico

En la sección anterior se definieron las relaciones estáticas entre varios componentes del sistema. En la presente sección describiremos el comportamiento dinámico de partes relevantes del sistema. Estas partes comprenden los pasos de inicialización de la aplicación y de un proyecto y la secuencia seguida para compilar un proyecto. La mayor parte del

comportamiento que se presenta durante la etapa de edición de código es manejada por Scintilla y por lo tanto no se incluye una descripción de la misma.

Inicialización del IDE:

La inicialización de la aplicación es un punto crítico en la vida de la aplicación. Durante la inicialización de la aplicación el marco de trabajo crea las ventanas principales de la aplicación y nos da la oportunidad de crear nuestras propias ventanas entre las cuales se encuentran los paneles flotantes. Si la aplicación se esta inicializando porque se hizo doble clic sobre un archivo asociado a la aplicación, entonces el marco de trabajo proveerá la ruta del archivo seleccionado y se encargara de llamar a la función apropiada para abrir el archivo. Se considera importante para la interacción con el usuario que la aplicación cree y mantenga las asociaciones de los archivos, ya que esto acelera el inicio de una sesión de trabajo.

Otra actividad que ocurre durante la inicialización de la aplicación es la carga de la configuración global de la

aplicación, la revisión de las asociaciones de los archivos y la configuración de la interfase de usuario.

A continuación se detalla la secuencia de inicialización de la aplicación que se puede observar en el diagrama de la figura 4.19:

1. El objeto aplicación (CLIDE2App) es siempre el primer objeto en instanciarse. En primer lugar el objeto lee el archivo de configuración común del IDE.
2. Pasa a revisar que las asociaciones de los archivos no hayan sufrido cambios. En caso de ser diferentes a la ultima vez que inicio la aplicación, la aplicación mostrara un dialogo donde se puedan configurar las asociaciones.
3. Se lee el archivo que contiene los prototipos de las funciones conocidas de la librería de C para tener datos para la función de auto-completar.
4. La librería de interfase, ProfUI, necesita un conjunto de paso para configurarse de acuerdo a nuestras necesidades. Este paso se asila dentro de un método del objeto aplicación que es llamado durante la inicialización.

5. El objeto aplicación a continuación crea un objeto de tipo CMultiDocTemplate. Este objeto conoce como instanciar los objetos Documento (CLIDE2Doc) y vista (CLIDE2View).
6. La construcción del objeto de clase CMultiDocTemplate inicia la construcción de un objeto de tipo CChildFrame.
7. Ya que el objeto vista esta contenido dentro del objeto CChildFrame, la construcción del último causa que se cree el objeto vista.
8. La construcción del objeto vista inicia la construcción del objeto editor (CSintillaProxy), esto se debe a que la vista contiene un objeto editor.
9. El objeto editor es inicializado por la clase Vista.
10. El objeto de tipo CMultiDocTemplate crea el objeto Documento y asocia el objeto vista con el objeto Documento para que sea posible navegar de uno al otro.
11. Se construye el objeto de la ventana principal de la aplicación (CMainFrame)
12. El objeto aplicación invoca LoadFrame() sobre el objeto ventana principal. Esto invoca las funciones de inicialización de la ventana principal de la aplicación y en

este punto procedemos a la crearon de los paneles flotantes que se presentan en la interfase de usuario.

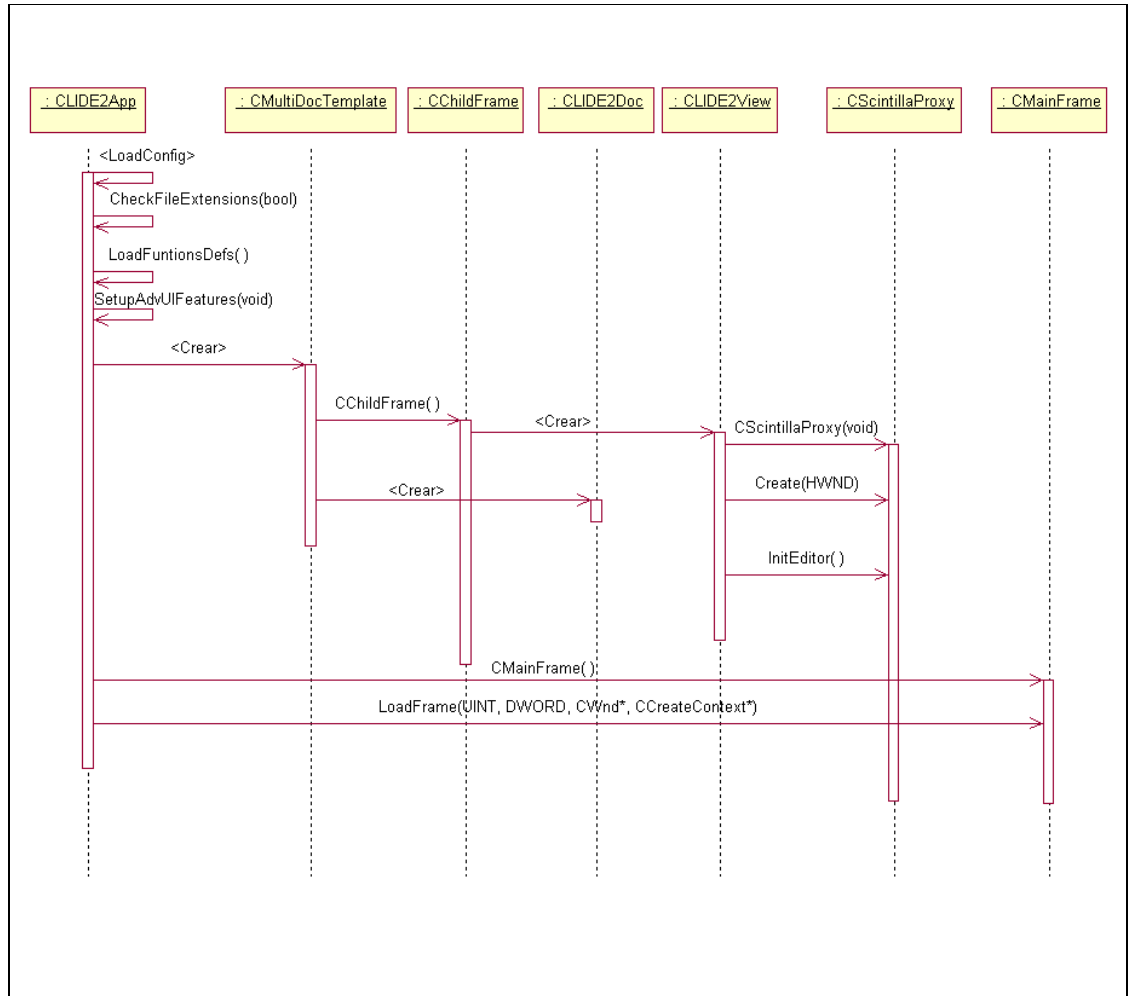


Figura 4.19. Diagrama de secuencia de la Inicialización del IDE

Compilación de una aplicación

La compilación de una aplicación es la secuencia de pasos seguidos por el IDE para interactuar con el componente del compilador para obtener el código ejecutable. La secuencia incluye la recopilación de información del proyecto para poder construir la línea de comando correcta para compilar, la ejecución del compilador, la redirección de la entrada/salida del mismo y la recolección de los datos de salida del compilador.

Ya que las clases CProject y CCompiler (ver figura 4.18) son clases abstractas que solo definen las interfase para el uso del compilador, se derivaron las clases CConsoleGCCProject y CGCCCompiler respectivamente (ver figura 4.20). A continuación se describen los pasos de la interacción de objetos mostrada en la figura 4.20:

1. En respuesta a un evento originado en la interfase de usuario se llama la función OnBuildBuild del objeto aplicación. La cual realiza una serie de pasos previos, tales como el guardado de los archivos de código que hayan sido modificados y aun no se hayan guardado.

Esto último se hace para que los datos del usuario no se pierdan en caso de que el compilador falle, lo cual es posible ya el código del componente no esta bajo nuestro control,.

2. Se invoca a Build del objeto proyecto. La primera tarea de esta función es eliminar cualquier archivo producto de una compilación anterior para evitar que el compilador omita el procesamiento de algún archivo de código fuente.

   Normalmente los compiladores tratan de ahorrar tiempo al comparar las fechas de modificación de los archivos. Si un archivo de código fuente no ha sufrido ninguna modificación entonces no es necesario volverlo a compilar pero existen situaciones causadas por cambios en la hora del sistema, u otros que pueden producir inconsistencias que es mejor evitar.

3. El objeto proyecto inicializa el objeto compilador con los archivos que conforman el proyecto y las opciones del proyecto. Finalmente pide que se compile la aplicación y espera a que termine el proceso.

4. El objeto compilador crea la línea de comando apropiada para el compilador que se va a usar.
5. La tarea de ejecución del compilador externo y la redirección de la entrada y salida es delegada al objeto CSpawn.
6. Una vez terminada la compilación se notifica al objeto aplicación que puede procesar la información de salida del compilador. En esta etapa se filtra la salida del compilador para identificar los errores y advertencias del compilador.
7. El objeto aplicación agrega el texto en crudo de la salida del compilador al panel de salida del compilador y la lista de errores y advertencias al panel de errores y advertencias.

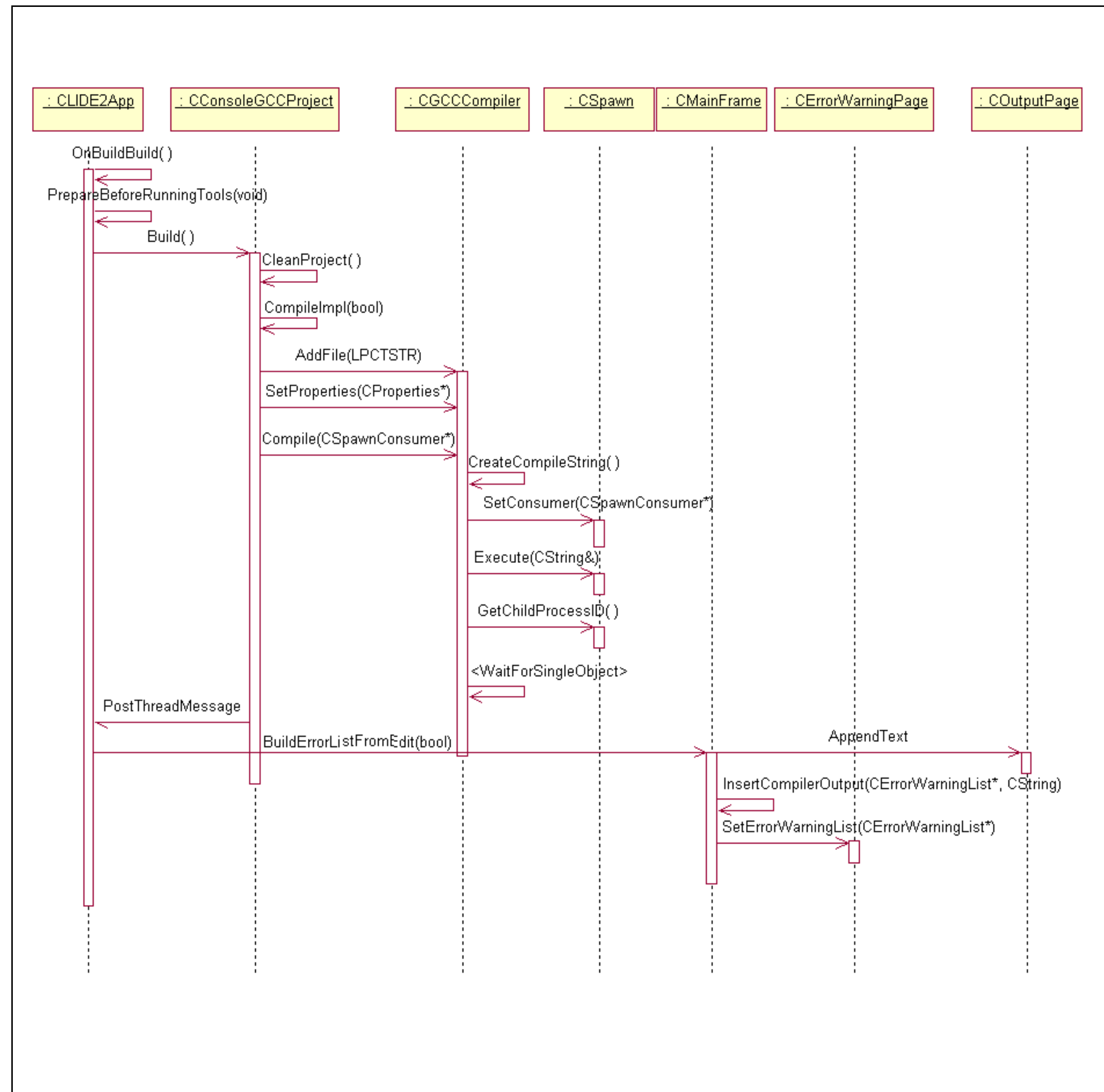


Figura 4.20. Diagrama de secuencia de la Compilación de un aplicación

4.4 Arquitectura de la Interfase con el Depurador Externo

Un componente sumamente importante en el editor es el componente de la interfase con el depurador. La interfase con el depurador permite observar la ejecución paso a paso de las aplicaciones desarrolladas en el IDE y asimismo es la fuente de información que permite implementar el componente de visualización de estructuras.

Debido a la estrecha relación que existe entre ambos componente se los describe juntos en la presente sección.

4.4.1 Modelo Estático

Componente de la Interfase con el Depurador:

De igual manera que en la arquitectura de la aplicación se diseñó para poder emplear otros compiladores (ver la figura 4.18), la arquitectura del componente de depuración brinda la posibilidad de emplear otros depuradores.

El componente de interfase con el depurador tiene como pilares a dos clases: la clase abstracta CDebugger y la clase CDebugSession.

La clase CDebugSession es la responsable de crear el objeto depurador correcto, administra el tiempo de vida del objeto depurador y sirve como un punto de acceso al objeto depurador desde otras partes de la aplicación. Otra tarea de la clase CDebugSession es mantener informaron generada mientras se edita el código y que será necesaria durante la depuración (como por ejemplo la localización de los puntos de ruptura). Por ultimo, la clase CDebugSession se utiliza como almacenamiento intermedio (cache) de los resultados enviados por el depurador y de esta manera reducir el número de comandos enviados al depurador.

La clase abstracta CDebugger define una interfase común para controlar el componente externo del depurador. Si deseamos utilizar un depurador diferente a GDB esperamos que el nuevo depurador permita:

- Iniciar y terminar el programa que se depura.
- Insertar y remover puntos de ruptura.
- Ejecutar paso a paso la aplicación.
- Evaluar una expresión.

- Obtener la dirección de memoria y el tamaño de una expresión.
- Obtener la pila de llamadas a funciones.

Por expresión se entiende una variable simple o variables y operadores bajo la sintaxis permitida por el lenguaje C para una expresión. Permitir variables junto con operadores es una característica poderosa. Por ejemplo si se tiene una variable te tipo puntero podría obtenerse el valor al que apunta el puntero utilizando el operador de desreferencia de C (*) .

Ya que se desea que la interfase siempre responda al usuario y no se detenga hasta que algún proceso largo termine, se decidió diseñar la comunicación con el depurador de manera asincrónica. Para que esto sea posible el depurador no procesa de manera inmediata los comandos, sino que los ubica en una cola de trabajos. CJobQueue es la clase que modela la cola de trabajos.

Los comandos que se envían al depurador externo son iguales a los que enviaría un usuario que estuviera usando directamente el depurador. Desde el punto de vista de diseño

estos comandos no son más que cadenas de texto que se envían al depurador. Como el objeto depurador no esta interesado en el contenido de las cadenas, se definió una interfase muy simple para enviar comandos hacia el depurador externo. Esta interfase crea objetos trabajo, que contienen los comandos en forma de cadenas de texto, y los coloca en la cola de trabajos mencionada en el párrafo anterior.

La clase CDebugger (o sus clases derivadas) deberá procesar la salida del depurador y con esta salida crear estructuras o lanzar eventos a otras partes del IDE. Al aislar esta función se intenta simplificar el diseño del resto del sistema y minimizar el impacto de un cambio de depurador.

Para ejecutar el depurador externo y redirigir la entrada/salida, la clase CDebugger depende de la clase CSpawn y de la clase utilitaria CGDBConsumer. La clase CSpawn inicia el proceso externo y redirige la entrada/salida. La clase CGDBConsumer es una especialización de la clase CSpawnConsumer para dirigir la salida del depurador hacia la clase CGDBDebugger. CSpawnConsumer ya fue definida en la sección 4.3.1.

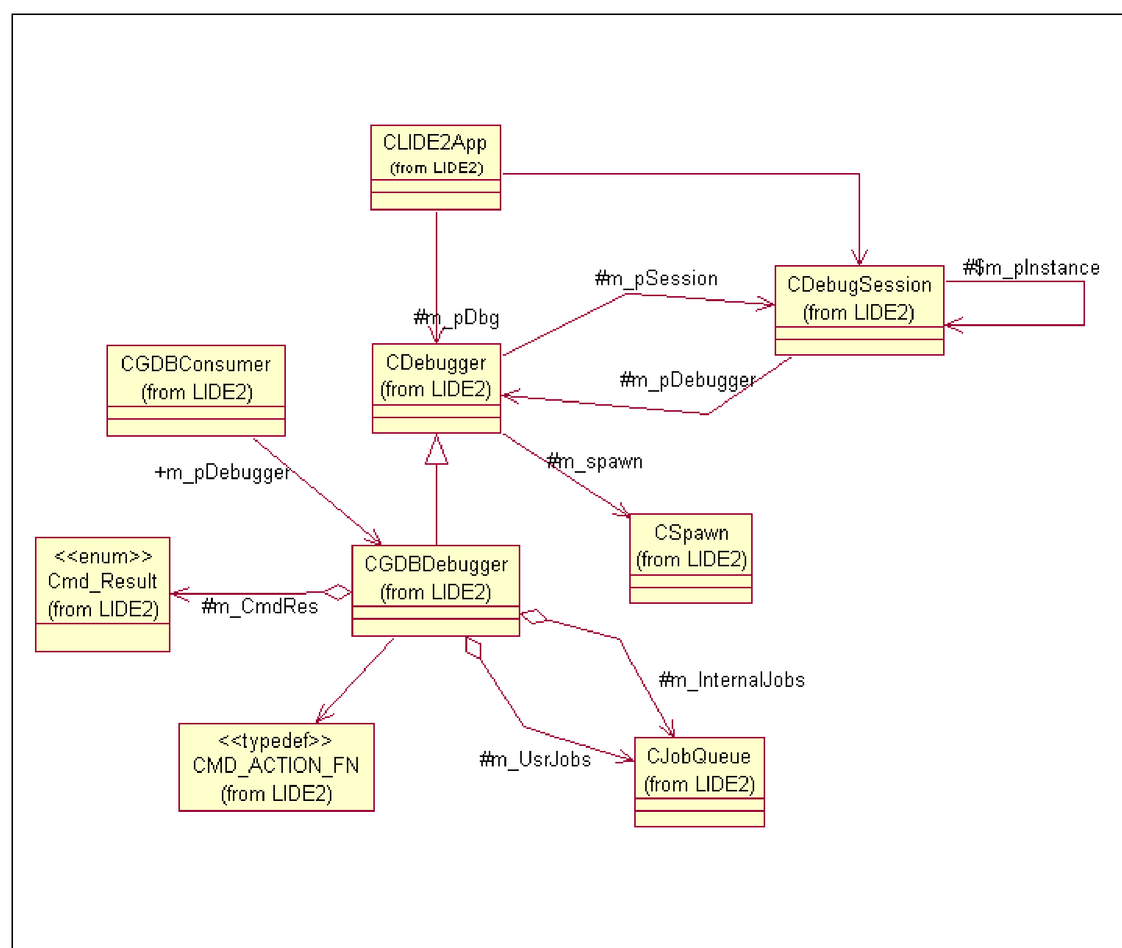


Figura 4.21. Componente de la Interfase con el Depurador

Interacción de la Interfase del Depurador con la Interfase de Usuario:

Las clases CDebugger y CDebuggerSession deben interactuar con otras partes del sistema para poder enviar las notificaciones de los eventos producidos por el depurador.

En el diseño actual se notifican dos tipos de eventos: la parada de la aplicación por haber encontrado un punto de ruptura y la parada de una aplicación por la terminación de la aplicación.

Cuando la aplicación se detiene por haber encontrado un punto de ruptura se notifica al objeto aplicación, a la clase que contienen la lista de variables y a la clase que muestra la pila de llamadas a funciones.

El objeto aplicación al recibir la notificación de parada de la aplicación utiliza el nombre del archivo y la línea donde se detuvo la aplicación para abrir el archivo respectivo y colocar una marca para indicar la línea donde se encuentra detenida la aplicación.

El objeto que presenta la pila de llamadas a funciones cuando recibe la notificación de parada interroga al objeto CDebugSession para obtener la pila de llamadas a funciones. El objeto CDebugSession a su vez emplea el objeto depurador creado para obtener la pila de llamadas a funciones y retornarlo al objeto que lo necesita.

Finalmente la clase que representa la lista de variables al recibir la notificación de parada utiliza la clase CDebugSession para que le indique que variables han cambiado de valor. CDebugSession delega esta tarea a la instancia CDebugger y almacena el resultado en una lista de objetos de tipo CVarUpdateResult. Los objetos CVarUpdateResult contienen el nombre, tipo y nuevo valor de las variables actualizadas. La clase CDebugSession almacena estos resultados en un cache ya que es posible que sean necesitados por más de una clase. De acuerdo al diseño actual estos datos son necesitados por el panel que muestra la lista de variables, por el panel de visualización de estructuras y por el panel que muestra la representación tabular de la visualización de estructuras.

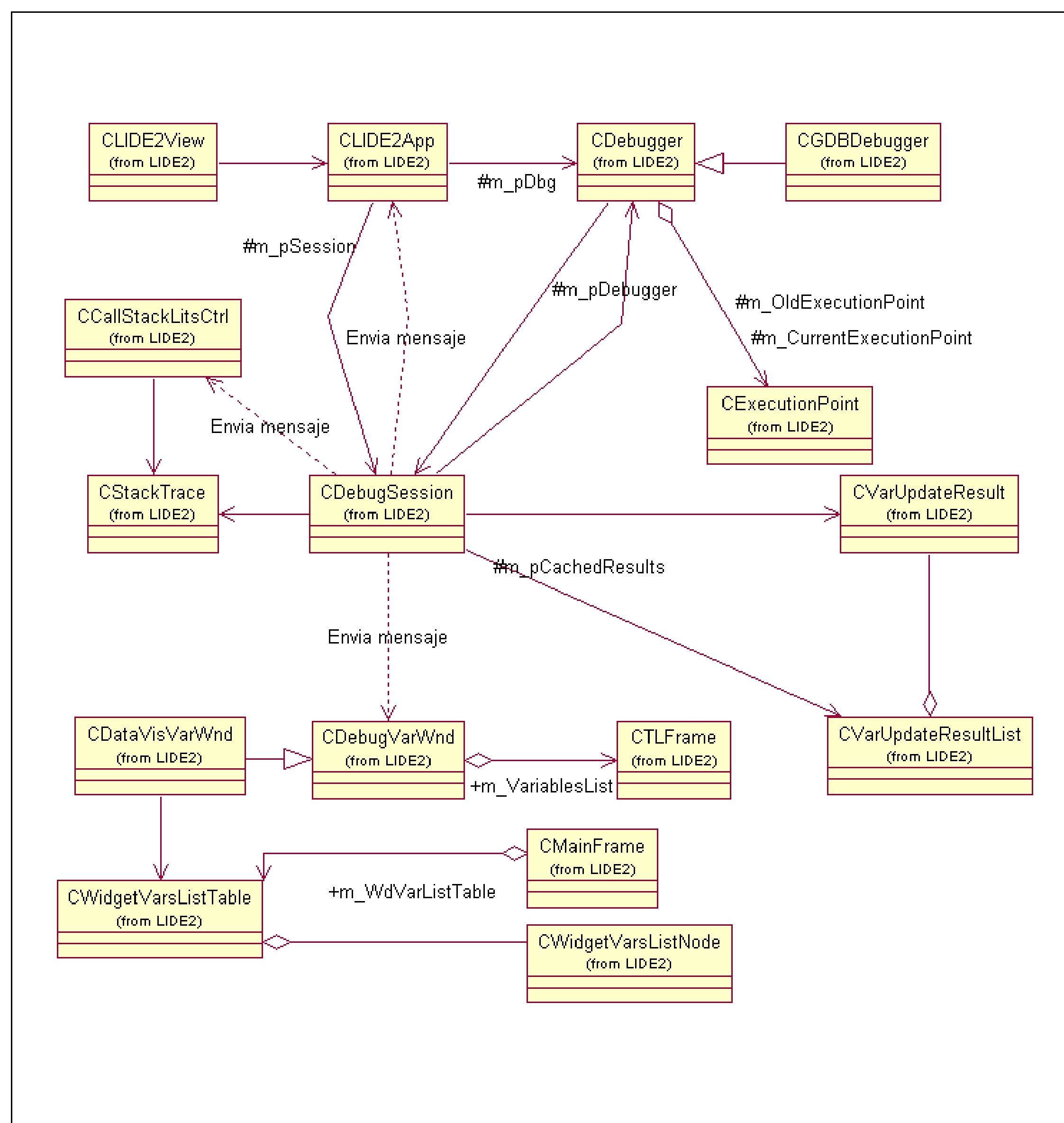


Figura 4.22. Interacción de la Interfase del Depurador con la Interfase de Usuario

4.4.2 Modelo Dinámico

Es necesario describir la secuencia de acciones que ocurren en las partes más importantes del componente de interacción con el depurador. En esta sección se describe el inicio de una sesión de depuración, la inserción de un variable para poder observar su valor, el manejo de la parada de una aplicación por haber alcanzado un punto de ruptura y la actualización del valor de las variables y de la pila de llamadas a funciones.

La exploración de las variables, estructuras y la pila de llamadas a funciones solo puede realizarse cuando la aplicación depurada se encuentra suspendida a causa del encuentro de un punto de ruptura. Esto no es una limitación del sistema, es en realidad la manera como los depuradores de aplicaciones trabajan.

El diseño de la secuencia de pasos durante la sesión de depuración esta dictado por el depurador que se utiliza como back-end. Dicho de otra manera, el componente de interacción con el depurador simula la secuencia de pasos que seguiría un usuario si usara directamente el depurador.

Inicio de una Sesión Depuración:

Una sesión de depuración se inicia con la construcción del objeto depurador correcto por parte de la clase CDebugSession. Una particularidad de la inicialización de una sesión de depuración es la obtención de los puntos de ruptura. Los puntos de ruptura se almacenan en la clase CDebugSession justo antes de crear el objeto depurador, ya que su posición dentro de un archivo de código fuente puede cambiar durante la etapa de edición de código. Es por esto que el único objeto que conoce todo el tiempo la posición de los puntos de ruptura es el control de edición y es a este control al que se pide la ubicación de los puntos de ruptura.

De manera más precisa, se describen los pasos de la inicialización de una sesión de depuración:

1. El objeto aplicación recibe un evento originado en la interfase de usuario. El primer paso que se ejecuta es la verificación de que no se este depurando actualmente.

2. Se obtiene un objeto proyecto que representa el proyecto actual. Este objeto mantiene la ruta donde se encuentra el ejecutable compilado.
3. Se pide al objeto vista que inicializa los puntos de ruptura en la clase CDebugSession.
4. Se presentan los paneles que tienen relación con la depuración y se ocultan los paneles relativos a la compilación para ahorrar espacio de pantalla.
5. El objeto aplicación pide la clase CDebugSession que cree un objeto depurador. La clase CDebugSession conoce que tipo de depurador crear ya que tiene una referencia al proyecto y puede preguntar al objeto proyecto cual es su tipo.
6. El objeto aplicación pide que se ejecute el depurador externo. En respuesta a esto el objeto depurador crea la línea de comando para ejecutar el depurador. La línea de comando contiene el nombre de la aplicación que se desea depurar.
7. El objeto depurador invoca LoadDebugger() para ejecutar el depurador externo. El objeto depurador delega la tarea de ejecutar el depurador y redirigir la entrada y salida a la

clase CSpawn. Lo último opera de manera similar a la ejecución del compilador externo.

8. Una vez que el depurador externo esta ejecutándose, el objeto aplicación pide que se inicialicen los puntos de ruptura en el depurador externo.
9. Los puntos de ruptura se inicializan por medio de la función InsertBreakPoint(). La cual necesita conocer el archivo de código fuente y la línea donde se desea insertar el punto de ruptura.
10. La función InsertBreakPoint(), al igual que la gran mayoría de funciones expone la interfase del objeto depurador, construye una cadena de texto para el comando pedido y lo envía por una interfase común. La interfase común es la función SendCommand(). Una vez enviado el comando, se sienta a esperar que el comando sea procesado para poder obtener el resultado del comando.
11. Finalmente el objeto aplicación le pide al objeto depurador que empiece a ejecutar la aplicación que se quiere depurar.

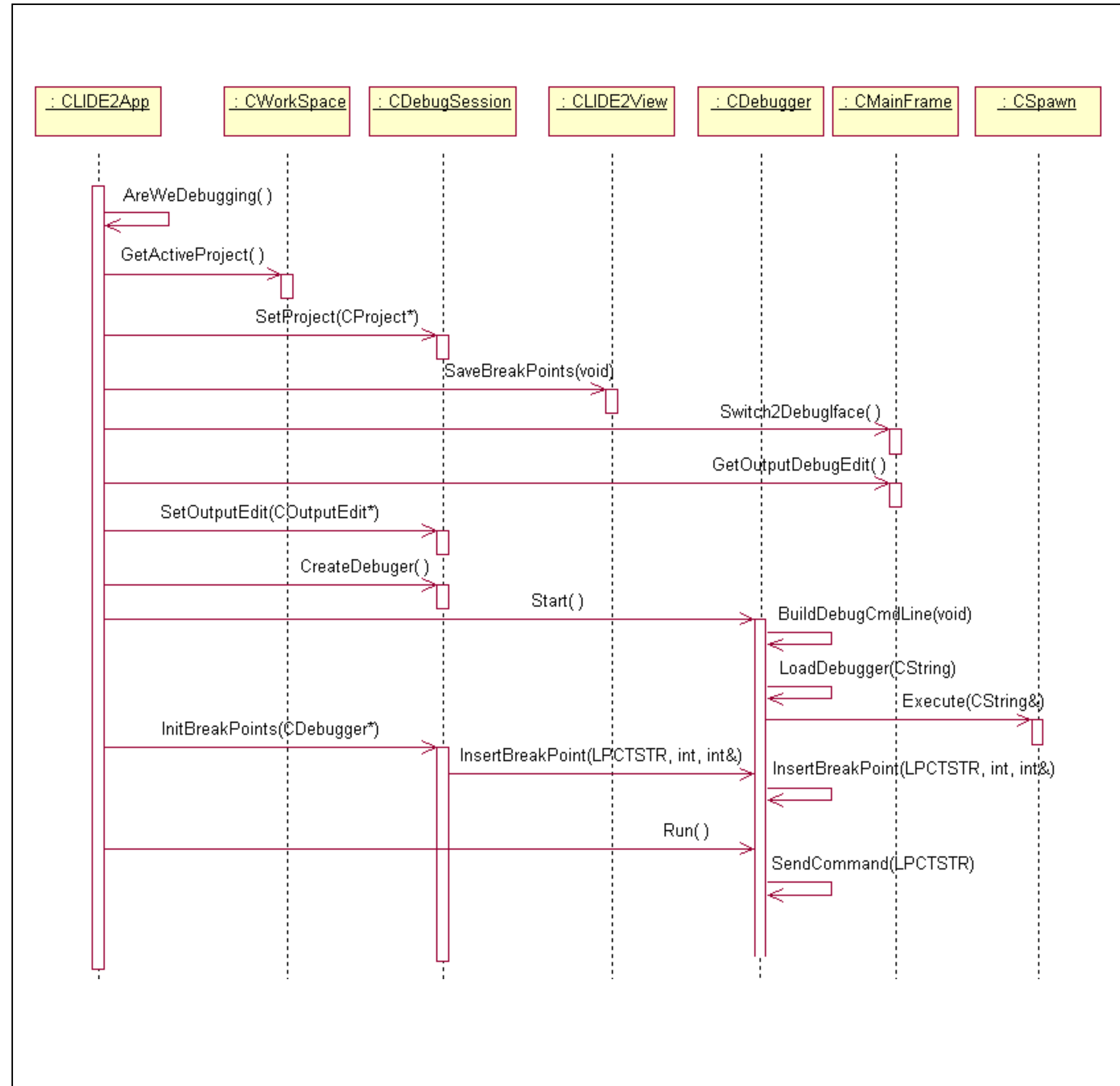


Figura 4.23. Diagrama de Secuencia del Inicio de una sesión depuración

Inserción Simple de una Variable:

Para observar el valor de una variable es necesario indicarle al depurador externo que nos interesa una variable o una expresión en especial. La inserción de una variable solo puede efectuarse cuando la aplicación depurada se encuentra suspendida por el encuentro de un punto de ruptura, ya que este es el único instante en el que el depurador externo acepta este tipo de comandos.

Las variables que se desean explorar son recordadas por el objeto de tipo CDebugVarWnd. Este objeto se comunica con el depurador a través del objeto sesión de depuración.

De las variables siempre se desea saber su tipo, su valor, su estructura (variables miembro), la dirección de memoria donde se encuentran y su tamaño en bytes. El tipo, el valor y la estructura de las variables se presentan el usuario en la interfase grafica. La dirección de memoria y su tamaño se emplean en la visualización de estructuras para ayudar a detectar cambios en los valores de los punteros y actualizar la visualización de estructuras de manera automática.

Es necesario indicar que para el depurador seleccionado es necesario mantener alguna estructura que permita hacer traducciones del nombre de la variable. Esto se debe a que el depurador emplea un nombre genérico para cada variable que se desea explorar y se debe emplear este nombre genérico para cualquier llamada sucesiva al depurador. Para ilustrar esto supongamos que se tiene una variable de tipo entero cuyo nombre es nIndex. Para observar el valor de esta variable se debe informar al depurador externo que estamos interesados en esta variable. Para GDB esto se logra por medio del comando -var-create nIndex. Este comando retornara el nombre genérico autogenerado de la variable, supongamos que el GDB retornó como nombre genérico var1. Ahora se desea obtener el valor de la variable nIndex. Para obtener el valor de la variable se emplea la instrucción -var-evaluate-expression var1. Es decir GDB no conoce de la variable nIndex, solo conoce de var1. Por lo tanto, es tarea del IDE mantener la relación entre las variables que se tienen en el código fuente con las variables que conoce el GDB. Todo esto con el objetivo de reducir la carga cognitiva del usuario final. A continuación se describe la

secuencia de interacción para la inserción simple de una variable:

1. El objeto ventana de variables le pide al objeto sesión de depuración que cree una variable y retorne el nombre generado por el depurador externo. El objeto sesión delega esta tarea al objeto depurador.
2. Con el nombre generado por el depurador el objeto ventana de variables procede a obtener el valor, el tipo de dato, la dirección de memoria y el tamaño de la variable.
3. Si la variable resultara ser de un tipo de dato estructurado, es decir una clase o una estructura, el objeto ventana de lista de variables pedirá al objeto sesión de depuración la descripción de la estructura. Este paso funciona de manera recursiva hasta que se llegan a variables miembro con tipos de dato primitivos. Se consideran como tipos de datos primitivos los mismos tipos de dato que el compilador considera como primitivos, es decir carácter entero, doble, puntero, etc.
4. Una vez que se obtiene toda esa información la ventana de variable procede a actualizar la interfase de usuario.

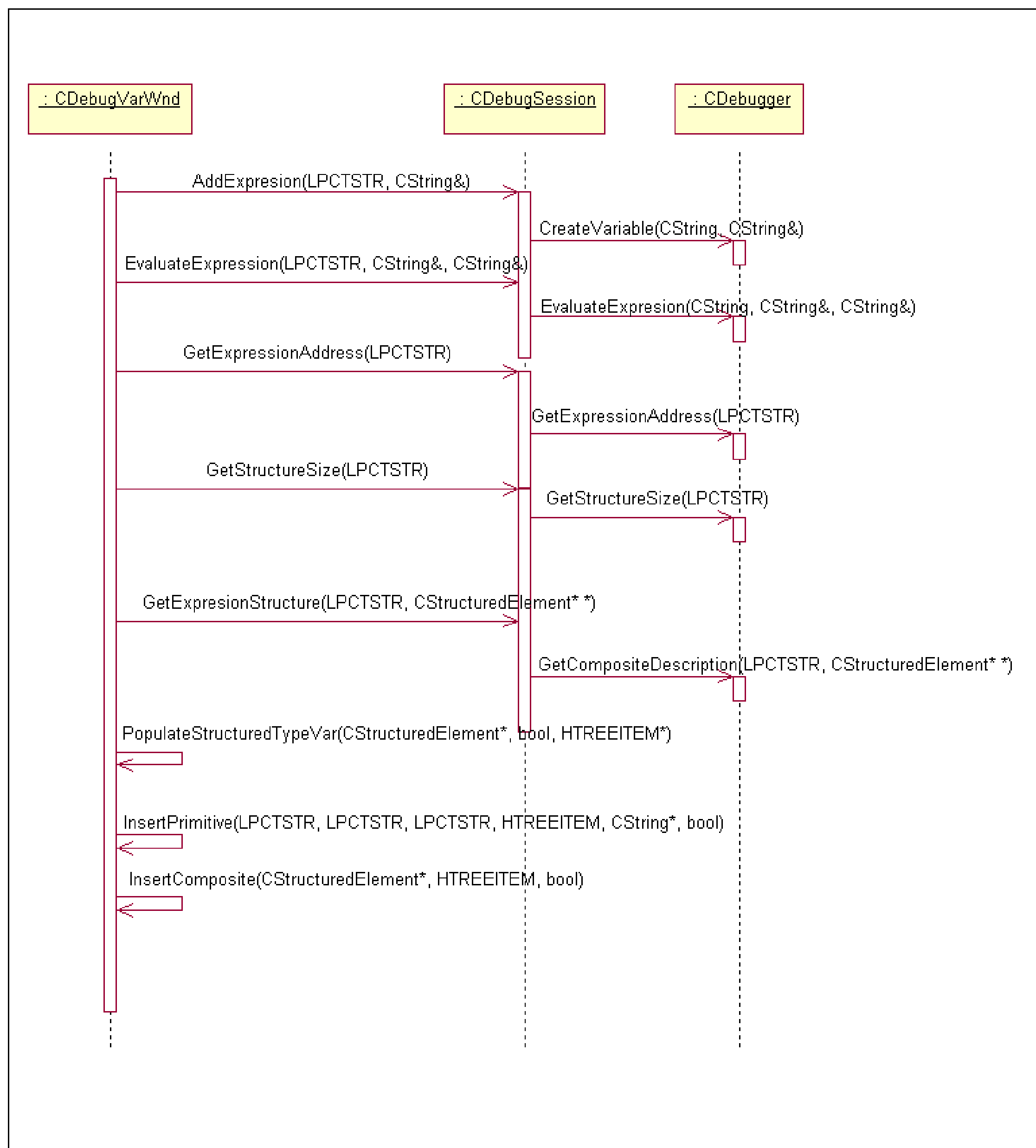


Figura 4.24. Diagrama de Secuencia de la Inserción de una Variable

Inserción de una Variable para Visualizarla como Estructura:

Existe una estrecha relación entre la inserción simple de variable descrita en la sección anterior y la inserción de una variable para visualizarla como una estructura. De hecho se puede ver la segunda (visualización como estructura) como una extensión de la primera (simple).

Además en el diagrama de clases de la figura 4.21 se puede observar que la clase que presenta de manera tabular las variables (la cual es utilizada en la visualización como estructura) que se están visualizando hereda de la clase que presenta la lista variables (la cual se utiliza en la inserción simple de variable).

Este es el antecedente para poder describir la inserción de una variable para mostrarla como estructura. La inserción ocurre de manera similar a la inserción simple de una variable, con la diferencia de que existe un desvío en la lógica de ejecución. Antes de presentar la variable en la lista de variables se notifica a la clase CDataVisVarWnd para que pueda comunicar al panel de visualización de estructuras (descrito en la sección 4.3.1

Modelo Estático de la Arquitectura del Sistema) que se tiene una nueva variable para insertar. El panel de visualización de estructuras crea los elementos necesarios para mostrar la variable en la interfase como una estructura. Esta secuencia de eventos ocurre de manera independiente de si la variable fue agregada por medio del panel de visualización de estructuras o la lista de variables. A continuación se describe la secuencia de interacción para la inserción de una variable para visualizarla como estructura:

1. El objeto de tipo CDataVisVarWnd, la lista de variables, es notificado que una variable ha sido agregada.
2. La lista una vez que tiene información de la variable que se desea insertar, notifica al panel de visualización de estructuras, pasando toda la información relativa a la variable.
3. El panel de visualización de estructuras delega la tarea de visualización al objeto de tipo CDataVisCanvas.
4. La clase CDataVisCanvas determina alguna ubicación para el nuevo elemento (CDataVisWidget) que se va a presentar. Inicializa al elemento que representa la

estructura y lo almacena dentro de una lista para posterior referencia.

5. Si es que se debe enlazar el nuevo elemento con otro anterior (por ejemplo, si se esta desreferenciando un puntero), se crean un enlace por medio de la clase CWidgetLinkMgr. La clase CWidgetLinkMgr crea un objeto que representa la relación entre los dos elementos y realiza todas las inicializaciones necesarias.

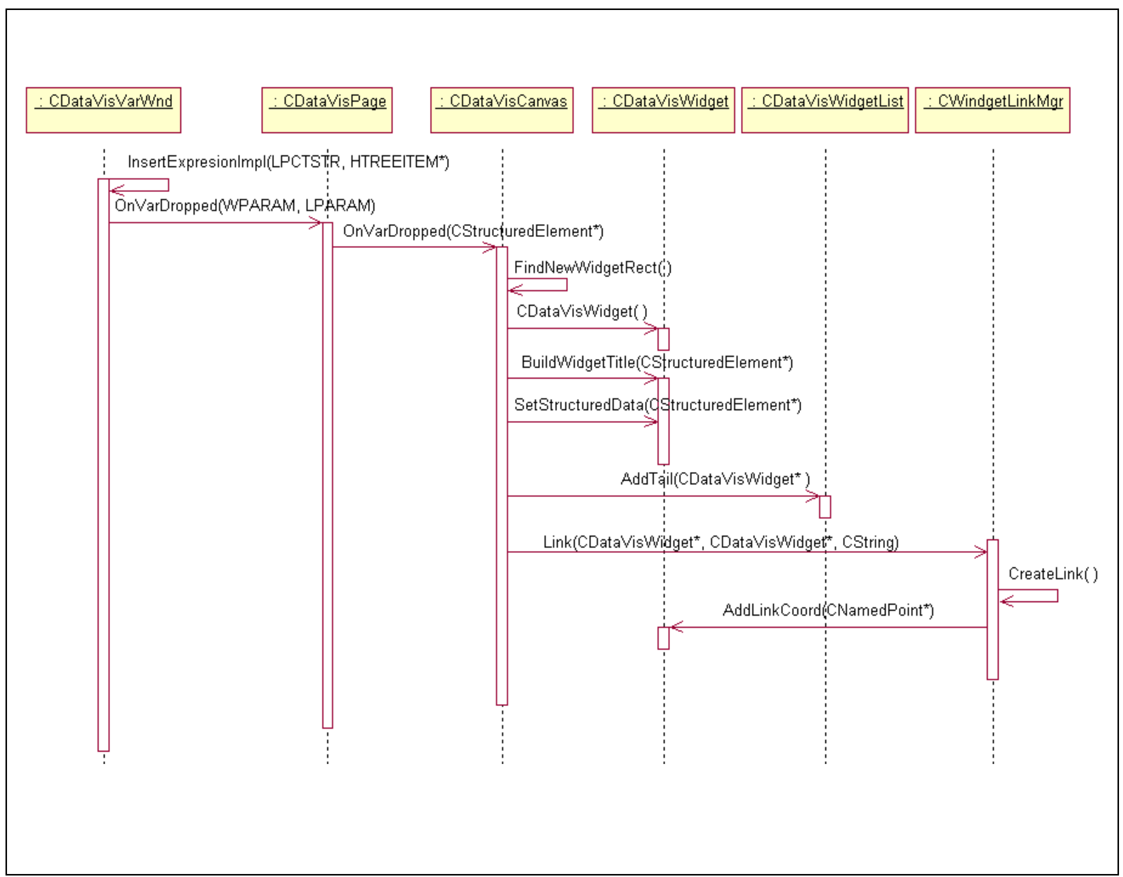


Figura 4.25. Diagrama de Secuencia de la Inserción de una Variable para Visualizarla como Estructura

Encuentro de un Punto de Ruptura:

El punto de inicio de la notificación del encuentro de un punto de ruptura es el depurador externo. Cuando el depurador externo encuentra un punto de ruptura suspende la aplicación que se esta depurando. Luego de suspender la aplicación imprime en la pantalla por que fue suspendida la aplicación, en este caso por un punto de ruptura.

El objeto depurador permanentemente esta leyendo e interpretando la salida estándar del depurador externo. Cuando el depurador externo imprime la notificación del encuentro de un punto de ruptura, el objeto depurador crea las estructuras de datos necesarias para notificar a otros componentes del sistema que podrían estar interesados en esta notificación. La ruta del archivo de de código fuente y el numero de línea donde se encontró el punto de ruptura es la información mas importante asociada con un punto de ruptura. A continuación se describe la secuencia de interacción del encuentro de un punto de ruptura:

1. El objeto depurador invoca el método ProcessOutput() cada vez que se recibe texto desde el depurador externo. Si durante la ejecución de ProcessOutput() se encuentra la salida de un comando que requiere atención, pro ejemplo cuando se encuentra un punto de ruptura, el objeto depurador elige el método apropiado para manejar la salida del depurador externo.
2. De acuerdo a la manera en el que el depurador externo seleccionado (GDB) interactúa, primero se notifica que la aplicación depurada ha sido suspendida, luego se informa la razón, en este caso por que se alcanzo un punto de ruptura.
3. El método HandleStopped() es el responsable por crear las estructuras de datos que contendrán la información sobre la razón de la suspensión de la aplicación e información relevante al estado de la aplicación suspendida (por ejemplo en que línea y archivo de código fuente se encuentra suspendida).
4. El objeto depurador siempre actualiza el punto actual de ejecución de la aplicación cuando la aplicación depurada es suspendida.

5. El objeto depurador notifica al objeto sesión de depuración para que se este ultimo el responsable de notificar a otras partes de la aplicación que se encontró un punto de ruptura.
6. El objeto sesión de depuración pide al objeto aplicación que cambie el punto actual de ejecución.
7. Luego, el objeto sesión de depuración notifica a la ventana de variables y a la ventana de la pila de llamada de funciones.

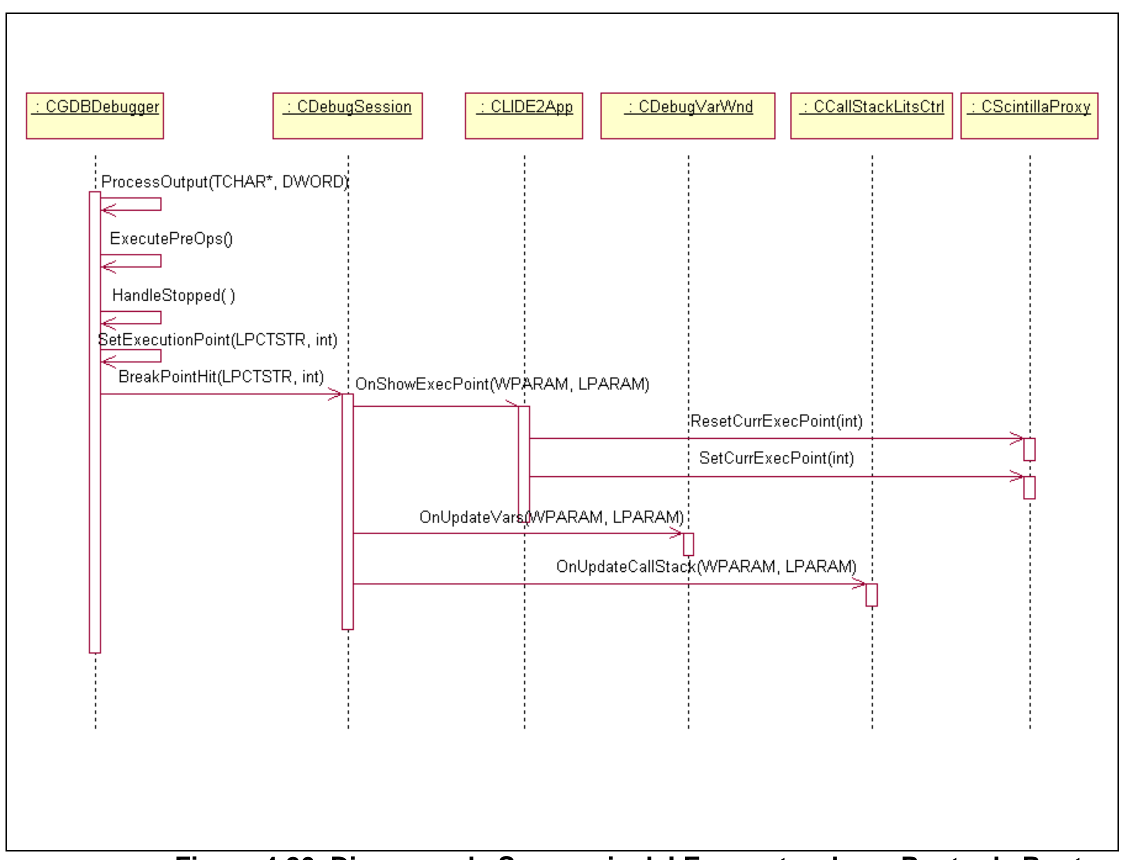


Figura 4.26. Diagrama de Secuencia del Encuentro de un Punto de Ruptura

Actualización del Valor de una Variable:

El IDE guarda el nombre de las variables que el usuario desea explorar. Cuando se suspende la aplicación el IDE intenta actualizar los valores de estas variables. Decimos que intenta dado que es posible que la variable no se encuentre definida en el contexto actual.

El objeto de tipo CDebugVarWnd es el encargado de administrar y presentar las variables de la aplicación depurada. Cuando recibe la notificación de suspensión de la aplicación depurada le pregunta al objeto sesión de depuración si el valor de alguna de las variables ha cambiado. Para simplificar el diseño se aprovechó de la existencia de la capacidad del depurador seleccionado de suministrar la lista de variables que han cambiado de valor desde la última vez que la aplicación fue suspendida. Esto es importante para el diseño ya que el depurador notifica solamente una vez el cambio de valor de las variables. Es decir, la primera vez que se pide la lista de variables que han cambiado su valor, el depurador responde de la manera esperada. Sin embargo, invocaciones siguientes

retornaran una lista vacía, indicando que ninguna variable ha sido modificada. Este comportamiento se mantienen hasta que la aplicación vuelva ha ser suspendida. Sabiendo que posiblemente exista más de un componente del sistema que pudiera requerir la lista de variables modificadas cada vez que se suspende la aplicación, se incorporó un cache de la lista de variables modificadas. Este cache esta contenido dentro del objeto sesión de depuración. A continuación se describe la secuencia de interacción de la actualización del valor de una variable:

1. Al ser notificado de la suspensión de la aplicación el objeto de tipo de CDebugVarWnd invoca el método UpdateVars del objeto sesión de depuración, este último delega la tarea al objeto depurador creado.
2. El objeto depurador crear el comando correcto para obtener el lista de variables modificadas y la retorna al objeto sesión de depuración. El objeto sesión de depuración guarda la lista retornada por el depurador en un cache para futuros accesos por parte de otros componentes.

3. Una vez que el objeto de tipo CDebugVarWnd conoce cuales han sido las variables que han cambiado de valor, este objeto emplea el nombre de las variables para obtener el nuevo valor y actualizar la interfase de usuario.

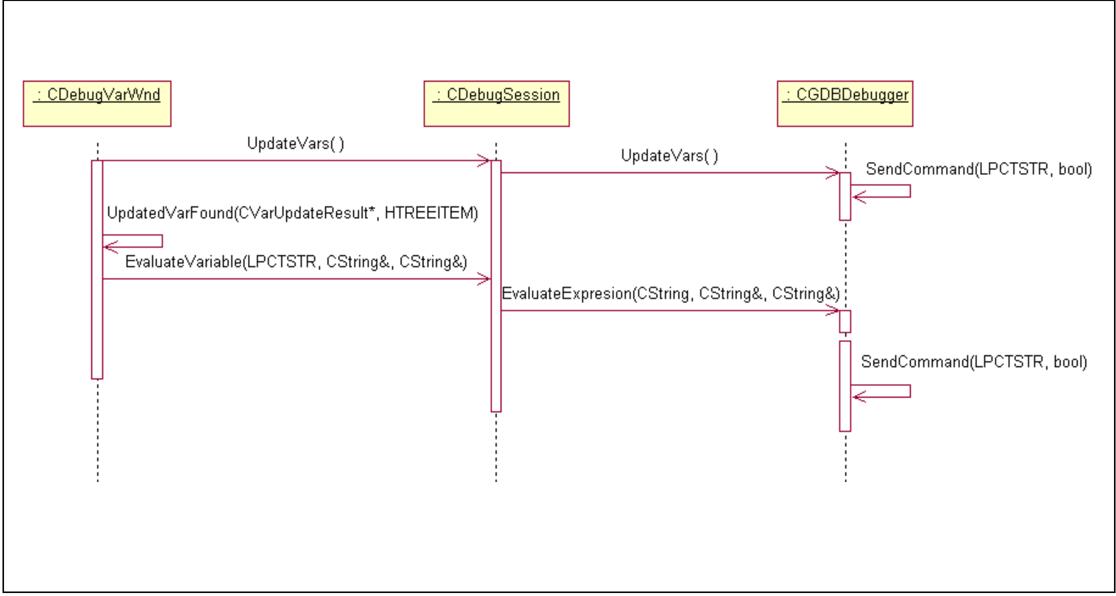


Figura 4.27. Diagrama de Secuencia de la Actualización del Valor de una Variable

4.4.3 Protocolo de comunicación entre el IDE y el depurador

La presente sección describe el diseño de los comandos enviados al depurador y el procesamiento de la salida, así como la interfase que provee el depurador externo seleccionado.

El protocolo de comunicación con el depurador ejecutado por el componente de la interfase con el depurador, de forma mas especifica, es ejecutado por una clase depurador derivada de la clase abstracta CDebugger.

EL protocolo a diseñarse tiene como objetivo fundamental enviar los comandos hacia el depurador y obtener los valores de respuesta. Otro objetivo es facilitar la implementación de manera extremadamente confiable. La razón de tal necesidad de confiabilidad yace en las posiblemente complejas interacciones con el depurador y el deseo de presentar el IDE sólidamente integrado con el depurador externo.

Para el actual diseño del IDE se considera únicamente un depurador externo, en secciones anteriores se selecciono a

GDB como el depurador externo. De acuerdo a esto, el resto de la presente sección trata el caso específico de GDB.

Como se mencionó en la sección 2.3, los depuradores modernos de línea de comandos consideran en su diseño que, además poder ser controlados por una persona, puedan ser controlados por otra aplicación. En el caso de GDB esto se manifiesta con la interfase GDB/mi, la cual se define como una interfase de texto basada en líneas orientada hacia maquinas. Actualmente la interfase GDB/mi se encuentra bajo desarrollo y esta sujeta a cambios.

El objetivo de la interfase GDB/mi es simplificar el control de GDB por parte de otras aplicaciones. Esto se intenta lograr definiendo una gramática para las expresiones de entrada y salida del depurador.

Gramática de los Comandos:

Para describir los comandos de entrada y la salida del depurador se utilizara la siguiente notación:

| separa dos alternativas

[*algo*] indica que *algo* es opcional

(*grupo*)* indica que *grupo* puede repetirse cero o mas veces.

(*grupo*)+ indica que *grupo* puede repetirse una o mas veces.

“cadena” indica una cadena de texto

Cuadro 6. Notación empleada

Sintaxis de Comandos:

La sintaxis de los comandos mostrada a continuación fue extraída del manual de GDB, no esta descrita de la forma EBNF pero para fines prácticos sirve como gramática.

(P1) comando → [token] ”-“ operacion (“ “ opcion)* [“-”] (" " parametro)* nl

(P2) token → cualquier secuencia de dígitos

(P3) opcion → "-" parametro [" " parametro]

(P4) parametro → secuencia-no-en-blanco | c-string

(P5) operacion → cualquier operación permitida por la interfase GDB/mi

(P6) secuencia-no-en-blanco → cualquier cosa que no contenga caracteres especiales tales como "-", nl, """ y " "

(P7) c-string → " cadena_ascii “

(P8) nl → CR | CR-LF

token es cualquier secuencia de dígitos que se pasa junto al comando y es devuelta en la respuesta del comando.

Cuadro 7. Sintaxis de Comandos de GDB/mi

Como se puede observar la interfase de comandos es bastante general y sencilla. Luego de definir la sintaxis de salida se presentaran algunos ejemplos.

Sintaxis de Salida:

(P1) output → (out-of-band-record)* [result-record] "(gdb)" nl
(P2) result-record → [token] "^" result-class ("," result)* nl
(P3) out-of-band-record → async-record | stream-record
(P4) async-record → exec-async-output | status-async-output | notify-async-output
(P5) exec-async-output → [token] "*" async-output
(P6) status-async-output → [token] "+" async-output
(P7) notify-async-output → [token] "=" async-output
(P8) async-output → async-class ("," result)* nl
(P9) result-class → "done" | "running" | "connected" | "error" | "exit"
(P10) async-class → "stopped" | otros (donde otros serán añadidos de acuerdo a las necesidades – aun bajo desarrollo).
(P11) result → variable "=" value
(P12) variable → string
(P13) value → const | tuple | list
(P14) const → c-string
(P15) tuple → "{}" | "{" result ("," result)* "}"
(P16) list → "[]" | "[" value ("," value)* "]" | "[" result ("," result)* "]"
(P17) stream-record → console-stream-output | target-stream-output | log-stream-output
(P18) console-stream-output → "~" c-string
(P19) target-stream-output → "@" c-string
(P20) log-stream-output → "" c-string
(P21) c-string → " cadena_ascii “
(P22) nl → CR | CR-LF
(P23) token → cualquier secuencia de dígitos

Cuadro 8. Sintaxis de la Salida de GDB/mi

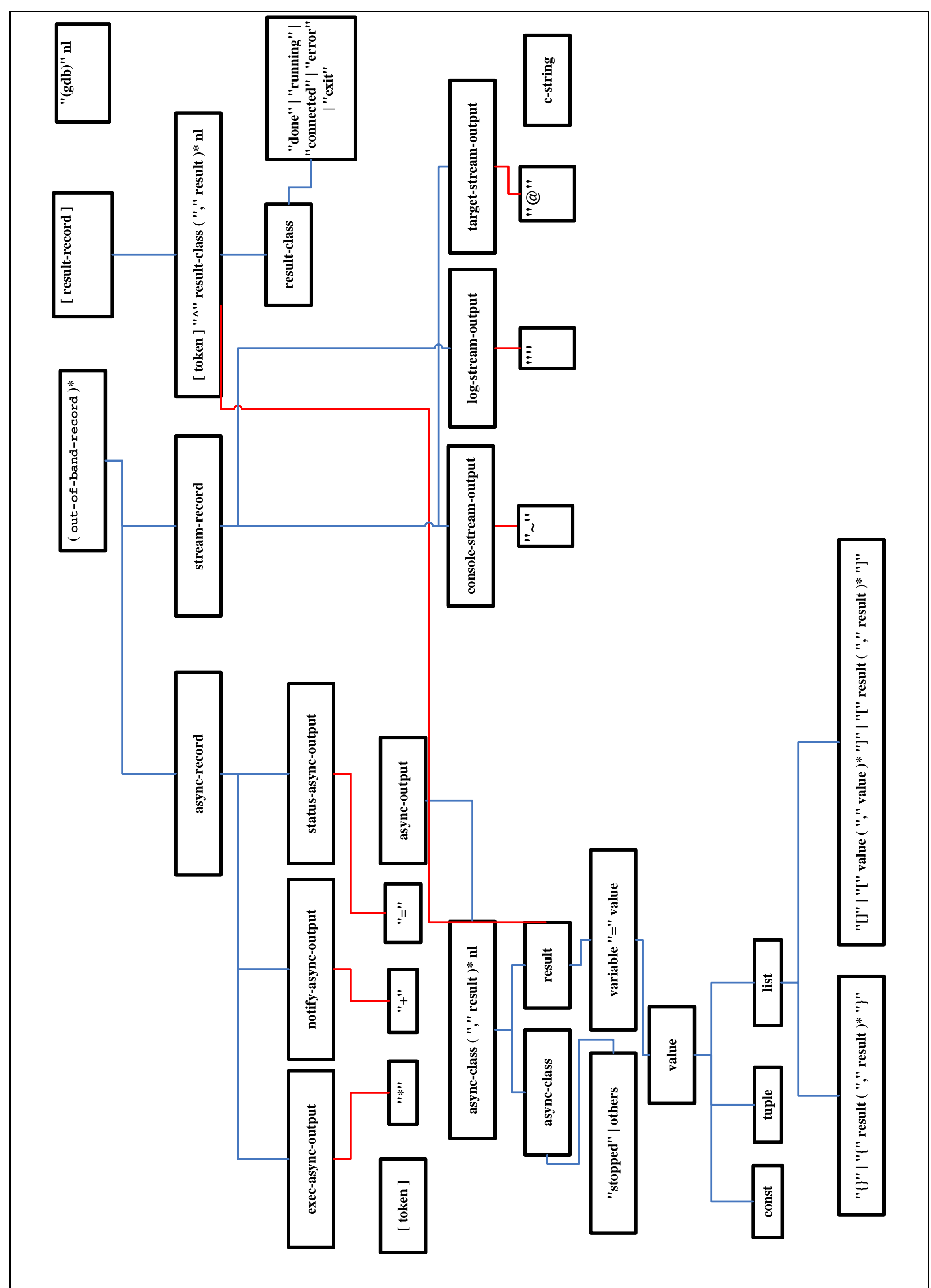
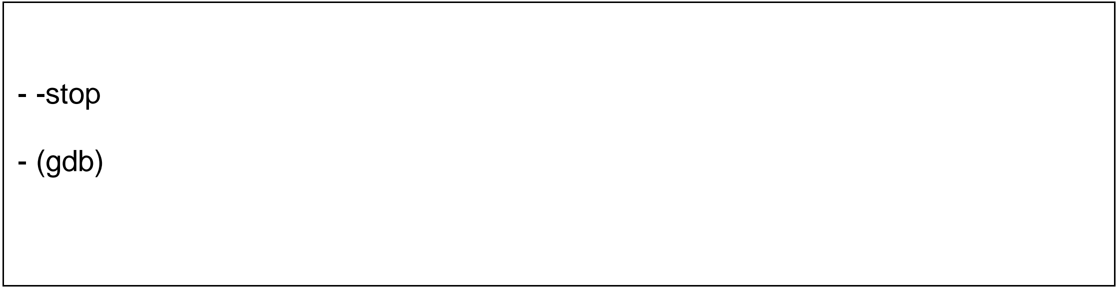


Figura 4.28. Diagrama de bloques de la gramática del depurador

La figura 4.28 muestra un diagrama de bloques de la sintaxis de la salida de GDB/mi, en el diagrama los bloques representan símbolos de la gramática (terminales y no-terminales) y las líneas muestran la jerarquía de manera descendente. El diagrama no esta completo ya que no muestra las propiedades recursivas de la gramática, sin embargo ayuda a comprender la estructura de la gramática.

El siguiente es un ejemplo de la suspensión del programa depurado:



```
- -stop
- (gdb)
```

Cuadro 9. Suspensión del Programa Depurado

A este comando el depurador responde con luego con:

```
- *stop,reason="stop",address="0x123",source="a.c:123"
- (gdb)
```

Cuadro 10. Respuesta del Depurador a un Comando

Luego resume el programa depurado hasta que encuentra un punto de ruptura previamente insertado:

```
-exec-continue
^running
(gdb)
@Hello world
*stopped,reason="breakpoint-hit", bkptno="2", ↳frame={func="foo",args=[] ,
↳file="hello.c",line="13"}
(gdb)
```

Cuadro 11. Respuesta del Depurador a un Comando (continuación)

Como se puede observar la sintaxis de la salida del depurador es significativamente más compleja que la sintaxis de los comandos. La información más relevante para el diseño que podemos obtener de la sintaxis de la salida es la siguiente:

- La salida de un comando termina con la cadena “(gdb)” seguida de un salto de línea.
- El depurador clasifica la salida en varios tipos de registros (records).

- Cada línea comienza por un carácter que identifica el tipo de record.
- Los registros que más interesantes son el registro de resultado (result-record), porque contiene los valores retornados por algún comando y los registros de salida asincrónica del ejecutable (exec-async-output) que indican que la aplicación depurada ha sido suspendida.

El hecho de que los resultados de los comandos terminen de manera común sirve como una bandera para indicar el fin del procesamiento de un resultado de un comando.

En lo referente a los registros de salida, el depurador define los siguientes registros de salida: registros de resultado, registros de flujo (stream) y registros fuera de banda (out-of-band).

Los registros de flujo representan respuestas textuales a ciertos comandos o texto enviado por la aplicación depurada y mensajes de depuración del GDB. Se considera que estos registros deben ser mostrados textualmente y no pasar por ningún procesamiento debido a que solamente informan estados.

Los registros de fuera de banda indica la suspensión del programa depurado. Esto puede ocurrir debido a que el programa depurado termino, en tal caso se debe notificar al IDE para que reconfigure la interfase, o por que se alcanzo un punto de ruptura, así mismo se debe notificar al punto al IDE para que actualice la interfase de usuario y notifique a otras partes del sistema este hecho.

Finalmente los registro de resultado contiene el resultado del comando ejecutado por el GDB, además indican el si la operación fue exitosa o no. Estos registros deberán ser procesados para obtener los valores que retorno el GDB.

Una vez descrita la interfase GDB/mi del depurador externo podemos pasar a la descripción de la manera en que el IDE aprovechara la misma.

El diagrama de la figura 4.29 muestra el esquema del envío de comandos al depurador externo y la recuperación de los

resultados a partir del flujo de caracteres enviados desde el depurador.

El envío de comandos lo realizara algún componente del IDE. Este componente debe generar la cadena de texto del comando que se desea enviar, encapsular el comando en un objeto que represente un trabajo y colocarlo en una cola de trabajos. Posteriormente otro componente retirara el trabajo de la cola, extraerá el comando y lo enviara hacia el GDB.

La construcción del comando no es especialmente interesante, pues simplemente consiste en el crear una cadena de texto ceñida a cada comando. El problema interesante consiste en mantener activa la interfase de usuario cuando se esta enviando un comando. Debido a que los comandos pueden ser enviados en cualquier orden y con cualquier frecuencia se considero que el componente que envía los comandos no debe ser el mismo componente que espera la respuesta del comando, por que de otra manera un comando que se ejecute por largo tiempo congelaría la interfase y esto produciría confusión al usuario. Por esta razón el comando se coloca en una cola hasta que sea su turno de ejecutarse. En la práctica

no se espera que la cola tenga muchos elementos así que se piensa que no existirá una demora significativa.

Por otro lado, la recuperación de los valores de retorno de un comando se ejecuta de manera asincrónica. Existe un componente encargado de obtener el flujo de caracteres, interpretar los caracteres y notificar a algún componente del sistema que ya se tiene la respuesta de un comando enviado con anterioridad. Para poder representar los valores de retorno de manera genérica se emplean cadenas de texto organizadas como tuplas de la forma (Nombre,Valor), para así poder recuperar un valor de retorno especifico y, debido a que se almacenan como cadenas de texto, poder contener cualquier tipo de dato.

El componente encargado del procesamiento del flujo de caracteres de la salida del depurador externo abarcará la lógica de la sintaxis de la salida del depurador y la maquina de estados necesaria para procesar tal sintaxis.

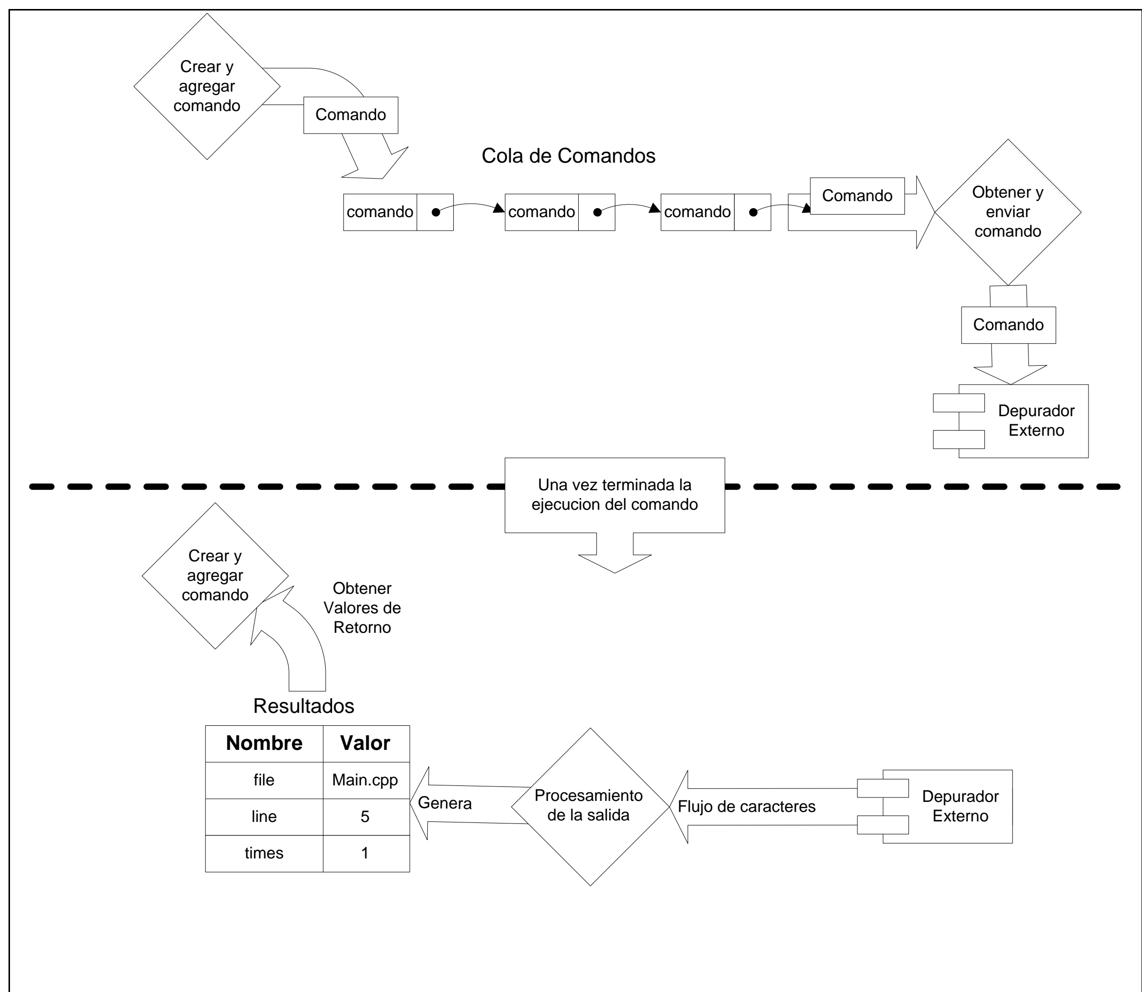


Figura 4.29. Diagrama de la Comunicación con GDB

En este punto resta por definir que comandos provistos por GDB se emplearan para implementar las funciones que la interfase con el depurador externo brinda al resto del sistema. A continuación se describen los comandos de GDB que se deben emplear para implementar las funciones de depuración del IDE.

Inicio / Fin de la sesión de depuración:

```
gdb.exe -interpreter=mi --silent "c:\la_ruta\la_app.exe"
```

Cuadro 12. Inicio de la sesión de depuración

Se inicia la sesión de depuración invocando el depurador pasando como argumento el la ruta y el nombre del ejecutable a depurar. Además del nombre de ejecutable se especifica que se quiere que el depurador presente una interfase apropiada para ser controlado por una aplicación externa con el argumento: -interpreter=mi. El argumento –silent le indica al depurador que no presente mensajes informativos, como por ejemplo las librerías que se están usando o la versión del depurador, con el objetivo de simplificar el procesamiento de la salida del depurador y reducir la carga de información que se le presenta al usuario.

```
-gdb-exit
```

Cuadro 13. Fin de la sesión de depuración

Para terminar la sesión de depuración se invoca el comando –gdb-exit dentro de la sesión interactivo con el depurador. Esto comando termina el programa depurador, en caso de que aun se este ejecutando, y termina el depurador.

Arranque / Terminación del programa depurado:

```
-exec-run
```

Cuadro 14. Arranque del programa depurado

El ejecutable especificado en la línea de comando se empieza a ejecutar cuando se invoca el comando -exec-run.

```
-kill
```

Cuadro 15. Terminación del programa depurado

Termina la aplicación que se esta depurado. Este comando se puede invocar solamente cuando la aplicación que estamos depurando se encuentra suspendida.

Inserción / Eliminación de un punto de ruptura:

```
-break-insert archivo:Linea
```

Cuadro 16. Inserción de un punto de ruptura

El comando de inserción de puntos de ruptura necesita de dos argumentos: La ruta del archivo de código fuente y la línea donde se desea insertar el punto de ruptura. El comando retorna un identificador generado por el depurador del punto de ruptura recientemente insertado.

```
-break-delete Identificador
```

Cuadro 17. Eliminación de un punto de ruptura

Para eliminar un punto de ruptura insertado se pasa como argumento el identificador retornado por el comando de inserción.

Inserción / Eliminación de una variable

```
-var-create - * variable
```

Cuadro 18. Inserción de una variable

Para crear una variable, además del nombre de la variable, se necesita indicar el nombre que deseamos que tenga la variable y el nivel de la pila de llamadas de funciones donde se quiere crear la variable.

Es posible pedirle al depurador que genere automáticamente el nombre de la variable al pasar '-' como argumento. El argumento * le indica al depurador que la variable debe ser creada en el nivel de pila de llamadas actual. El uso de las dos últimas características mencionadas reduce el esfuerzo de desarrollo ya que ahora el depurador es el encargado de manejar esos detalles.

Este comando retorna el nombre generador por el depurador. Este nombre se emplea mas adelante en todas las funciones de consulta de información sobre la variable.

```
-var-delete NombreDeVariable
```

Cuadro 19. Eliminación de una variable

Para eliminar una variable se pasa como argumento el nombre de la variable generado por el depurador.

Consulta información sobre una variable:

```
-var-evaluate-expression NombreDeVariable
```

Cuadro 20. Consulta del tipo y valor de una variable

El tipo de dato y el valor de una variable se obtienen de manera simultánea con el comando -var-evaluate-expression. De igual

manera como otros comandos de consulta, este comando necesita el nombre generador por el depurador.

```
-data-evaluate-expression &(NombreDeVariable)
```

Cuadro 21. Consulta de la dirección de memoria de una variable

```
-data-evaluate-expression sizeof(NombreDeVariable)
```

Cuadro 22. Consulta del tamaño en memoria de una variable

Para obtener la dirección de memoria y el tamaño de una variable se aprovecha que el comando -data-evaluate-expression puede trabajar sobre expresiones de variables. En el caso de la consulta de la dirección de memoria se emplea el operador de dirección de C (&) para pedir la dirección de la variable. Y en el caso de la consulta del tamaño de la variable se utiliza el operador de consulta de tamaño (sizeof).

```
-var-list-children NombreDeVariable
```

Cuadro 23. Consulta de la estructura de una variable

La estructura de una variable se obtiene por medio del comando -var-list-children. Este comando retorna el nombre, el nivel de acceso (público o privado en el caso de clases) y el nombre de las variables que constituyen un arreglo o una variable de tipo de dato estructurado (tales como estructuras, uniones y clases).

Consulta de la pila de llamadas:

```
-stack-list-frames
```

Cuadro 24. Consulta de la pila de llamadas

Este comando retorna una lista ordenada que representa la pila de llamadas de funciones. Cada ítem contienen el nivel en la

pila, nombre de la función, la dirección de memoria desde donde se invoco a la función, archivo de código fuente y línea.

CAPÍTULO 5

5. IMPLEMENTACIÓN

5.1 Componentes de Terceros

A continuación se describirá la manera como se realizó la implementación para el uso de los componentes de terceros. Se tratarán temas tales como los pasos necesarios para integrar los componentes a nuestro proyecto y las interfaces de programación de estos componentes.

Scintilla

Para poder utilizar Scintilla es necesario compilar la librería primero ya que se distribuye en forma de código fuente solamente. Para compilar Scintilla debemos abrir la ventana del intérprete de

comandos de Visual Studio .Net, este interprete de comandos tiene definidas variables de entorno necesarias para compilar desde una línea de comandos. Luego debemos ubicarnos en la carpeta win32. En dicha carpeta debemos ejecutar la utilería make de Microsoft:

nmake -f scintilla.mak

Si la compilación fue exitosa encontraremos en la carpeta bin de las fuentes de scintilla los siguientes archivos: SciLexer.dll, Scintilla.dll, SciLexer.lib, Scintilla.lib. Los cuatro archivos son necesarios para utilizar scintilla en cualquier proyecto.

Uno de las interfases de programación de scintilla es orientada a usuarios de la API de Windows. Una ventana del editor de crear de igual manera que se crearía una ventana de otro control del sistema operativo. La diferencia esta en el nombre de la clase que se utiliza para crear la ventana. En el cuadro 25 se muestra la creación de la ventana. El parámetro resaltado en color azul indica que se desea crear una ventana de Scintilla.

```
m_hWnd = ::CreateWindow(
     "Scintilla",
     "Source",
     WS_CHILD | WS_VSCROLL | WS_HSCROLL |
     WS_CLIPCHILDREN,
     0, 0,
     0, 0,
     hParent,
     NULL,
     AfxGetInstanceHandle(),
     0);
```

Cuadro 25. Creación de una Ventana Scintilla

Para darle a la librería una interfase mas adecuada para su uso con C++ se decidió tomar la manera como MFC encapsula los controles de Windows. MFC crea un objeto derivado de ventana e implementa un método de creación (Create). Este método es responsable de llamar a la función ::CreateWindow con los parámetros correctos e inicializar el miembro CWnd::m_hWnd para que tener un objeto ventana funcional. A este nuevo objeto que deriva de CWnd y contiene una ventana de Scintilla lo llamaremos CSintillaProxy. CSintillaProxy es nuestra interfase con Scintilla, de esta manera podemos exponer una interfase de programación al resto del sistema

diferente de la provista por scintilla. Además CSintillaProxy tienen el comportamiento de una ventana normal, ya que deriva de CWnd, es decir podemos mover una ventana del editor de la misma manera que moveríamos una ventana de una botón.

Control de Scintilla:

Existen dos maneras de controlar a Scintilla. La primera es enviando mensajes a scintilla y la segunda consiste en invocar directamente la función dentro del control.

Para enviar un mensaje a Scintilla se utiliza la función ::SendMessage de la API de Windows. El cuadro 26 muestra un ejemplo del envío de un mensaje a Scintilla.

```
SendMessage(hwndScintilla,SCI_CREATEDOCUMENT, 0, 0);
```

Cuadro 26. Envío de Mensajes a Scintilla

Para invocar directamente una función del control, primero debemos obtener un puntero a la tabla de funciones y luego usar este puntero para invocar las funciones. El cuadro 27 muestra un ejemplo de la invocación directa de una función de scintilla.

```
int (*fn)(void*,int,int,int);
void * ptr;
int canundo;

fn = (int (__cdecl *)(void *,int,int,int))SendMessage(
           hwndScintilla,SCI_GETDIRECTFUNCTION,0,0);
ptr =(void*)SendMessage(hwndScintilla,SCI_GETDIRECTPOINTER,0,0);

canundo = fn(ptr,SCI_CANUNDO,0,0);
```

Cuadro 27. Comunicación Directa con Scintilla

El segundo método, invocación directa de funciones, tiene un procesamiento más rápido ya que si se envía un mensaje hay que esperar a que este mensaje sea procesador por el control. Para el IDE se selección el segundo método ya que se obtiene un tiempo de respuesta mas rápido y la complejidad de la invocación se puede esconder dentro del objeto CSintillaProxy.

Notificaciones de Scintilla:

Cada vez que el Scintilla quiera notificar algún evento lo hace por medio del mensaje WM_NOTIFY de Windows. Este método es similar al usado por otros controles del sistema operativo. El mensaje WM_NOTIFY tiene entre sus parámetros una estructura genérica en la que se puede indicar el tipo de notificación. El cuadro 28 muestra como debe procesar este mensaje.

```
NMHDR *lpnmhdr;

[...]

case WM_NOTIFY:
        lpnmhdr = (LPNMHDR) lParam;

        if(lpnmhdr->hwndFrom==hwndScintilla)
        {
                switch(lpnmhdr->code)
                {
                        case SCN_CHARADDED:
                                /* Hey, Scintilla just told me that a new */
                                /* character was added to the Edit Control.*/
                                /* Now i do something cool with that char. */
                        break;
                }
        }
break;
```

Cuadro 28. Manejo de Notificaciones de Scintilla

Las funciones y eventos de Scintilla relevantes en la implementación del diseño se discuten en la sección Arquitectura del Sistema del presente capitulo.

ProfUI:

Se distribuye en forma de código fuente y es necesario compilar la librería antes de su uso. ProfUI viene con un archivo de espacio de trabajo de Visual Studio. Esto hace más fácil la compilación ya que se puede compilar desde Visual Studio directamente.

La compilación produce varias librerías de enlace dinámico (dll) de acuerdo a diferentes configuraciones. Las diferentes configuraciones producen varias versiones de las librerías. Se tienen versiones con información de depuración, para uso en aplicaciones multihilos y combinaciones de estas.

Las librerías contienen la implementación de las clases de ProfUI. Para usar la librería es necesario incluir en nuestro proyecto los archivos de cabecera que vienen en la distribución de ProfUI y enlazar con la librería de ProfUI. Las clases especificas que se usan

de esta librería se describen en la sección Arquitectura del Sistema del presente capitulo.

MinGW – GCC/GDB:

MinGW se distribuye de manera binaria y con código fuente. Se prefirió utilizar la distribución binaria ya que compilar MinGW es un proceso largo y complejo donde es fácil cometer errores. La distribución binaria incluye el código fuente de las librerías estándares de C para propósitos de depuración.

El cuadro 29 muestra la línea de comando necesaria para compilar una aplicación usando MinGW.

gcc.exe *archivos_fuentes* –o *nombre_ejecutable* [*parametros*]

archivos_fuentes – Ruta y nombre del archivo o los archivos de código fuente que se quieren compilar.

nombre_ejecutable – Ruta y nombre del archivo producido en la compilación.

parametros – Los argumentos necesarios para compilar. Los parámetros son opcionales, si no se especifica ninguno el compilado utiliza parámetros por defecto. Los parámetros indican si se quiere hacer una librería, una aplicación de consola, una aplicación multihilos, solo por mencionar algunos.

Cuadro 29. Uso de GCC

Parámetros del compilador empleados en el sistema:

Los siguientes parámetros son empleados directamente por el sistema, es decir tienen su representación en la interfase de usuario del sistema. Cabe recalar que el IDE permite el ingreso manual de parámetros. El objetivo del ingreso manual es dar la suficiente

flexibilidad al sistema cuando se desee realizar especial con el compilador.

-g3

Esta opción hace que el compilador agregue información de depuración compatible con GDB. El numero 3 indica que se quiere la mayor cantidad posible de información de depuración. Se no se agrega información de depuración ejecutable producido por el compilador no se puede realizar depuración a nivel de código fuente.

-O0 -O1 -O2 -O3 –Os

Estos parámetros controlan el nivel de optimización que el compilador realiza sobre el ejecutable. Cuando nos encontramos de depurando no es recomendable realizar ninguna optimización ya que el compilador reordena el código generado de manera que ya no existe correspondencia entre el ejecutable y su código fuente. Una vez terminada la depuración y cuando se desee distribuir la aplicación generado posiblemente se desee que el compilador optimice el ejecutable generado. Existen varios niveles de

optimización y se puede optimizar con una orientación hacia reducir el tamaño del ejecutable o mejorar su velocidad. –O0 desactiva las optimizaciones. –Os optimiza el tamaño del ejecutable. -O1 -O2 -O3 controlan el nivel de la optimización de la velocidad del ejecutable.

-Wall –Werror –w

Estos parámetros controlan el nivel de advertencias que lanza el compilador. –Wall activa todas las advertencias del compilador. El parámetro –Wall podría parecer excesivo sin embargo esto es deseable cuando se quiere que el compilador detecte cualquier posible fuente de un error en tiempo de ejecución. Es probable que este parámetro sea demasiado para un usuario novato, así que también se incluyo el parámetro –w. El parámetro –w activa las advertencias de los potenciales errores mas comunes de programación. Las advertencias exactas que estos parámetros activan se pueden encontrar en el manual de referencia de GCC. –Werror es un parámetro que pone al compilador en un modo muy estricto. Cuando esta definido –Werror todas las advertencias son tratadas como errores, lo que ocasiona que no se genere ningún

ejecutable si se encuentran alguna advertencia durante la compilación.

-fmessage-length=0

Este parámetro se emplea para que el compilador presente los errores en una solo línea. Normalmente la salida del compilador se presenta en líneas de 80 caracteres. Cuando estamos compilando desde una línea de comando esto es útil ya que nos permite leer completamente los errores reportados por el compilador. Sin embargo como en el sistema el compilador es un programa externo este no es el comportamiento que se desea ya que se quiere poder procesar la salida del compilador. Para procesar la salida del compilador se utiliza el carácter de fin de línea, que normalmente se coloca después de cada error reportado, para detectar los límites de los errores reportados por el compilador. Por otro lado, ya que el IDE es el encargado de presentar la salida del compilador, es posible presentar líneas de más de 80 caracteres. Esto hace que sea más fácil para el usuario la identificación de un error ya que cada línea contiene exactamente un error.

-Xlinker -Map -Xlinker archivo_map

Este parámetro le instruye al compilador que genere un archivo .map. Los archivos .map contienen la dirección de memoria de todas las variables y los métodos del ejecutable. Estos archivos son útiles para depurar aplicaciones. Cuando una aplicación falla suele presentar un cuadro de dialogo con la dirección de memoria del fallo. Usando el archivo .map y la dirección de memoria se puede tener una idea de cual fue el método donde la aplicación fallo. archivo_map es el nombre del archivo .map generado.

-L –I

Estos parámetros indican directorios adicionales donde el compilador puede buscar librerías y archivos de cabecera. –L indica directorios adicionales de librerías. –I indica directorios adicionales de archivos de cabecera.

-l

Indica librerías adicionales que se deben enlazar. Por ejemplo si se desea una aplicación que utilice OpenGL se pasa como argumento -lglut

-D

Permite definir constantes del preprocesador que afectan a todos los archivos de código fuente. Por ejemplo si se pasa al compilador –DDEPURACION es equivalente a escribir #define DEPURACION en todos los archivos de código.

GraphApp:

GraphApp se distribuye también en forma de código fuente. La diferencia de en el uso de este componente radica en que no se la emplea para generar el sistema, si no que se integra al compilador. Es decir la empleamos para que nuestro sistema genere ejecutables que utilicen GraphApp. Para que se pueda emplear esta librería con MinGW es necesario que haya sido compilada con MinGW. El problema consiste en que cada compilador crea las librerías de manera un poco diferente, es decir una librería generada con Borland C++ no es exactamente igual que una librería generada con Visual C++. Por ejemplo si se emplea una librería generada con Borland C++ en enlazador de Visual C++ no va ha entender el formato del archivo de la librería y la compilación fallara. En nuestro

caso compilamos GraphApp con MinGW para que los usuarios la utilicen con MinGW.

GraphApp se compila desde una línea de comandos. En el directorio src\rules se encuentran los archivos .mak necesarios para compilar la librería. Para compilarla se ejecuta:

```
nmake –f msvc.mak
```

Si la compilación fue exitosa se crear en la carpeta src de las fuentes el archivo libappw32.a. Este archivo es la librería de GraphAppp compilada para su uso con MinGW. La librería compilada, junto con sus fuentes se va ha incluir con el instalador del IDE para su uso por parte de los estudiantes.

5.2 Arquitectura del sistema

La presente sección describe como se implementan las diferentes funciones del sistema a partir del diseño. Como se mencionó en el capitulo 4 se seleccionó como el componente del entorno para el IDE a la librería de clases MFC y como el patrón principal de diseño a la arquitectura Documento/Vista. Alrededor de esta arquitectura se

implementarán los requerimientos del sistema para sus diferentes elementos que son: el entorno, los paneles flotantes, el panel de visualización de datos, los datos de la aplicación, los archivos de proyecto, la compilación, y las ayudas a la edición. A continuación se detallan los aspectos de implementación de cada uno de ellos:

Entorno:

Entre las clases principales de la arquitectura, se repartieron las responsabilidades, relacionadas al entorno, de la siguiente manera:

La clase aplicación: Esta clase se encarga de crear, modificar y guardar toda la información referente a proyectos y la creación de los nuevos archivos. Además, controla la compilación y la depuración de los programas desarrollados en el IDE.

- La clase documento: Contiene el texto escrito en el editor, inicializa el control del editor y administra el guardado de los archivos.

- La clase vista: Contiene una instancia del control de edición de Scintilla y actúa como un intermediario entre el sistema y el control de edición del Scintilla.

- La clase ventana principal: Esta clase maneja los eventos tanto del menú principal como de la barra de herramientas. Crea y contiene los paneles flotantes.

- Los paneles flotantes: Cada panel realiza una función diferente y contiene los controles, los agrupa y administra su posición.

La clase aplicación sobre escribe dos métodos llamados al inicio y al fin de la aplicación. Estos métodos son CLide2App::InitInstance y CLide2App::ExitInstance.

Durante el proceso de terminación de la aplicación se guarda el proyecto y los archivos que se encuentren abiertos y se destruyen los objetos creados en la inicialización.

Durante la inicialización de la aplicación se cargan la configuración del entorno, se verifican las extensiones de los archivos que según la configuración están asociados con el IDE, y se inicializan componentes del sistema tales como Scintilla y ProfUI.

Para manejar la configuración se emplea una tabla de dispersión. La tabla de dispersión se encuentra encapsulada dentro de la clase CProperties. Esta tabla guarda la tupla clave,valor en forma de cadenas de texto. El hecho de que se guarden como cadenas de texto facilita la serialización de la configuración. Para esto se implementaron las clases especializadas que serializan la tabla de dispersión en archivos .ini, en el registro de Windows o en formato XML. La clase CProperties provee una interfase de programación que convierte automáticamente la cadena de texto al tipo de dato que se desea. Esto se logra por medio de métodos sobrecargados. En el IDE se utilizo un patrón de singleton para tener acceso global al objeto CProperties, lo cual facilita su acceso desde cualquier parte de la aplicación.

Otra tarea del objeto aplicación es presentar diálogos de creación de archivos y proyectos al usuario. El usuario puede seleccionar que tipo específico de proyecto o archivo desea crear. El IDE creará un proyecto preconfigurado o un archivo con la extensión apropiada. Esto ahorra trabajo al usuario y ayuda a que el usuario pase más rápido a la tarea que le interesa. La figura 5.1 y la figura 5.2 muestran el dialogo de creación de un nuevo proyecto y de un archivo nuevo respectivamente.

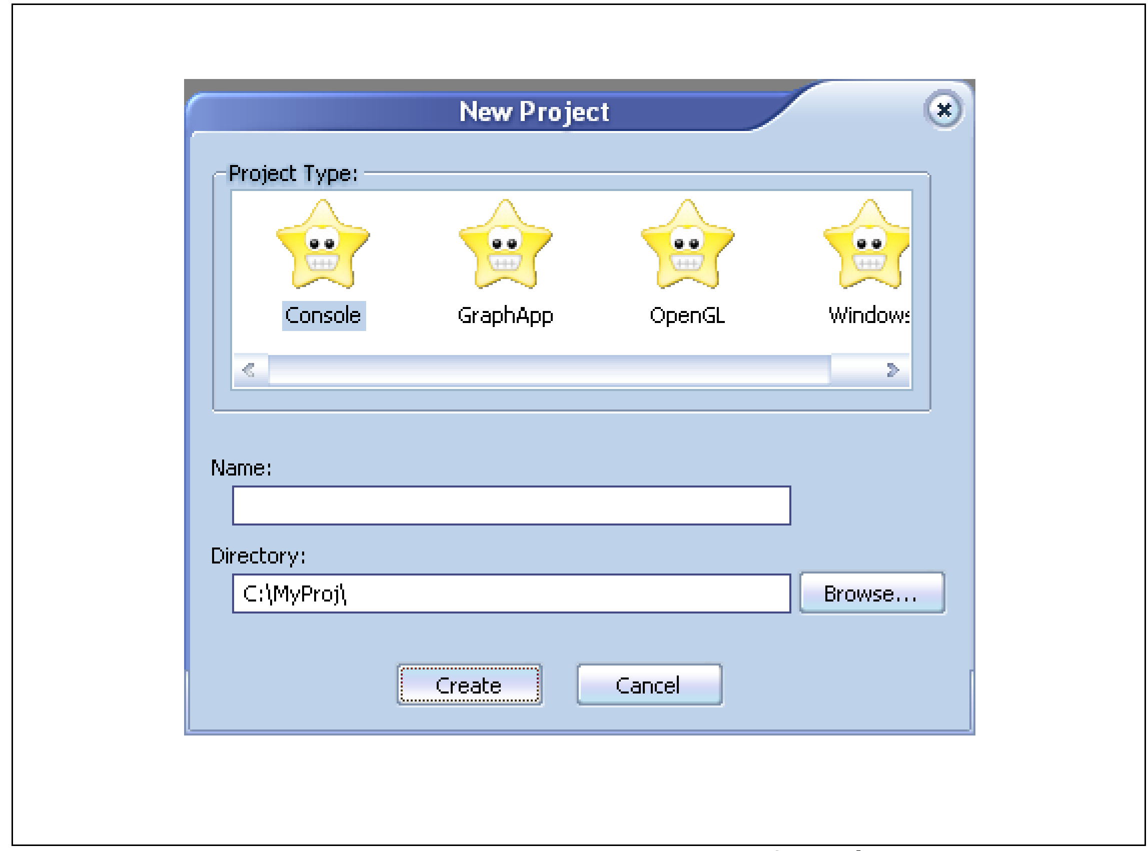


Figura 5.1. Dialogo de Creación de Nuevo Proyecto

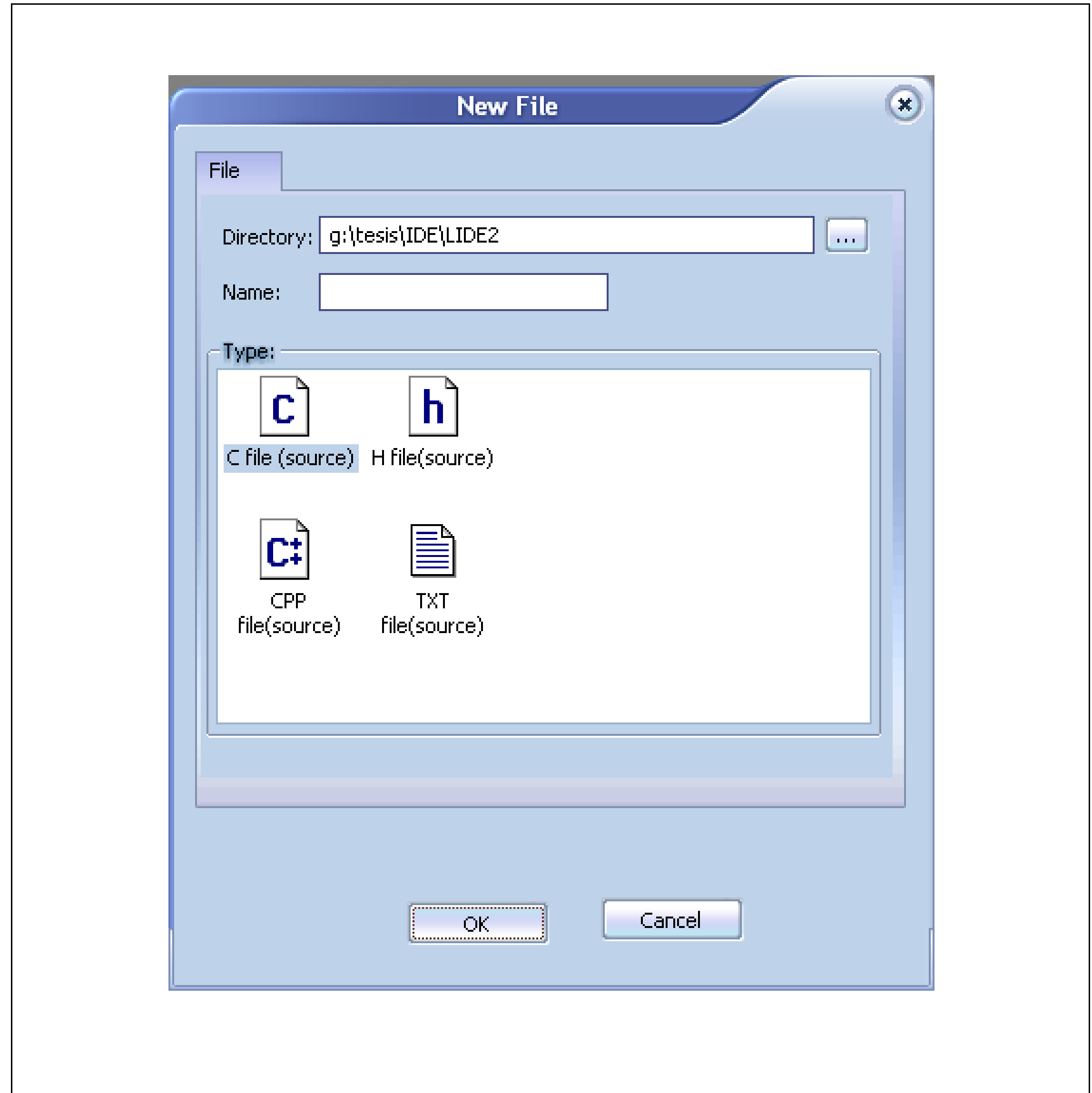


Figura 5.2. Dialogo de Creación de Nuevo Archivo

Para simplificar la activación y desactivación de los menús y las barras de herramientas se empleó el mismo identificador para ambos. Luego se escribieron manejadores ON_UPDATE_COMMAND_UI que recibían notificaciones de la barra

de herramientas y del menú. Usando este mecanismo se centraliza la lógica del estado de estos elementos de la interfase de usuario. Los menús y las barras de herramientas utilizan las clases CExtMenuControlBar y CExtToolControlBar respectivamente. Estas clases pertenecen a ProfUI. Son especializaciones de las clases de menú y barra de herramientas de MFC. Su uso es sencillo ya que conservan el comportamiento de los que tiene las clases de menú y barra de herramientas de MFC.

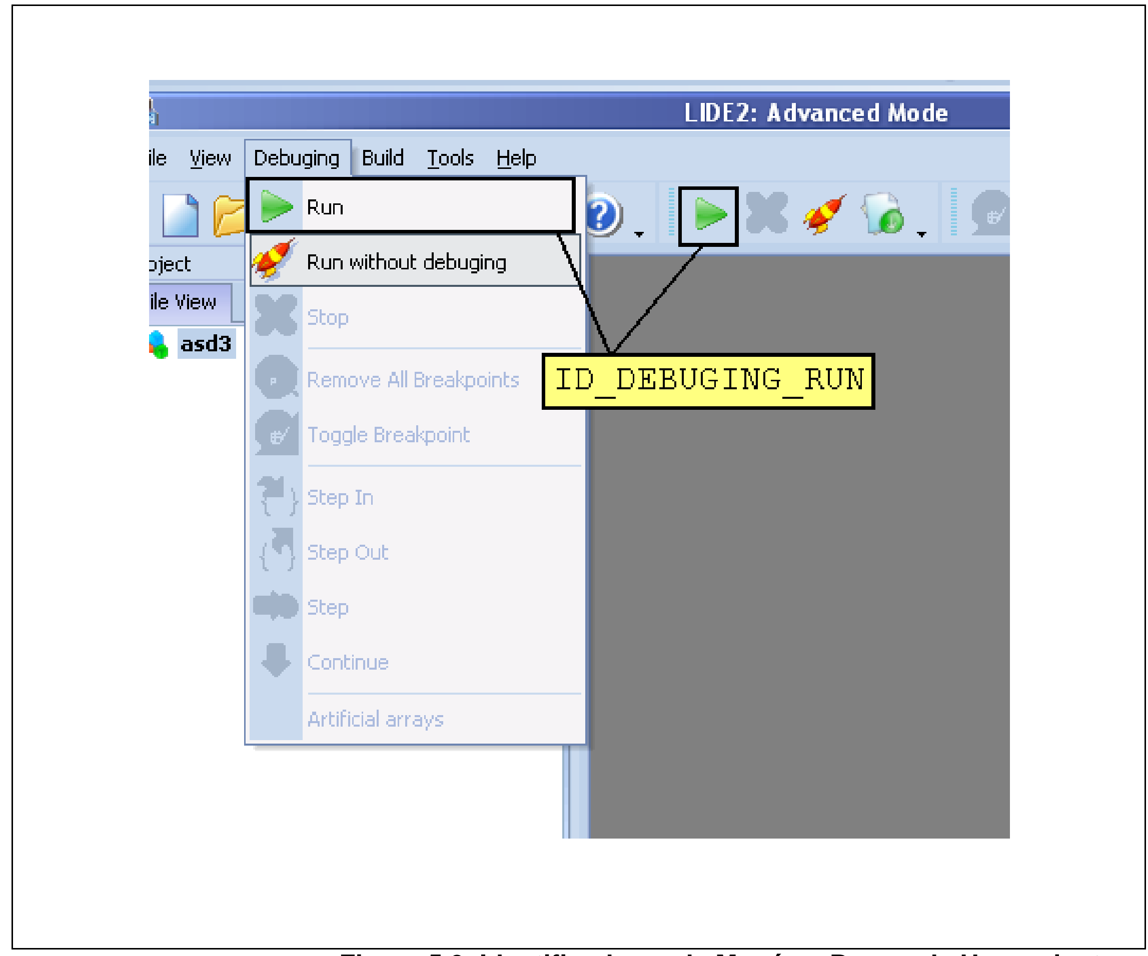


Figura 5.3. Identificadores de Menús y Barras de Herramientas

Paneles flotantes:

En la implementación de los paneles flotantes entran en juego 3 clases: CExtChildResizablePropertySheet, CExtControlBar, CExtResizablePropertyPage. Estas tres clases forman parte de ProfUI. La figura 5.4 muestra la relación de las tres clases con la arquitectura de la aplicación.

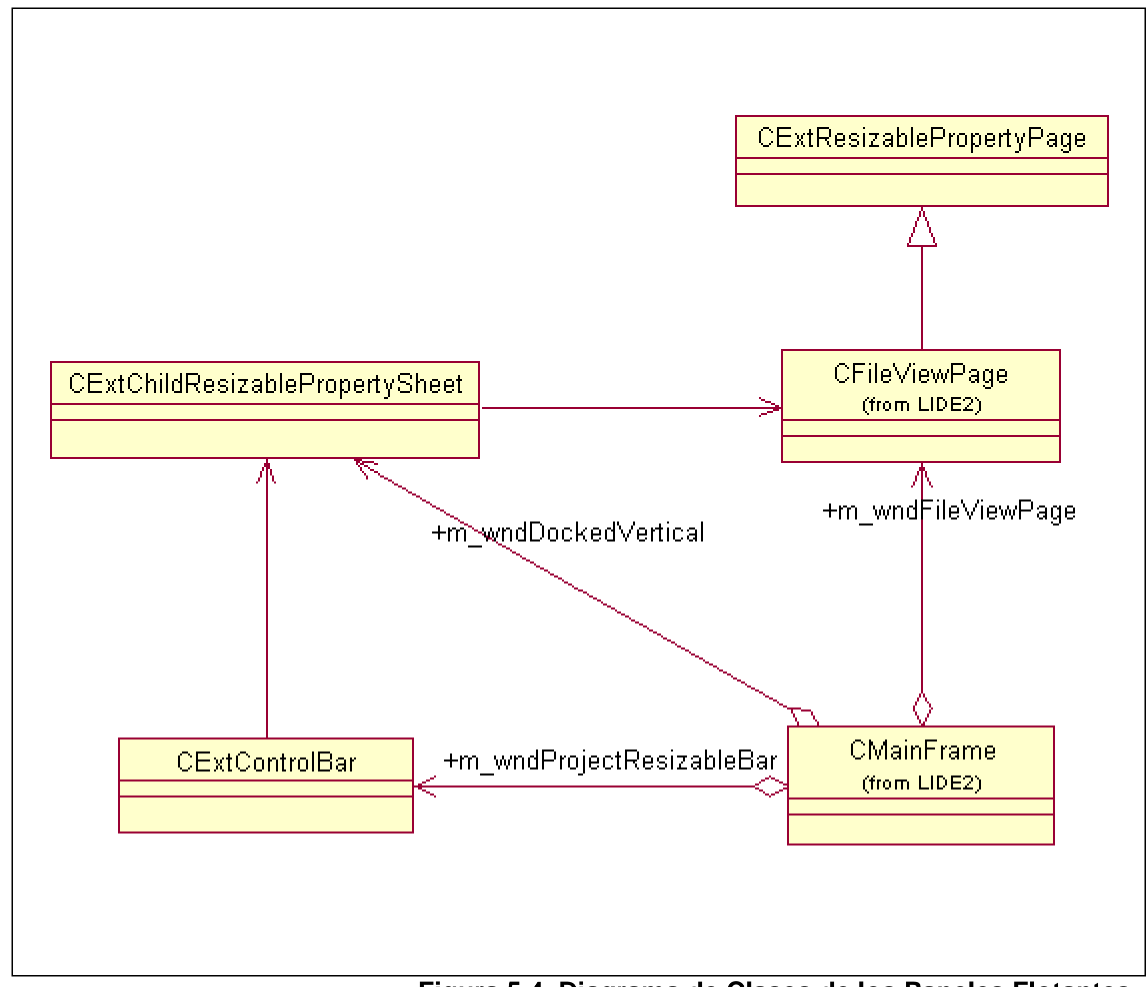


Figura 5.4. Diagrama de Clases de los Paneles Flotantes

CExtControlBar es el reemplazo de la clase CControlBar de MFC. CExtControlBar agrega comportamiento de anclaje, ocultación y serialización de la posición del panel.

CExtChildResizablePropertySheet es la clase que contiene a las clases derivadas de CExtResizablePropertyPage. La primera clase presenta los tabuladores para seleccionar el panel interior correspondiente. Y la segunda es el panel que contiene los controles. Es sobre las clases derivadas de CExtResizablePropertyPage que se implementan las diferentes funciones del sistema, funciones tales como la visualización de estructuras o la salida en crudo del compilador.

Para ocultar o presentar se utilizan los mismos mecanismos que en MFC, esto es la función CFrameWnd::ShowControlBar. La serialización de la posición de los paneles es administrada por ProfUI de manera automática.

Panel de Visualización de datos:

CDataVisPage es la clase derivada de CExtResizablePropertyPage. CDataVisPage contiene dos ventanas: un cuadro de ingreso manual y la ventana de visualización propiamente dicha. Además, es un intermediario entre la visualización y el resto del sistema.

Para el cuadro de ingreso manual se opto por utilizar un control de caja de combo que imita el comportamiento de la barra de direcciones del Internet Explorer. El combo recuerda lo que el usuario recientemente escribió y además tiene capacidades de autocompletar automáticamente mientras se ingresa texto.

Por otro lado, la clase utilizada para la ventana de visualización es la clase CDataVisCanvas. Esta clase contiene a todos los elementos que representan visualmente las estructuras e implementa la lógica de la visualización.

Los elementos que representan estructuras de datos son implementados por la clase CDataVisWidget, la cual deriva de la clase ventana (CWnd). Existe una relación padre hijo entre las instancias de esta clase (CDataVisWidget) y la clase

CDataVisCanvas, donde CDataVisWidget es hijo de CDataVisCanvas.

La razón para implementar CDataVisWidget como una clase derivada de CWnd yace en el hecho de que necesitamos mostrar información que posee estructura dentro de los elementos. Como se menciono en el diseño, para mostrar de manera tabular una estructura de datos se optó por utilizar un control hibrido de árbol/lista. De acuerdo a la arquitectura de Windows un control no puede, sin tener que implementar una gran cantidad de código, estar contenido en algo que no sea una ventana o una clase derivada de ventana.

Otra razón para implementar CDataVisWidget como una ventana, es que hace posible agregar menús contextuales y tips emergentes, los cuales solo pueden agregarse a ventanas o a clases derivadas de ventana.

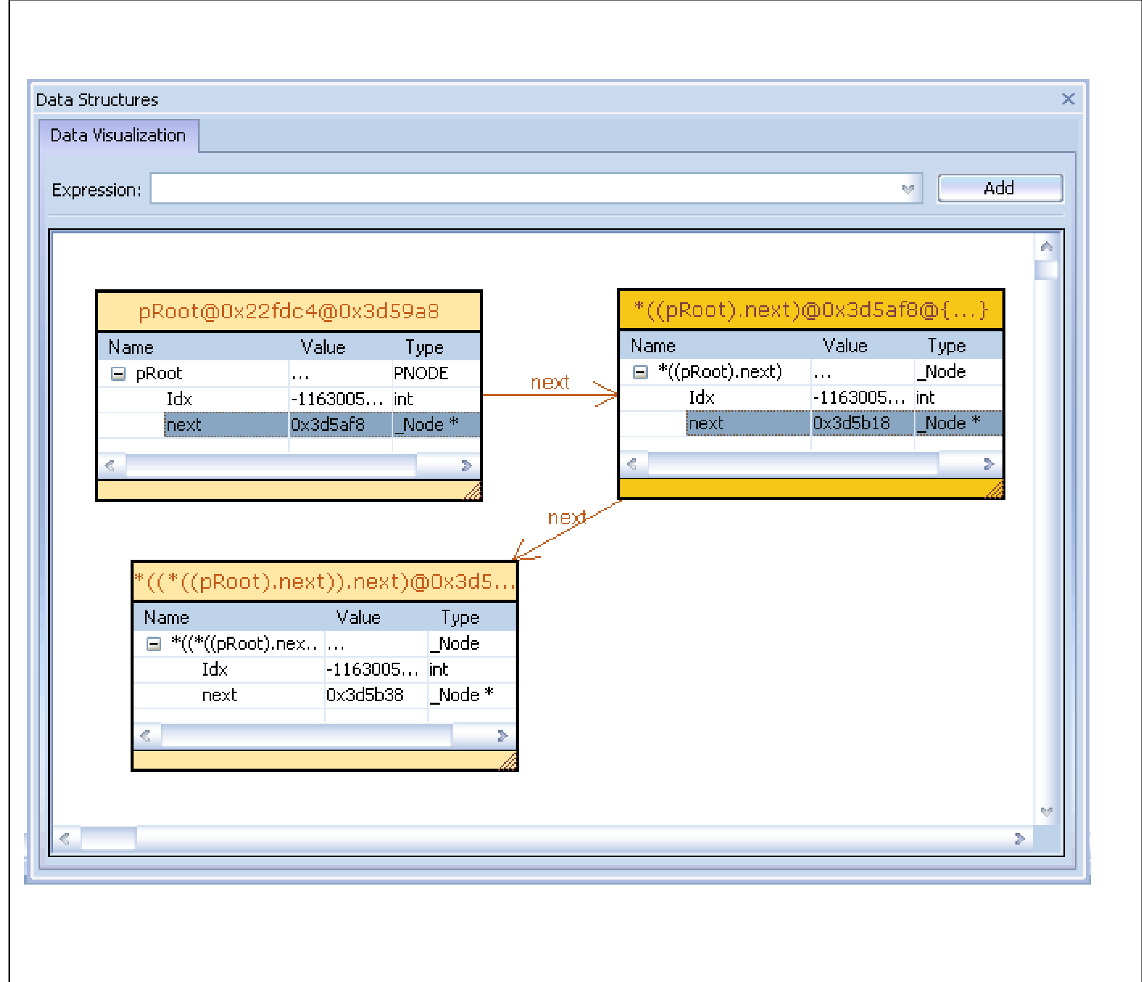


Figura 5.5. Elementos de Visualización y Enlaces

Cada instancia de CDataVisWidget se comporta igual que una ventana. Es posible moverla desde la barra de titulo y cambiar su tamaño arrastrando la esquina inferior derecha. Además se implemento la selección múltiple de los CDataVisWidget para que puedan ser movidos en grupos.

Con el objetivo de reducir la carga visual, las instancias de CDataVisWidget pueden compactarse hasta quedar visible solamente la barra de titulo. Además en caso de que se quieran dejar de ver pero no se desee eliminarlos de la visualización, las instancias de CDataVisWidget pueden ser ocultadas. Esto se implementa usando los mecanismos usuales que se usan sobre las ventanas, tales como ::SetWindowPos() o ::ShowWindow().

Otra técnica empleada para reducir la sobrecarga de información es la configuración del titulo CDataVisWidget. El titulo se puede configurar para que presente el nombre, el tamaño y dirección de memoria, el tipo de dato y valor de la variable o cualquier combinación de esta información. Sin embargo la información completa de la variable siempre estará presente cuando se presente el tip emergente de la variable.

Los enlaces que representan las relaciones creadas por los punteros, son elementos dibujados directamente sobre CDataVisCanvas. Los enlaces están implementados en la clase CWidgetLink y son almacenados en la clase CWindgetLinkMgr. CWindgetLinkMgr es la clase responsable de agrupar todos los enlaces presentes, además de permitir su rápida recuperación y

dibujar los enlaces empleando los datos de CWidgetLink. CWidgetLink tiene un par de punteros de tipo CDataVisWidget para poder navegar hacia el origen y el destino del enlace.

Sin embargo, como la clase CDataVisWidget no tiene referencias a las instancias de CWidgetLink, es necesario proveer un mecanismo para que cuando por ejemplo se cambie la posición de un CDataVisWidget, se pueda hacer el reajuste en las posiciones de los enlaces que apuntan a él. Para esto, es necesario proveer una clase adicional. Esta clase es CNamedPoint la cual deriva de CPoint y contiene un identificador. El identificador de CNamedPoint es igual al identificador del enlace al que corresponde. Los objetos CNamedPoint guardan el punto inicial o final de un enlace. De esta forma, CDataVisWidget mantiene una lista de punteros CNamedPoint, a través de la cual, tiene acceso a las coordenadas de todos los enlaces que llegan o parten de el. De esta manera si CDataVisWidget cambia de posición, únicamente tiene que actualizar su lista de puntos y pedir que se redibuje la ventana, y automáticamente los links recalcularán su posición y se dibujarán en el lugar correcto.

El identificador de los puntos es también usado en casos de ocultación o eliminación de un CDataVisWidget. Cuando se elimina un CDataVisWidget se recorre la lista de puntos para obtener todos los enlaces que llegan o salen de él. En este caso, el identificador se usa para extraer un objeto CWidgetLink de CWindgetLinkMgr.

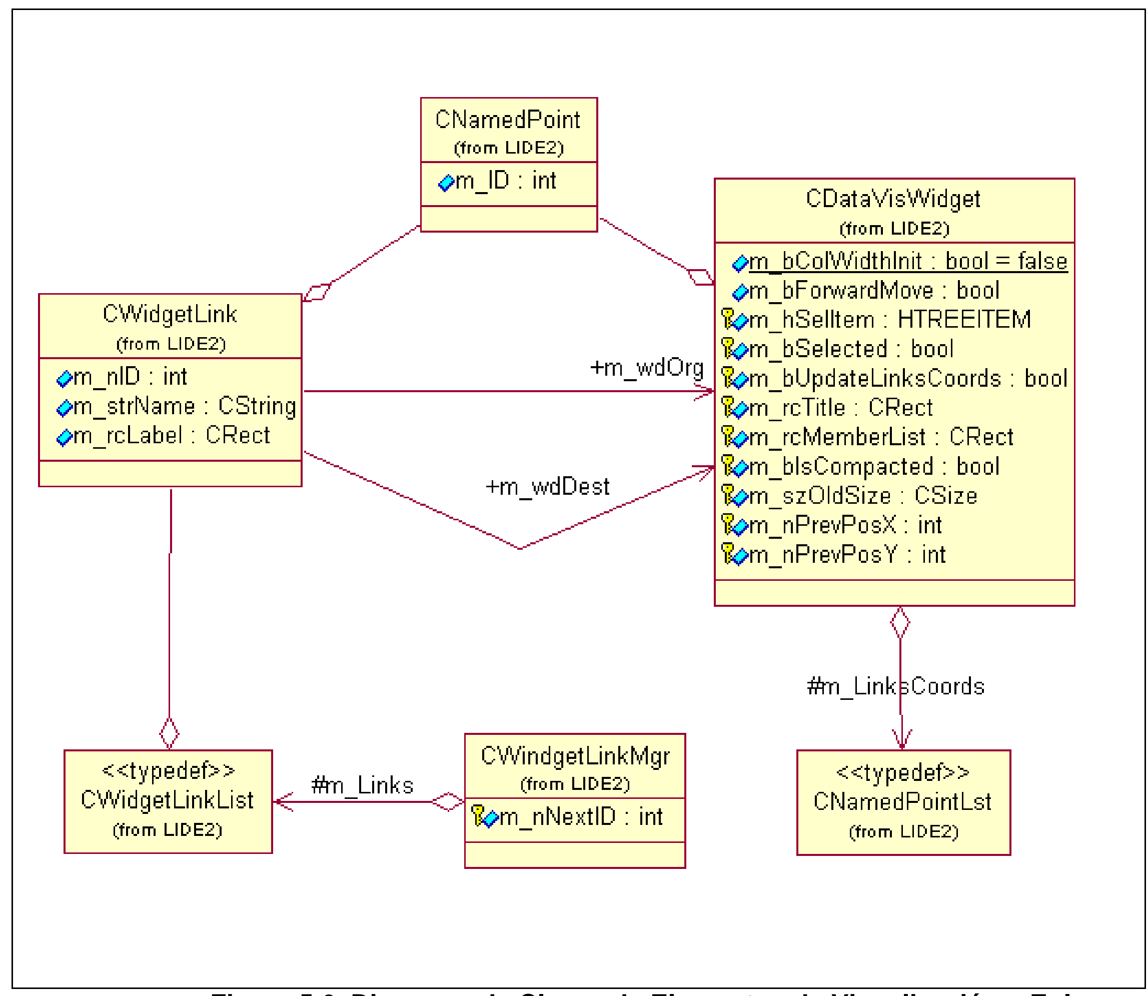


Figura 5.6. Diagrama de Clases de Elementos de Visualización y Enlaces

Un problema que se puede presentar cuando se están explorando las estructuras es la acumulación de objetos en la ventana. Un mecanismo, explicado en párrafos anteriores, es la compactación y la ocultación de los objetos CDataVisWidget. Otra estrategia es implementar la capacidad de scroll en la ventana de visualización. Esto se logra derivando CDataVisCanvas de la clase CExtScrollWnd de ProfUI. La ventaja de la implementación de CExtScrollWnd es el manejo del redibujo. La CExtScrollWnd trata de reducir el redibujo al mínimo al utilizar técnicas de doble buffer y redibujar solo una porción de la ventana.

Un inconveniente en la implementación actual de CDataVisCanvas es que el área de visualización no se redimensiona de manera automática. Es posible configurar cuan grande queremos que sea, aunque una vez que existen ítems dentro del área de visualización no es posible cambiar su tamaño.

Los datos de la aplicación y los modos del IDE:

Los diferentes tipos de proyectos preconfigurados que puede crear el IDE se determinan en tiempo de desarrollo. Los tipos de proyectos se definen directamente en el código fuente del IDE. Cada especialización de la clase CProject realiza inicializaciones específicas de acuerdo al tipo de proyecto.

El modo novato se implementa dentro del IDE como una especialización de la clase proyecto, para ser mas precisos se implementa en la clase CNovProject. Un proyecto de tipo novato tiene preconfiguradas las librerías de GraphApp y crea el ejecutable en la misma carpeta donde se encuentra el archivo de código fuente. Otra diferencia entre un proyecto CNovProject y los otros tipos de proyectos es el número de archivos de código fuente que contienen. Los proyectos de tipo CNovProject tienen únicamente un archivo de código fuente.

La clase CWorkspace mencionada en el capítulo de diseño puede contener una o más clases CProject. Para la actual implementación del IDE no se explota la capacidad de contener más de un proyecto de CWorkspace, únicamente se utiliza un objeto CProject a la vez.

Se tomó esta decisión ya que se cree que un espacio de trabajo multi-proyecto podría resultar muy complejo para los estudiantes. Sin embargo se permite una futura extensión en caso de que lo amerite.

La configuración de las opciones del proyecto es responsabilidad de un objeto de tipo CProperties. Este objeto es miembro de CProject. Y es el mismo tipo de objeto que se emplea para guardar la configuración del IDE.

La serialización de los proyectos se lleva acabo con la colaboración de todos los objetos relacionados (CWorkspace, CProject, CFileGroup y CLideFile) coordinados por CWorkspace. Las clases CProject, CFileGroup y CLideFile (mencionadas en el diseño) pueden serializarse en forma de cadenas de texto. CWorkspace es responsable de la apertura y escritura de los archivos de configuración.

Con el objetivo de tener un formato bien estructurado de archivo de proyecto se decidió utilizar XML para el archivo de configuración. Además nos permite agregar elementos a la configuración sin tener que modificar el código antiguo. El cuadro 30 muestra el contenido de un archivo de proyecto.

```
- <WorkSpace>
  - <Project name="asd3" type="1">
      <FileGroup name="Header Files" filter=".h;.hpp" />
    - <FileGroup name="Source Files" filter=".c;.cpp">
        <File path=".\main.cpp" />
      </FileGroup>
      <option Name="Project.Linker.ExtraOps" Value="-mconsole" />
      <option Name="Project.Linker.LibsDirs" Value="" />
      <option Name="Project.Compiler.PreprocessorDefs" Value="" />
      <option Name="Project.OutputDir" Value="C:\MyProj\asd\asd3\" />
      <option Name="Project.WarningAsErrors" Value="0" />
      <option Name="Project.Compiler.HeadersDirs" Value="" />
      <option Name="Project.WarningLevel" Value="1" />
      <option Name="Project.Linker.GenerateMapFile" Value="1" />
      <option Name="Project.Compiler.ExtraOps" Value="" />
      <option Name="Project.OptimizeLevel" Value="0" />
      <option Name="Project.OutputName" Value="asd3.exe" />
      <option Name="Project.IntDir" Value="C:\MyProj\asd\asd3\objs\" />
      <option Name="Project.Compiler.CompilationType" Value="1" />
      <option Name="Project.Linker.Libs" Value="" />
      <option Name="Project.Linker.IgnoreStdLibs" Value="0" />
      <option Name="Project.Compiler.AddDebugInfo" Value="1" />
    </Project>
  </WorkSpace>
```

Cuadro 30. Archivo de Proyecto

Para leer los archivos de proyecto se utilizo Expat. Expat es un parser SAX no validador de XML. Expat es mas simple de utilizar que los parsers de Microsoft o de del proyecto Apache y satisface nuestras necesidades de manera bastante completa. Se utiliza SAX ya que solamente necesitamos leer secuencialmente el archivo de

configuración y no es necesario navegar por el modelo DOM del documento XML. Un beneficio del uso de SAX es el poco consumo de memoria en comparación con DOM, lo cual es parte de nuestro requerimientos ya que no se quiere que el IDE requiera de una maquina poderosa para ejecutarse. Al usar Expat nos libramos de la responsabilidad de la lectura del archivo y el procesamiento (parsing) del contenido del mismo.

Para leer un archivo en XML con Expat debemos registrar una serie de funciones (callbacks) en los eventos que lanza el parser. En el IDE nos interesa saber cuando se encuentra el comienzo y el fin de un elemento. Cuando el parser detecta el comienzo de un elemento se leen los atributos del elemento y se construye el objeto. Este es el mecanismo empleado para cargar los proyectos desde el archivo de proyecto.

Si se observa el cuadro 30 se puede ver al final un grupo de ítems de tipo “option”. Eso es la serialización del objeto CProperties que contiene CProject. Como ya se mencionó, existe varias especializaciones del objeto CProperties, cada una con un diferente método de serialización. En el caso de la serialización hacia XML se usa la clase CPropertiesXML.

Cargado de un Archivo de Proyecto:

El cargado de un proyecto es uno de los pasos de inicialización del IDE. Por medio del cargado de proyecto se lee el archivo de configuración y permite preparar el IDE para la edición, compilación y depuración de código. En el diagrama de la figura 4.19 se muestra la secuencia de pasos requeridos para leer y terminar de inicializar el IDE.

Los pasos en orden cronológico seguidos para cargar un archivo de proyecto son:

1. El objeto aplicación, en respuesta de un mensaje enviado por el sistema operativo, llama a OpenProject(). Esta función realiza validaciones de los argumentos.
2. Se llama a la función Load del objeto CWorkSpace.
3. Se inicializa el objeto CWorkSpace en caso de que no este inicializado.
4. Se hacen varias llamadas a funciones para inicializar el parser de XML e iniciar el proceso de lectura del archivo.

5. Se invoca a AddProject cuando el parser encuentra un elemento específico en el archivo de proyecto.
6. Se crear el objeto derivado de CProject de tipo correcto.
7. Se invoca a la función AddFileGroup y esta crea un objeto de tipo CFileGroup y lo agrega al objeto proyecto recientemente creado.
8. Se invoca a AddFile. AddFile crea un objeto de tipo CLideFile y lo agrega al CFileGroup recientemente creado.
9. El objeto CLideFile aprovecha para inicializar un valor de dispersión (hash) que identifica de manera única al archivo dentro del IDE.
10. Cuando se encuentran las opciones del proyecto dentro del archivo de configuración se invoca a SetProjectOption. Esta función inicializa las opciones del último objeto proyecto creado.
11. La función GetProperties, obtiene el objeto CProperties que contiene el objeto proyecto y este último es modificado a través de PutProperties.

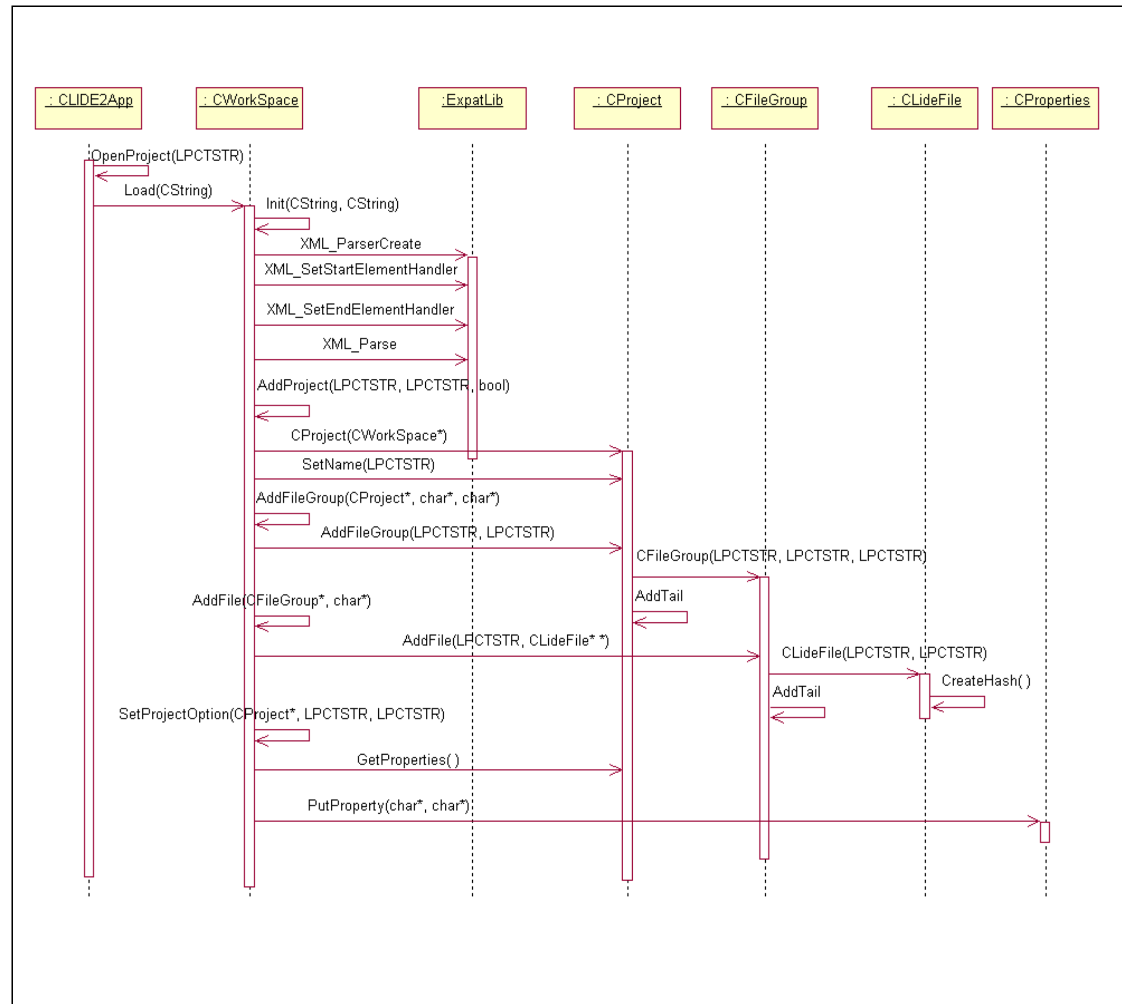


Figura 5.7. Diagrama de secuencia del Cargado de un Proyecto

Compilación:

El primer paso para implementar la compilación es la generación de la línea de comando del compilador. Para generar la línea de comandos del compilador necesitamos: la ruta del compilador, la ruta de los archivos de código fuente que se piensan compilar y las opciones del compilador.

La ruta del compilador se obtiene de la configuración general del IDE. La ruta de los archivos de código fuente y las opciones del compilador se obtienen del objeto proyecto contenido dentro del objeto CWorkspace.

Con el objetivo de poder agregar otros compiladores a futuro se definió una clase abstracta base, CCompiler, que define la interfase de programación de la compilación. Los detalles específicos de cada compilador son manejados por las clases derivadas de CCompiler. Para esta versión del IDE se emplea CGCCCompiler, para compilar con GCC. El objeto CGCCCompiler traduce las opciones configuradas por el usuario en argumentos para GCC.

Una vez que se tiene la información necesaria, la generación de la línea de comando no es mas que el formateo de una cadena de texto con una función similar a sprintf.

El objeto CGCCCompiler crea el proceso del compilador, pasando como argumento la línea de comandos. Para crear un proceso en Windows se utiliza la función ::CreateProcess. El cuadro 31 presenta un ejemplo de la invocación de la función ::CreateProcess.

```
PROCESS_INFORMATION m_piProcInfo;
STARTUPINFO                       m_siStartInfo;

// Parametros de la ejecucion del proceso.
m_siStartInfo.dwFlags = STARTF_USESHOWWINDOW;
m_siStartInfo.wShowWindow = SW_HIDE;
m_siStartInfo.cb = sizeof(m_siStartInfo);

TCHAR shellCmd[4096];
_tcscpy(shellCmd, m_szExecutable);
TRACE(_T("Trying: %s\r\n"), shellCmd);

// Crear el proceso hijo.
BOOL bRet = ::CreateProcess(
        NULL,
        shellCmd,          // nombre de la aplicacion
        NULL,    // atributos de seguridad del proceso
        NULL,    // atributos de seguridad del hilo principal
        TRUE,              // heredar handles?
        CREATE_NEW_CONSOLE, // banderas de la creacion
        NULL,              // ambiente del proceso
        NULL,              // directorio actual del proceso
        &m_siStartInfo,  // STARTUPINFO
        &m_piProcInfo);  // PROCESS_INFORMATION
```

Cuadro 31. Creación de un Proceso

El segundo argumento de la función ::CreateProcess es la línea de comando que deseamos ejecutar. Windows no puede manejar líneas

de comando de más de 4096 caracteres. Esto limita el número de archivos que se puede compilar simultáneamente. Y este hecho afecta el número de archivos que un proyecto puede tener. El número máximo de archivos que puede tener un proyecto no es fácil de determinar, ya que este número depende de la longitud de las rutas de los archivos. La limitación del número de archivos no es problema critico por el momento, ya que no se espera que los proyectos creados por los estudiantes consistan de un gran cantidad de archivos.

Durante la creación del proceso se realiza la redirección de la entrada y salida. La redirección de la entrada y salida se implementa por medio de pipes en Windows. Un pipe es una canal de comunicación full-duplex orientado a flujo de caracteres. Desde el punto de vista del programador un pipe esta representado por dos manijas (handles). Existen dos handles ya que se tiene separada la escritura y la lectura de un pipe. Un handle se emplea para solo leer del pipe y el otro solo se emplea para escribir en el pipe. Para leer y escribir sobre un pipe se usan las mismas funciones que se emplean para los archivos, (::ReadFile y ::WriteFile).

El algoritmo de redirección de la entrada y salida en Windows es le siguiente:

1. Guardar los handles de la entrada y salida estándar del padre
2. Crear el pipe para la salida del hijo (el proceso hijo) usando ::CreatePipe, hacer los handles heredables
3. Cambiar la salida estándar del padre por el handle de escritura del pipe de la salida del hijo.
4. Duplicar el handle de lectura del padre para eliminar la capacidad de heredarlo.
5. Cerrar el handle heredable de lectura del pipe de la salida del hijo
6. Crear el pipe para la entrada del hijo (el proceso hijo) usando ::CreatePipe, hacer los handles heredables
7. Cambiar la entrada estándar del padre por el handle de lectura del pipe de la salida del hijo.
8. Duplicar el handle de escritura del padre para eliminar la capacidad de heredarlo.
9. Cerrar el handle heredable de escritura del pipe de la salida del hijo
10. Invocar a ::CreateProcess pasando en la estructura STARTUPINFO los handles de la entrada y salida de la siguiente manera:

Entrada estándar ⇔ handle de lectura de la entrada

Salida estándar ⇔ handle de escritura de la salida

Error estándar ⇔ handle de escritura de la salida

El cuadro 32 muestra la implementación del algoritmo de redirección en Windows.

```
HANDLE hChildStdoutRdTmp, hChildStdinWrTmp;
SECURITY_ATTRIBUTES saAttr;
BOOL bSuccess;

saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
saAttr.bInheritHandle = TRUE;
saAttr.lpSecurityDescriptor = NULL;


// Redirección de la salida

m_hSaveStdout = GetStdHandle(STD_OUTPUT_HANDLE);

CreatePipe(&hChildStdoutRdTmp, &m_hChildStdoutWr, &saAttr, 0);

SetStdHandle(STD_OUTPUT_HANDLE, m_hChildStdoutWr);

DuplicateHandle(
      GetCurrentProcess(),hChildStdoutRdTmp,
      GetCurrentProcess(), &m_hChildStdoutRd,
      0, FALSE, DUPLICATE_SAME_ACCESS
      );

CloseHandle(hChildStdoutRdTmp);

// Redirección de la entrada

m_hSaveStdin = GetStdHandle(STD_INPUT_HANDLE);
```

```
CreatePipe(&m_hChildStdinRd, &hChildStdinWrTmp, &saAttr, 0);

SetStdHandle(STD_INPUT_HANDLE, m_hChildStdinRd);

DuplicateHandle(
      GetCurrentProcess(), hChildStdinWrTmp,
      GetCurrentProcess(), &m_hChildStdinWr,
      0, FALSE, DUPLICATE_SAME_ACCESS
      );

CloseHandle(hChildStdinWrTmp);


PROCESS_INFORMATION m_piProcInfo;
STARTUPINFO             m_siStartInfo;

// Parametros de la ejecucion del proceso.
m_siStartInfo.dwFlags = STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW;
m_siStartInfo.wShowWindow = SW_HIDE;
m_siStartInfo.hStdInput = m_hChildStdinRd;
m_siStartInfo.hStdOutput = m_hChildStdoutWr;
m_siStartInfo.hStdError = m_hChildStdoutWr;
m_siStartInfo.cb = sizeof(m_siStartInfo);

// ...
// Se continua con la creación del proceso normalmente
```

Cuadro 32. Redirección de la E/S en Windows

Una vez redirigida la entrada y la salida del proceso, si el padre desea leer de la salida del hijo se llama a ::ReadFile sobre el handle de lectura de la salida estándar. Si por el contrario, el padre desea escribir en la entrada del hijo, deberá llamar a WriteFile sobre el handle de escritura de la entrada estándar.

Cuando se tiene la salida del compilador, se ejecuta un simple algoritmo para tratar de encontrar dentro de la salida las líneas que contienen advertencias y errores. Debido a que pasamos -fmessage-length=0 al compilador, obteneos cada mensaje de información del compilador en una sola línea. Este es el primer criterio para descomponer la salida en líneas. Dentro los primeros caracteres de cada línea se buscan las palabras ‘error’ o ‘warning’, si se encuentran dichas palabras se pasa a la siguiente etapa. Ya que se desea que se pueda saltar directamente a la línea que contiene el error, debemos extraer el nombre del archivo y el número de línea. Los nombres de los archivos y los números de línea están separados por el carácter ‘:’, así que para obtener la información necesaria basta con separar en tokens la cadena.

Ayudas a la edición:

Como se menciono en la sección 5.1, Scintilla necesita ser inicializado para operar correctamente. Por inicialización nos referimos a la configuración de los caracteres permitidos por el control, la lista de palabras claves del lenguaje, los colores de los elementos del lenguaje, entre otros.

Los elementos que se configuraron en Scintilla para su uso en el IDE son:

- Lexer a utilizarse
- Caracteres permitidos
- Palabras claves
- Fuente del texto escrito.
- Colores de los elementos del lenguaje
- Palabras claves o caracteres que determinan el fin de un bloque
- Formato de los marcadores

Para Scintilla un lexer es un componente que indica si determinada palabra o cadena de texto tiene un significado especial en el lenguaje, tal como un comentario o una cadena de texto. Scintilla permite el desarrollo de lexers personalizados en caso de que se necesite. En nuestro caso no es necesario desarrollar un lexer ya que Scintilla provee un lexer para C y para C++. Nuestra responsabilidad es indicar el color de los elementos del lenguaje. Con esto se logra que el editor coloree elementos del código como líneas con instrucción de #define, comentarios y cadenas de texto.

Para colorear las palabras claves del lenguaje necesitamos indicarle a Scintilla por medio del mensaje SCI_SETKEYWORDS cuales son las palabras claves.

Los marcadores son elementos gráficos que se ubican en el margen del texto. Es posible definir hasta seis marcadores. El IDE emplea los marcadores para indicar: puntos de ruptura, marcadores de posiciones, líneas con errores y el punto de ejecución actual.

Los marcadores de posiciones (bookmarks) se emplean para regresar rápidamente a alguna parte del código que nos interese.

La tabla 2 presenta los elementos gráficos que se emplearon en el IDE.

Imagen	Uso	Constante de Scintilla
	Punto de Ruptura	SC_MARK_CIRCLE
	Marcador de posición	SC_MARK_ROUNDRECT
	Línea de error	SC_MARK_ARROW
	Punto de ejecución	SC_MARK_SHORTARROW

Tabla 2. Iconos de los Marcadores

Además de los elementos gráficos, otro aspecto importante del Scintilla que nos permite implementar las funciones necesarias para la edición, son las notificaciones enviadas por este. Estas son:

SCN_MODIFIED – Enviado cuando el texto o el estilo del documento cambian.
SC_MOD_CHANGEFOLD – Es una especialización de SCN_MODIFIED. Enviado cuando ha cambiado el doblado de código.
SCN_UPDATEUI – Enviado cuando cambia el estilo, la selección o el texto del documento. Esta es una oportunidad para actualizar elementos de interfase de usuario que dependen del documento.
SCN_CHARADDED – Enviado cuando el usuario ingresa un carácter.

A continuación se detalla el uso que el IDE le da a cada una de a las notificaciones descritas en el párrafo anterior.

La notificación SCN_MODIFIED se emplea para dar retroalimentación al usuario, de manera que el usuario este conciente que el documento ha sido modificado y aun no ha sido guardado. En la interfase de usuario un documento modificado se

indica por la adición de un asterisco al final del nombre del archivo. La figura 5.8 presenta la implementación de este elemento de la interfase de usuario.

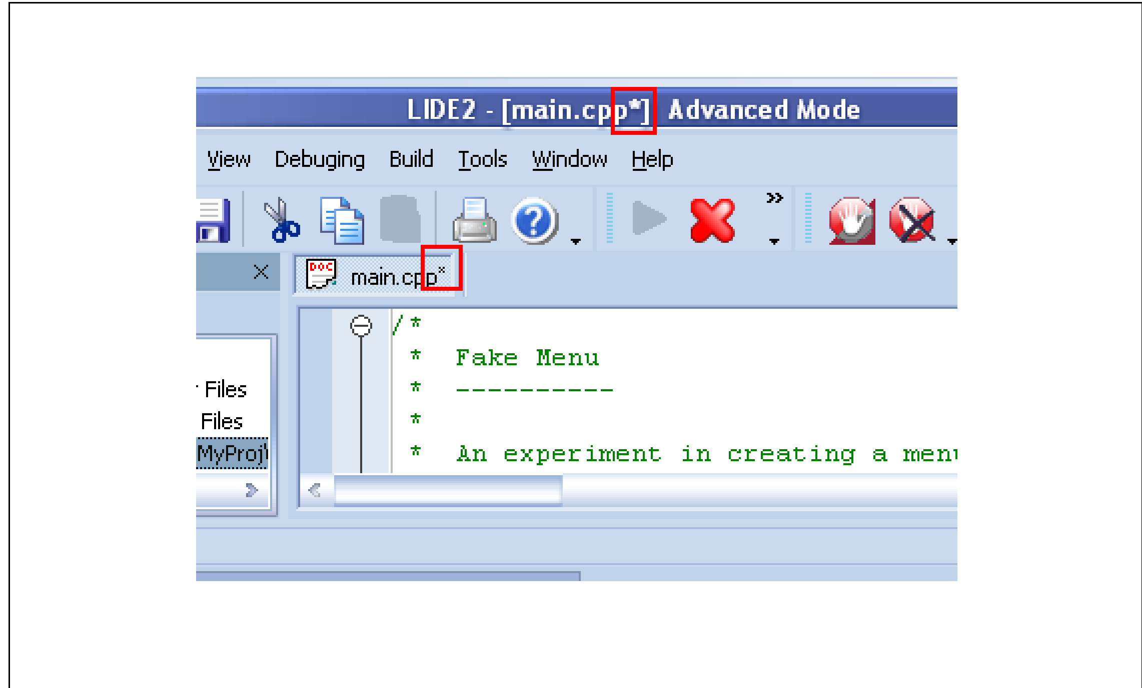


Figura 5.8. Identificación de un Archivo Modificado

Scintilla puede determinar que partes del código fuente pueden ser dobladas y puede colocar iconos en el margen que indican estas posiciones. Sin embargo es responsabilidad del programador implementar el doblado del código fuente. Detalles como estos le

dan mucha flexibilidad a Scintilla, ya que el programador puede controlar de que manera, oculta el código.

Para implementar el doblado del código fuente se deben detectar los cambios en el estilo del doblado. Concretamente nos referimos a detectar el cambio de los iconos que se encuentran en el margen del documento. Para detectar el cambio en el estilo de doblado empleamos la notificación SC_MOD_CHANGEFOLD.

Cuando se recibe la notificación SC_MOD_CHANGEFOLD se pregunta cual es el nivel de doblado de la línea actual y luego se llama sucesivamente por cada nivel al mensaje SCI_SETFOLDEXPANDED para expandir o compactar cada nivel.

Cuando se recibe la notificación de SCN_UPDATEUI del Scintilla, se aprovecha para actualizar el numero de línea y columna donde esta el cursor. Esta información se presenta en la barra de estado del IDE. Otra acción ejecutada cuando se recibe esta notificación es la verificación del emparejamiento de los paréntesis. La verificación del emparejamiento de los paréntesis es una ayuda a la edición. Con esta ayuda se pretende mostrar, de una manera no obstrusiva, al

estudiante que existe un paréntesis sin su pareja. Las figuras 5.9 y 5.10 muestran, respectivamente, la manera en que se presenta al usuario un par de paréntesis con pareja y un paréntesis sin pareja.

Paréntesis emparejado:

```
printf("hola mundo\n");
int a = ( ( (int)cos( (PI*0.35)/2.0 ) )*15 +16;
```

Figura 5.9. Paréntesis con pareja

Paréntesis sin pareja:

```
printf("hola mundo\n");
int a = ( ( (int)cos( (PI*0.35/2.0 ) )*15 +16;
```

Figura 5.10. Paréntesis sin Pareja

Para implementar la validación de paréntesis, se espera a que el cursor se encuentre junto a un paréntesis o una llave. Cuando se encuentra junto a un paréntesis o llave se empieza a explorar, hacia a delante o hacia atrás se según sea un paréntesis de apertura o no, y se cuentan paréntesis hasta que se encuentra la pareja. Si se encuentra la pareja se colorea la pareja de azul. Si no se encuentra la pareja se lo colorea de rojo.

La notificación SCN_CHARADDED nos permite implementar el sangrado automático. Cuando se recibe un carácter de nueva línea se explora hacia atrás para determinar si se tiene una llave. Si se tiene una llave repregunta a la línea anterior cual era su nivel de sangrado con el mensaje SCI_GETLINEINDENTATION. Luego se incrementa el sangrado y se actualiza la nueva línea (usando SCI_SETLINEINDENTATION) con el nivel de sangrado incrementado.

El componente de listas de palabras y la función de auto completar:

Existen otras ayudas a la edición que se implementaron con la ayuda de Scintilla y de componentes adicionales. Ayudas tales como el listado de los parámetros de las funciones de la librería estándar de C y terminación automática de palabras (autocompletar).

Desde el punto de vista de scintilla, para presentar la lista de palabras que puede ser seleccionadas basta con enviar el mensaje SCI_AUTOCSHOW a Scintilla pasando como argumento las palabras candidatas. Las palabras candidatas deberán estar separadas por un carácter de espacio. Scintilla se encarga de la presentación de la lista, la selección e inserción de la palabra, y la interacción con el usuario. La figura 5.11 muestra la lista de autocompletar que presenta Scintilla.

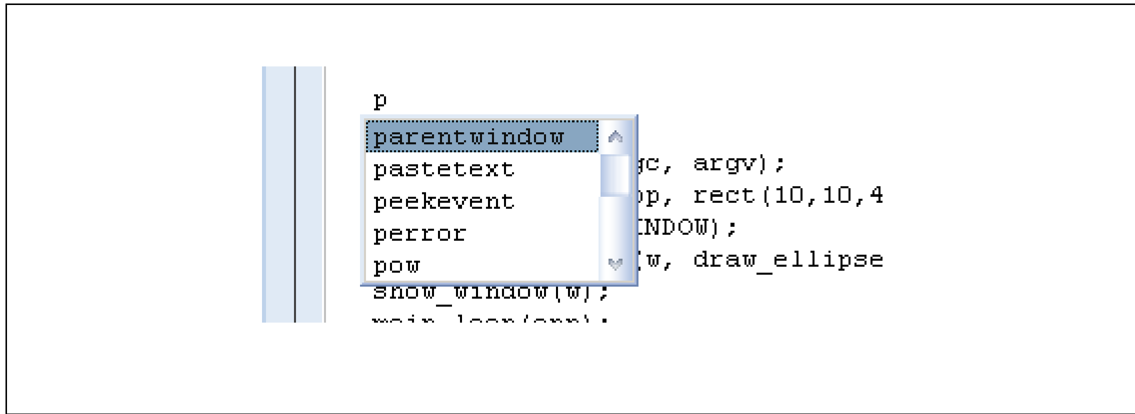


Figura 5.11. Lista de Autocompletar

La tarea del programador es generar la lista de palabras. Para hacer más fácil la tarea se creo un objeto para contener las palabras posibles. Llamaremos a este objeto WordList. Este objeto puede retornar una cadena con todas las cadenas que empiecen con determinado prefijo.

WordList se inicializa usando de dos fuentes de datos diferentes. La primera fuente de datos es un archivo de texto plano que contiene el nombre de las funciones de la librería estándar de C y de la librería de GraphApp.

La segunda fuente de datos es el propio archivo de código fuente. Ya que se desea que se puedan autocompletar también nombres de variables y funciones definidas por el usuario. Para implementar esta última característica es necesario extraer los nombres de variables y funciones del código fuente. Los nombres de variables y funciones se denominan identificadores de acuerdo a la defección que el lenguaje C le da a sus elementos.

Ya que el usuario podría pedir que el IDE complete automáticamente una palabra en cualquier momento durante la edición de código se

decidió que la mejor estrategia seria extraer los identificadores cada vez que se necesita autocompletar. Esto garantiza que siempre se tienen todos los identificadores presentes en el documento. Como se desea mantener corto el tiempo de respuesta de la interfase se necesita que el proceso de extraer los identificadores sea sumamente rápido.

Para implementar el extractor de identificadores se decidió utilizar Flex. Como se recordará de lo tratado en la sección 2.4 Flex es un generador de lexers basado en expresiones regulares. Un lexer es un componente de software que toma un flujo de caracteres y retorna los elementos que cumplen con las expresiones regulares. La estabilidad y la velocidad son los dos criterios que nos llevaron a tomar la decisión de usar Flex. Si bien es cierto que, definir las expresiones regulares puede ser más complejo que escribir código que directamente procese la cadena de texto, no tenemos garantía de que el código escrito directamente no contenga errores. Esto difiere del código autogenerado por Flex, ya que Flex es una herramienta que lleva algunos años en uso y es muy poco probable que genere código poco estable. Además, desde el punto de vista del mantenimiento del código fuente, es más simple cambiar una

expresión regular y volver a generar el código que encontrar, cambiar y probar algunas líneas de código C.

En lo referente a la velocidad de ejecución del código del lexer podemos decir que es posible escribir código que procese el flujo de caracteres igual de rápido o incluso más rápido que el código generado por Flex. Sin embargo esta tarea requeriría de una cantidad significativa de horas de trabajo.

Para generar un lexer con Flex es necesario crear un archivo que contenga una serie de expresiones regulares que definen los elementos que se desean extraer. No fue necesario escribir un archivo con las expresiones regulares ya que debido a al popularidad de C es posible encontrarlos desarrollados en Internet.

Dentro del código generado por Flex se define la función yylex().La función yylex retorna un numero entero que identifica el tipo de elemento encontrado. En nuestro caso nos interesa únicamente cuando encuentra identificadores. Para obtener el valor, es decir el texto, del identificador se utiliza la variable global yylval, la cual es un puntero a una cadena de caracteres.

Flex tiene una interfase de programación orientada a la programación estructurada. El diseño de Flex se debe a que se basa (y es compatible) con Lex y este ultimo componente se diseño alrededor de los años 70. El ejemplo mas obvio de lo antiguo del diseño de Flex es el uso de una variable global para pasar el valor del elemento encontrado.

Para mantener consisten el diseño del IDE se decidió encapsular el uso de flex dentro de una clase. La clase que encapsula a Flex es CWordListBuilder. El método CWordListBuilder::Build toma la cadena de texto que se desea analizar y carga el objeto con los identificadores encontrados. Internamente CWordListBuilder guarda los identificadores en un objeto de tipo CWordList.

Finalmente cuando el usuario pide que se complete automáticamente una palabra se pasa todo el texto del documento a la clase CWordListBuilder para que obtenga los identificadores. Luego se utiliza esta clase CWordListBuilder para indicarle a Scintilla cuales son las palabras candidatas que se vana presentar en la lista de autocompletar.

La otra ayuda a la edición es la presentación de los parámetros de las funciones de la librería estándar de C y de GraphApp. La figura 5.12 muestra la forma como Scintilla presenta los argumentos de una función.

```
printf(
       int printf (const char*, ...)
app = new_app(argc, argv);
w = new_window(app, rect(10,10,448,400),
      STANDARD_WINDOW);
on window redraw(w, draw ellipses);
```

Figura 5.12. Tip de los Parámetros de una Función

Para presentar con Scintilla los parámetros de una función basta con enviar a Scintilla el mensaje SCI_CALLTIPSHOW pasando como argumentos la posición donde se desea que se presente el cuadro y el texto que se quiere presentar. Adicionalmente se puede indicar el color del cuadro usando los mensajes SCI_CALLTIPSETBACK y SCI_CALLTIPSETFORE.

Cuando el usuario pide que se presenten los argumentos de una función se explora hacia atrás de la posición actual buscando una cadena de texto. La cadena de texto que este antes de un paréntesis

abierto y después de cualquier carácter que no sea letra o número deberá ser el nombre de la función.

Resta por indicar el mecanismo por el cual se obtienen los argumentos de la función. Ya que para esta implementación hemos decidido presentar únicamente los parámetros de las funciones de la librería estándar de C y de GraphApp, no es necesario encontrarlos durante la edición de código. Los nombres de estas funciones y sus parámetros son fijos, así que ponemos pre-cargarlos durante la inicialización del IDE. Los parámetros se almacenan en la clase CWordList. La clase CWordList nos sirve para buscar rápidamente la función.

El nombre de las funciones y sus parámetros están previamente escritos en un archivo de texto plano. El cuadro 33 muestra un extracto del archivo mencionado.

```
fullpath (char*, const char*, size_t):char*
_itoa (int, char*, int):char*
_ltoa (long, char*, int):char*
_ultoa(unsigned long, char*, int):char*
_itow (int, wchar_t*, int):wchar_t*
_ltow (long, wchar_t*, int):wchar_t*
_ultow (unsigned long, wchar_t*, int):wchar_t*
itoa (int, char*, int):char*
ltoa (long, char*, int):char*
_cgets (char*):char*
_cprintf (const char*, ...):int
_cputs (const char*):int
_cscanf (char*, ...):int
_getch (void):int
_getche (void):int
```

Cuadro 33. Archivo de los Prototipos de Funciones

En el archivo de las definiciones de las funciones, cada línea contiene una función. El nombre de la función aparece al principio de la línea para que sea más fácil buscarla. El tipo de dato del valor de retorno de la función aparece después del caracter ‘:’ en cada línea.

Cuando se encuentra la función dentro de la clase CWordList, se obtiene toda la línea. Luego es necesario separar la definición del valor de retorno y volver a construir la cadena colocando el tipo de dato al principio de la definición. Esta tarea es bastante sencilla y se ejecuta con rapidez.

5.3 Arquitectura de la Interfase con el Depurador Externo

En esta sección vamos de describir como se implementaron los mecanismos de comunicación con el depurador externo. No se describen todos los posibles comandos que se pueden enviar al depurador externo. En su lugar se describe el mecanismo común de comunicación que emplean todos los comandos.

Inicio de la aplicación que se va ha depurar:

Los mecanismos del inicio de la aplicación que se piensa depurar no difieren significativamente de la ejecución del compilador externo. De igual manera que en la ejecución del compilador, se utiliza la función del sistema operativo ::CreateProcess() y se redirige la entrada y salida tal como se describe en la sección 5.2. El trabajo de ejecutar

el depurador y de redirigir la entrada y salida es responsabilidad de la clase CSpawn.

Envío de comandos:

Como se explico en la sección 4.4.3 los comandos que se envían al depurador son simples cadenas de texto. Para facilitar el procesamiento de los comandos se decidió encapsularlos dentro de una clase, llamaremos a esta clase CCmdJob.

El objeto de CDebugger mantiene una lista enlazada de objetos CCmdJob. Cuando encuentra algún elemento en la lista, CDebugger lo retira y procede a ejecutarlo. El único punto de entrada para insertar un comando en la lista de comandos es CDebugger:: SendCommand(). SendCommand toma la cadena de texto que se desea enviar al depurador, crea un objeto CCmdJob alrededor de la cadena y coloca el nuevo objeto CCmdJob en la cola.

Se deseaba que el procesamiento de los comandos sea serializado, es decir que se ejecuten de manera sucesiva uno tras otro. Si se implementa este comportamiento de manera directa, es posible que la interfase de usuario no responda por un periodo de tiempo. Esto

no es para nada recomendable, ya que el usuario podría pensar que algo esta mal con la aplicación cuando la misma no responda a los eventos del ratón o del teclado.

Para resolver el problema del congelamiento de la interfase se empleó un sencillo modelo de productor/consumidor con dos hilos de ejecución. El primer hilo es el hilo principal de la aplicación y el segundo hilo es el hilo consumidor de comandos. El hilo principal de al aplicación responde a los eventos del usuario, tales como un click del ratón. El hilo consumidor extrae los objetos CCmdJob de la lista y los envía al depurador.

Al tener dos hilos un productor y otro consumidor introducimos el problema de la espera ocupada. La espera ocupada ocurre cuando tenemos un hilo que esta iterando en espera de algo para realizar algún trabajo útil. El problema de la espera ocupada yace en el hecho de que consume todo el tiempo del procesador, lo cual afecta el tiempo de respuesta de todos los hilos de la computadora.

La solución del problema de la espera ocupada yace en mandar al ‘dormir’ al hilo mientras espera por el evento. Un hilo ‘dormido’ es un hilo al que no se le asigna tiempo del procesador. En Windows un

hilo se duerme cuando espera por un objeto del kernel utilizando la función de la API ::WaitForSingleObject. Un objeto del kernel puede ser un proceso, un hilo, un semáforo o un evento, por mencionar algunos.

La función ::WaitForSingleObject duerme al hilo por un tiempo determinado hasta que el objeto por el que esta esperando se señalice. El tiempo de espera es definido por el programador. Al decir que un objeto del kernel se ‘señalice’ nos referimos a un estado interno del objeto del kernel. El comportamiento de señalización de los objetos del kernel depende del tipo de objeto que sea.

En el caso específico de la interfase con el depurador externo se utilizó un objeto de sincronización de tipo evento. Se señaliza utilizando la función ::SetEvent() y se deseñaliza después de que :: ::WaitForSingleObject retorna. Usando el objeto evento del kernel podemos hacer que el hilo se duerma por un intervalo de tiempo antes de revisar si existen comandos en la cola de comandos.

Como se recordará, de acuerdo a nuestros requerimientos se desea que la aplicación consuma pocos recursos computacionales y además que tenga tiempos de repuesta cortos. Pero al utilizar

::WaitForSingleObject para reducir el uso del procesador debemos especificamos un tiempo. Si se especifica un tiempo muy largo el IDE responderá con lentitud a al usuario durante la depuración. Por otro lado si se especifica un tiempo demasiado corto el uso del procesador aumenta dado que el hilo itera con más frecuencia.

Debido a que queremos que la aplicación responda de manera rápida al usuario se utilizaron tiempos de espera cortos en la función ::WaitForSingleObject. El resultado inicial fue mejor que sin la función ::WaitForSingleObject. Sin embargo aun se consumían bastante tiempo del procesador (aproximadamente 40% del tiempo del procesador de manera sostenida). Para reducir a un nivel aceptable de consumo de procesador se utilizo la función de la API ::Sleep(). Esta función duerme un hilo por el tiempo que se especifica en su argumento, no obstante si se pasa ‘0’ como argumento, el hilo cede el resto del tiempo del procesador que le correspondía. Tras revisar el código fuente del hilo consumidor se pudo determinar que era posible el uso de ::Sleep(0) para reducir a menos del 1% el uso del procesador.

Recuperación de la respuesta del Depurador:

Otra parte del diseño de la comunicación con el depurador externo es la recuperación de la respuesta del compilador. De acuerdo a lo diseñado en la sección 4.4.3 la respuesta del depurador se obtiene en forma de tuplas de la forma clave/valor. Estas tuplas se almacenan un una tabla de dispersión. Para la tabla de dispersión de utilizo la clase de MFC CMapStringToString. La clase CMapStringToString almacena pares clave/valor dentro de una tabla de dispersión.

Un elemento crucial para generar esta tabla de respuestas es la generación de claves únicas para la tabla de dispersión a partir de los identificadores encontrados en la salida. Se deseaba que el procesamiento de la salida no sea dependiente del tipo de comando enviado, para de esta manera poder agregar más comandos en caso de que se necesitara sin afectar al resto del código. Para ilustrar el problema vamos a presentar en primer lugar un caso sencillo de recuperación de la respuesta y luego un caso mucho mas complejo. Primero tomemos la salida del comando de inserción de puntos de ruptura (-break-insert):

bkptno="number",func="funcname", file="filename",line="lineno"

Se desea poder recuperar los valores usando claves generadas a partir de los identificadores en los resultados devueltos por el depurador. Es decir a partir del resultado del comando -break-insert quisiéramos tener una tabla similar a la tabla 3:

Clave	Valor
bkptno	5
func	Mover
file	c:\dir\file.c
line	27

Tabla 3. Tabla de Resultados del Depurador

En este punto aun no se presentan ninguna complicación. Únicamente basta con obtener el nombre antes del carácter =. Los valores están siempre delimitados entre un par de comillas dobles, así que su recuperación es sencilla.

Para ilustrar un caso más complejo tomemos el listado de elementos de una variable. Para listar los elementos de una variable se utiliza el comando –var-list-children del depurador. El comando puede retornar resultados con listas de listas o con nombres duplicados.

Por ejemplo, tomemos el caso del listado de elementos de una variable que es un arreglo de estructuras. El cuadro 34 muestra algunas definiciones necesarias para el ejemplo y el cuadro 35 muestra el resultado del comando.

```
typedef struct Punto
{
      float x,y;
}PUNTO,*PPUNTO;

typedef struct Rectangulo
{
      float x,y;
}RECTANGULO,*PRECTANGULO;

RECTANGULO rects[5];
```

Cuadro 34. Definiciones de Estructuras

```
numchild="5",
children={
       child={
             name=" ",exp="0",numchild="1",type="RECTANGULO"
             },
       child={
             name="var1.1",exp="1",numchild="1",type="RECTANGULO"
             },
       child={
             name="var1.2",exp="2",numchild="1",type="RECTANGULO"
             },
       child={
             name="var1.3",exp="3",numchild="1",type="RECTANGULO"
             },
       child={
             name="var1.4",exp="4",numchild="1",type="RECTANGULO"
             }
}
```

Cuadro 35. Salida de un Comando Complejo de GDB

Del cuadro 35 se puede observar que en la salida se tienen identificadores repetidos (child se repite 5 veces) y además se encuentran anidados (child esta dentro de children). Para lidiar con el anidamiento se empleo un esquema de nombres planos en el que cada nivel se separa por un carácter de punto. Es decir children.child.name indica que name esta dentro de child y child

dentro de children. Para implementar esto basta con hacer recursivo nuestro algoritmo de recuperación de resultados.

El segundo problema, la duplicación de identificadores, produce identificadores duplicados en la tabla de dispersión, lo cual produce perdida de información ya que los elementos sobrescribirían sus valores. La tabla 4 ilustra el problema.

Clave	Valor
Numchild	5
children.child.name	var1.0
children.child.exp	0
children.child.numchild	1
children.child.type	RECTANGULO
children.child.name	var1.1
children.child.exp	1
children.child.numchild	1
children.child.type	RECTANGULO
Etc…	

Tabla 4. Tabla de Resultados con Claves Duplicadas

La duplicación de identificadores se solucionó al mantener una cuenta de repetición de los nombre de los resultados. Cada vez que un nombre se repite se agrega el numero de la cuenta de repetición la nombre. De esta manera se generaría una tabla con claves únicas. La tabla 5 muestra la tabla con claves únicas.

Clave	Valor
Numchild	5
children.child0.name	var1.0
children.child0.exp	0
children.child0.numchild	1
children.child0.type	RECTANGULO
children.child1.name	var1.1
children.child1.exp	1
children.child1.numchild	1
children.child1.type	RECTANGULO
Etc…	

Tabla 5.Tabla de Resultados Mejorada

Procesamiento de la salida del depurador

El procesamiento de la salida del depurador como tal lo realiza un procesador de la gramática de la salida del depurador. Este componente funciona para cualquier tipo de salida del depurador y es independiente del contenido de la salida. Como ya se mencionó en el capitulo de diseño, el uso de una gramática nos da mayor estabilidad y tolerancia a los cambios.

El procesador de la gramática se implementó como un autómata de estado finito (FSA) basado en tabla. El cuadro 36 muestra el algoritmo del autómata empleado:

1) Leer la tabla del autómata en la estructura del autómata.
2) EstadoActual ← 1
    Terminado Falso
    Éxito ← Falso
3) Repetir
    3a) Encuentra todas las entradas en la tabla para EstadoActual
    Por cada entrada encontrada en 3a
        3b-i) Encontrado ← Falso
        3b-ii) Si el carácter y el test para la transición son verdaderos para el caracter actual entonces:
            Ejecutar la acción asociada a la transición
            EstadoActual ← EstadoDestino de la tabla
            Encontrado ← Verdadero
            Break
    3c) Si no Encontrado y EstadoActual es Estado Final entonces
            Terminado ← Verdadero
            Éxito ← Verdadero
    3d) Si no Encontrado y EstadoActual no es Estado Final entonces
            Terminado ← Verdadero
            Éxito ← Falso
4) Hasta Terminado

Cuadro 36. Seudo código de un Autómata de Estado Finito

Las figuras X y X muestran el FSA que se empleo. Se ha separado en dos diagramas para mejorar la claridad. La figura 5.13 muestra la parte más general del FSA. La figura 5.14 muestra la parte del FSA encargada de la recuperación de los resultados del depurador.

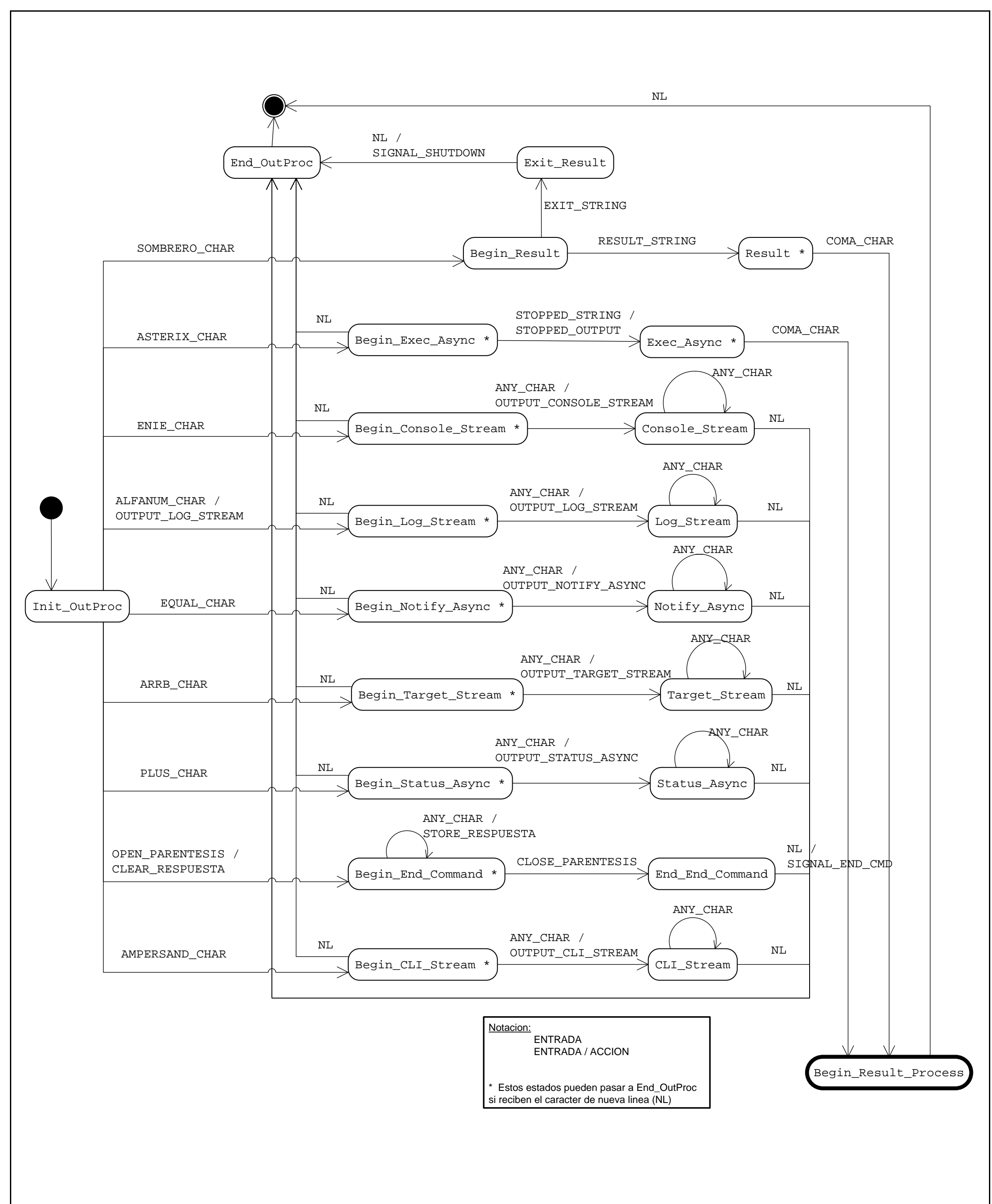


Figura 5.13. Diagrama de Estados del FSA del Depurador

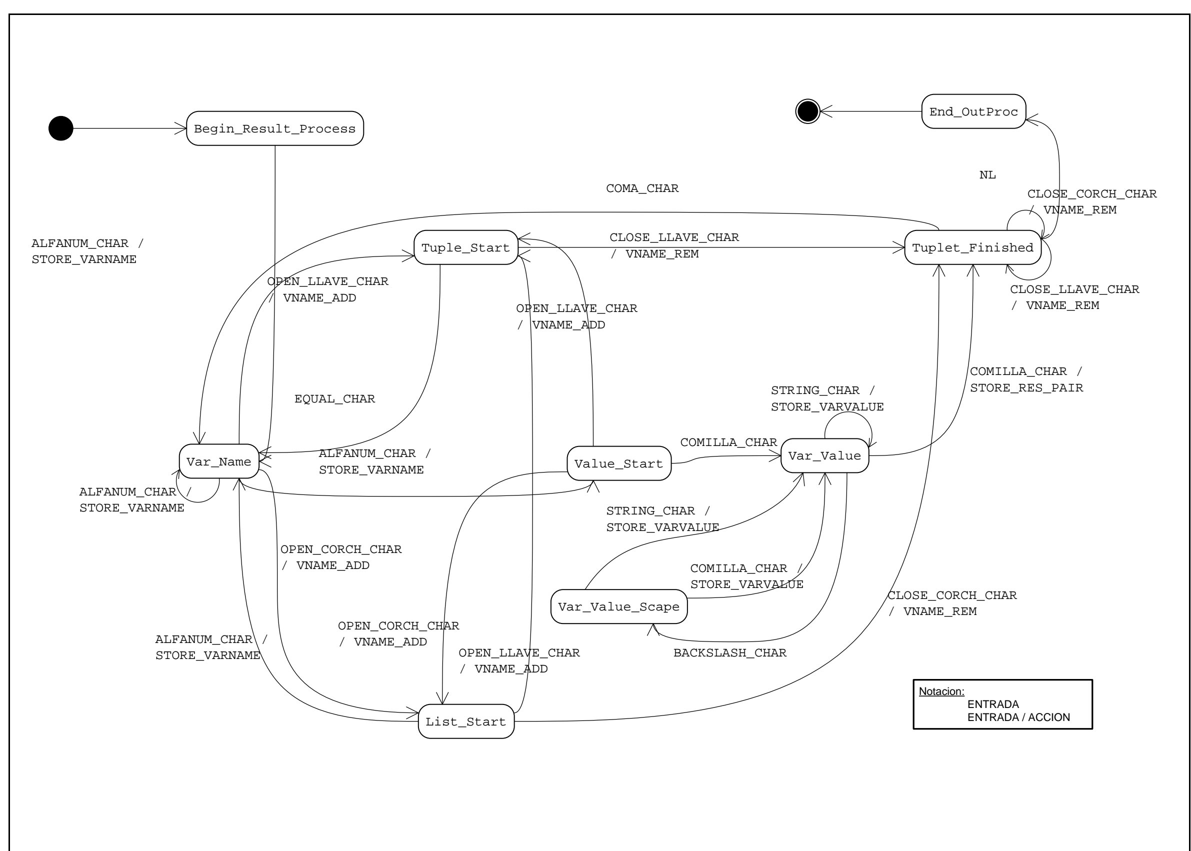


Figura 5.14. Detalle del diagrama de Estados de la Recuperación de Valores del Depurador

En ambas figuras las condiciones de transición están definidas como:

Condición	Caracteres Validos
ALFANUM_CHAR	A – Z, a – z, 0 – 9.
AMPERSAND_CHAR	&
ANY_CHAR	Caracteres ASCII entre 32 – 126 excepto los paréntesis
ARRB_CHAR	@
ASTERIX_CHAR	*
BACKSLASH_CHAR	\
CLOSE_PARENTESIS	)
CLOSE_LLAVE_CHAR	}
CLOSE_CORCH_CHAR	]
COMILLA_CHAR	“
EQUAL_CHAR	=
ENIE_CHAR	~
EXIT_STRING	‘exit’
PLUS_CHAR	+
RESULT_STRING	‘done’ , ‘running’ , ‘connected’ , ‘error’ o ‘exit’
OPEN_CORCH_CHAR	[
OPEN_LLAVE_CHAR	{
OPEN_PARENTESIS	(

NL	Carácter ASCII 13
NONE_CONDITION	Ninguna condición
SOMBRERO_CHAR	^
STOPPED_STRING	'stopped'
STRING_CHAR	Caracteres ASCII entre 32 – 126 excepto comillas dobles y backslash
COMA_CHAR	,

Tabla 6. Detalle de las Condiciones del FSA

Las acciones que toma el FSA cuando realiza una transición de estados se definen como:

NONE

El autómata de estado finito no realiza ninguna acción.

OUTPUT_LOG_STREAM

Presenta en el panel de salida del depurador el texto tal como lo envía el depurador.

OUTPUT_TARGET_STREAM

Presenta en el panel de salida del depurador el texto tal como lo envía el depurador.

OUTPUT_CLI_STREAM

Este flujo de caracteres no contiene información útil para el usuario. Por esta razón el FSA no hace nada con la salida. Sin embargo, es posible que a futuro sea necesaria si se hacen algunas mejoras al IDE.

OUTPUT_CONSOLE_STREAM

Presenta en el panel de salida del depurador el texto tal como lo envía el depurador.

OUTPUT_STATUS_ASYNC

Esta acción se ejecuta cuando el depurador informa del estado de un comando de larga ejecución. Como el depurador que estamos empleando no soporta aun esta característica no realiza nada en esta acción. Sin embargo la consideramos con fines de compatibilidad a futuro.

OUTPUT_NOTIFY_ASYNC

Esta acción se ejecuta cuando el depurador desea informar de algún error interno. Esta información se guarda en un archivo de bitácora para una depuración posterior del IDE y no se presenta el usuario.

CLEAR_RESPUESTA

Reinicializa las variables usadas en la acción STORE_RESPUESTA.

STORE_RESPUESTA

Esta acción maneja el fin de la respuesta de un comando.

SIGNAL_END_CMD

Realiza acciones necesarias cuando la aplicación depurada se detiene. La acciones especificas dependen se si la aplicación se detuvo por un punto de ruptura o por el fin de la aplicación.

STOPPED_OUTPUT

Cambia la bandera que indica la razón de la parada de la aplicación que se esta depurando. La aplicación que se esta depurando podría haberse detenido por un punto de ruptura, por una excepción o por terminación normal.

SIGNAL_SHUTDOWN

Detiene el hilo consumidor de objetos CCmdJob de la clase CDebugger. Es uno de los pasos de la terminación de una sesión de depuración.

STORE_VARNAME

Almacena los caracteres del nombre de un identificador de la salida de los resultados. En este punto no se realiza ningún manejo de duplicación ya que aun no se tiene el nombre completo del identificador.

VNAME_ADD

En esta acción se generan los nombres de las claves para la tabla de resultados. En este punto se realiza el manejo de identificadores duplicados y anidados.

VNAME_REM

Esta acción elimina e inicializa cualquier variable necesaria para la generación de las claves de la tabla de resultados.

STORE_RES_PAIR

Esta acción guarda en la tabla de resultados la tupla clave/valor.

STORE_VARVALUE

En esta acción se obtienen el valor de algún identificador de la salida de los resultados del depurador.

CONCLUSIONES Y RECOMENDACIONES

De acuerdo a lo desarrollado a lo largo de esta Tesis de Grado podemos concluir y recomendar lo siguiente:

1. El uso de componentes de terceros redujo significativamente el tiempo de desarrollo del proyecto. El proyecto del IDE tiene actualmente 35.000 líneas de código (sin considerar el código del compilador y tampoco el código de las librerías) y tomó aproximadamente 12 meses de desarrollo y una persona/mes. De acuerdo a datos históricos se estima [referencia al libro de connell] que un proyecto como el IDE tomaría 8 meses de desarrollo y 24 personas/mes; y el desarrollo de un compilador de código C/C++, el cual necesitaría entre 50.000 y 60.000 líneas de código, tomaría 16 meses y 190 personas/mes.

2. Si bien es cierto que los componentes de terceros redujeron el tiempo de desarrollo, también es necesario mencionar que su utilización conlleva el resolver problemas técnicos sobre todo en su adaptación a una solución en particular. Por ejemplo en este proyecto fue necesario escribir interfaces de programación orientadas a objetos para el código ato generado por Flex, el cual es orientado hacia la programación estructurada, de igual manera el componente del editor al ser orientado hacia dos plataformas diferentes (Windows y GTK) tenia una interfase de programación neutral que fue necesaria adaptar; Por lado, los controles desarrollados por terceros tienen una interfase de programación nueva la cual se debe aprender antes de utilizar algún nuevo control.

3. El uso de máquinas de estado finito para implementar el protocolo de comunicación con el depurador demostró ser una desciñó ventajosa porque agregó estabilidad al permitir cubrir todos los casos posibles de la salida y entrada del depurador externo.

4. De acuerdo a las evaluaciones preliminares hechas por los profesores de los cursos de Fundamentos de Programación y Estructuras de Datos que se incluyen en el anexo C, existen claros indicios de que la usabilidad de la herramienta desarrollada se ha mejorado en relación a la herramienta actual utilizada en esos cursos, la cual es LCC versión 3.
Las mejoras en la interacción con el usuario se deben principalmente a:

Al dar a la aplicación un apariencia y un comportamiento más consistente con el “look & feel” actual del sistema operativo. Al tener una apariencia más acorde a la versión actual del sistema operativo, el usuario posiblemente tenga una predisposición más favorable a la aplicación. Además se incorporan comportamiento enriquecido que se encuentra en las aplicaciones modernas tales como, la capacidad de “arrastrar/soltar”, la capacidad autocompletar y recordar el texto escrito por el usuario.

La creación de dos modos de la aplicación, un modo novato para el usuario que desea hacer programas sencillos y un modo avanzado para situaciones en las que se quiera crear aplicaciones más complejas.

La implementación de la depuración interactiva, lo cual permite que el usuario observe paso a paso la ejecución de la aplicación.

La visualización de estructuras permite crear otro modelo mental de las estructuras de datos presentes en una aplicación. Ya que el estudiante puede observar de una manera gráfica las estructuras y sus relaciones; Además de poder manipularlas para mejorar la claridad del concepto detrás de la implementación.

5. Se recomienda continuar con el desarrollo del proyecto ya que a pesar de ser un producto funcional existe una gran variedad de elementos de interacción que podrían agregarse. Por ejemplo la corrección en tiempo de edición del código fuente o el manejo automático de idiomas de programación.

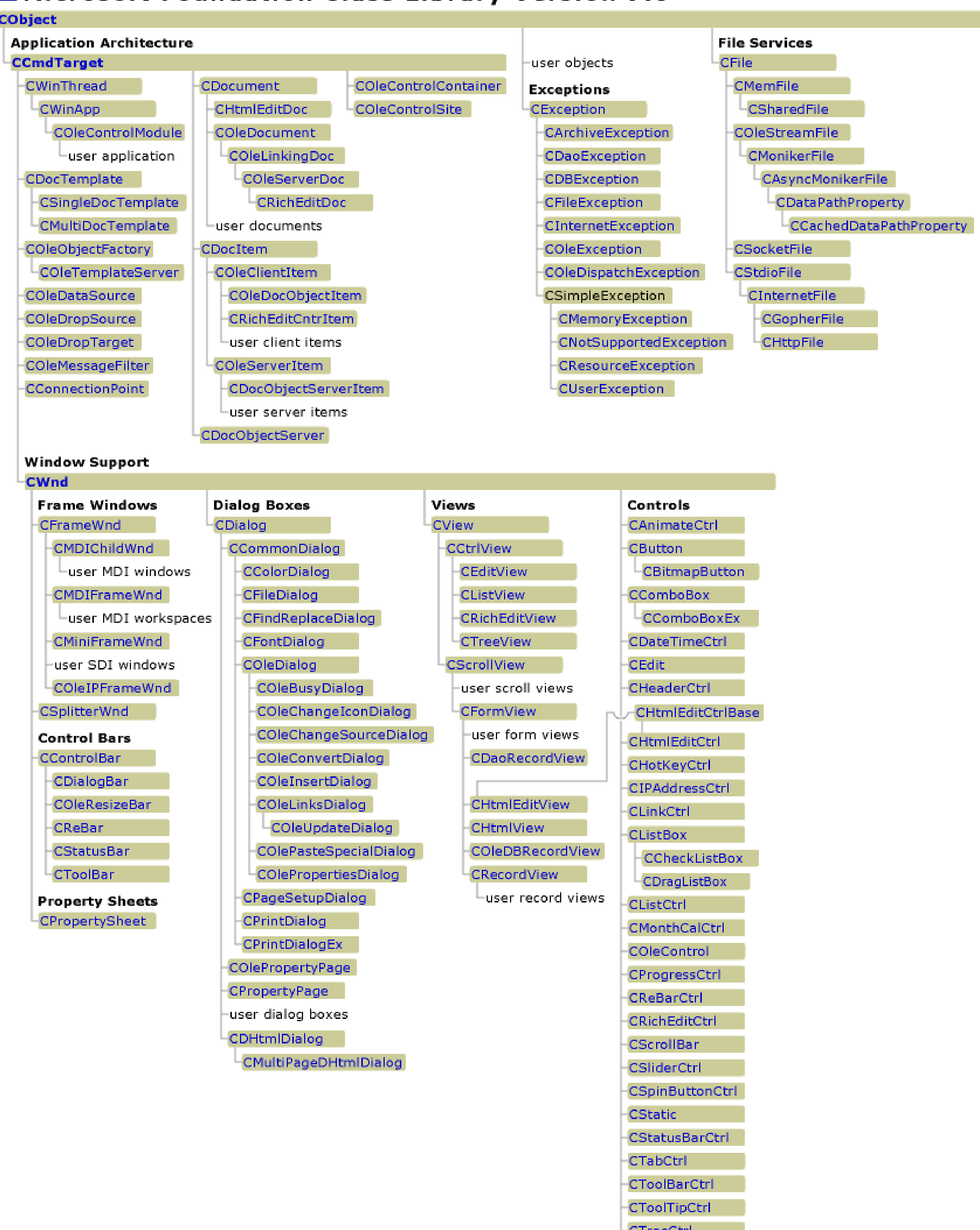
6. Se recomienda hacer una evaluación extensiva de la usabilidad de la herramienta en el ambiente educativo de la carrera de Ing. en Computación de la ESPOL para determinar de manera más certera su impacto en el proceso de aprendizaje-enseñanza.

7. Se recomienda implementar algún mecanismo para ordenar automáticamente los elementos de la visualización de estructuras ya que puede presentar problemas de sobre carga de información. Esto se requiere ya que existe un área limitada donde se presentan las estructuras de datos y no debe existir ninguna limitación en el número de visualizaciones, este problema se describe en la sección 4.3.

8. Se recomienda crear mecanismos para la visualización a un nivel conceptual del código, facilitar la navegación y la comprensión de la lógica de las aplicaciones. Mecanismos tales como una vista de las clases y estructuras disponibles al desarrollador, definiciones y constantes.

APÉNDICE A

Jerarquía de Clases de MFC


Microsoft Foundation Class Library Version 7.0
CObject
Application Architecture
CCmdTarget
CWinThread
CWinApp
COleControlModule
user application
CDocTemplate
CSingleDocTemplate
CMultiDocTemplate
COleObjectFactory
COleTemplateServer
COleDataSource
COleDropSource
COleDropTarget
COleMessageFilter
CConnectionPoint
CDocument
CHtmlEditDoc
COleDocument
COleLinkingDoc
COleServerDoc
CRichEditDoc
user documents
CDocItem
COleClientItem
COleDocObjectItem
CRichEditCntrItem
user client items
COleServerItem
CDocObjectServerItem
user server items
CDocObjectServer
COleControlContainer
COleControlSite
user objects
Exceptions
CException
CArchiveException
CDaoException
CDBException
CFileException
CInternetException
COleException
COleDispatchException
CSimpleException
CMemoryException
CNotSupportedException
CResourceException
CUserException
File Services
CFile
CMemFile
CSharedFile
COleStreamFile
CMonikerFile
CAsyncMonikerFile
CDataPathProperty
CCachedDataPathProperty
CSocketFile
CStdioFile
CInternetFile
CGopherFile
CHttpFile
Window Support
CWnd
Frame Windows
CFrameWnd
CMDIChildWnd
user MDI windows
CMDIFrameWnd
user MDI workspaces
CMiniFrameWnd
user SDI windows
COleIPFrameWnd
CSplitterWnd
Control Bars
CControlBar
CDialogBar
COleResizeBar
CReBar
CStatusBar
CToolBar
Property Sheets
CPropertySheet
Dialog Boxes
CDialog
CCommonDialog
CColorDialog
CFileDialog
CFindReplaceDialog
CFontDialog
COleDialog
COleBusyDialog
COleChangeIconDialog
COleChangeSourceDialog
COleConvertDialog
COleInsertDialog
COleLinksDialog
COleUpdateDialog
COlePasteSpecialDialog
COlePropertiesDialog
CPageSetupDialog
CPrintDialog
CPrintDialogEx
COlePropertyPage
CPropertyPage
user dialog boxes
CDHtmlDialog
CMultiPageDHtmlDialog
Views
CView
CCtrlView
CEditView
CListView
CRichEditView
CTreeView
CScrollView
user scroll views
CFormView
user form views
CDaoRecordView
CHtmlEditView
CHtmlView
COleDBRecordView
CRecordView
user record views
Controls
CAnimateCtrl
CButton
CBitmapButton
CComboBox
CComboBoxEx
CDateTimeCtrl
CEdit
CHeaderCtrl
CHtmlEditCtrlBase
CHtmlEditCtrl
CHotKeyCtrl
CIPAddressCtrl
CLinkCtrl
CListBox
CCheckListBox
CDragListBox
CListCtrl
CMonthCalCtrl
COleControl
CProgressCtrl
CReBarCtrl
CRichEditCtrl
CScrollBar
CSliderCtrl
CSpinButtonCtrl
CStatic
CStatusBarCtrl
CTabCtrl
CToolBarCtrl
CToolTipCtrl
CTreeCtrl

BIBLIOGRAFÍA

1. Kaplan Randy M., Constructing Language Processors for Little Languages (John Wiley & sons, 1994)
2. Levine John R., Tony Mason, Doug Brown. Lex & Yacc,(O'Reilly, 1995)
3. McConnell Steve, Code Complete (Microsoft Press, 1993)
4. Prosise Jeff, Programming Windows with MFC (Microsoft Press, 1999)
5. Richter Jeffrey, Programming Applications for Microsoft Windows (Microsoft Press, 1999)
6. Simple API for XML (SAX), http://www.xml.org/xml/resources_focus_sax.shtml
7. Tenenbaum Andrew, Sistemas Operativos Modernos (Prentice Hall, 1992)