INSTITUTO TECNOLÓGICO DE TUXTLA GUTIÉRREZ

INGENIERIA EN SISTEMAS COMPUTACIONALES

RESIDENCIA PROFESIONAL

"HERRAMIENTA DE MODELADO VISUAL ORIENTADO A OBJETOS PARA LA GENERACIÓN AUTOMÁTICA DE CÓDIGO EN LENGUAJE C++ PARA EL DISEÑO DE LOS MÉTODOS, UTILIZANDO LA NOTACIÓN WARNIER"

12 de enero del 2009

INSTITUTO TECNOLÓGICO DE TUXTLA GUTIÉRREZ

INGENIERIA EN SISTEMAS COMPUTACIONALES

RESIDENCIA PROFESIONAL

"HERRAMIENTA DE MODELADO VISUAL ORIENTADO A OBJETOS PARA LA GENERACIÓN AUTOMÁTICA DE CÓDIGO EN LENGUAJE C++ PARA EL DISEÑO DE LOS MÉTODOS, UTILIZANDO LA NOTACIÓN WARNIER"

ALUMNO
Hugo Enrique Morales Maldonado

ASESOR
Ing. José Alberto Morales Mancilla

REVISOR
Ing. José Alberto Morales Mancilla

Índice

Página

1. Resumen ---1

2. Introducción --2

3. Justificación --4

4. Objetivo General y objetivos específicos--5

4.1 Objetivo General --5

4.2 Objetivos específicos --5

5. Caracterización del área en que se participó -------------------------------------6

6. Problemas a resolver priorizándolos --7

7. Alcances y limitaciones--8

8. Fundamento teórico--9

8.1 Marco teórico conceptual ---9

8.1.1 Diagrama de flujo del diseño detallado---10

8.1.2 MFC ---10

8.1.3 Clases de la arquitectura de una aplicación------------------------------------12

8.1.4 Clase del objeto de aplicación--12

8.1.5 Clases para plantillas de documentos --14

8.1.6 Clase de documento---15

8.1.7 Clases para el soporte de vistas ---16

8.2 Marco teórico específico ---17

8.2.1 MFC ---18

8.2.2 Los diagramas de Warnier ---18

8.2.3 C++ --19

9. Procedimiento y descripción de las actividades realizadas----------------------20

9.1 Búsqueda de Información para la Generación de Código de Métodos -----20

9.2 Análisis de Herramientas Case---20

9.3 Análisis de Requisitos---21

9.4 Diseño de la Herramienta--22

9.5 Desarrollo de la Herramienta---22

9.6 Pruebas a la Herramienta--23

9.7 Conclusiones, Aportaciones y Trabajos Futuros-------------------------------23

10. Resultados, planos, gráficas, prototipos y programas--------------------------24

10.1 Casos de Uso --24

10.2 Plantillas de Casos de Uso ---25

10.3 Arquitectura de DDVi ---28

10.4 Funcionamiento de DDVi --30

10.5 Resultados--34

11. Conclusiones y recomendaciones --40

12. Referencias Bibliográficas --42

13. Anexos ---44

1. Resumen

Existen hoy en día gran cantidad de lenguajes de programación y muchas veces la generación de código es tarea habitual de los programadores y gracias a ello existe también gran cantidad de herramientas CASE, las cuales permiten la generación de código en distintos lenguajes a partir de modelos previamente establecidos, estos ayudan tanto a nuevos programadores, como a experimentados a facilitarles de cierta forma la generación de código, en este caso se realizaron mejoras a la herramienta CASE llamada DDVi que genera código de los métodos en lenguaje C.

Muchas herramientas Case toman como base los diagramas UML, sin embargo este maneja modelos estáticos y no permitía el uso de estructuras de control, para resolver este problema la herramienta DDVi utiliza diagramas de Warnier, que es la notación base a partir de los cuales se generó el diseño detallado de los métodos. Las mejoras que se realizaron fueron diversas ya que la herramienta DDVi carecía de varias acciones indispensables y presentaba muchos errores que de cierta manera dificultaban al usuario la utilización de dicha herramienta, todo esto trabajando bajo Microsoft Visual C++ 6.0 que es como fue creado DDVi.

Se pretendía corregir en mayor parte muchos errores que se presentaron, para ello se necesito el estudio y análisis de la herramienta, para así de esta forma lograr la comprensión de su funcionamiento, se logró corregir la manera en la cual realizaba las distintas acciones y de esta forma encontrar donde se generaban los errores para encontrar la mejor solución posible a ellos, así como el localizar las áreas que se podían mejorar en el código para implementarlas en DDVi.

2. Introducción

El grupo de Ingeniería de Software del CENIDET (Centro Nacional de Investigación y Desarrollo Tecnológico) ha desarrollado una suite de modelado basado en UML, la cuál es conocida como MANGO. Éste trabajo tiene como antecedentes la definición e implementación de un Ambiente Integrado de Soporte para la Administración y Desarrollo de Sistemas de Software (AMASS) y el SCUML (SUITE CENIDET UML). El objetivo de MANGO es la construcción de una plataforma metodológica para la producción de sistemas de software de calidad, así como enfoque de reutilización de componentes.

Dentro de MANGO se encuentran las herramientas SOODA (Sistema Orientado a Objetos para Diseño y Análisis) y DDVi (Diseño Detallado Visual). La herramienta DDVi en cooperación con SOODA permiten manejar un nivel más bajo de abstracción, el cuál ayuda a entender de manera gráfica el comportamiento de los métodos de una clase. El diseño detallado que se utiliza está basado en la notación de Warnier y está reforzado con un aspecto visual que tiene un mayor impacto en su comprensión y entendimiento. Otro beneficio importante es que la implementación de los métodos utilizando un diseño detallado nos coloca a un paso de su implementación. Finalmente, los algoritmos que hacen operar a los sistemas de software orientados a objetos, están dentro de los métodos y por tal razón hay que diseñarlos.

El trabajo de investigación que se presenta se propone a partir del DDVi teniendo en cuenta todos los aspectos que hacen falta, completando y mejorando en la mayor medida la herramienta tanto en la generación del código, como en el funcionamiento interno que maneja al desarrollar el código para los métodos. Entre los errores que se pretenden resolver y son de primera necesidad encontramos

Las herramientas de desarrollo de software tradicionales sólo incorporan conocimiento sobre el código fuente, pero ya que el análisis y diseño orientados a objetos ponen el énfasis en abstracciones y mecanismos clave, se necesitan herramientas que puedan concentrarse en semánticas más ricas. Algunas herramientas analizadas fueron las siguientes:

Class Designer: Esta herramienta viene integrada con Visual Studio 2005 y permite crear los diagramas de clases y generar código a partir de las clases, los diagramas y el código están sincronizados ya que cualquier cambio en los diagramas se ve reflejado en el código generado.

NClass versión 1.0 es una herramienta que utiliza los diagramas de clase de UML, genera código en C# y Java a partir del modelo y también utiliza ingeniería inversa para generar el modelo a partir del código fuente. La interfaz de usuario es muy simple y amigable lo que agiliza en gran medida el desarrollo de las aplicaciones

MagicDraw versión 15.1 es una herramienta CASE de modelado visual desarrollada en Java que usa la notación UML. La herramienta facilita el análisis y el diseño de los sistemas orientados a objetos y bases de datos. MagicDraw provee los mecanismos de ingeniería directa para generar código en Java, C++, C#, CORBA IDL y WSDL así como ingeniería inversa para crear el modelo a partir de los lenguajes antes mencionados

3. Justificación

En los últimos años se ha trabajado en el área de desarrollo de sistemas para encontrar técnicas que permitan incrementar la productividad y el control de la calidad en cualquier proceso de elaboración de software, y hoy en día la tecnología CASE (Computer Aided Software Engineering) reemplaza al papel y al lápiz por la computadora para transformar la actividad de desarrollar software en un proceso automatizado, la nueva generación de herramientas CASE permiten al ingeniero de software modelar gráficamente una aplicación de ingeniería y después generar el código fuente a partir del modelo gráfico.

DDVi es un sistema de software cuyo objetivo es ayudar en la elaboración de diagramas de diseño detallado, utilizando una notación basada en diagramas de Warnier, ofrece al usuario plantear sus diseños detallados de una manera gráfica haciendo uso del ratón y el teclado. Pero carece de muchas características que pueden de una forma volver más robusta a la herramienta y así poder integrarla con algún otro sistema

Las principales razones por las cuales el desarrollo de este proyecto es importante, es debido a que:

- Complementar el proceso que es parte importante del modelo de desarrollo de software, el cuál es el diseño de la arquitectura por medio del diseño detallado de los métodos por medio de diagramas de Warnier.
- A veces el código que se genera no se aproxima al nivel de abstracción del modelo. Es necesario crear una herramienta CASE que permita generar código derivado del modelo.
- Se invierte más tiempo en la codificación que en el análisis y diseño del software. Por ello es necesario crear una herramienta CASE que permita crear el modelo y a partir de ahí generar de manera automática el código.
- Cuando se hace la codificación manual se generan errores de sintaxis, por lo que es necesario automatizar la generación del código.

4. Objetivo General y objetivos específicos

4.1 Objetivo General

Optimizar la herramienta CASE DDVi complementándola en la mayor medida posible, corrigiendo errores de diseño y agregando funcionalidades que hagan robusta y completa la generación de código de los métodos detallados en lenguaje C++.

4.2 Objetivos específicos

- Corregir errores en la generación de código para el diseño detallado de los métodos.
- Resolver problemas cuando no se ha realizado algún movimiento en la herramienta y se intenta generar código ya que genera excepciones.
- Implementar el uso de variables locales.
- Facilitar la implementación de código por medio de la herramienta y sus diferentes opciones, sin tener la necesidad de tener que introducir manualmente dicho código.

5. Caracterización del área en que se participó

El Instituto Tecnológico de Tuxtla Gutiérrez es una institución educativa que pretende lograr el máximo desarrollo en la enseñanza en sus diferentes carreras para formar profesionales, que apoya proyectos que los estudiantes realizan con la finalidad de aplicarlos en la vida diaria y en los distintos medios en los que se pueda aprovechar al máximo su utilización.

Misión: Formar de manera integral profesionistas de excelencia en el campo de la ciencia y la tecnología con actitud emprendedora, respeto al medio ambiente y apego a los valores éticos.

Visión: Ser una Institución de excelencia en la educación superior tecnológica del Sureste, comprometida con el desarrollo socioeconómico sustentable de la región

Valores: El ser humano, El espíritu de servicio, El liderazgo, El trabajo en equipo, La calidad y El alto desempeño.
El Instituto Tecnológico de Tuxtla Gutiérrez se encuentra en Carretera Panamericana Km. 1080 en Tuxtla Gutiérrez, Chiapas, México.

6. Problemas a resolver priorizándolos

Inicialmente fue necesario realizar un estudio y análisis de la herramienta, para poder detectar los problemas más relevantes y las necesidades básicas. Los problemas encontrados se mencionan a continuación:

- Se desconocía en gran parte el cómo estaba integrada la herramienta y cuál era la forma en la que trabajaba.
- Falta de experiencia en algunas clases para el manejo del lenguaje de programación Microsoft Visual C++.
- Se necesitaba encontrar la mejor solución para poder optimizar el código del programa de la forma más eficiente.

Tomando en cuenta las situaciones anteriores se proponen las siguientes alternativas de solución.

- Realizar un estudio y análisis de la herramienta y de cada uno de los componentes que la integran.
- Conocer cómo actúan los distintos métodos que permiten el correcto funcionamiento de la herramienta.
- El estudio del lenguaje Visual C++ para poder implementar los métodos que fueran necesarias para poder hacer que el sistema funcione como se requiere, resolviendo los problemas que se tienen planteados.

7. Alcances y limitaciones

La herramienta a mejorar para la generación de código de los métodos detallados tendrá los alcances y limitaciones siguientes.

1. Optimizar las funciones existentes para eliminar errores en la generación de código y evitar los problemas que surgen en la realización de este.

2. Implementar un método que para la captura de las variables locales e insertarlo en los elementos gráficos que representan los diagramas de Warnier para las variables especificas.

3. Aunque se logro la corrección de múltiples errores y la optimización de la herramienta, aun quedan muchos problemas por resolver con respecto a procesos de almacenamiento del código ya generado.

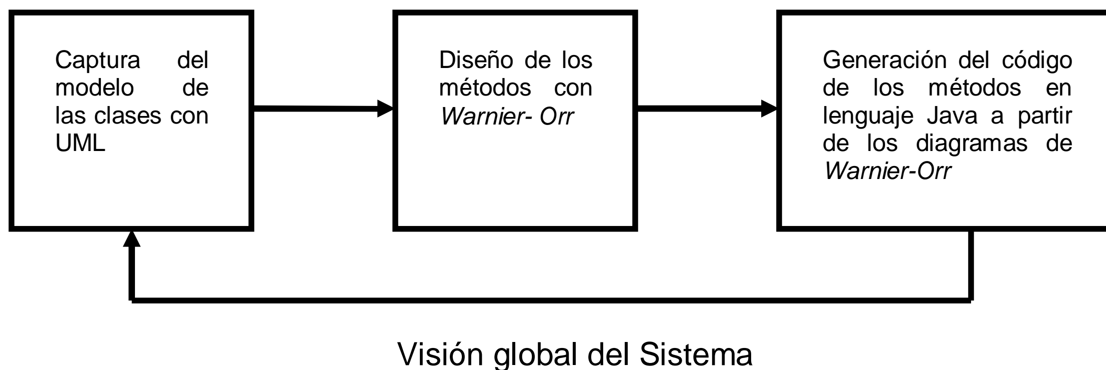
8. Fundamento teórico

8.1 Marco teórico conceptual

La metodología que aquí se propone comienza desde el proceso de analizar las clases ya creadas en las herramienta DDVi, que forma parte de la suite CENIDET UML MANGO, para posteriormente hacer el diseño de las clases y métodos, los que servirán para que pueda generar el código del diseño detallado de los métodos de las clases utilizando los diagramas de Warnier y terminar hasta generar completamente el código de las clases con el código de sus métodos.

En este proceso se contempla el hecho de que no se tiene una generación de código en el diseño detallado y en el código de clases generado, el cual considere las diferentes sentencias existentes en C++

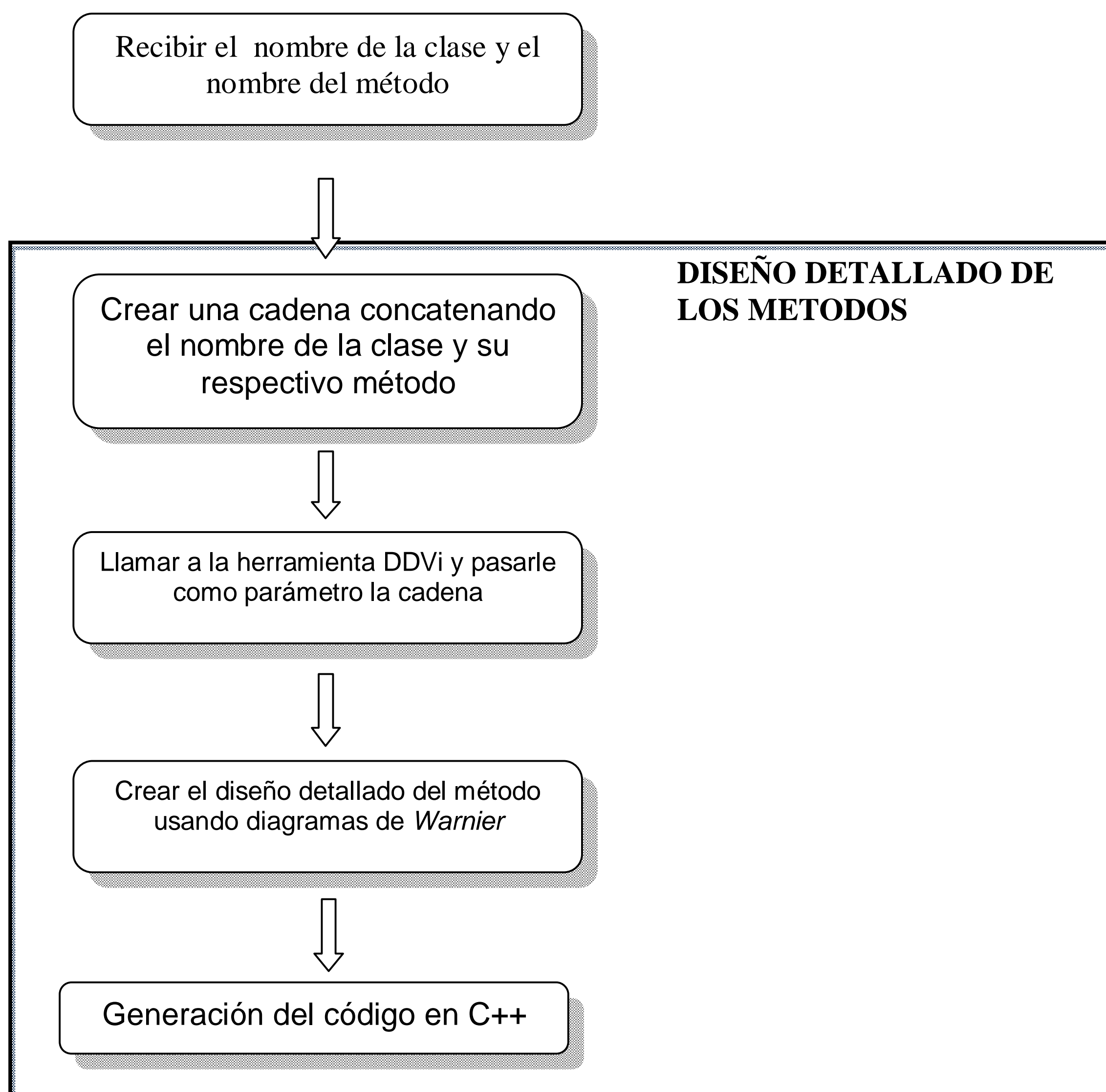
La capa de interfaz de usuario proporciona un conjunto de herramientas de interfaz que sirven para la comunicación entre usuarios y la aplicación, en donde el usuario puede interactuar con la aplicación, generando de manera automática modelos con diagramas de clases y diagramas de Warnier para el diseño detallado de los métodos de las clases.



Visión global del Sistema

8.1.1 Diagrama de flujo del diseño detallado

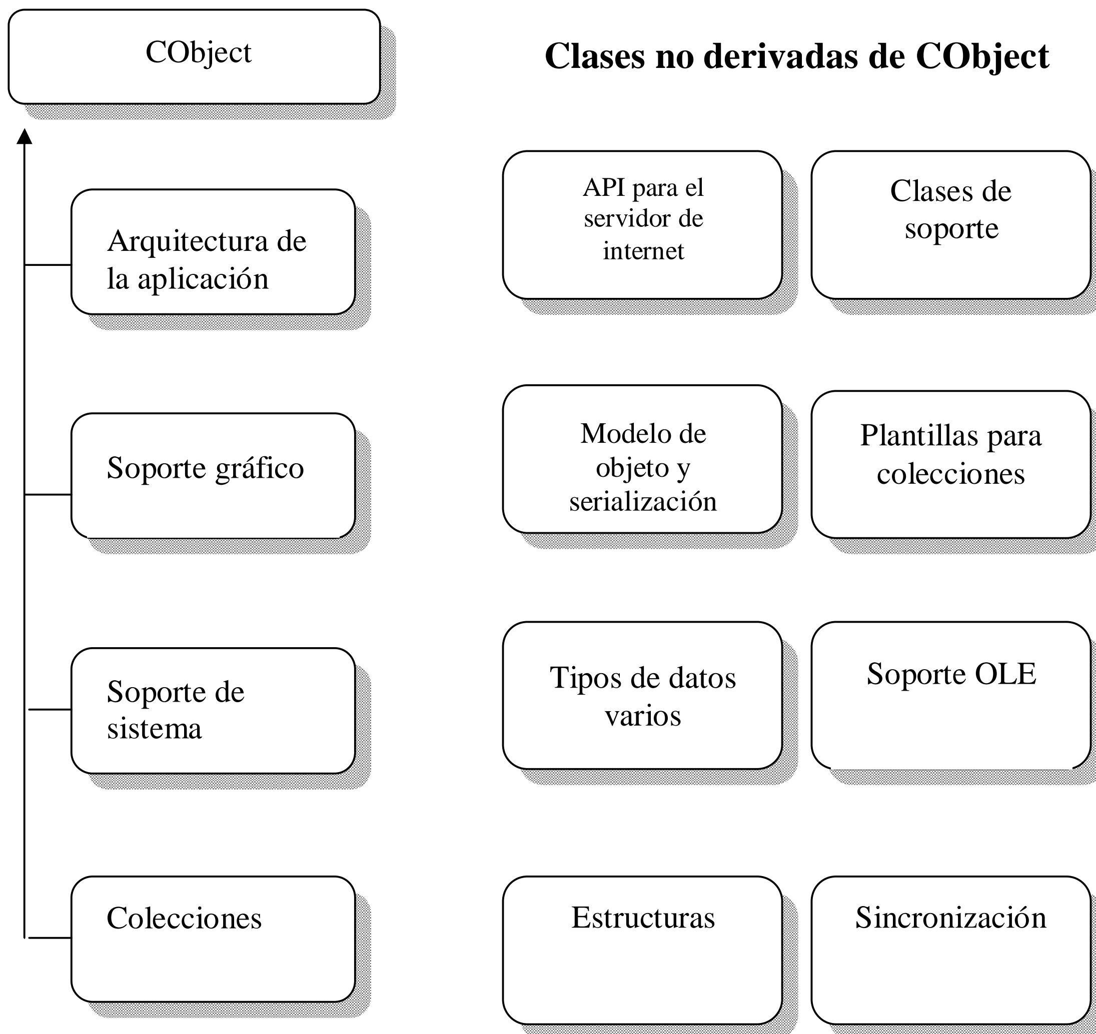
La herramienta DDVi recibe el nombre del método y su clase correspondiente, para saber la clase a la que corresponde dicho método. En la herramienta DDVi se procede a crear el diseño del método de manera visual usando los diagramas de Warnier. El diagrama de flujo que se incluye en la siguiente figura, nos muestra el conjunto de procesos que se realizarán para generar el diseño detallado de los métodos.



8.1.2 MFC

La biblioteca de clases base de Microsoft de la MFC por sus siglas en inglés Microsoft Foundation Classes es una interfaz orientada a objetos que permite desarrollar aplicaciones para Windows. DDVi basaron su desarrollo con la MFC, por lo que es importante conocer como está construida.

Las clases que proporciona la MFC se pueden clasificar de la siguiente manera y su jerarquía se muestra en la siguiente figura:



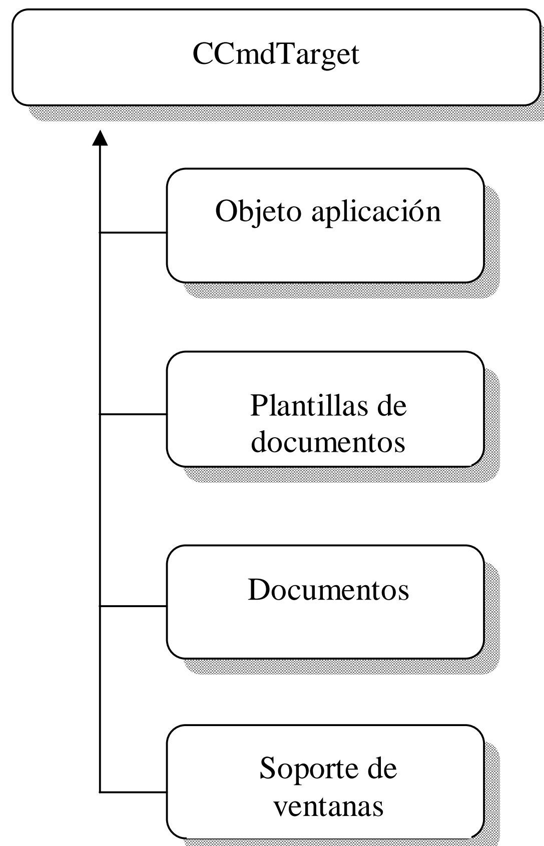
Jerarquía de clases de la MFC

CObject: Es una clase abstracta y es la clase raíz de la jerarquía de clases de la biblioteca MFC, proporciona varios servicios útiles como soporte de diagnóstico para ayudar a depurar, creación y borrado de objetos, soporte para serialización (guardar y recuperar objetos), información de tiempo de ejecución para determinar la clase de un objeto.

8.1.3 Clases de la arquitectura de una aplicación

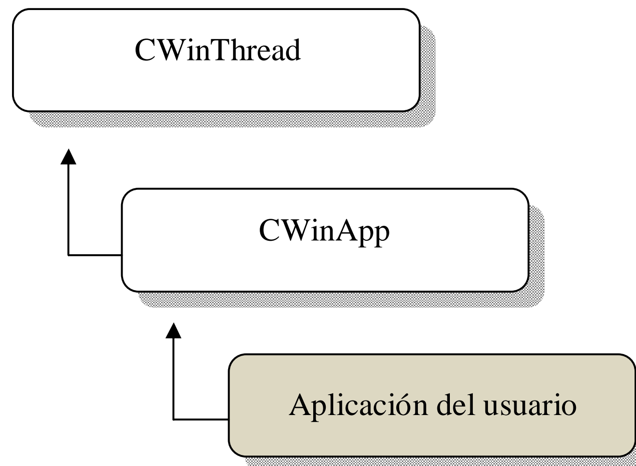
Las clases que intervienen en la arquitectura de una aplicación, están todas derivadas de la clase CCmdTarget. Un objeto CCmdTarget es un objeto que tiene un mapa de mensajes y puede procesarlos.

CCmdTarget: Esta clase permite manejar mapas de mensaje de Windows. Cualquier clase derivada de CCmdTarget puede manejar su propio mapa de mensajes, esto permite que cada clase maneje de la forma en que lo desee, los mensajes en los que tenga interés.



8.1.4 Clase del objeto de aplicación

La clase del objeto aplicación representa procesos e hilos. Cada aplicación en el marco de trabajo de las MFC tiene un objeto de aplicación de una clase derivada de CWinApp



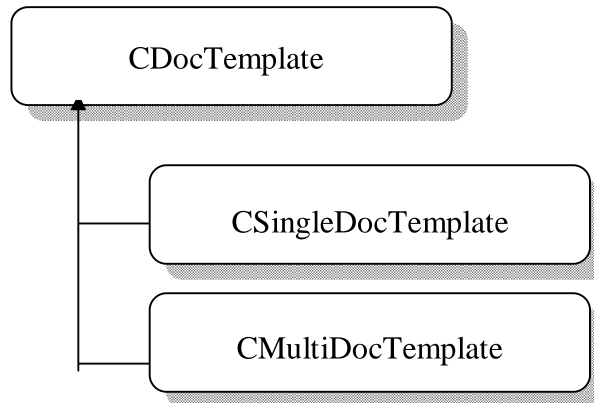
Arquitectura de un objeto aplicación.

CWinThread; Esta clase define un hilo de ejecución para la aplicación y proporciona soporte para la multitarea mediante el uso de subprocesos basada en hilos, lo cual provoca que se puedan ejecutar varias tareas al mismo tiempo.

CWinApp: Esta clase encapsula la aplicación basada en la MFC, controla el arranque, la inicialización, ejecución creación de su propia ventana, operación de un bucle de mensaje para obtener mensajes desde el sistema operativo de Windows y distribuirlos a las ventanas del programa, cierre de la aplicación y hacer limpieza después de la aplicación.

8.1.5 Clases para plantillas de documentos

Una plantilla de documento, describe la relación del documento definido por el usuario y las clases de vista utilizadas para mostrar el documento. MFC utiliza las plantillas de documento para crear y administrar la ventana de aplicación, documento y objetos vista.



Arquitectura de una plantilla de documento.

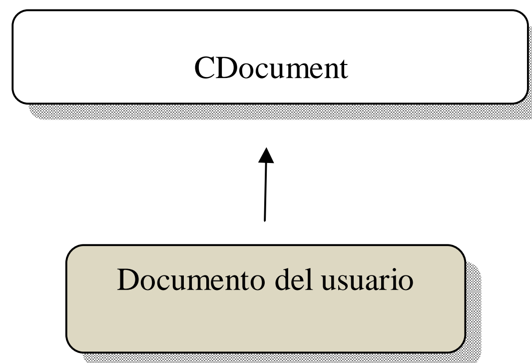
CDocTemplate: Es una clase base abstracta que define la funcionalidad básica para las plantillas de documento. Por ser una clase base abstracta, no se puede utilizar directamente.

CSingleDocTemplate: Esta clase define una plantilla de documento que implementa una Interfaz Sencilla de Documento, por sus siglas en inglés SDI (sólo un documento puede ser abierto a la vez).

CMultiDocTemplate: Esta clase que define una plantilla de documento que implementa una Interfaz Múltiple de Documentos, por sus siglas en inglés MDI (varios documentos pueden estar abiertos a la vez).

8.1.6 Clase de documento

Un documento representa una unidad de datos que el usuario puede abrir, manipular y guardar. Los objetos documento que guardan los datos del usuario están relacionados estrechamente con los objetos vista que presentan dichos datos en pantalla. Cada aplicación en el marco de trabajo de las MFC, tiene al menos un objeto documento derivado de la clase CDocument.

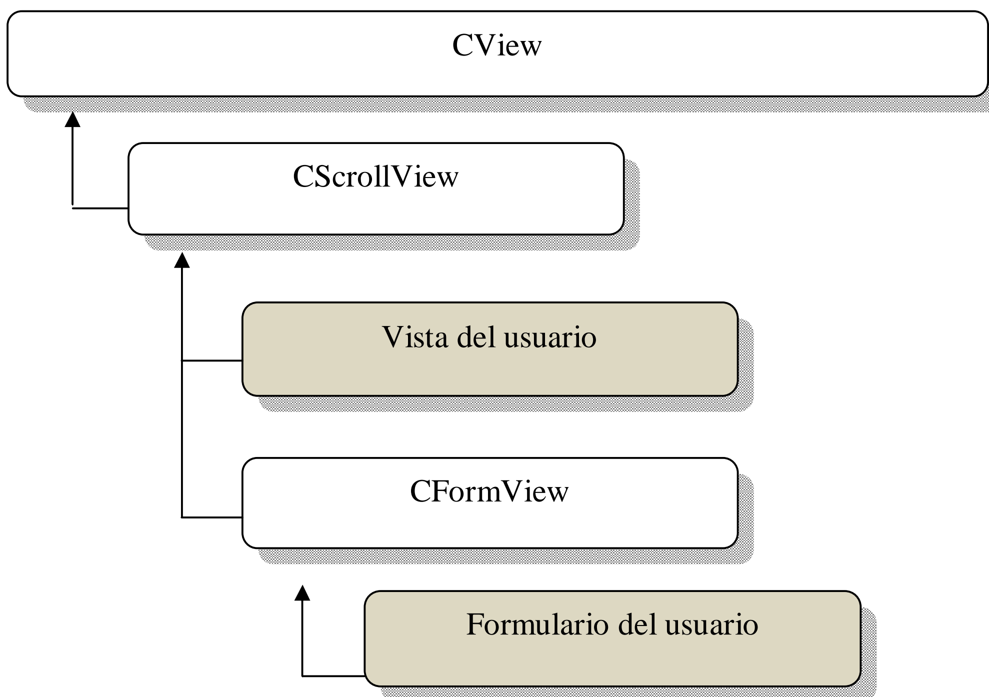


Arquitectura de la clase documento.

CDocument: Esta clase proporciona la funcionalidad para las clases documento definidas por el usuario. Un documento es un objeto para almacenar los datos de un usuario, el puede abrir y guardar los datos usando el menú Archivo.

8.1.7 Clases para el soporte de vistas

Una vista se asocia con un documento y actúa como interfaz entre el documento y el usuario. Una vista es una ventana en el diseño de una aplicación dentro del marco de trabajo de las MFC y se usa para presentar de manera gráfica el contenido del documento al usuario y de interpretar las operaciones del usuario sobre los datos del documento. La clase CView proporciona la funcionalidad para las clases vista definidas por el usuario.



Arquitectura de las clases vista.

CView: Proporciona la funcionalidad básica para la visualización de los datos.

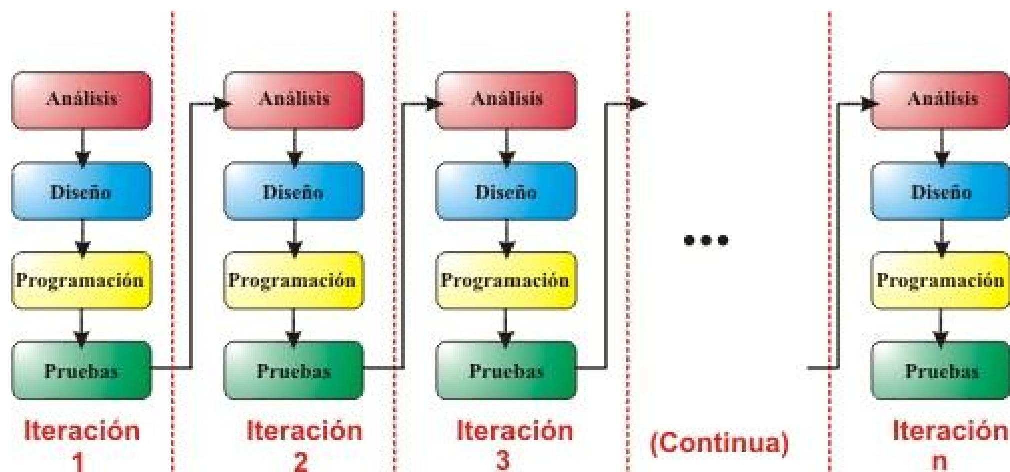
CScrollView: Esta clase permite utilizar barras de desplazamiento automático.

CFormView: Esta clase permite utilizar ventanas de diálogo como vistas lo que hace posible que la ventana principal de la aplicación tome el aspecto de una caja de diálogo.

8.2 Marco teórico específico

Basándose en el análisis previo del proyecto y en cumplimiento a las normas de la ingeniera de software el modelo que se esta utilizando para el desarrollo del mismo es el Modelo de desarrollo incremental

El modelo incremental combina elementos del modelo en cascada aplicado en forma iterativa, aplica secuencias lineales de manera escalonada conforme avanza el tiempo en el calendario, cada secuencia lineal produce incrementos del software. Al utilizar un modelo incremental el primer incremento es un producto esencial y este queda a manos del cliente. El modelo incremental es iterativo por naturaleza y se enfoca en la entrega de un producto operacional con cada incremento, los primeros incrementos son versiones "incompletas" del producto final, pero proporcionan al usuario la información que necesita y una plataforma para evaluarlo.



Modelo Incremental

8.2.1 MFC

La MFC es un sistema de clases C++ diseñado para facilitar y agilizar la programación de Windows. La MFC consiste en una jerarquía de clases de multicapas que definen aproximadamente 200 clases. Estas clases permiten construir una aplicación de Windows utilizando principios orientados a objetos. La MFC ofrece la ventaja de código reutilizable, donde las aplicaciones pueden heredar la funcionalidad de la MFC.

Otro modo en que la MFC simplifica la programación de Windows es la organización de la Interfaz de Programación de Aplicación de Windows, por sus siglas en inglés API. Los programas de aplicación se relacionan con Windows a través de la API. Las MFC encapsulan gran parte de la API en un conjunto de clases, por lo que son más fáciles de manejar.

Algunas clases definidas por la MFC no se relacionan directamente con la programación de Windows, es el caso de algunas clases de MFC que se utilizan para el manejo de cadenas, archivos y manejo de excepciones, a estas clases se les conocen como de propósito general, pueden utilizarse o no en programas de Windows.

8.2.2 Los diagramas de Warnier

También conocidos como construcción lógica de programas/construcción lógica de sistemas, fueron desarrollados inicialmente en Francia por Jean Dominique Warnier y en los Estados Unidos por Kenneth Orr. Este método ayuda al diseño de estructuras de programas identificando la salida y resultado del procedimiento, y entonces trabaja hacia atrás para determinar los pasos y combinaciones de entrada necesarios para producirlos. Los sencillos métodos gráficos usados en los diagramas de Warnier hacen evidentes los niveles en un sistema y más claros los movimientos de los datos en dichos niveles.

Los diagramas de Warnier muestran los procesos y la secuencia en que se realizan. Cada proceso se define de una manera jerárquica; es decir, consta de conjuntos de subprocesos que lo definen, en cada nivel, el proceso se muestra en una llave que agrupa a sus componentes. Puesto que un proceso puede tener muchos subprocesos distintos, un diagrama de Warnier/Orr usa un conjunto de llaves para mostrar cada nivel del sistema.

8.2.3 C++

C++ es un lenguaje de programación diseñado a mediados de los años 1980 por Bjarne Stroustrup. La intención de su creación fue el extender al exitoso lenguaje de programación C con mecanismos que permitan la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido.

Posteriormente se añadieron facilidades programación genérica, que se sumó a los otros dos paradigmas que ya estaban admitidos (programación estructurada y la programación orientada a objetos). Por esto se suele decir que el C++ es un lenguaje multiparadigma.

Una particularidad del C++ es la posibilidad de redefinir los operadores (sobrecarga de operadores), y de poder crear nuevos tipos que se comporten como tipos fundamentales.

9. Procedimiento y descripción de las actividades realizadas

9.1 Búsqueda de Información para la Generación de Código de Métodos

Para la generación de código de métodos, se hizo un análisis previo de como se representan los diagramas de Warnier, cuál es su función y su estructura. Esto mismo se aplicó a la estructura de la herramienta DDVi a modo que se observara como realizaba su funcionalidad para generar código automático.

Se analizaron las clases empleadas, métodos implentados, eventos generados de parte de la herramienta, y se observaron cómo estos eran representados en pantalla, además de que se examinó cómo el usuario interactuaba con la aplicación para que de acuerdo a las diversas opciones se pudiera elaborar un código 100% funcional.

9.2 Análisis de Herramientas Case

Durante el desarrollo de las nuevas mejoras que se realizaron a la herramienta, se realizo un análisis a las herramientas Case que existen actualmente, se consideraron las herramientas Class Designer, NClass, MagicDraw de las cuales, ninguna de ellas está en función a los diagramas de Warnier en la generación de código por diagramas de Warnier, todas las mencionadas están basadas en notación UML, pero a su vez carecen de la utilización de muchas opciones importantes en la generación que son completadas gracias a los diagramas de Warnier, lo más destacable es que se pudo tener un mejor entendimiento de cómo funcionan y reaccionan los diagramas de Warnier ante distintos aspectos que el usuario contempla para poder generar el código.

9.3 Análisis de Requisitos

Durante ésta actividad, se requirió que la herramienta DDVi, estuviera en funcionamiento ya que para la generación de código en C++ se le pudieran implementar distintas opciones que permitan al usuario final hacer uso más práctico y sencillo, así como se necesitaba la opción de poder importar atributos, corregir los errores que tenia al generar código, errores generados para las opciones de almacenamiento, el poder implementar una nueva clase para la generación de variables locales y que de manera automática facilitara el poder elegir el tipo de dato, nombre y valor sin la necesidad de escribirlo en pseudocódigo y este insertarlo en los elementos gráficos que representa el diagrama de Warnier para las variables especificas y corregir excepciones que generaba de manera frecuente con algunas opciones.

Se observaron que algunas clases requerían ajustes, e incluso se requería una clase más, se agregaron métodos.

Se solicitó que el programa no permitiera DDVi generar código sino se guarda antes ya que es la forma que la otra herramienta SOODA puede recuperar la información que genera DDVi, para que lo incluya como código dentro del método que esta declarado dentro del SOODA.

Se observa que con los símbolos de Warnier se puede insertar estructuras de control, mandar a imprimir o pedir un dato pero no existe manera de declarar variables, una opción para escribir seudo código por ejemplo: int x = 10.

9.4 Diseño de la Herramienta

Con base al diseño anterior que con el que se contaba de la herramienta DDVi, se realizaron algunos ajustes al diseño, como el sistema utilizaba un mismo método y una misma clase para todo las opciones (if for while), esto representaba un impedimento para poder agregar el modulo para las variables, por lo que se decidió hacer una nueva clase específica para las variables y además una nueva interfaz gráfica.

Se tuvo que diseñar un método que mandara a llamar esta clase y mostrara la ventana así como este mismo método recibiría los valores retornados por esta ventana y una ve tomados los valores recibiera el tipo de dato y si tiene un valor o no y se agrega a la lista de elementos gráficos que se ven en pantalla, es decir, las variables y estructuras de control y otras cosas que se ven representadas en pantalla gráficamente.

9.5 Desarrollo de la Herramienta

En esta fase, se desarrolló el código correspondiente a las modificaciones desarrolladas en la fase de diseño.

Posteriormente se alteró el método OnGeneracodigoc y se agregó a la lista para generar código, las variables, la clase que se diseñó para ver la ventana se llama: Variables_AKIRAFLAME y el método que se diseñó, es el que manda a llamar esta clase y recibe los parámetros devueltos, se llama ObtenVariable_AKIRAFLAME.

Ese método recibiría los datos que se escribieron en la ventana, después se tuvo que modificar el método Invariables, se le agregaron instrucciones para que llamara el método (obtenVariable_AKIRAFLAME) y además lo agregara a la lista de símbolos que se ven en pantalla representados.

Con respecto a OnGeneracodigoc se agregó a la lista de código a generar las variables, en este último método se le corrigieron muchos bugs (errores) ya que no permitía generar código, a menos que se guardara en la clase desarrollada, ya que en la ventana tiene un combobox que permite seleccionar el tipo de dato, un campo para escribir el nombre de la variable y otro campo por si se le quiere poner un valor.

9.6 Pruebas a la Herramienta

Las pruebas realizadas a la herramienta, se basaron en los resultados obtenidos de los de las pruebas realizadas a la herramienta, explicando los alcances obtenidos como resultado de los análisis de las pruebas, el objetivo de estas pruebas es constatar que es posible generar el código en C++ a partir de los diagramas de clases del UML y código en C++ para el diseño detallado de los métodos de las clases a partir de los diagramas de Warnier.

9.7 Conclusiones, Aportaciones y Trabajos Futuros

En esta actividad, se obtuvieron las conclusiones del desarrollo del proyecto, se elaboró el reporte del proyecto y se realizó la proyección de las posibles mejoras y un desarrollo futuro.

10. Resultados, planos, gráficas, prototipos y programas

10.1 Casos de Uso

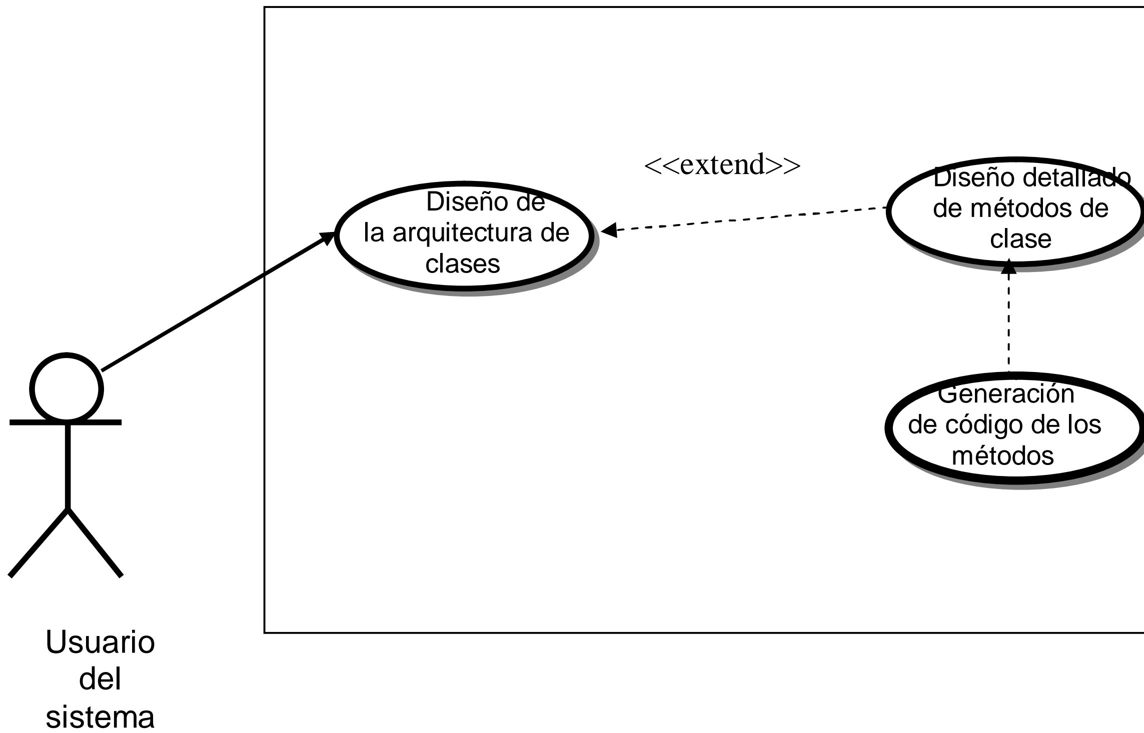


Diagrama de casos de usos del sistema.

10.2 Plantillas de Casos de Uso

Caso de uso diseño de la arquitectura de clases

Nombre del caso de uso	Diseño de la arquitectura de clases
Descripción	Permite capturar el modelo de la arquitectura de clases usando la notación del UML y crear una lista con doble ligadura.
Actores participantes	Usuario del sistema
Precondición	
Condición inicial	1. El actor activa el botón de dibujo de clases de la barra de herramientas de diseño.
Flujo de eventos	2. El sistema muestra el botón seleccionado y el actor puede ubicarse en la posición que desee de la ventana principal para dibujar cada clase. 3. Haciendo click en la posición seleccionada de la ventana principal se dibuja la clase. Si desea agregar más clases, hace clic en otra posición de la ventana principal. Si ya no desea agregar más clases oprime el botón apuntador que se encuentra en la barra de herramientas. 4. Haciendo doble click sobre la clase dibujada, se despliega el cuadro de diálogo propiedades. 5. El actor captura el nombre, atributos y nombres de los métodos de la clase en el cuadro de diálogo y se guardan los datos de la clase en una lista con doble ligadura. 6. Si el actor desea agregar más clases, oprime el botón de dibujo de clases que se encuentra en la barra de herramientas y repite los pasos 2, 3, 4, 5 y 6. 7. Si el actor desea agregar, generalización, agregación o composición, puede seleccionar los botones de la barra de herramientas respectivamente para su diseño.
Condición de salida	1. El modelo creado con el diagrama de clases del UML se muestra en pantalla.
Poscondición	1. El modelo con diagrama de clases se guarda en una lista con doble ligadura.

Caso de uso diseño detallado de métodos de clase

Nombre del caso de uso	Diseño detallado de métodos de clase
Descripción	Permite capturar el modelo con diagrama de Warnier y crear una lista con doble ligadura.
Actores participantes	Usuario del sistema
Precondición	Haber construido su clase contenedora.
Condición inicial	1. Desde el cuadro de diálogo donde se capturan el los atributos y métodos de las clases, el actor selecciona el nombre del método y oprime el botón derecho del mouse, aparece una lista con la opción diseño, selecciona diseño y llama automáticamente a la herramienta DDVi para su diseño.
Flujo de eventos	2. El sistema muestra la barra de herramientas que contiene el conjunto de símbolos de la notación del diseño detallado de Warnier. 3. El actor puede seleccionar en la posición que desee de la ventana principal para dibujar cada símbolo. 4. El símbolo elegido por el actor es insertado al final de la lista de objetos. 5. Cada vez que se inserta un símbolo en el diseño, se verifica el anidamiento de las funciones. 6. Si el actor inserta un símbolo que representa un else, se hace un recorrido en la lista para verificar que debe haber antes un if. Por cada sentencia de control se verifica que exista un fin correspondiente. 7. Después de un símbolo fin del sistema el actor ya no puede agregar más símbolos.
Condición de salida	1. El modelo creado con el diagrama de Warnier se muestra en pantalla.
Poscondición	1. El modelo creado con la notación del diseño detallado de Warnier, se guarda en una lista con doble ligadura.

Caso de uso para la generación de código detallado de métodos

Nombre del caso de uso	Generación de código detallado de métodos.
Descripción	Permite generar el código de los métodos de las clases en C++ a partir de un modelo con diagramas de Warnier.
Actores participantes	Usuario del sistema.
Precondición	Lista con doble ligadura en donde se guarda el modelo creado con el diagrama de Warnier.
Condición inicial	1. El actor selecciona el botón de generación de código en C++.
Flujo de eventos	2. El sistema recorre la lista con doble ligadura y va generando automáticamente el código en C++ en un archivo de texto. 3. El actor puede abrir el archivo de texto donde se encuentra el código en C++.
Condición de salida	1. Archivo de texto generado, donde se encuentra el código en C++ del diseño detallado del método.
Poscondición	1. El código generado en C++ se guarda en un archivo de texto con el nombre del método. 2. El modelo del diseño detallado de Warnier se guarda en un archivo de diseño.

10.3 Arquitectura de DDVi

La herramienta DDVi sirve para generar código en C++ a partir del diseño detallado de los métodos, para ello utiliza símbolos gráficos que sirven para representar de manera visual las estructuras y datos de los programas. En la clase CLevissDoc se encuentra definida la lista mListaSimbolos que es de tipo CTypedPtrList, que es una clase de la MFC, que sirve para manejar la programación genérica mediante plantillas de colecciones (templates). En la lista se guardan objetos de la clase CSimbolo, como se muestra en las siguientes líneas de código:

```
CTypedPtrList<CObList, CSimbolo*> mListaSimbolos;
```

La clase CSimbolo sirve para representar los elementos que forman parte del diagrama del diseño detallado, esta clase se deriva de la clase CObject de la MFC para manejar serialización. En la clase CLevissDoc también se encuentra el método InsertaSimbolo() el cual crea un apuntador a un objeto oSimbolo de la clase CSimbolo y por medio de este apuntador se hace referencia a los datos del objeto de la clase para guardar los datos, una vez obtenidos estos, se almacenan en la lista como se muestra en las siguientes líneas de código:

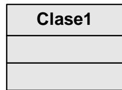
```
mListaSimbolos.AddTail(oSimbolo);
```

La clase CLevissView contiene los manejadores de mensaje de cada uno de los iconos de figuras de la barra de herramientas de DDVi, por ejemplo para el icono que representa el ciclo for tiene el manejador de mensaje Onfor() y así sucesivamente para los demás iconos. En esta clase también se encuentran los métodos para el recorrido de la lista como se muestra a continuación:

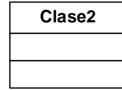
```
POSITION pos = pDoc->mListaSimbolos.GetHeadPosition();
while (pos!= NULL){
        CSimbolo* pSimbolo = pDoc->mListaSAimbolos.GetNext(pos);
        pSimbolo->Dibuja(pDC);
}
```

Por medio del método OnDraw() de la clase CLevissView se hace el recorrido en la lista como se explicó anteriormente y desde ahí se llama al método Dibuja() que pertenece a la clase CSimbolo, el cual sirve para dibujar en la pantalla el mapa de bits correspondiente a cada figura para ello se usa un objeto de la clase CBitmap.

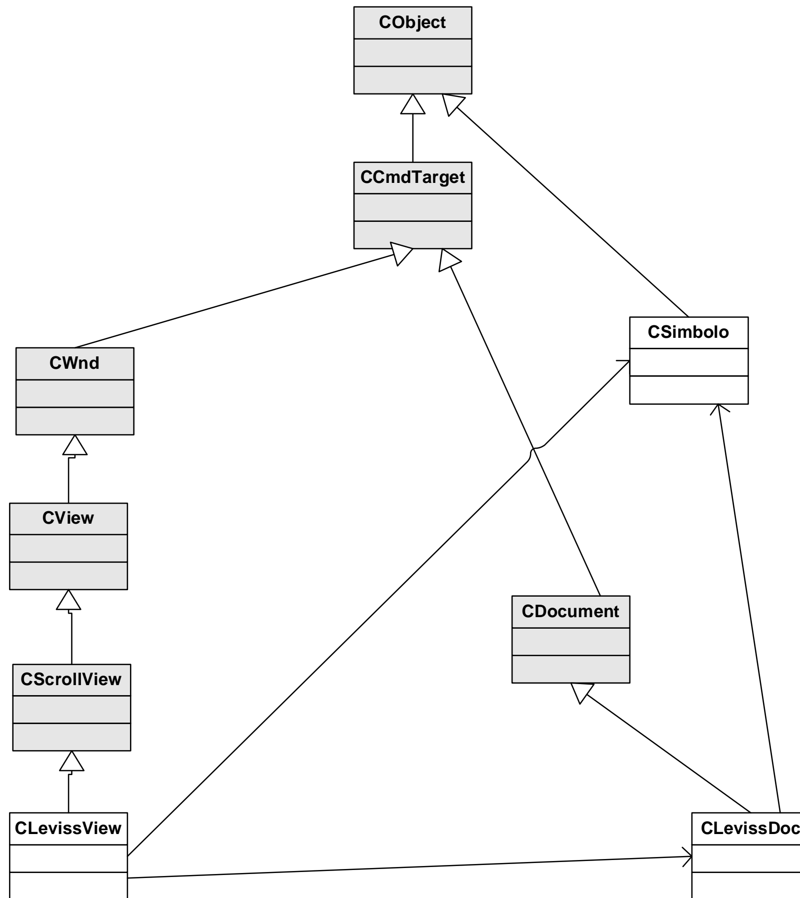
En la clase CLevissView se encuentra el método OnGeneracodigo() que sirve para generar código en C++, para llamar a este manejador de mensaje se oprime el icono que se encuentra en la barra de herramientas de la interfaz de DDVi y que sirve para tal propósito. La Jerarquía de clases del documento/vista de DDVi se presenta a continuación:



Clases de la MFC



Clases de la herramienta DDVi



Jerarquía de clases del documento/vista de DDVi.

10.4 Funcionamiento de DDVi

Cuando se llama a la herramienta DDVi desde la herramienta SOODA con la instrucción WinExec(cadena, SW_SHOWNORMAL) en DDVi desde el método InitInstance() de la clase CLevissApp se crea el archivo, esto se logra usando la variable lpCmdLine como parámetro pasado por Windows a WinMain. Para apuntar a la

línea de comando de la aplicación se usa a m_lpCmdLine para acceder a los argumentos de la línea de comando, la variable m_lpCmdLine es una variable pública de tipo LPTSTR. En las siguientes líneas se muestra el código:

```
BOOL CLevissApp::InitInstance()
{
        if (m_lpCmdLine[0] == ´ ´)
        {
                //crea un documento vacío
                OnFileNew();
        }
        else
        {
                //Abre un archivo pasado como parámetro en el primer //argumento
                //de la línea de //comando
                CString sCmdLine(m_lpCmdLine)
                I
                if (sCmdLine.Find(“.dis”)== -1){
                        OnFileNew();
                }
                else
                        OpenDocumentFile(m_lpCmdLine);
        }
}
```

Cuando se presiona un icono de la barra de herramientas de la interfaz de DDVi, se ejecuta el manejador de mensajes respectivo que se encuentra en la clase CLevissView, dentro de este manejador por medio del apuntador pDoc de la clase CLevissDoc se llama la método InsertaSimbolo() al cuál se le envía como parámetro el símbolo seleccionado, dentro de este método se crea un objeto de la clase CSimbolo y

se guardan en él los datos del símbolo seleccionado, y una vez creado el objeto y almacenado los datos se inserta en la lista como se explico anteriormente.

Para mostrar en la interfaz de DDVi el icono seleccionado, se usan dos métodos de la clase CDocument, GetFirtViewPosition() y GetNextView(), el primer método obtiene la primera posición de la vista del documento y el segundo método sirve para iterar en todas las vistas del documento. La función regresa la vista identificada por la posición pos que es de tipo POSITION como se muestra en las siguientes líneas de código:

```
void CLevissDoc::InsertaSimbolo(UINT nBitMap, int Nivel, CString Descrip){
        CSimbolo *oSimbolo = new CSimbolo;
        POSITION pos = GetFirstViewPosition();
        CView *pView = GetNextView(pos);
}
```

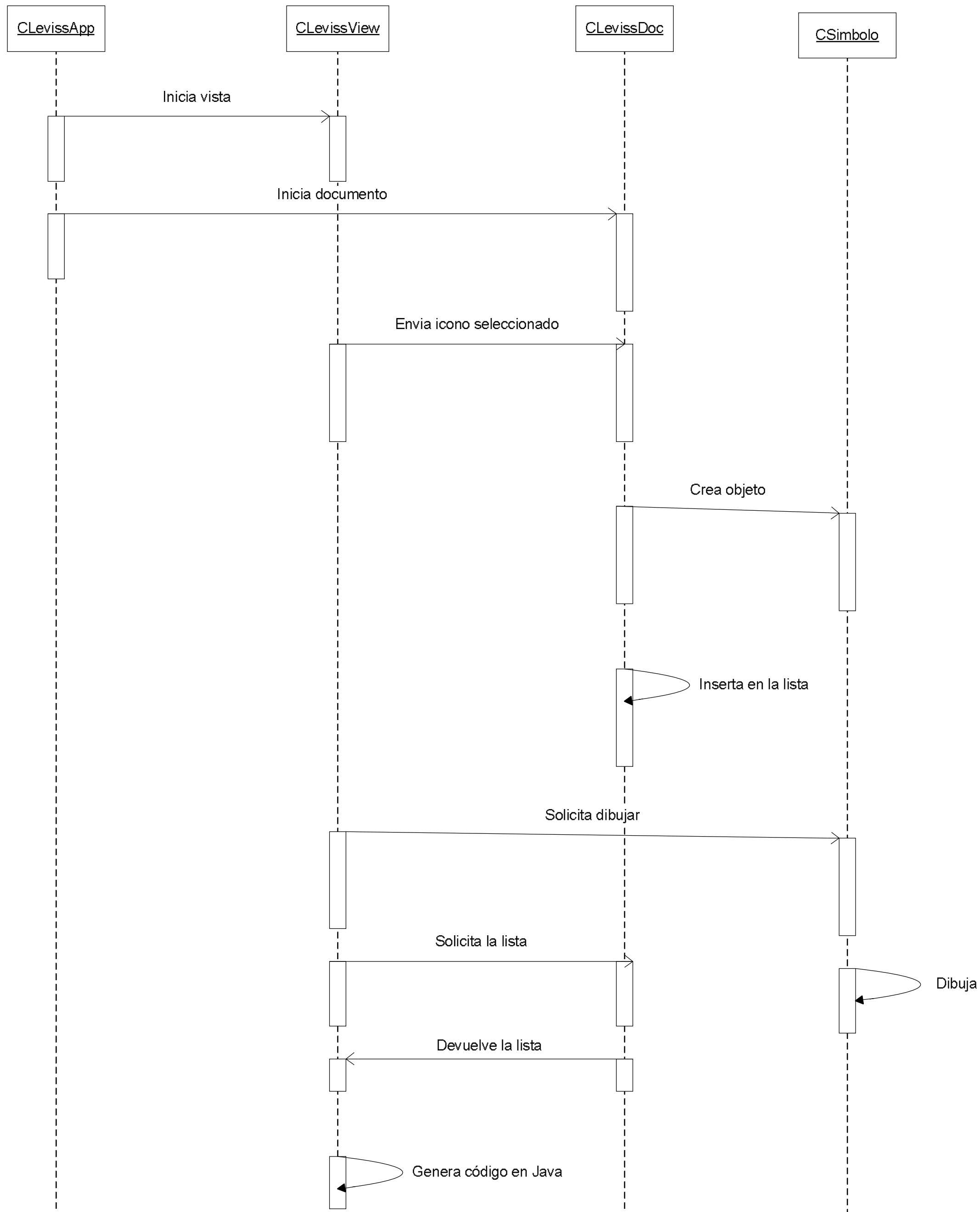
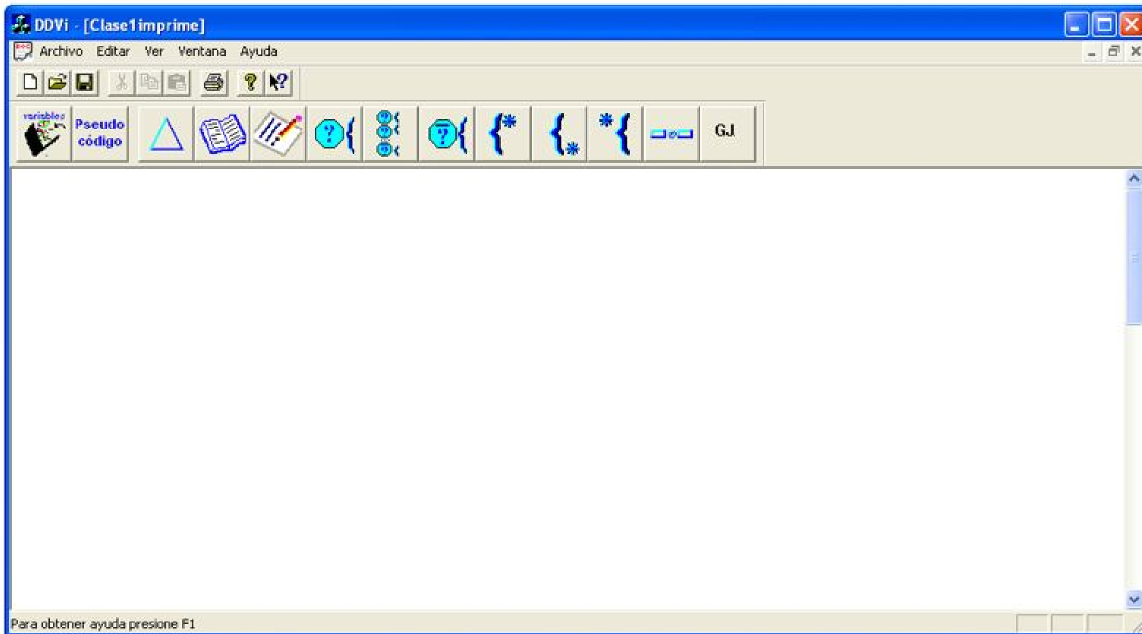


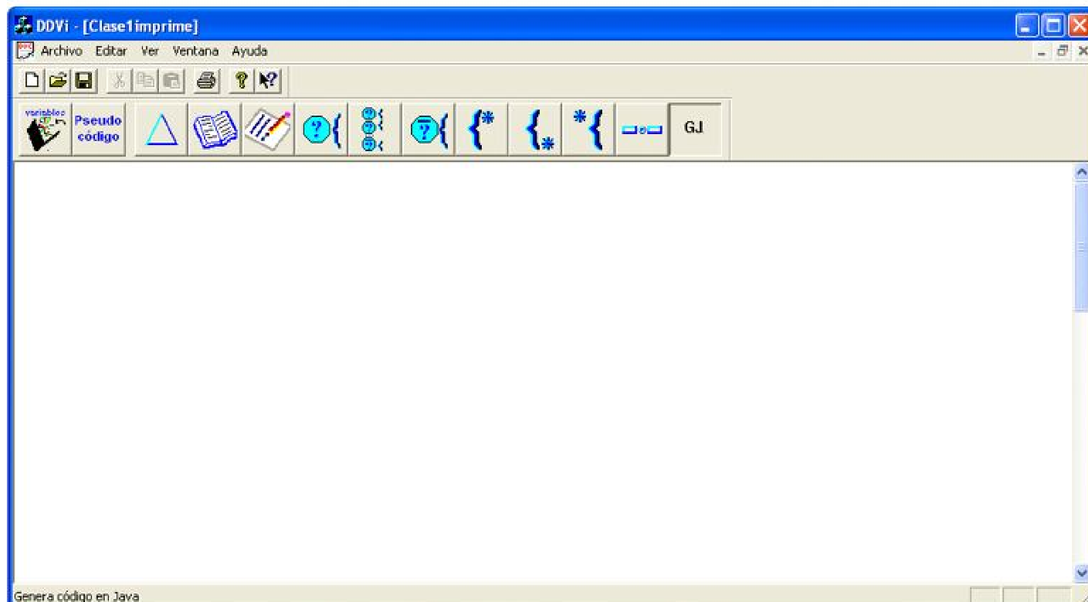
Diagrama de secuencias para la creación del diseño de los métodos.

10.5 Resultados

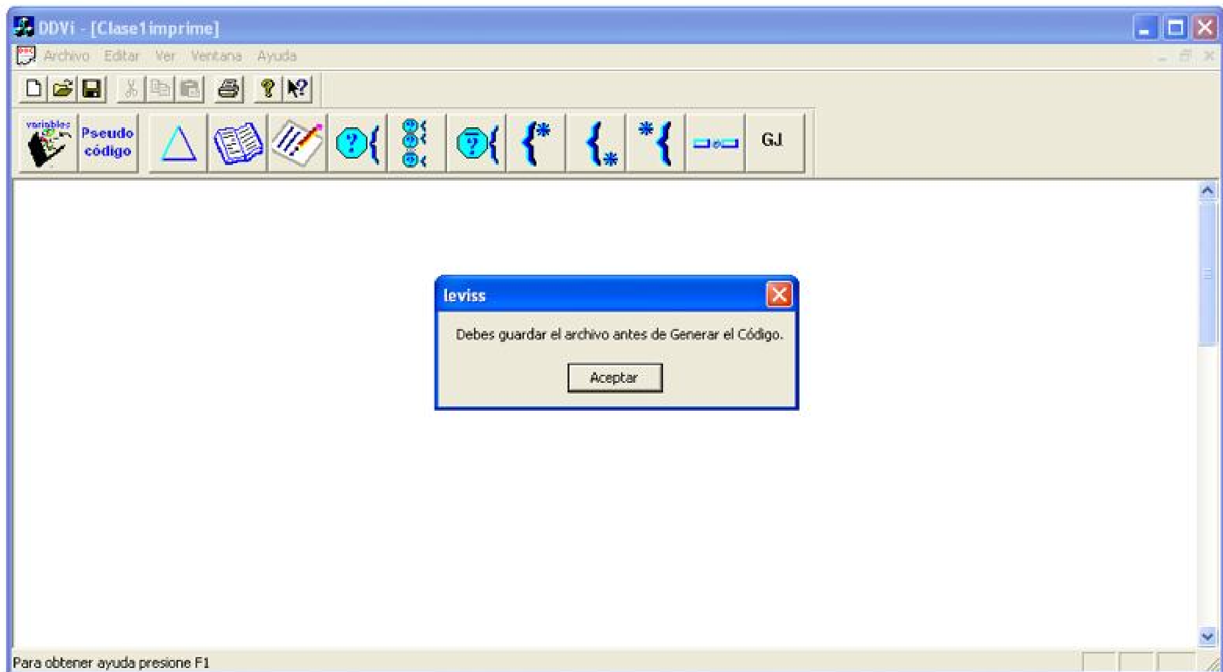
De acuerdo con los ajustes que se requerían para el correcto funcionamiento de la herramienta, se muestran los resultados obtenidos, esta es la pantalla principal:



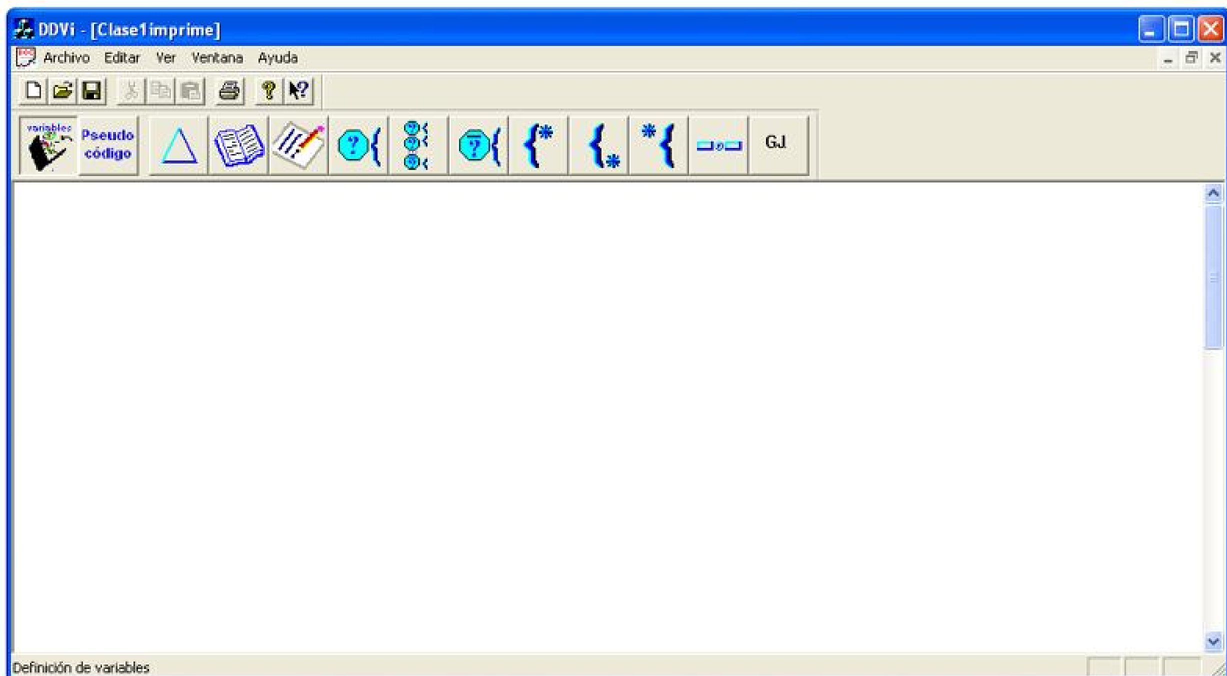
Esta parte se muestra el funcionamiento de DDVi, esta imagen enseña como se debe guardar el código antes, para que con ayuda de la herramienta SOODA, pueda emplearse. A continuación se presiona GJ para generar el código una vez terminado de introducir los distintos datos



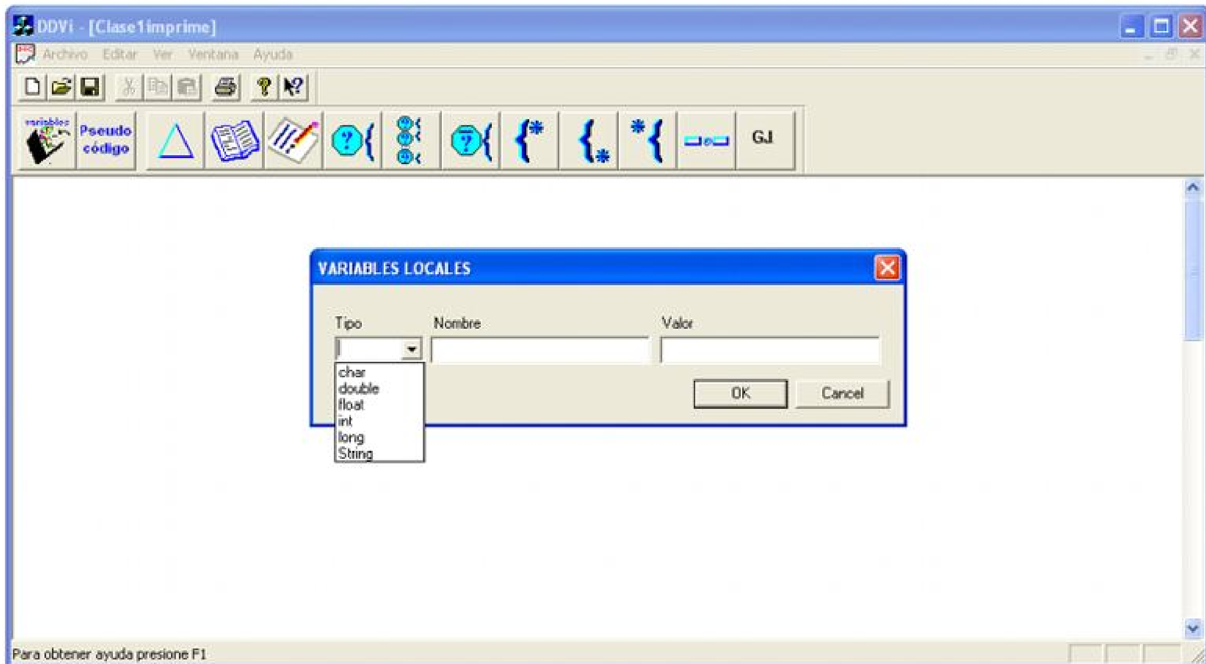
En esta parte, el software cumple con lo que nos solicitó el cliente, que guarde el archivo antes de generar el código, incluso, le da indicaciones al usuario para que guarde el archivo antes de que se genere el código.



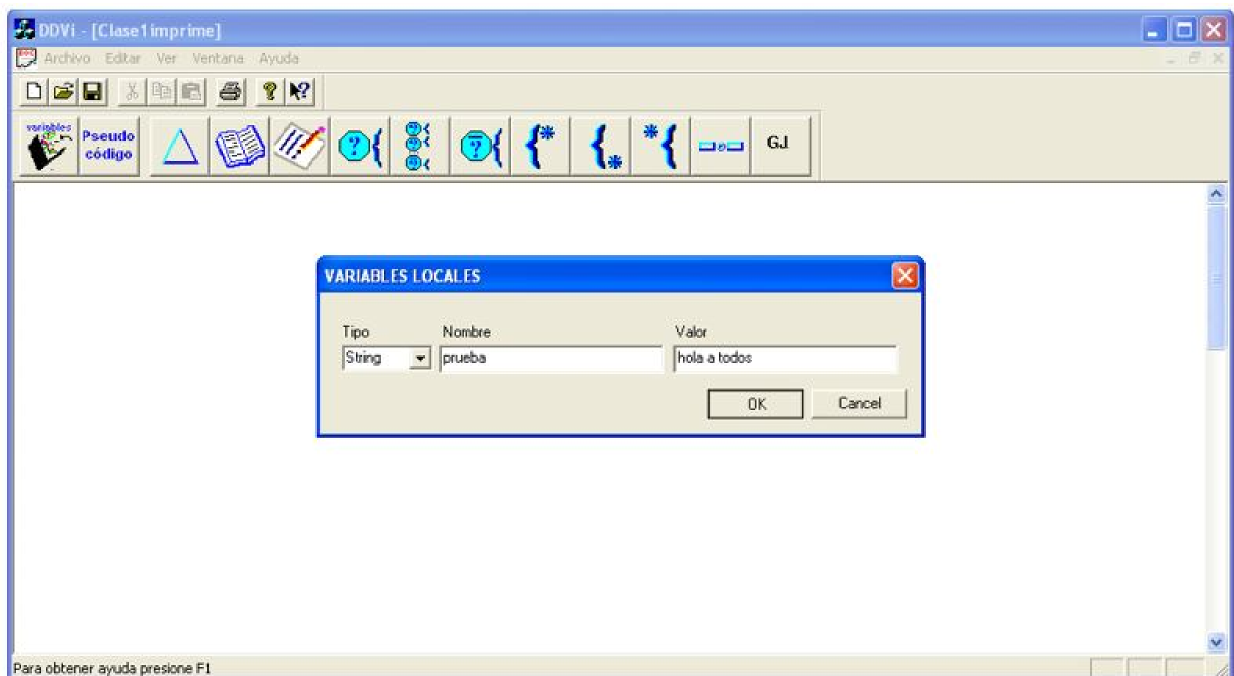
Aquí se muestra la opción que el usuario debe seleccionar para que pueda definir variables.



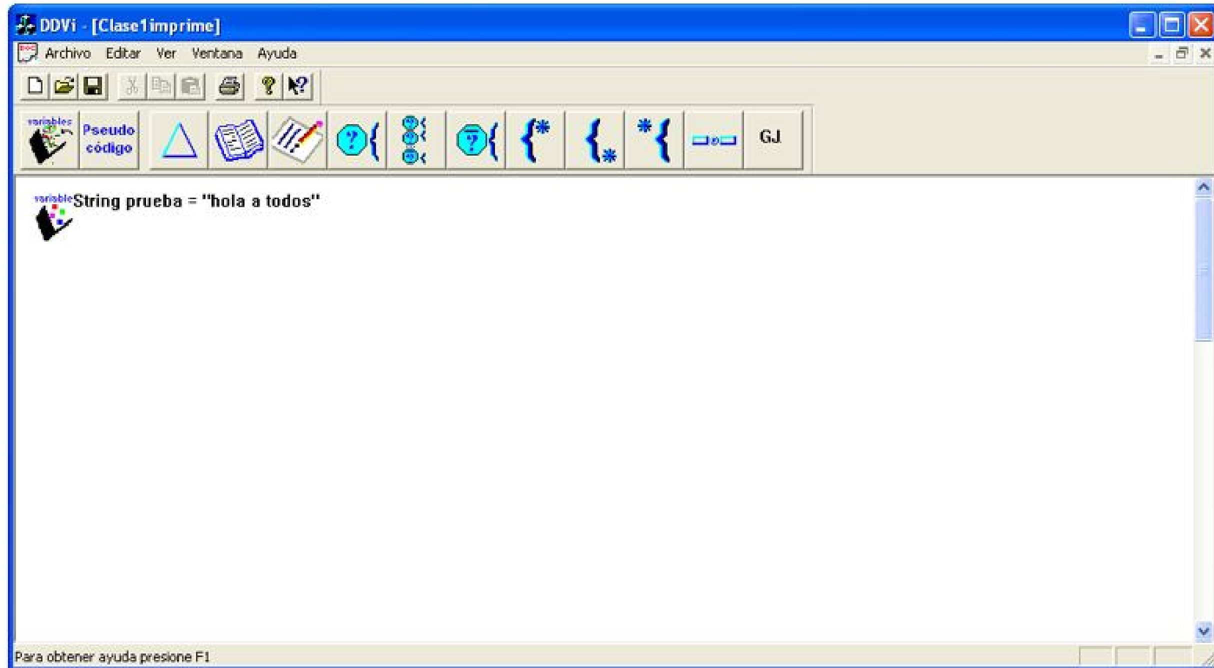
El sistema muestra una ventana emergente en la cual se puede declarar el tipo de variable, nombre y el valor.



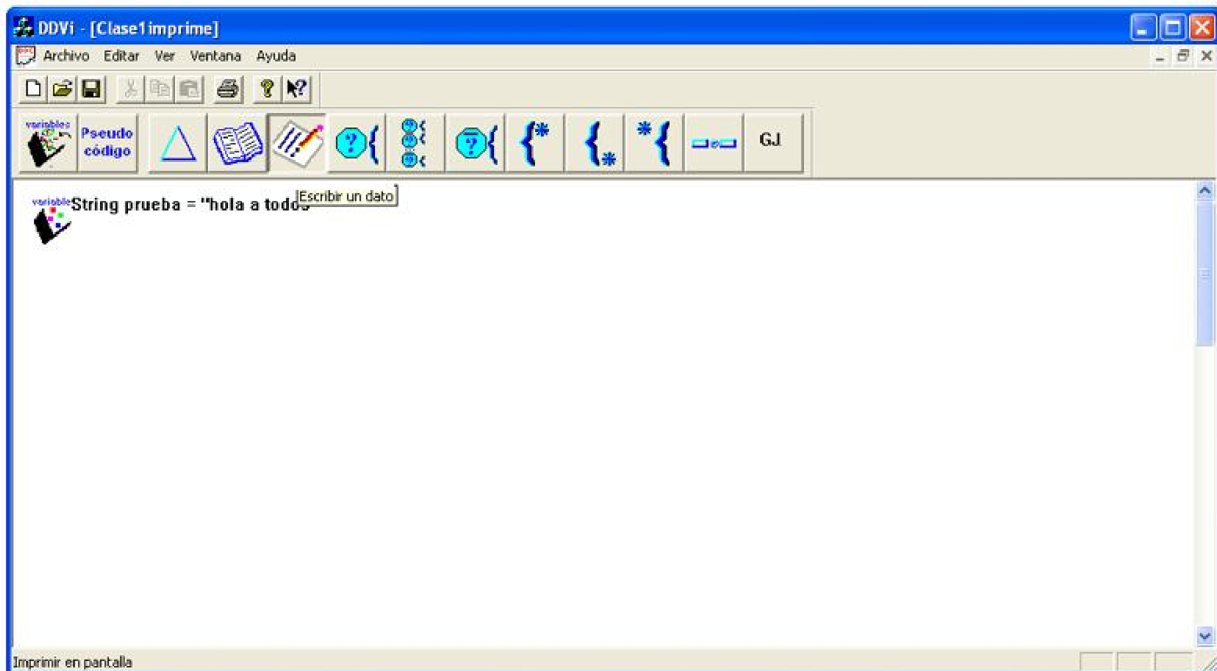
Aquí se ilustra un ejemplo de una variable local de tipo cadena, con nombre prueba y con el valor = “hola a todos”.



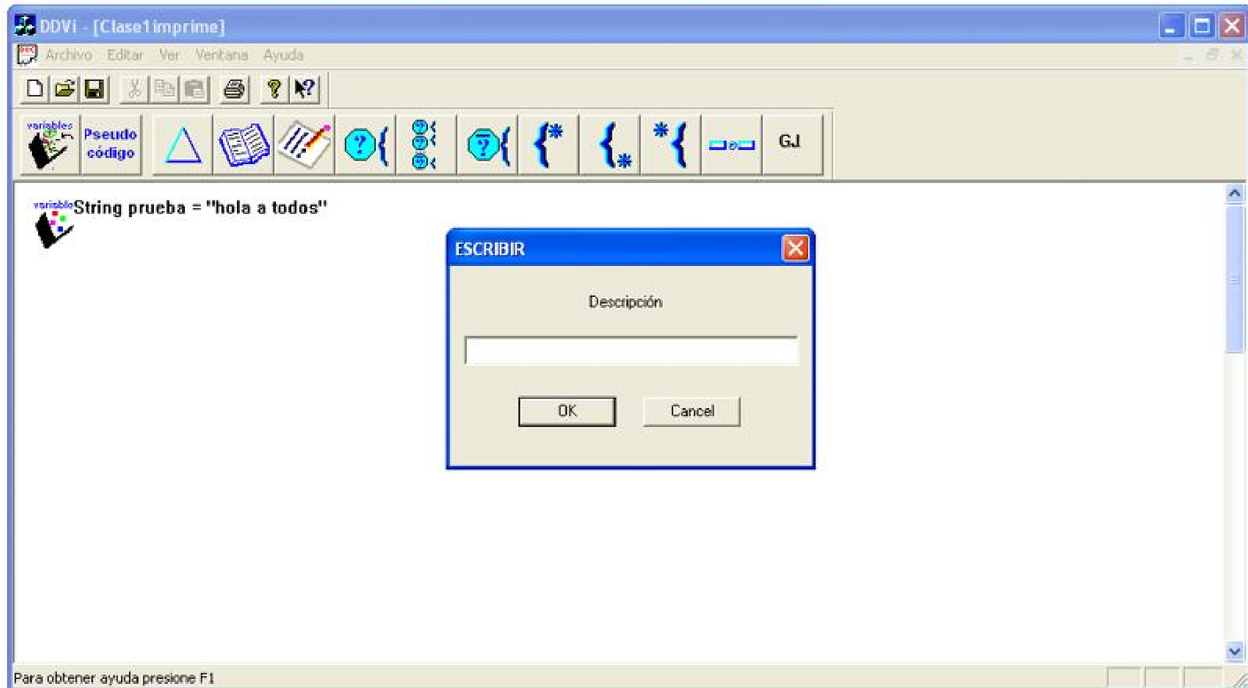
A continuación se puede observar que en la pantalla se ve la variable mediante la notación Warnier, como se puede ver, no existe simbología para poder ilustrar a las variables, sin embargo, se muestran como sigue en la siguiente imagen:



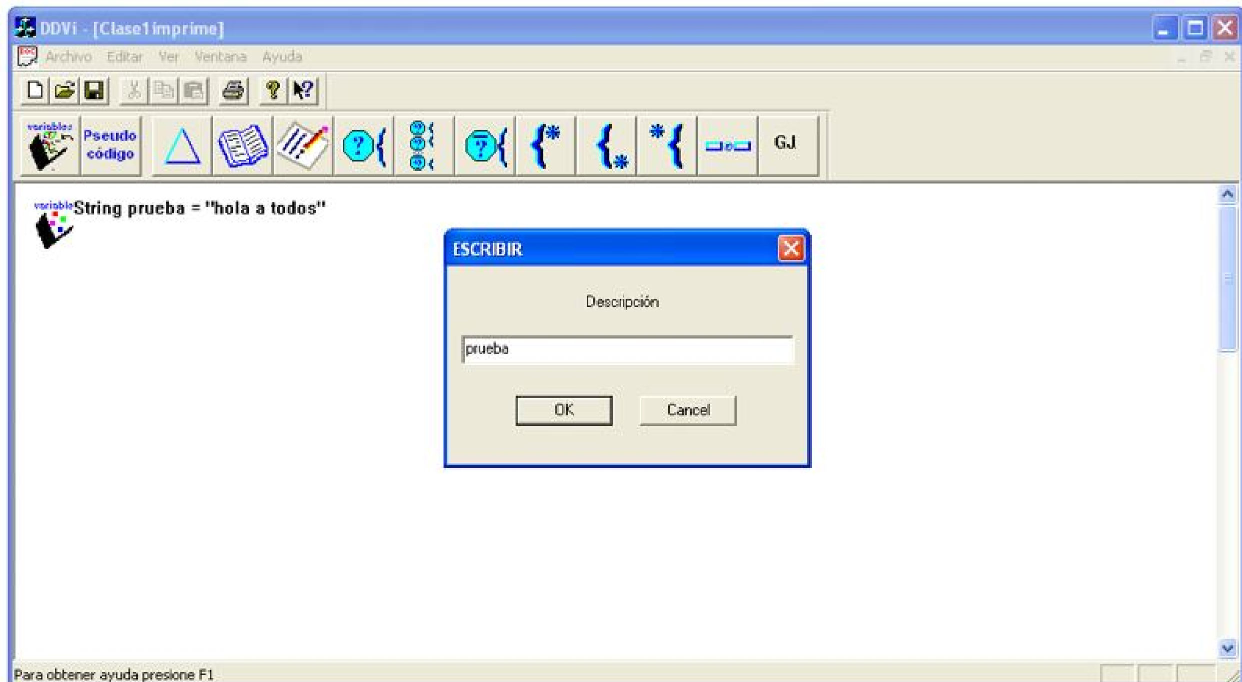
Una vez declarada una variable, se pueden introducir datos, seleccionando una opción como se ilustra en la siguiente imagen:



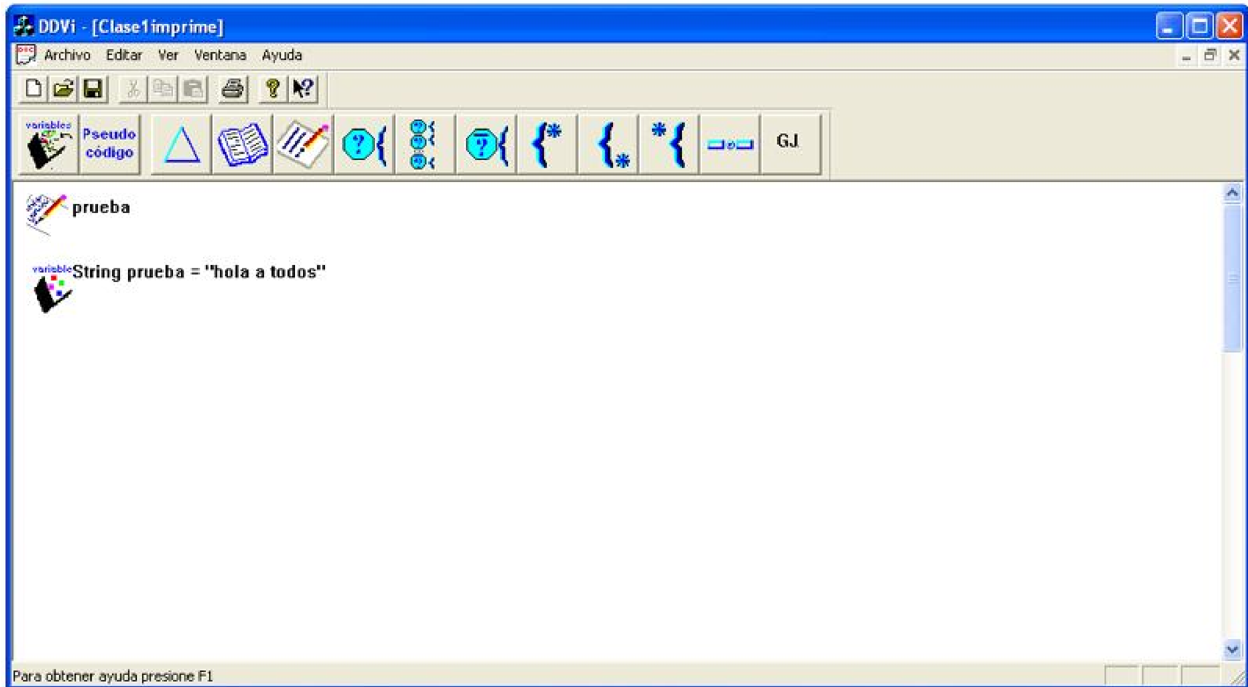
Para comenzar a escribir un dato, el sistema presenta una ventana emergente como la que se presenta a continuación:



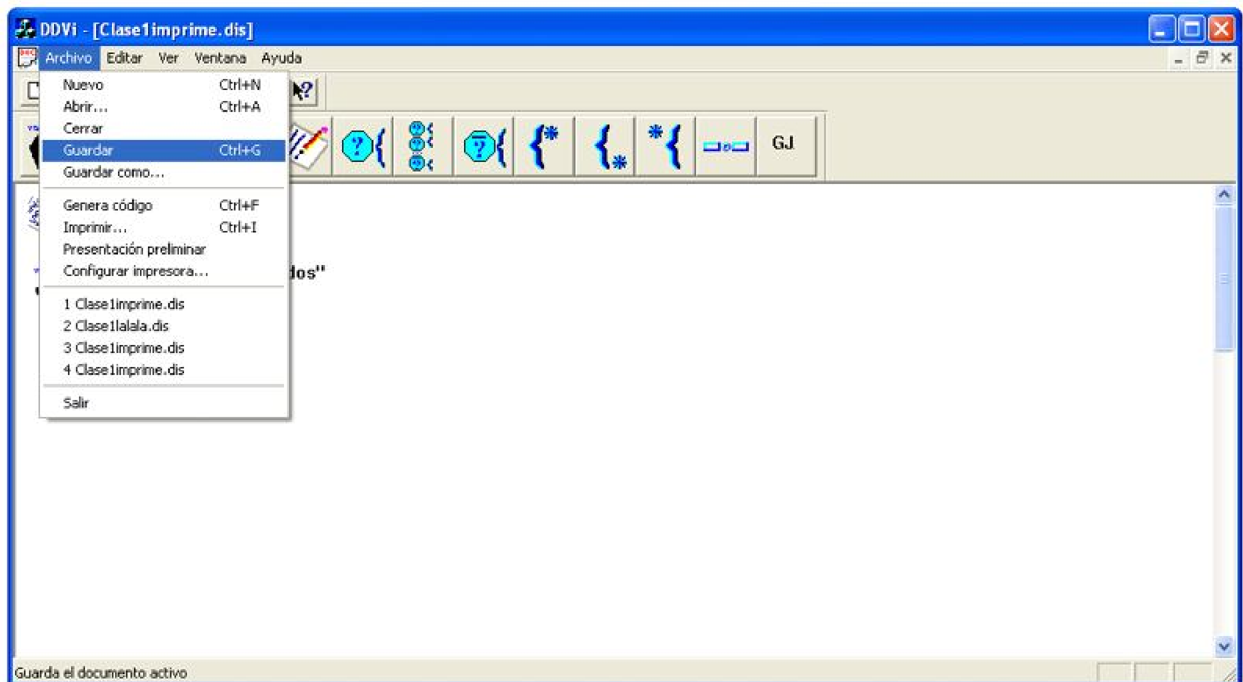
En este caso, se introduce una descripción llamada prueba

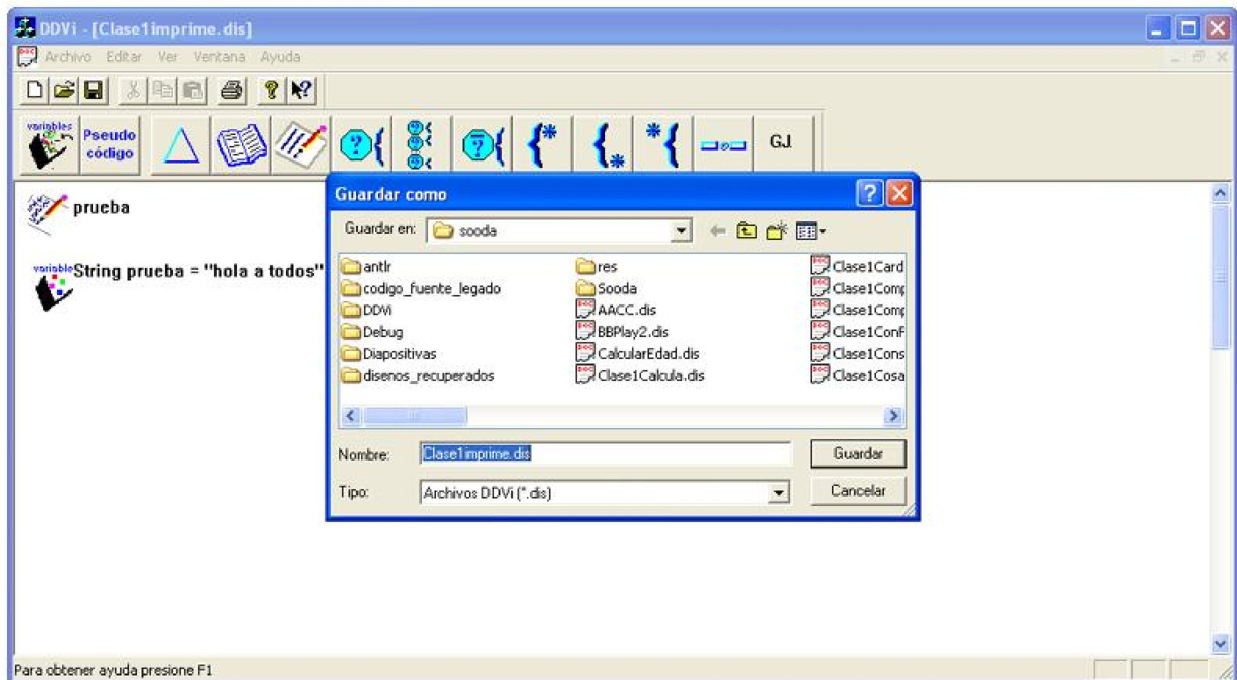


Ahora se muestra en pantalla como se muestran los datos y las variables agregadas



En la imagen se observa que el usuario debe guardar primero en un archivo el "diseño de los métodos" para que proceda a generar el código.





11. Conclusiones y recomendaciones

De la generación del código se puede concluir lo siguiente:

• Con DDVi usando los diagramas de Warnier se genera código en C++ dentro de los métodos de las clases.

• Con DDVi usando los diagramas de Warnier se puede representar de manera visual las estructuras de control que forman parte del método de la clase.

• Representando las estructuras de control con los diagramas de Warnier dentro de los métodos de las clases, se puede generar código en C++ a partir del modelo.

• Al generar código en C++, se crea un archivo texto con la extensión .cpp. La extensión de este archivo permite que los IDEs puedan interpretar el código en C++.

Los beneficios obtenidos del presente trabajo fueron:

1. Se ha demostrado que el código generado a partir de los diagramas de Warnier esta libre de errores de sintaxis, al igual que el código obtenido mediante los diagramas de clases.
2. El diseño de los métodos con diagramas de Warnier permite representar las estructuras básicas de secuenciación, repetición y selección en código C++.
3. Se obtuvo una forma de comparar que el código fuente de las estructuras básicas de secuenciación, repetición y selección generada contra el modelo propuesto es casi el mismo.
4. Las herramientas DDVi fueron un medio para automatizar el proceso de generación de código en C++ a partir de un modelo.

Durante el desarrollo de la herramienta se ha podido comprobar la importancia del papel de la ingeniería de software en el desarrollo de los sistemas de software, así como la necesidad para crear estándares que sirvan para el desarrollo de sistemas más eficientes, de la importancia de seguir un proceso formal utilizando una notación de Warnier. Este proyecto de residencia abre áreas de conocimiento para continuar con trabajos futuros que aporten los beneficios que ayudarán a complementar el perfeccionamiento de la herramienta. Dentro de los trabajos futuros se considera lo siguiente:

1. Que la herramienta pueda generar código en otros lenguajes como C#, o Delphi.
2. Que la herramienta sea compatible con otras herramientas CASE existentes.
3. Que la herramienta pueda crear paquetes agrupando las clases generadas.

12. Referencias Bibliográficas

Bernd Bruegge, *“Ingeniería de Software Orientado a Objetos”*, Prentice Hall, 1ª. Edición, 2002.

Chuck Spar, *“Aprenda Microsoft Visual C++ 6.0”*, Mc Graw Hill, 1ª. Edición, 1999.

Eric J. Braude, *“Ingeniería de Software Una perspectiva orientada a objetos”*, Alfaomega, 1ª. Edición, 2003.

F. Javier Ceballos, *“Microsoft Visual C++”*, Alfaomega Ra-Ma, 2ª. Edición, 2004.

Grady Booch, *“Analisis y Diseño Orientado a Objetos con Aplicaciones”*, Addison Wesley Longman, 2ª. Edición, 1996.

Herbert Schildt, *“Programación con MFC 6.0”*, Mc Graw Hill, 1ª. Edición, 1999.

Herbert Schildt, *“Java 2”*, Mc Graw Hill, 4ª. Edición, 2001.

Ian Sommerville, *“Ingeniería de software”,* Addison Wesley, 6ª. Edición, 2002.

I. A. Parra R., *“ModeladoVisual Orientado a Objetos”,* Tesis de Maestría, Centro Nacional de Investigación y Desarrollo Tecnológico *cenidet*, 2000.

Ivar Jacobson, *“El Proceso Unificado de Desarrollo de Software”*, Addison Wesley, 1ª. Edición, 2000.

James Rumbaugh, *“Modelado y diseño orientados a objetos”*, Prentice Hall, 3ª. Edición, 1999.

Presman S. Roger, *“Ingeniería del Software un enfoque práctico”*, Mc Graw Hill, 5ª. Edición, 2002.

S. Stelting, *“Patrones de diseño aplicados a Java”*, Prentice Hall, 1ª. Edición, 2003.

Rober C. Martin, *“UML para Programadores Java”*, Prentice Hall, 1ª. Edición, 2004.

R. Hernández S., *“Metodología de la Investigación”*, Mc Graw Hill, 3ª. Edición, 2003.

V. Aho Alfred, *“Compiladores principio, técnicas y herramientas”*, Addison Wesley Iberoamericana, 1ª. Edición, 1998.

BJORK, RUSSELL C. ATM Simulation. Gordon College. Copyright 2004, http://www.math-cs.gordon.edu/courses/cs211/ATMExample/ Consultado el 22 de septiembre del 2008.

Gustavo Torossi, *Modelado de Objetos con UML*. http://www.chaco.gov.ar/UTN/disenodesistemas/apuntes/oo/ApunteUML.pdf, 9 Enero 2008

Marc Gilbert G. *Ingeniería del Software en Entornos de SL* http://www.uoc.edu/masters/oficials/img/917.pdf, 9 Enero 2008

Eckel B. (2003). *Thinking in Java* (3.ª ed.). Upper Saddle River: Prentice Hall
http://www.mindview.net/Books/TIJ/, 6 Enero 2008

Grady Booch. (2003). *El Lenguaje Unificado de Modelado* http://elvex.ugr.es/decsai/java/pdf/3E-UML.pdf, 12 Octubre 2007

Object Management Group. http://www.omg.org/, 3 Diciembre 2007

Java Technology. http://java.sun.com. 18 Diciembre 2007

Microsoft.NET. http://www.microsoft.com/net/, 26 Septiembre 2007

Free UML Class Designer, http://nclass.sourceforge.net/, 1 Noviembre 2007

IBM, http://www.ibm.com/software/awdtools/developer/rose/*, 2 Noviembre 2007*
Wikipedia, http://en.wikipedia.org/wiki/Rational_Software, 2 Noviembre 2007

MagicDraw, http://www.magicdraw.com/, 2 Noviembre 2007

Plasticsoftware, http://plasticsoftware.com/, 3 Noviembre 2007

University of Paderborn, Software Engineering Group, http://www.fujaba.de/, 3 Noviembre 2007

Visual Paradigm, http://www.visual-paradigm.com/, 3 Noviembre 2007

13. Anexos

Cabe aclarar que la herramienta DDVi en la cual se basa este proyecto, es llamada por la herramienta SOODA para el diseño detallado de los métodos de las clases, en el siguiente plan de pruebas no se pretende poner a prueba la herramienta SOODA, sin embargo es necesario considerarla para demostrar el verdadero funcionamiento de la herramienta DDVi para este proyecto.

Plan de pruebas

Convención de nombres

El presente plan de pruebas se basa en el estándar IEEE Std 829-1998. A continuación se describen las claves para los documentos del plan de pruebas. El documento se identifica con la siguiente convención de nombres:

- Clave del sistema: SOODA corresponde al nombre del procedimiento para la generación automática de código en C++.
- Laboratorio: El laboratorio que se describe es el Laboratorio de Ingeniería de Software (LIS).
- Los documentos se identifican con una clave:
  - PP Plan de pruebas
  - CP Especificaciones de casos de prueba
  - EP Especificación de procedimientos de prueba
  - RI Reporte de incidentes de prueba
  - RR Reporte de resumen de prueba
- Número consecutivo: Es el número obtenido para cada prueba y documento.

Para describir el primer documento del plan de pruebas se utilizará el siguiente número que corresponde con la convención de nombres:

SOODA LIS PP 01

Elementos de pruebas

- Diagramas de Warnier.
- Código generado a partir de los diagramas de Warnier.
- Herramienta.

Características que serán probadas

Se probarán los diagramas de Warnier que representan el diseño detallado de los métodos de las clases con las siguientes características:

- Se probará que existan variables, estructuras de secuencia, condición y repetición dentro de los diagramas de Warnier.

Código generado:

- Se probará que el código generado del método tenga correspondencia con el diagrama de Warnier.
- Se probará que los códigos generados de los métodos estén libres de errores de sintaxis.

Características que no se probarán

Diagramas de Warnier:

- No se probará que los diagramas Warnier estén semánticamente correctos.

Código generado:

- Probar la lógica del código.
- Probar la compilación del código.

Enfoque

El procedimiento para la generación automática de código en C++ a partir de un modelo con diagramas de clases y de los diseños de los métodos con diagramas de Warnier, serán probados comparando los códigos de un modelo existente contra el código generado por las herramientas SOODA y DDVi.

Criterios de éxito o fracaso

Los criterios de éxito

El éxito es generar código en lenguaje C++ del diseño detallado de los métodos a partir de diagramas de Warnier que estén libres de errores y tengan correspondencia con sus respectivos modelos.

Los criterios de fracaso

Se consideraron fracasos los archivos de código que tenían errores de sintaxis y que no tenían similitud con los modelos creados con diagramas de Warnier.

Criterios de la suspensión y requisitos para la reanudación

Detectar errores de sintaxis en la generación de código en lenguaje C++.

Entregables de pruebas

Los documentos entregables son los siguientes

a) Plan de pruebas
b) Especificaciones de casos de pruebas
c) Especificación de procedimientos de pruebas
d) Reporte de incidentes de prueba
e) Reportes de resumen de prueba

Tareas de pruebas

1. Seleccionar el caso de estudio: el sistema simulador del cajero automático.
2. Seleccionar los casos de prueba.
3. Realizar el procedimiento de prueba
4. Comparar resultados
5. Obtener conclusiones.

Necesidades del entorno

Los sistemas elaborados en Visual C++ trabajan en el Sistema Operativo Windows. RAM 256 como mínimo. El sistema está instalado en un equipo marca Toshiba con las siguientes características:

- Intel® Pentium(R) 4 CPU 3.00 GHz
- Velocidad de bus 800 MHz
- Memoria RAM: 1024 MB
- Disco duro: 74.53 GB

Riesgos y contingencias

Riesgos:

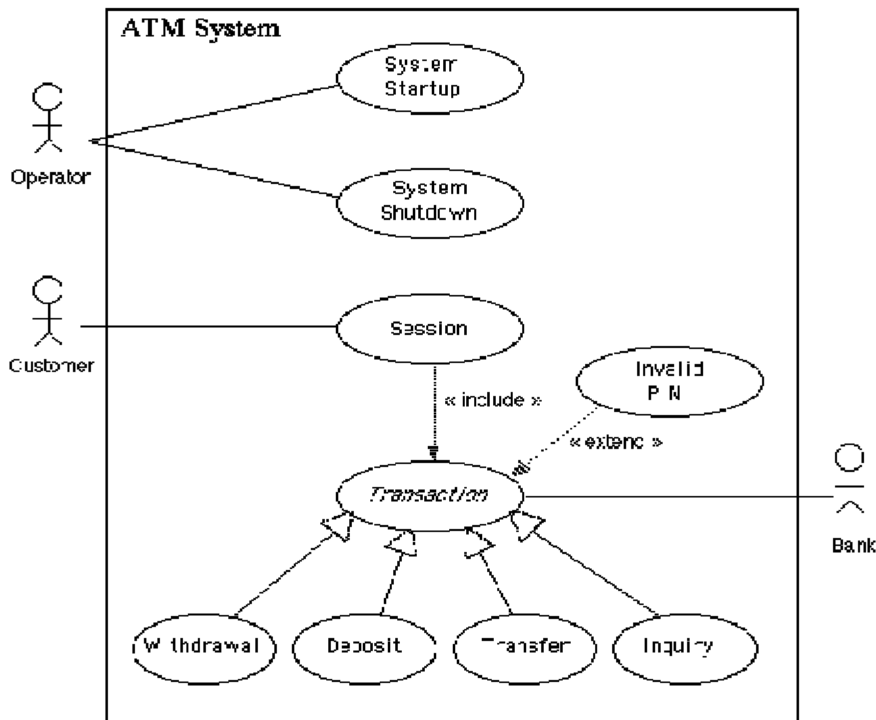
- Entrega retrasada del sistema.
- Que los diagramas de actividad del caso de prueba estén mal elaborados.

Aprobaciones

Para implementar el presente plan de pruebas, es necesario que tenga la aprobación del asesor del proyecto Ing. José Alberto Morales Mancilla

Especificaciones de casos de prueba (CP)

Se seleccionó como caso de estudio un sistema simulador de cajero automático (ATM, Automated Teller machine) del departamento de matemáticas y ciencias de la computación por el profesor Rusell C. Bjork en la materia de desarrollo orientado a objetos. Este sistema proporciona funcionalidades como las mostradas en la siguiente figura y consisten en iniciar sistema (System startup), cerrar sistema (System Shutdown), iniciar sesión (Session), realizar transacciones (Transaction), retirar (Withdrwall), depositar (Deposit), transferir (Transfer) y consultar (Inquiry).



Casos de uso del caso de estudio: Cajero automático

El sistema completo esta desarrollado en Java y para el lenguaje C++ estándar no está completo. El código posee documentación UML de los diagramas de clases, interacción, estados, entre otros.

Los diagramas de clases sirven para representar los métodos de las clases. Los diagramas de clases del cajero permiten conocer la estructura del sistema.

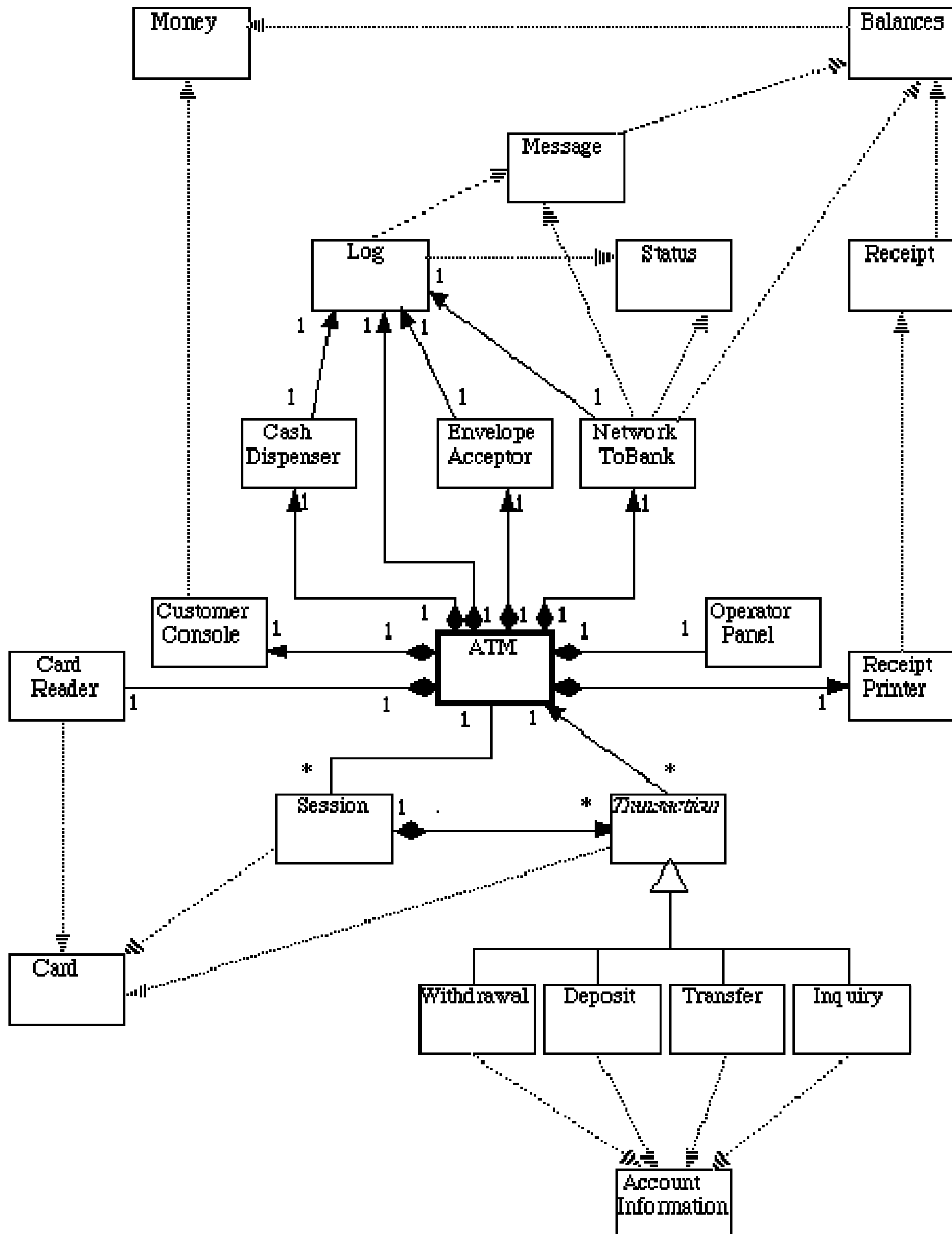


Diagrama de clases del cajero automático

La tabla describe los casos de prueba que fueron seleccionados de las clases del cajero automático. Las características corresponden al tipo de flujo y el tipo de estructuras que serán probadas por método.

Caso de prueba	Clase	método	Características
CP1	Card	Card()	Flujo secuencial y retorno de valores.
		getNumber()	Flujo secuencial sin retorno de valores.
CP2	Status	toString()	Contiene condicionales anidadas.
CP3	ReceiptPrinter	PrintReceipt()	Contiene estructuras de repetición.

Casos de prueba definidos para la herramienta SOODA y DDVi.

Especificación de procedimientos de prueba (EP)

Los pasos a seguir en el procedimiento de prueba son los siguientes:

1. Definir los casos de prueba que contengan estructuras básicas de secuenciación, repetición y selección para ser modeladas con diagramas de clases. En SOODA-LIS-CP-00 se definieron los casos de prueba CP1, CP2, CP3.
2. Obtener el código original en C++ de los casos de prueba que se van a modelar.
3. Generar el diagrama de Warnier que corresponde al diseño detallado de los métodos de los casos de prueba con la herramienta DDVi.
4. Generar el código completo en C++ (clases y métodos) con la herramienta SOODA y DDVi.

5. Evaluar los resultados obtenidos del código generado en C++ a partir de los modelos con diagramas de clases y diagramas de Warnier.
6. Revisar la sintaxis.

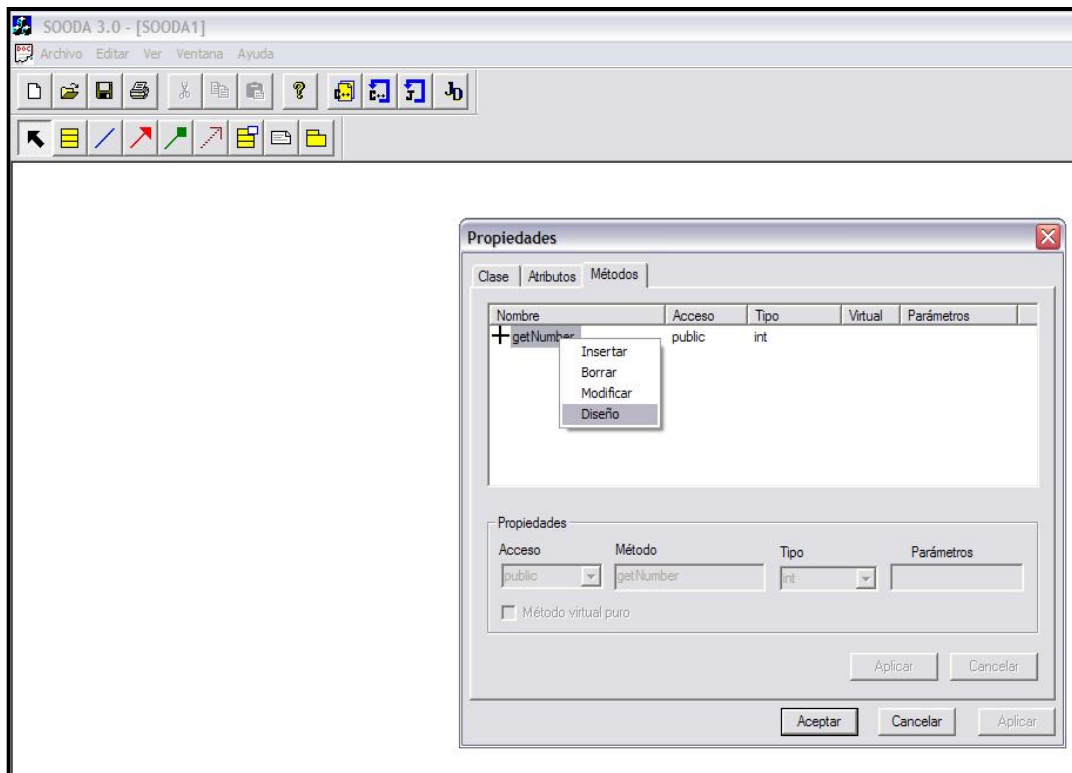
EP01. Clase Card: Métodos Card() y getNumber()

Paso 1. El paso uno del procedimiento de prueba fue realizado en la tabla.

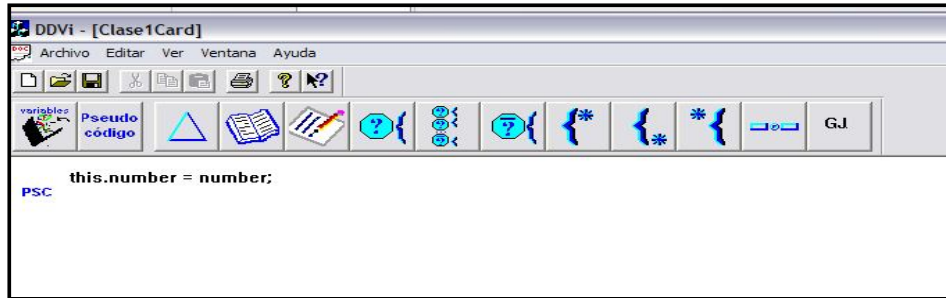
Paso 2. Utilizar el código original de la clase Card se encuentra a continuación.

Paso 3 y 4. Modelado y generación de código con SOODA y DDVi de la clase Card con sus métodos Card() y getNumber()

Una vez que se ha capturado el modelo con el diagrama de clases con SOODA, se procede a seleccionar el método para iniciar con el diseño detallado de los métodos getNumber() y Card(). Haciendo clic en la opción diseño se llama, se observa la interfaz de usuario para el diseño del método getNumber() con DDVi.

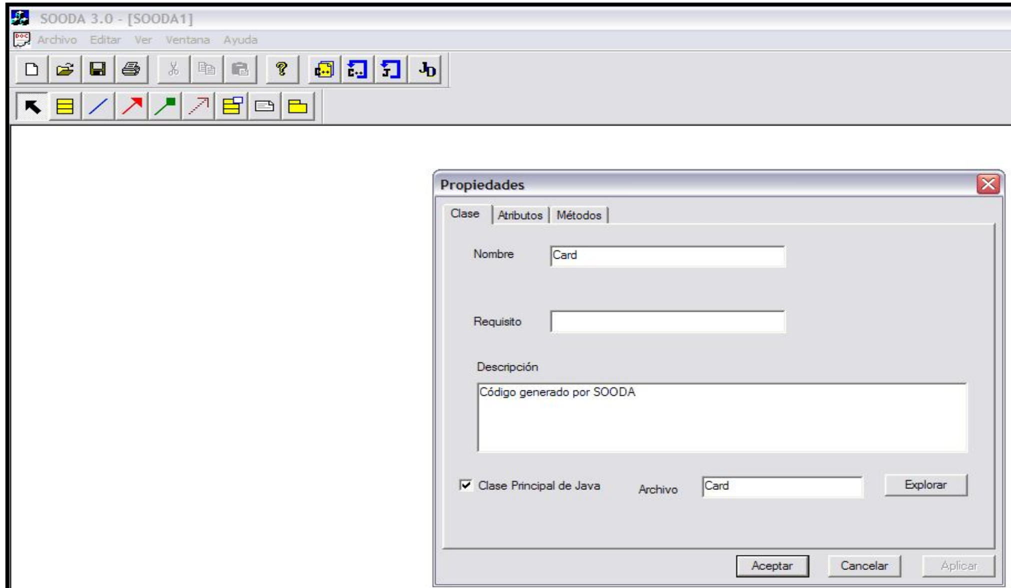


Cuadro de diálogo para llamar a DDVi.



Diseño del método getNumber() con DDVi

Paso 5. Desde la herramienta SOODA se puede crear la clase principal para realizar la ejecución del código



Cuadro de diálogo para la creación de la clase principal

Se ejecutó la compilación del archivo para mostrar los errores de sintaxis y se puede observar el resultado. Se observa que en el proceso de compilación no se detectaron errores de sintaxis.

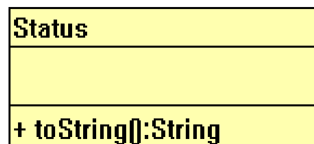
Prueba Manual del código generado

1. Se crea un objeto usando la referencia ob y se le envia al constructor del objeto un valor 10.
2. Se ejecutó el constructor Card(int number) que recibe un parámetro y ejecutó la instrucción this para inicializar el dato del objeto con un valor 10.
3. El atributo number es inicializado con un valor 10.
4. Se invoca al método get Number() para obtener el valor del atributo number.

Ejecución del programa: Se probó la ejecución del código para conocer paso a paso la ejecución del programa y poder realizar las pruebas y se agregó la impresión del mensaje en la ejecución de la instrucción

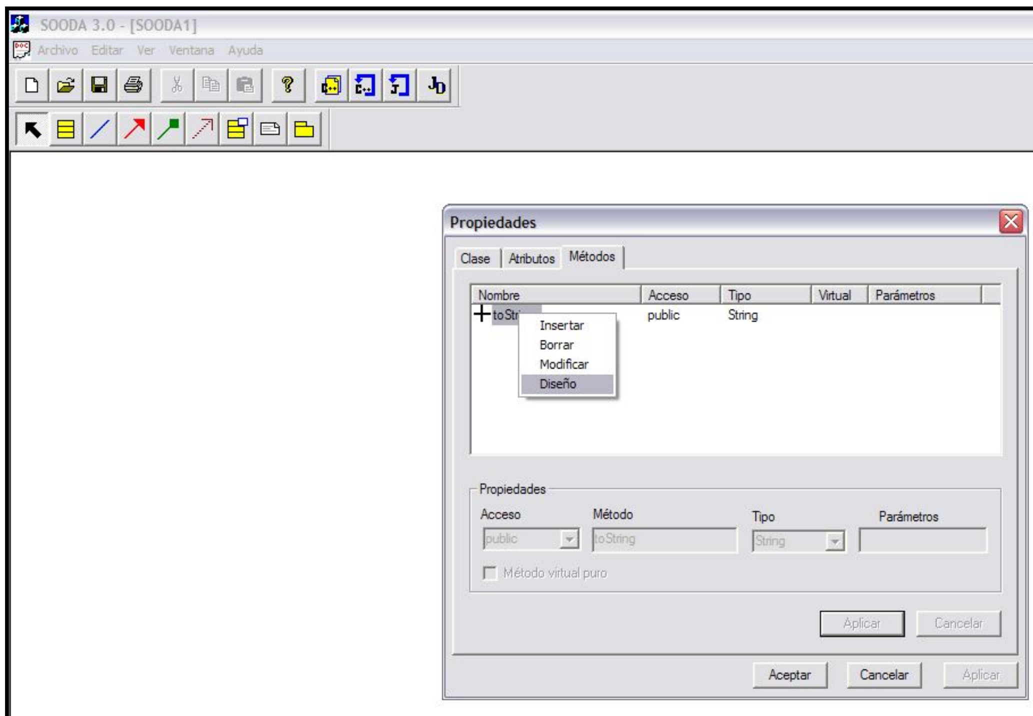
EP02. Clase Status: Método toString()

Paso 1. El paso uno del procedimiento de prueba fue realizado en la tabla.
Paso 2. Se trabaja con el código original de la clase Status
Paso 3 y 4. Modelado y generación de código con SOODA y DDVi del método toString() de la clase Status.

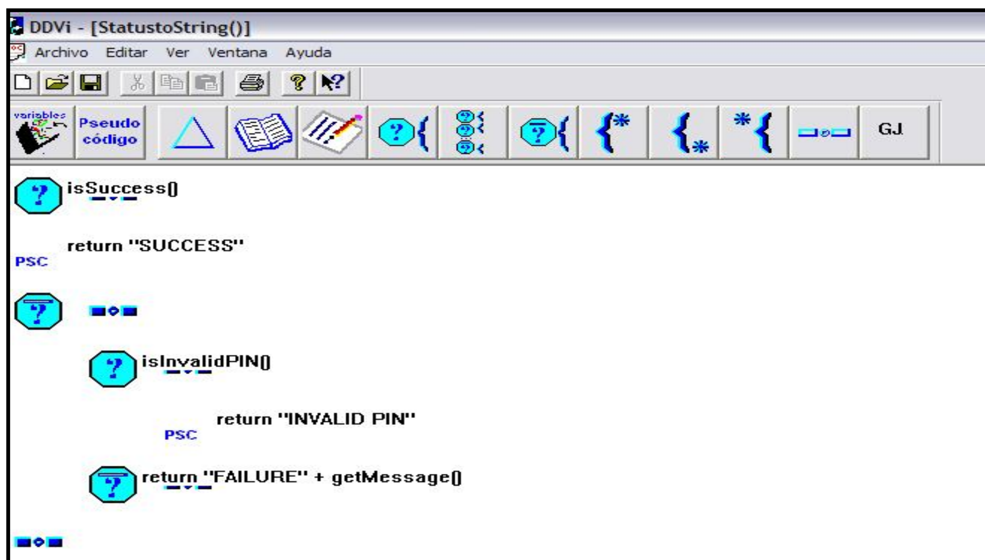


a. Diagrama de clase.

Una vez que se ha capturado el modelo con el diagrama de clases con SOODA, se procede a seleccionar el método para iniciar con el diseño detallado del método toString(). Haciendo clic en la opción diseño se llama a DDVi y se observa el diseño del método toString() con DDVi.



Cuadro de diálogo para llamar a DDVi.



Diseño del método toString() con DDVi

Se observa el código generado para el método toString() de la clase Status a partir de los modelos creados con SOODA y DDVi.

Paso 5. Desde la herramienta SOODA se puede crear la clase principal para realizar la ejecución del código. El propósito de estas pruebas no es compilar el código generado por SOODA y DDVi sino más bien mostrar que el código generado por SOODA y DDVi está libre de errores.

Reporte de incidentes de prueba (RI)

Se encontró que en el código generado por SOODA en la declaración de las clases definidas por el usuario no comienza con la palabra reservada public.

En el código generado por SOODA las clases no pueden ser declaradas con la palabra clave abstract.

En el código generado por SOODA los métodos no pueden ser declarados con la palabra clave abstract.

La herramienta IDE permite identificar errores de sintaxis al momento, es por ello que se utilizo esta herramienta para la revisión de sintaxis del código generado con SOODA y DDVi.

En el código del cajero automático no se encuentra disponible para descarga el código del paquete simulation.

Reportes de resumen de pruebas (RR)

En el modelado del método toString de la clase Status con DDVi, se encontró que no se pueden escribir sentencias else if. Sin embargo en el modelado de las

sentencias else if se puede adaptar con una sentencia if anidada dentro de una sentencia else.

Conclusiones del capítulo

En el caso de prueba CP1 el tipo de estructuras de control que se modeló es de tipo secuencial, por lo tanto la prueba resultó un éxito porque se creó el método main para comprobar la ejecución del programa.

El caso de prueba CP2 tiene estructuras de condición anidadas, las cuales fueron creadas con DDVi, los errores de sintaxis se deben a que en la herramienta SOODA no se pueden declarar clases y métodos abstractos, se concluye que la representación de las estructuras en el código fuente generado en C++ fue correcto.