





CINVESTAV-IPN

Biblioteca de Ingeniería Eléctrica



FB000014124

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA



CENTRO DE INVESTIGACIÓN Y DE
ESTUDIOS AVANZADOS DEL INSTITUTO
POLITÉCNICO NACIONAL

DEPARTAMENTO DE INGENIERÍA
ELÉCTRICA
SECCIÓN COMPUTACIÓN

Diseño e Implementación de un Sistema para
Edición de Flujogramas

Tesis que presenta

Roberto Tecla Parra

Para Obtener el Grado de

MAESTRO EN CIENCIAS

En la Especialidad de Ingeniería Eléctrica con Opción en
Computación

Director de tesis: Dr. Sergio V. Chapa Vergara

México, Distrito Federal

Abril, dos mil uno

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA



XM

01.10
01-1000
X
1000

RESUMEN

Un programa describe un algoritmo en una notación formal, un lenguaje de programación, para su ejecución. En la mayoría de los lenguajes de programación los programas son descritos por una cadena de caracteres básicamente unidimensional. Los lenguajes de programación visuales, en cambio, usan una notación principalmente gráfica para codificar un algoritmo.

La ventaja de usar un lenguaje visual es que el ser humano procesa las imágenes más fácil y rápido que los textos. EL texto tiene una naturaleza secuencial, mientras que las representaciones visuales proveen acceso aleatorio a cualquier parte de la imagen, así como vistas detalladas y generales.

Un programa visual, o flujograma, es una distribución espacial de iconos conectados entre si, donde los iconos representan operaciones sobre datos y las interconexiones el flujo de los mismos. El desarrollo de un programa visual requiere de un editor de flujogramas cuya operación determina la facilidad de la programación.

Esta tesis presenta un editor de flujogramas flexible que permite manipular fácilmente iconos y especificar operaciones entre ellos. Para probar nuestro editor, lo hemos integrado al Lenguaje Iconográfico para el desarrollo de Aplicaciones (LIDA) en el cual se desarrollan programas visuales que definen y realizan consultas a bases de datos. Con nuestro editor, al desarrollar programas LIDA es posible:

- Superponer una cuadrícula al lienzo para alinear los nodos del diagrama.

Agrupar varios nodos del diagrama para copiarlos, cortarlos, pegarlos, moverlos y guardarlos como si fueran un solo nodo. La operación de pegado se puede realizar también entre lienzos diferentes.

Aumentar el número de nodos en el diagrama mientras haya memoria disponible.

- Cambiar la imagen predeterminada por otra que sea más descriptiva de la información almacenada en una relación.

Usar pixmaps en lugar de dibujos vectoriales para disponer de un número mayor de fuentes de imágenes.

CENTRO DE INVESTIGACION Y DE
ESTUDIOS AVANZADOS DEL
I. P. N.
BIBLIOTECA
INGENIERIA ELECTRICA

ÍNDICE

INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN Y GENERACIÓN DE LENGUAJES VISUALES	6
1.1 Definición de Lenguajes visuales	6
1.1.1 Diccionario de Iconos	6
1.1.2 Diccionario de Operadores	7
1.1.3 Base de Datos de Acciones Semánticas	7
1.2 Generación Automática de Lenguajes Visuales.	8
1.2.1 Interprete de Iconos	9
1.3 Breve panorama de los Lenguajes Visuales	9
1.4 Lenguajes de Flujo de Datos	11
1.4.1 Extensiones al flujo de datos puro	11
CAPÍTULO 2: ENTORNOS DE PROGRAMACIÓN VISUAL	12
2.1 Lenguajes de Definición de Iconos	12
2.2 El lenguaje VODL	12
2.3 Definición de LIDA	15
2.3.1 Arquitectura de LIDA	16
2.3.2 LIDA como lenguaje visual	16
2.3.3 Interpretación Semántica	18
2.4 Diagrama de clases de un flujograma	19
CAPÍTULO 3: UML Y PATRONES DE DISEÑO	21
3.1 UML	21
3.2 Patones de diseño	23
3.2.1 Patones de diseño y frameworks	24
3.2.2 Patones de diseño en Smalltalk-MVC	24
CAPÍTULO 4: VISUALWORKS	30
4.1 Características de VisualWorks	30
4.2 Estructura de aplicaciones en VisualWorks	30
4.2.1 Modelos de Dominio	31
4.2.2 Modelos de Aplicación	31
4.3 Modelo de aspecto	32
4.4 Especificaciones de interfaz de usuario	32
4.5 Pintor y UIBuilder	32
4.6 Componentes visuales y vistas	33
4.7 Sublienzo	35
4.8 Control	35
4.8.1 El Flujo de Control	35
4.8.2 La secuencia de control	35

CAPÍTULO 5: INTERFAZ GRÁFICA DE LIDA	37
5.1 Paleta	38
5.2 Operación del Lienzo	40
5.3 Dibujo de líneas entre pseudoventanas	40
5.4 Menú del Lienzo	42
5.5 Operación de la herramienta del lienzo	43
5.6 Capturando información asociada con los elementos del diagrama	46
5.6.1 Relación	46
5.6.2 Proyección	47
5.6.3 Diferencia, Intersección, Junta Natural y Unión	48
5.6.4 Ordenación Ascendente y Descendente	48
5.6.5 Selección	49
5.6.6 Proceso	50
5.6.7 Función	51
CAPÍTULO 6: SEUDOVENTANAS	52
6.1 Características de nuestras pseudoventanas	53
6.2 Clases usadas en la implementación de las pseudoventanas	54
6.3 Estructura de las pseudoventanas en el lienzo	54
6.4 Estructura de la especificación de una pseudoventana	57
6.5 Iconización de pseudoventanas	58
CAPÍTULO 7: IMPLEMENTACIÓN DE NUESTRO EDITOR DE FLUJOGRAMAS	60
7.1 Pintor	61
7.2 PintorController	61
7.3 Clases de control	61
7.3.1 SelectModo	63
7.3.2 SelecciónDragModo	64
7.3.3 DragPlacementModo	65
7.4 Diseño de la Paleta	66
7.5 La secuencia de control en el editor de flujogramas	71
7.6 La secuencia de control en la respuesta de los controles a la entrada del usuario	71
CAPÍTULO 8: CONCLUSIONES Y TRABAJO FUTURO	73
8.1 Conclusiones	73
8.2 Trabajo futuro	73
BIBLIOGRAFÍA	76

INTRODUCCIÓN

Un programa describe un algoritmo en una notación formal, un lenguaje de programación, para su ejecución. En la mayoría de los lenguajes de programación los programas son descritos por una cadena de caracteres básicamente unidimensional. Los lenguajes de programación visuales, en cambio, usan una notación principalmente gráfica para codificar un algoritmo.

Un lenguaje visual esta compuesto por la terna Diccionario de Iconos (DI), Diccionario de Operadores (DO) y Base de Datos de Acciones Semánticas (BDAS). El Diccionario de Iconos es una base de datos que corresponde a un conjunto de iconos elementales. El Diccionario de Operadores es un conjunto de operadores genéricos que se aplican a iconos para crear iconos complejos, también llamados proposiciones visuales. La BDAS contiene la interpretación semántica que esta asociada a los elementos del Diccionario de Iconos.

Un sistema para desarrollar programas en un lenguaje visual consiste de un Diccionario de Iconos, un Diccionario de Operadores, una BDAS y conceptualmente dos editores. Uno de los editores se utiliza para definir y editar los iconos del lenguaje visual. Este proceso implícitamente crea y actualiza el Diccionario de Iconos, el Diccionario de Operadores, y la BDAS. La definición/ edición de iconos especifica/ modifica los atributos de los iconos tales como forma, color, tamaño, etcétera. La función del otro editor es editar los valores de los atributos de los iconos, y manipular los iconos disponibles para especificar un programa visual. A este editor nos referiremos como editor de flujogramas, o distribuciones espaciales de iconos interconectados, donde los iconos representan operaciones sobre datos y las interconexiones el flujo de los mismos.

La flexibilidad de un editor de flujogramas es un factor determinante para facilitar la programación visual. La falta de flexibilidad de algunos sistemas se debe a que tienen las limitaciones siguientes:

El cursor de edición solo se puede mover un número fijo de unidades cada vez. No se puede elegir un icono y colocarlo donde se desea o moverlo con solo arrastrar y soltar. El número de nodos en el diagrama no puede sobrepasar un máximo aun cuando haya todavía memoria disponible en el sistema. Los nodos del diagrama no se pueden agrupar, para realizar operaciones en conjunto, ni alinear unos respecto de otros.

En esta tesis se presenta un editor de flujogramas flexible que permite:

Mover el cursor de edición un número arbitrario de unidades. Colocar por primera vez o reubicar iconos en el lienzo con solo arrastrar y soltar. Aumentar el número de nodos en el diagrama mientras haya todavía memoria disponible en el sistema. Superponer una cuadrícula al lienzo para alinear los nodos del diagrama. Agrupar varios nodos del diagrama para copiarlos, cortarlos, pegarlos, moverlos y guardarlos como si fueran un solo nodo.

El diseño de nuestro editor de flujogramas está especificado en UML (Unified Modeling Language) y patrones de diseño ya que el UML es un estándar de facto de la industria para construir modelos orientados a objetos y los patrones de diseño sirven para guiar y documentar el diseño de sistemas de software orientado a objetos. Para su implementación se utilizó la herramienta de desarrollo VisualWorks de ParcPlace Systems. VisualWorks es un ambiente totalmente orientado a objetos para construir aplicaciones, utilizando el lenguaje SmallTalk de ParcPlace.

La arquitectura utilizada por las aplicaciones en el lenguaje Smalltalk es la arquitectura Modelo-Vista-Controlador (MVC). El modelo es un objeto aplicación, la vista es la presentación en pantalla, y el controlador define cómo la IGU reacciona a la entrada de usuario.

La arquitectura MVC hace uso de varios patrones de diseño. Entre los más importantes para entender MVC están: Observador, Compuesto y Estrategia. MVC separa vistas y modelos estableciendo un protocolo de suscripción /notificación entre ellos. Una vista debe asegurarse de que su apariencia refleja el estado del modelo. Siempre que los datos del modelo cambian, el modelo notifica a las vistas que dependen de este. En respuesta, cada vista tiene una oportunidad para actualizarse ella misma. El patrón de diseño observador describe el diseño más general donde: se separan objetos para que los cambios que afectan a uno puedan afectar cualquier número de otros sin requerir que el objeto cambiado conozca detalles de los otros. Otra característica de MVC es que las vistas se pueden anidar. El patrón de diseño Compuesto permite crear una jerarquía de clases en la cual algunas subclases definen objetos compuestos que ensamblan los primitivos en objetos más complejos. MVC permite cambiar la forma como una vista responde a la entrada del usuario sin cambiar su presentación visual. MVC encapsula el mecanismo de respuesta en un objeto controlador. Una vista usa una instancia de una subclase controlador para implementar una estrategia de respuesta en particular; para implementar una estrategia diferente, simplemente se reemplaza la instancia con otra instancia de una subclase de controlador diferente. La relación Vista-Controlador es un ejemplo del patrón de diseño estrategia [9].

Nuestro editor consta de los módulos siguientes: la paleta, el lienzo, los controladores y el módulo de captura de los valores de los atributos de los iconos ver figura 1.

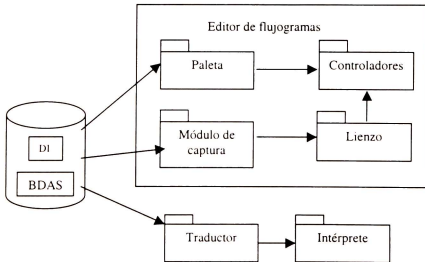


Fig. 1 Arquitectura del Editor de flujogramas

La paleta es donde se eligen los iconos que aparecerán en el diagrama que constituye un programa en un Lenguaje de Flujo de Datos (LFD). Uno de tales programas se conoce como flujograma. Estos flujogramas se editan y ejecutan en el lienzo. El módulo de captura sirve para establecer los valores de los atributos de la parte lógica del icono que este seleccionado en el flujograma editado en el lienzo.

Cada controlador en el módulo de controladores responde de diferente forma a la entrada del usuario. La paleta modifica el comportamiento del lienzo cuando cambia el controlador del lienzo.

Para probar la flexibilidad de nuestro editor, lo hemos integrado al Lenguaje Iconográfico para el desarrollo de Aplicaciones (LIDA) en el cual se desarrollan programas visuales que definen y realizan consultas a bases de datos.

LIDA cuenta con un lenguaje visual basado en el álgebra relacional donde cada operador, relación, y función agregada se representan mediante iconos. Y para realizar una consulta se construye un flujograma. Por ejemplo dados los esquemas de relación siguientes:

Proveedor(numS, Snom, edo, ciudad)

Orden(numS, numP, cantidad)

Y la consulta

Obtenga los nombres de aquellos proveedores que suministran la parte P2.

Es posible resolver dicha consulta utilizando el flujograma de la figura 2.

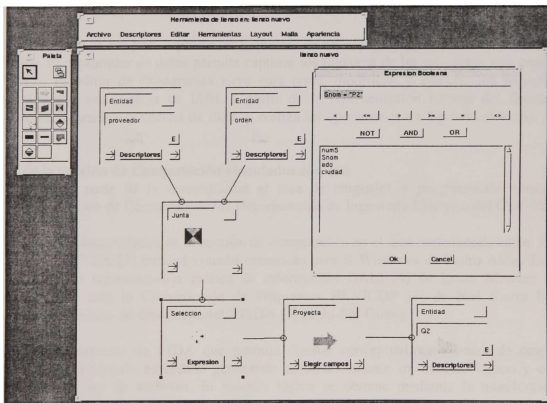


Fig. 2 Ejemplo de una consulta en LIDA utilizando nuestro editor.

En un flujograma las relaciones fluyen a través de las líneas de flujo para entrar a módulos de transformación que son los iconos. Este modelo de flujo de datos da a LIDA un enfoque de abstracción procedural debido a que el usuario solo trata con iconos que representan transformaciones de relaciones.

LIDA se compone de los siguientes módulos: editor de flujogramas, descriptor, capturador, traductor, intérprete de ISBL y álgebra relacional.

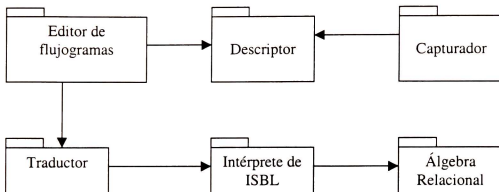


Fig. 6 Diagrama de componentes de LIDA

El módulo descriptor se usa para capturar los esquemas de relación. Además se utiliza para acceder a la base de datos. El capturador de datos permite capturar los registros de las relaciones que conforman la base de datos. El editor de flujogramas sirve para crear, modificar, y ejecutar los flujogramas. El traductor construye una cadena de ISBL a partir de la representación interna del flujograma. El intérprete de ISBL analiza la cadena de ISBL y realiza las operaciones de álgebra relacional indicadas por esta.

Proyectos de la Sección de Computación vinculados con este

Este trabajo forma parte de la investigación el área de lenguajes y programación visual que se desarrolla en la Sección de Computación del Departamento de Ingeniería Eléctrica del CINVESTAV.

Entre los proyectos desarrollados en la sección de computación en el área mencionada están: Diseño de base de datos con EVEX [3] entidad vinculo extendido para X-Windows de Pedro Alday Echavarría, Herramienta para la representación gráfica de información DALI [4] de Laura Mendez Segundo, Herramienta Visual para la Construcción de Programas HEVICOP [2] de Noé Sierra Romero y Visualización de una base de datos espacial VISDA [5] de Ju-Shi Guang.

EVEX es una herramienta de LIDA que permite diseñar conceptualmente bases de datos con la metodología entidad vinculo extendido generando automáticamente un modelo lógico y uno físico basado en el descriptor de archivos. El modelo lógico se obtiene mediante la transformación del diagrama eve a esquemas de relación y la verificación como esquemas en tercera forma normal.

DALI es otra herramienta de LIDA que permite visualizar la información contenida en una base de datos relacional ordenando y clasificando los datos para interpretarlos y comunicarlos al usuario de una manera gráfica fácil de comprender. DALI complementa a LIDA pues permite al usuario visualizar el comportamiento de los datos, que resultan de las operaciones del álgebra relacional, en el punto del flujograma que desee.

Hevicop surge de una propuesta de abstraer LIDA omitiendo las líneas de flujo de datos y mediante el ensamblaje de un conjunto de iconos elementales, tener procesos más complejos. Dichos iconos que son piezas de software y a partir de esta definición, son capaces de generar código en lenguaje C. La creación de un programa en Hevicop esta basado en el paradigma BLOX análogo a un juego de rompecabezas donde las piezas del mismo son segmentos del programa que se ensamblan para construir una estructura de control, siendo susceptibles de expansión en una nueva gráfica algunos de sus módulos de manera recursiva.

VISDA cuenta con un lenguaje visual de consulta para el procesamiento de información geográfica que en particular se aplica a la exploración petrolera y un gestor de base de datos espaciales el cual incluye un modelo lógico relacional de datos y un modelo físico de datos basado en arboles B y multilistas.

Estructura de la tesis

En el capítulo 1 se presentan los conceptos esenciales y las categorías de los lenguajes visuales, y los lenguajes de flujo de datos. Además se describe la generación automática de los lenguajes visuales.

En el capítulo 2 se presentan los Lenguajes de Definición de Iconos y se da un ejemplo de dichos lenguajes. Se describe la arquitectura de LIDA y su editor de flujogramas dentro del marco de los lenguajes visuales. También se explica como se realiza la interpretación semántica de los iconos y se muestra el diagrama de clases de los flujogramas.

En el capítulo 3 Se tratan el lenguaje unificado de modelado (UML, Unified Modeling Language) y los patrones de diseño que constituyen la metodología usada para describir el sistema. En particular se abordan los patrones de diseño en SmallTalk que son utilizados para la documentación del sistema.

En el capítulo 4 se describe el framework de desarrollo de aplicaciones de VisualWorks usado para implementar LIDA.

En el capítulo 5 se describe la interfaz gráfica de LIDA basada en nuestro editor. Se explica el funcionamiento de la paleta, el lienzo, el menú del lienzo, la herramienta del lienzo, la ventana de captura para los datos y la ventana de captura para los descriptores de relaciones. Además se muestra como se usa la Interfaz Gráfica de Usuario de las pseudoventanas para capturar los atributos lógicos de los iconos.

En el capítulo 6 se presenta la parte del diseño y de la implementación, en VisualWorks, de nuestro editor de flujogramas que tiene que ver con la captura de los valores de los atributos iconos.

En el capítulo 7 se presentan los diagramas de clases y los diagramas de estado de las clases más importantes para la implementación del editor de flujogramas.

En el capítulo 8 se presentan las conclusiones y se plantea la extensión del sistema con un editor de iconos y la modificación de nuestro editor gráfico para que incluya hojas de propiedades en lugar de pseudoventanas.

CAPITULO 1

DEFINICION Y GENERACION DE LENGUAJES VISUALES

Un programa describe un algoritmo en una notación formal, un lenguaje de programación, para su ejecución. En la mayoría de los lenguajes de programación los programas son descritos por una cadena de caracteres básicamente unidimensional. Los lenguajes de programación visuales, en cambio, usan una notación principalmente gráfica para codificar un algoritmo.

La ventaja de usar un lenguaje visual es que el ser humano procesa las imágenes más fácil y rápido que los textos. Una razón de esto es que el sistema sensorial humano puede procesar imágenes en tiempo real. El uso de una notación visual permite mover parte de la carga cognitiva del usuario del nivel consciente al de la corteza visual. EL texto tiene una naturaleza secuencial, mientras que las representaciones visuales proveen acceso aleatorio a cualquier parte de la imagen, así como vistas detalladas y generales.

Las representaciones visuales también proveen un lenguaje sintácticamente rico. Un texto es esencialmente unidimensional, mientras que las representaciones visuales pueden emplear dos o tres dimensiones espaciales así como otras propiedades como color y con la posibilidad de representar este tipo de visualizaciones en forma estática y dinámica.

La Programación Visual es cualquier sistema que permite al usuario especificar un programa de dos o tres dimensiones. Un Lenguaje Visual manipula información visual o soporta interacción visual, o permite programación con expresiones visuales.

1.1 Definición de Lenguajes Visuales

Un lenguaje visual esta compuesto por la terna Diccionario de Iconos (DI), Diccionario de Operadores (DO) y Base de Datos de Acciones Semánticas (BDAS).

El DI es una base de datos que corresponde a un conjunto de iconos elementales.

El DO es un conjunto de operadores genéricos que se aplican a iconos para crear iconos complejos, también llamados proposiciones visuales.

BDAS contiene la interpretación semántica que esta asociada a los elementos del DI.

1.1.1 Diccionario de Iconos (DI)

Los iconos del Diccionario de Iconos son iconos generalizados. Un icono generalizado es un objeto con la representación dual de una parte lógica (el significado) y una parte física (la imagen).

Un icono generalizado se denota por x o (x_s, x_i) donde:

x_s es la parte lógica del icono o su significado

x_i es la parte física del icono o su imagen

Los iconos generalizados pueden ser elementales o complejos. Los Iconos Elementales son de la forma (x_s, x_i) y son los iconos base del lenguaje. Estos se dividen en iconos objeto e iconos proceso. Los iconos objeto identifican objetos mientras que los iconos proceso expresan cálculos. El significado del icono proceso depende de la oración visual específica en la cual aparece. Los Iconos Complejos se obtiene a partir de un grupo de iconos elementales de la forma (x_{sk}, x_{ik}) , el icono resultante tiene la estructura siguiente:

$$(\{ OP, x_{s1}, \dots, x_{sn} \}, \{ OP', x_{i1}, \dots, x_{in} \})$$

donde

el operador OP opera sobre las partes lógicas de los iconos para obtener el significado del icono complejo.

OP' opera sobre las partes físicas de los iconos obteniendo la imagen del icono complejo.

Semántica vs. representación gráfica de un icono

La parte lógica de un icono, representa la acción a realizar o la interpretación que se le dará al icono dentro de uno complejo. Se puede especificar esta parte mediante acciones semánticas.

La representación gráfica esta compuesta por distintos patrones primitivos, los cuales se pueden generar mediante varias técnicas tales como:

- Uso de mapas pixeles
- Vectoriales, incluyendo el uso de atributos de color
- Uso de gramáticas gráficas

1.1.2 Diccionario de Operadores (DO)

Una proposición visual es un arreglo espacial de iconos que describe una secuencia de operaciones. En el caso de los lenguajes visuales bidimensionales se tienen al menos tres reglas de construcción para ser usadas en el arreglo espacial de iconos, que corresponden a los operadores espaciales siguientes:

- & concatenación horizontal
- ^ concatenación vertical
- + superposición espacial

1.1.3 Base de Datos de Acciones Semánticas (BDAS)

La interpretación de un icono complejo se hace usando los elementos que integran la Base de Datos de Acciones Semánticas.

Cada icono en el sistema posee información sobre como debe interpretarse, cualquier elemento icónico formado por un conjunto de iconos elementales, también cuenta con esta información, la cual se obtiene de los iconos que la conforman. La manera en que debe combinarse es especificada por este componente del lenguaje visual.

1.2 Generación Automática de Lenguajes Visuales.

La demanda de lenguajes de propósito especial y sus ambientes de programación dio origen a la investigación sobre la generación de entornos específicos al lenguaje basados en especificaciones de lenguajes formales. La especificación de un lenguaje consiste en la especificación de su sintaxis, su semántica y otras características del lenguaje. Dicha investigación se realiza en lenguajes textuales y visuales. En la especificación de un lenguaje visual se usa un lenguaje de definición de iconos (LDI).

Un Generador de Lenguajes Visuales (GLV) es un sistema de software para la especificación, interpretación, prototipo, y generación de sistemas orientados a iconos que tiene las siguientes características:

Especificación sintáctica: Usa una gramática formal para especificar el lenguaje visual.

Usado interactivo: Acepta las especificaciones del usuario de manera interactiva.

Ajustable: El usuario puede modificar el Diccionario de Iconos (DI), el diccionario de operadores (DO), y la gramática G para expandir o especializar el dominio de las aplicaciones.

El diagrama de arquitectura del GLV se muestra en la figura 1.1. El sistema incluye los componentes siguientes: DI, DO, BDAS, AE, el intérprete de iconos y el inicializador.

El inicializador (ajustador), que ayuda al usuario en su caracterización de su propio lenguaje visual mediante la especificación de: DI, DO, G y BDAS.

Cuando la fase de inicialización ha pasado, el sistema está listo para aceptar y ejecutar oraciones visuales consultando la gramática G y las bases de datos previamente creadas ID y BDAS. El analizador de estructura y el intérprete de iconos son los únicos módulos activos durante la fase de ejecución.

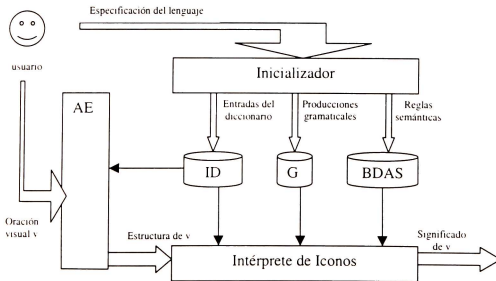


Fig. 1.1 Diagrama de un sistema GLV

1.2.1 Intérprete de Iconos

El diagrama de un intérprete de iconos se muestra en la figura 1.2; este incluye: DI, DO, BDAS y la gramática G que especifica como se pueden construir las oraciones visuales por medio de una distribución espacial de iconos. El analizador de estructura (AE) esta a cargo de transformar las oraciones visuales en arboles de estructura para permitir que el intérprete de iconos complete el análisis sintáctico de acuerdo a G, y realice la construcción del significado utilizando la BDAS.

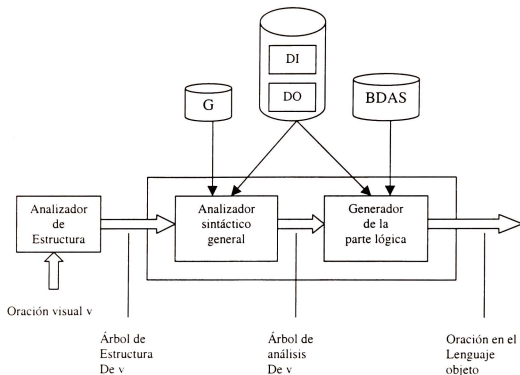


Fig. 1.2 Intérprete de Iconos

1.3 Breve panorama de los Lenguajes Visuales

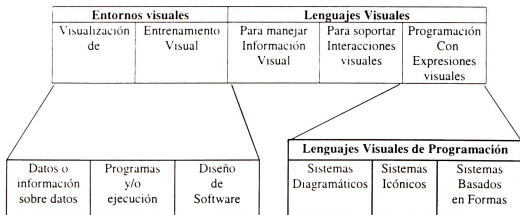


Fig. 1.3 Categorías de Lenguajes Visuales

1.3.1 Visualización

Visualización de datos o de información sobre datos

En esta clase de sistemas generalmente la información se almacena internamente en bases de datos tradicionales, pero el sistema la representa gráficamente y permite su análisis por parte del usuario.

Visualización de un Programa y/o su Ejecución

El objetivo de estos sistemas es usar un dispositivo gráfico de alta resolución para facilitar el desarrollo y prueba de programas que están escritos en lenguajes tradicionales de texto.

Visualización del Diseño de Software

Estos sistemas facilitan el desarrollo del software pues permiten visualizar los siguientes aspectos de dicho desarrollo: los requerimientos, las especificaciones, las decisiones de diseño y las estructuras del sistema.

1.3.2 Entrenamiento Visual

Son sistemas basados en programación gráfica que pretenden resolver problemas mediante una programación más rápida. Dentro del medio ambiente de estos sistemas el usuario no necesita visualizar mentalmente los efectos de sus instrucciones mientras construye un programa. En su lugar puede visualizar todas las acciones en la pantalla. La sintaxis del lenguaje en el sentido usual esta ausente desde el punto de vista del usuario.

1.3.3 Lenguajes para Manejar Información Visual

Estos sistemas surgen de la necesidad de tener lenguajes fáciles de usar para la manipulación y consulta de datos pictóricos. Generalmente el lenguaje permite referenciar directamente imágenes. Sin embargo, aunque los elementos que se manejan son gráficos el lenguaje en si es textual.

1.3.4 Lenguajes que soportan Interacción Visual

Estos lenguajes cuentan con representación e interacción visual, pero los lenguajes mismos son textuales, no visuales.

1.3.5 Lenguajes Visuales de Programación con Expresiones Visuales

La idea central de estos sistemas es permitir a los usuarios programar con expresiones visuales. Una expresión visual consiste en un arreglo espacial de iconos.

Sistemas Diagramáticos

Los diagramas que se usan en papel se incorporan en construcciones de programación como una extensión de los lenguajes de programación convencionales.

Sistemas Icónicos

El papel de los iconos esta limitado a la representación del ámbito de trabajo y a los comandos definidos dentro del mismo.

Sistemas Basados en Formas

Estos sistemas son fáciles de usar por usuarios novatos pues aprovechan la tendencia natural de la gente a usar estructuras regulares y/u organizar los datos en tablas.

1.4 Lenguajes de Flujo de Datos (LFD)

Dentro de los lenguajes diagramáticos están los lenguajes gráficos de flujo de datos.

Los LFD permiten visualizar el paralelismo y el flujo de control en un programa. Además este paradigma facilita la inserción de observadores en todo punto del flujo de datos.

Los programas de los LFD se representan con gráficas dirigidas donde cada nodo representa una función y cada arco un canal sobre el cual fluyen los datos. El flujo de control en la ejecución del programa esta determinado por los arcos dirigidos.

En un programa de un LFD el orden de ejecución esta dado por la disponibilidad de los datos que dicen quien se va a ejecutar: cuando todos los datos están presentes en un nodo este puede ejecutarse o dispararse para después del disparo enviar los resultados a otros nodos que requieren de dichos resultados. Además en un momento dado varios nodos pueden estar ejecutándose.

1.4.1 Extensiones al flujo de datos puro

Para que un LFD pueda manejar problemas grandes y complejos son necesarias las extensiones siguientes:

- Una colección grande (y extensible) de funciones predefinidas
- Abstracción procedural
- Elección condicional

Abstracción procedural

La abstracción encaja fácilmente en el modelo de flujo de datos: un procedimiento se define mediante una subgráfica con un nombre (o un icono) y se usa aplicando un nodo con el mismo nombre.

Elección condicional

La elección condicional se basa en los conceptos de distribuidor y selector.

Selector

Un selector tiene flujos de entrada i_2 , i_2 y c , y un flujo de salida o . Si c contiene true, entonces un token de i_1 es propagado a o , de otro modo de i_2 .

Distribuidor

Un distribuidor tiene flujos de entrada i y c , y flujos de salida o_1 y o_2 . Si c contiene true, entonces el token en i es propagado a o_1 , de otro modo a o_2 .

El paradigma de flujo de datos es un concepto muy simple y poderoso, con una representación visual intuitiva. Los LFD han sido más exitosos con programadores novatos y dominios de aplicación especializados.

CAPITULO 2

ENTORNOS DE PROGRAMACIÓN VISUAL

2.1 Lenguajes de definición de iconos

Como se menciono anteriormente dada la especificación de un lenguaje (la cual consiste en la sintaxis, semántica y otras características del lenguaje) se pueden generar las herramientas que constituyen un entorno para un lenguaje. Entre las herramientas que se pueden generar están los editores dirigidos por la sintaxis, los compiladores, los parsers y los verificadores de tipos. Dada la definición de la sintaxis de un Lenguaje Visual se puede generar un editor de oraciones visuales dirigido por la sintaxis.

En la especificación de un lenguaje visual se usa un Lenguaje de Definición de Iconos (LDI) el cual describe los tokens gráficos del lenguaje visual pero no tiene el poder para describir completamente el lenguaje visual. Lo anterior permite separar la descripción de tokens gráficos (la cual se puede usar en una variedad de aplicaciones) de la especificación del lenguaje visual. Los LDI incluyen lenguajes basados en gramáticas de atributos, descripciones basadas en procedimientos, o lenguajes declarativos basados en restricciones.

En este capítulo presentamos un ejemplo de un lenguaje de definición de iconos el cual fue propuesto por T.B. Dinesh del CWI (Centrum voor Wiskunde en Informatica) y S.M. Uskurdali de la Universidad de Ámsterdam. A dicho lenguaje lo llaman un lenguaje de definición de imágenes (Visual Object Definition Language VODL) el cual es declarativo basado en restricciones y sirve como una base gráfica para la especificación de lenguajes visuales y entornos de programación visual.

Las restricciones han sido usadas exitosamente en interfaces gráficas y son muy naturales para expresar relaciones espaciales entre elementos gráficos. En VODL cada imagen se define en términos de sus subcomponentes y de un conjunto de restricciones expresando las relaciones que existen entre ellos. Tales descripciones son fáciles de definir, ya que el especificador necesita describir solo subcomponentes y sus relaciones. En lugar de que el especificador tenga que describir algún procedimiento para lograr una cierta distribución espacial, el sistema subyacente es responsable de encontrar una solución (vía solución de restricciones).

Los autores citados han buscado una clara especificación de responsabilidades: objetos, restricciones, atributos (defaults/no defaults). Los objetos definen la estructura de la composición de una imagen, los atributos definen sus cualidades (para un gestor de recursos) y restricciones para definir su distribución espacial relativa (para un gestor de geometría).

2.2 El Lenguaje VODL

El lenguaje de definición de objetos visuales se diseño para especificar elementos visuales (imágenes). El lenguaje VODL especifica unidades visuales llamadas definiciones de objetos visuales (vods). Este provee un conjunto de vods primitivos junto con un conjunto de operaciones expresivas de vods para definir nuevos vods. También incluye una base gráfica que provee operaciones estándar gráficas definidas sobre todos los vods.

Los vods compuestos permiten la definición de nuevos vods en términos de otros vods (subvods) los cuales pueden tener restricciones espaciales entre ellos. Los sub-vods pueden ser primitivos, compuestos, u objetos emergentes. Cada vod puede tener atributos que definen sus propiedades

Ejemplo: SET

El lenguaje Set se usa como ejemplo para ilustrar la definición de tokens visuales y la especificación de un lenguaje.

La tabla siguiente muestra algunos tokens visuales para el lenguaje Set, donde Elems denota el parámetro.



Vod	representación
V-Set	
Union-Sym	

Fig. 2.1 Algunos tokens visuales para el lenguaje Set

El primer vod, V-Set, esta parametrizado (por elementos)

```
defv V-Set( Elems)
{ cir: circle(),
  elems: Elems }
<
{ cir contains elems }
```

Este vod describe un circulo conteniendo una colección de elementos. En esta definición se usa la abstracción **defv** la cual define un vod compuesto el cual consiste de dos subvods restringidos por la restricción **contains**.

La abstracción facilita la definición de vods (parametrizados), cuyo uso se puede diferir. Dicho uso puede incluir librerías de vods, definición de sintaxis, y definición de otros vods.

Los vods primitivos son los que se usan como bloques básicos de construcción en la construcción de vods compuestos. Hay varios tipos de vods primitivos tales como Punto, Línea, Circulo, Texto, Polígono, Lista y Colección.

El vod colección provee una forma de reunir varios vods, sin la necesidad de una figura específica que los contenga. Una colección es un conjunto de vods agrupados por una operación conmutativa y asociativa.

Los vods compuestos se construyen a partir de sub-vods y restricciones entre sub-vods. La construcción básica involucra la declaración de un conjunto de referencias-vods que se refieren a vods y especificar restricciones entre esos vods usando las referencias-vods.

Las restricciones son relaciones binarias entre vods. Se basan en las propiedades geométricas y los atributos de los vods. Las restricciones geométricas tratan con los tamaños relativos y posiciones de los vods. Otras clases de restricciones se definen en términos de los atributos de los vods tales como ancho y color.

Los objetos emergentes son los vods nuevos que aparecen como resultado de combinar dos vods donde ocurre una superposición. Por ejemplo, considere un vod que consiste en dos círculos que se traslapan. Esta composición da como resultado la emergencia de tres sub-vods uno definido por la región de la izquierda, otro por la del centro y uno más por la de la derecha. El reconocimiento y acceso a tales objetos emergentes puede ser muy útiles en la especificación de restricciones.

Definición del vod Two-Sets

Defv Two-Sets()

```
{ set1: circle () ⊕ [radius=15],
  set2: circle () ⊕ [radius=15],
  inter: overlap(set1,set2),
  diff1: difference(set1,set2),
  diff2: difference(set2,set1),
  border: rectangle() }
◁
{ set2.top = set1.top,
  set2.left = set1.left+set1.width*2/3,
  border contains set1
  border contains set2 }
```

La definición de Two-Sets consiste de seis sub-vods, tres de los cuales (set1, set2, y border) son sub-vods ordinarios predefinidos y los tres vods restantes () son vods emergentes identificados mediante el uso de las operaciones gráficas overlap y difference definidas en VODL. Aquellos vods emergentes son usados en las definiciones subsecuentes de vods (Intersect-Sym, Union-Sym, Diff-Sym).

Los atributos para los vods pueden ser especificados con la operación \oplus . Algunos atributos como ancho y altura se definen sobre todos los vods, mientras que otros son específicos a los vods.

Los vods nuevos se pueden definir usando operaciones de vods. Las operaciones de vods definen nuevos vods basados en los existentes extendiéndolos con atributos y/o restricciones, o combinando dos vods existentes.

El símbolo función para la intersección se define con la operación \oplus de vod, la cual redefine Two-Sets alterando el atributo de relleno del componente inter.

Defv Intersect-Sym ()

```
Two-Sets() ⊕ [inter.filled=true]
```

Similarmente el símbolo función para la unión es definido alterando los atributos de relleno de los componentes set1 y set2.

Defv Union-Sym()

Two-Sets() \oplus [set1.filled=true, set2.filled=true]

Similarmente el símbolo función para la diferencia es definido alterando el atributo de relleno del componente diff1.

Defv Diff-Sym()

Two-Sets() \oplus [diff1.filled=true]

En la especificación de un Lenguaje Visual VODL se define para describir tokens gráficos.

Los editores gráficos de vods permitirán la construcción gráfica de vods. La vista en la fig. muestra una definición gráfica del vod Intersect-Sym donde Two-Sets se extiende alterando un atributo (filled=true) del vod traslapado ("inter," en la definición textual). El vod que esta siendo extendido esta sobre el nuevo vod que esta siendo definido, estos están conectados con la operación de establecimiento de atributo. El especificador ha elegido el vod define del menú de operaciones para definir un nuevo vod y entonces ha elegido la operación de establecimiento de atributo, la cual requiere un vod y un conjunto de atributos.

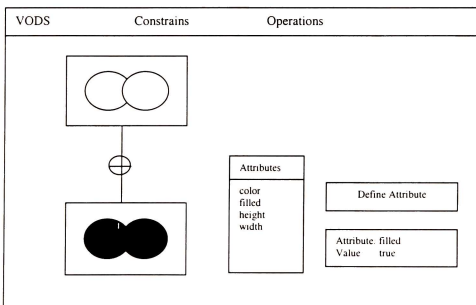


Fig. 2.2 El editor de VODL: Definiendo el vod Union-Sym

2.3 Definición de LIDA

LIDA es un lenguaje de flujo de datos donde las operaciones sobre los datos de entrada se representan mediante iconos. Las relaciones fluyen a través de las líneas de flujo para entrar a módulos de transformación que son los iconos.

El modelo de flujo de datos de LIDA da un enfoque de abstracción procedural debido a que el usuario solo trata con iconos que representan transformaciones de relaciones.

2.3.1 Arquitectura de LIDA

LIDA se compone de los siguientes módulos: editor de oraciones visuales, descriptor, capturador, traductor, intérprete de ISBL y álgebra relacional.

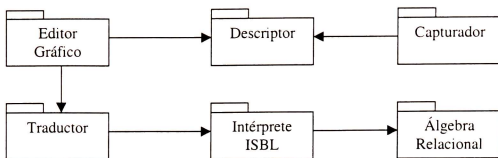


Fig. 2.3 Diagrama de componentes de LIDA

El módulo descriptor se usa para capturar los esquemas de relación. Además se utiliza para acceder a la base de datos.

El capturador de datos permite capturar los registros de las relaciones que conforman la base de datos.

El editor de oraciones visuales sirve para crear, modificar, y ejecutar los flujogramas.

El traductor construye una cadena de ISBL a partir de la representación interna del flujograma.

El intérprete de ISBL analiza la cadena de ISBL y realiza las operaciones de álgebra relacional indicadas por esta.

2.3.2 LIDA como lenguaje visual

A continuación se describen: el Diccionario de Iconos, el Diccionario de operadores y la Base de Datos de Acciones Semánticas de LIDA.

Diccionario de Iconos de LIDA

Los iconos de LIDA son iconos generalizados. La forma (reducida a lo básico) de una entrada en el Diccionario de Iconos es la siguiente:

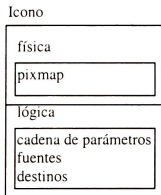


Fig. 2.4 Estructura de un icono en LIDA

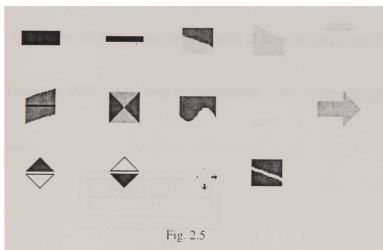
Cadena de parámetros: son los argumentos de la operación, del álgebra relacional, asociada al icono. Por ejemplo en el caso de la proyección los nombres de los campos a proyectar.

Operandos: son las relaciones que se usan como operandos de la operación, del álgebra relacional, asociada al icono.

La parte lógica de un icono corresponde a la semántica de un elemento del álgebra relacional de los que se muestran a continuación

Comando	Define	Diferencia	Entidad
Función	Intersección	Junta	Listado
Proceso	Proyecta	Ordena	OrdenaAsc
Selección	Unión		

Los pixmaps que corresponden a la parte física del diccionario iconos de LIDA se muestran a continuación.



Diccionario de Operadores de LIDA

LIDA cuenta con los operadores icónicos siguientes:

- Crea Icono
- Enlaza
- Borra
- Borra Enlace
- Reduce Icono
- Amplia Icono

Crea Icono. Crea un icono sin líneas de flujo de datos que salgan o entren en él.

Enlaza. Establece una línea de flujo de datos que va de un operando a un operador.

Borra. Elimina un icono así como todas las líneas de flujo de datos que salgan o entren en él.

Borra Enlace. Elimina una línea de flujo de datos entre dos iconos.

Reduce Icono. Aumenta el nivel de abstracción del icono mediante la eliminación de sus detalles (su IGU) dejando solo lo esencial (el pixmap).

Amplia Icono. Disminuye el nivel de abstracción pues restaura los detalles.

Base de Datos de Acciones Semánticas de LIDA

La Base de Datos de Acciones Semánticas contiene una interpretación para cada entrada del Diccionario de Iconos. Las acciones semánticas tienen la forma siguiente:

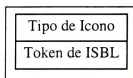


Fig. 2.6 Estructura de una acción semántica

El tipo de icono: es un número que identifica el tipo de icono

El Token de ISBL: es la cadena de ISBL que corresponde al tipo de elemento del álgebra relacional asociado con el icono.

El traductor utiliza el Token de ISBL, la cadena de parámetros y los operandos para generar el código de la consulta en ISBL.

Dada la acción semántica



Fig. 2.7

Si el valor de la cadena de parámetros es "A1,A2" y el nombre de la relación usada como operando es R1 entonces el traductor genera la cadena de ISBL siguiente: "R1%A1,A2"

2.3.3 Interpretación Semántica

La interpretación semántica de un icono depende de: el número de sus líneas de entrada, el número de sus líneas de salida y el tipo de icono.

Por ejemplo:

El significado de un icono cuyo tipo es entidad y de la cual sale una línea es leer la relación cuyo nombre se almaceno el la cadena de parámetros. En contraste el significado del mismo tipo de icono cuando la línea entra al es dar de alta el esquema de la relación en el archivo de descriptores y escribir la relación a un archivo de datos.

2.4 Diagrama de clases de un flujograma

Cada nodo en un flujograma es un objeto de la clase Elemento

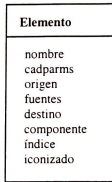


Fig. 2.8

El significado de cada atributo es el siguiente:

nombre: identifica el tipo de elemento

cadparms : son los argumentos de la operación asociada al icono.

origen: coordenadas de la pseudoventana

fuentes: es una colección de elementos que proveen información a este icono.

destinos: es una colección de elementos que reciben información de este icono.

componente: permite manipular el sublienzo que contiene la pseudoventana.

índice: identifica al elemento dentro del diagrama.

iconizado: indica si el elemento esta iconizado.

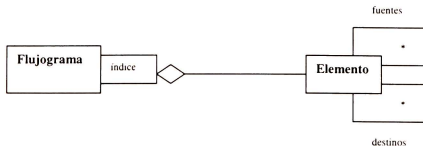


Fig. 2.9 Diagrama de clases de un flujograma

Del diagrama de clases se sigue que un elemento tiene como fuentes de la información que requiere y como destino de la información que produce a otros elementos.

Al flujograma siguiente:

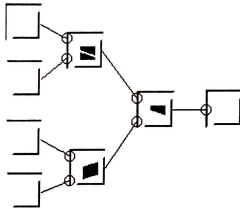


Fig. 2.10

corresponde el siguiente diagrama de clases

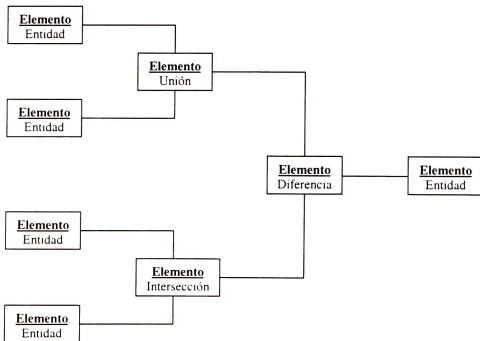


Fig. 2.11

CAPITULO 3

UML Y PATRONES DE DISEÑO

La descripción de nuestro sistema se hace mediante UML (Unified Modeling Language) y patrones de diseño, ya que el UML es un estándar de facto de la industria para construir modelos orientados a objetos y los patrones de diseño sirven para guiar y documentar el diseño de sistemas de software orientado a objetos. En este capítulo presentamos los conceptos de UML utilizados en el diseño de nuestro sistema. Y dado que el sistema se desarrollo utilizando el lenguaje SmallTalk de ParcPlace se describen también los patrones de diseño más importantes de SmallTalk-MVC.

3.1 UML

UML nació en 1994 por iniciativa de Grady Booch y Jim Rumbaugh para combinar sus dos conocidos métodos: el de Booch y el OMT (Object Modeling Technique, Técnica de Modelado de Objetos). Más tarde se les unió Ivar Jacobson, creador del metodo OOSE (Object Oriented Software Engineering, Ingeniería de Software Orientada a Objetos). En respuesta a una petición del OMG (Object Management Group, asociación para fijar los estándares de la industria) para definir un lenguaje y una notación estándar del lenguaje de construcción de modelos, en 1997 propusieron el UML como candidato. El UML abarca las mejores ideas tomadas de los métodos previos clásicos y orientados a objetos.

El UML define varios diagramas gráficos que ofrecen múltiples perspectivas del sistema bajo análisis y desarrollo. Esos diagramas se definen como siguen:

Diagrama de casos de uso: Muestra la relación entre actores y casos de uso dentro de un sistema. Un diagrama de casos de uso es similar en apariencia a aquellos de OOSE. El modelo de casos de uso representa la funcionalidad externa de un sistema o una clase desde el punto de vista de un agente externo que interactúa con el sistema. Un diagrama de casos de uso es una gráfica mostrando actores, un conjunto de casos de uso encerrado por la frontera del sistema, líneas de comunicación entre actores y los casos de uso, y generalizaciones entre casos de uso.

Diagrama de clases: Una fusión de los diagramas de clase de OMT, Booch y la mayoría de los otros métodos OO, los diagramas de clases muestran la estructura estática del modelo, especialmente las cosas que existen, tales como clases y tipos, sus estructuras internas, y sus relaciones con otras cosas.

Diagramas de comportamiento: Los diagramas de comportamiento muestran la evolución del sistema (su dinámica) y están clasificados como sigue:

Diagrama de estado: substancialmente basado en los diagramas de estado de David Harel, un diagrama de estado muestra la secuencia de los estados por los que pasa un objeto en interacción durante su vida como respuesta a los estímulos recibidos, junto con sus respuestas y acciones.

Diagrama de actividades: Un modelo de actividades es una forma de una maquina de estados en la cual los estados son actividades que representan la ejecución de operaciones. Las transiciones son disparadas por la finalización de las operaciones. Un diagrama de actividades representa la maquina de estado del procedimiento mismo; el procedimiento es la implementación de una operación de la clase a la que pertenece.

Diagramas de secuencia: Los diagramas de secuencia aparecieron en varios métodos OO con nombres tales como interacción, rastreo de mensajes, y rastreo de eventos. Un diagrama de secuencia muestra una interacción ordenada en una secuencia temporal. En particular, un diagrama de secuencia muestra los objetos que participan en la interacción por medio de sus líneas de vida y los mensajes que intercambian, ordenados en secuencia temporal. Un diagrama de secuencia no muestra las asociaciones entre objetos. Los diagramas de secuencia vienen en varios formatos ligeramente diferentes destinados a diferentes propósitos. Un diagrama de secuencia puede existir en una forma genérica para describir todas las secuencias posibles y en una forma instancia para describir una secuencia actual consistente con la forma genérica. En casos sin ciclos o ramas, las dos formas son isomorfas.

Diagramas de colaboración: Los diagramas de colaboración fueron adaptados de varias fuentes: el diagrama de objetos de Booch (1991), el método Fusion (gráfica de interacción de objetos)(Coleman et al, 1994), así como de otras. Un diagrama de colaboración muestra una interacción organizada alrededor de los objetos en la interacción y sus ligas uno con otro. Los diagramas de secuencia y los diagramas de colaboración expresan información similar pero la muestran en formas diferentes. Un diagrama de colaboración muestra las relaciones entre objetos y es mejor para entender todos los efectos en un objeto dado y para diseño procedural. Por otra parte, un diagrama de colaboración no muestra el tiempo como una dimensión separada, por tanto la secuencia de los mensajes y subprocessos concurrentes deben ser determinados usando números de secuencia.

Diagramas de implementación: Los diagramas de implementación muestran aspectos de la implementación tales como la estructura del código fuente y la estructura de la implementación de tiempo de ejecución. Se derivan de los diagramas de Booch de modulo y proceso, pero ahora están centrados en componentes en lugar de centrados en módulos y están mucho mejor interconectados. Vienen en dos formas: diagramas de componentes y diagramas de emplazamiento.

Diagramas de componentes: Un diagrama de componentes muestra las dependencias entre los componentes de software, incluyendo código fuente, código binario, y ejecutables. Un modulo de software puede ser representado como un tipo componente. Algunos componentes existen en tiempo de compilación; algunos existen en tiempos de ligado; y algunos existen en mas de uno de los tiempos. Un componente de solo compilación es significativo solo en tiempo de compilación; el componente de tiempo de ejecución en este caso será un programa ejecutable. Un diagrama de componentes tiene solo una forma tipo, no una forma instancia. Las instancias componentes se muestran usando un diagrama de emplazamiento, posiblemente uno degenerado sin nodos.

Diagrama de emplazamiento: Un diagrama de emplazamiento muestra la configuración del hardware y los componentes de software, procesos, y los objetos que viven en ellos. Las instancias de componentes de software representan manifestaciones en tiempo de ejecución de unidades de código. Los componentes que no existen como entidades de tiempo de ejecución (por que han sido compilados aparte) no aparecen en esos diagramas; deberán mostrarse en los diagramas; deberán mostrarse en los diagramas de componentes.

Entre otras características interesantes, UML toma en cuenta la noción de patrones de diseño a través de diagramas de colaboración, donde a las colaboraciones les ha sido otorgado el status de entidades de modelado de primera clase y pueden formar la base de patrones. Una colaboración es un contexto para interacciones. Una colaboración es un contexto para interacciones. Una colaboración puede de verdad ser usada para especificar la implementación de las construcciones del diseño. Una colaboración puede ser vista también como una entidad única desde fuera. Por ejemplo, una colaboración puede ser usada para identificar la presencia de patrones de diseño dentro del diseño del sistema. Un patrón es una colaboración parametrizada; en cada uso del patrón, las clases actuales son substituidas por los parámetros en la definición del patrón.

Un uso del patrón se muestra como una elipse punteada conteniendo el nombre del patrón. Una línea punteada conteniendo el nombre del patrón. Una línea punteada es dibujada del símbolo de colaboración a cada uno de los objetos o clases (dependiendo en si aparece dentro de un diagrama de objetos o un diagrama de clases) que participan en la colaboración. Cada línea es etiquetada con el papel del participante. Los papeles corresponden a los nombres de los elementos dentro del contexto para la colaboración; tales nombres en la colaboración son tratados como parámetros que son ligados para especificar elementos en cada aparición del patrón dentro de un modelo. Esta notación tiene la ventaja de hacer el uso de un patrón dado explícito y prominente en un diagrama de clases, sin entrar en los detalles de la implementación.

3.2 PATRONES DE DISEÑO

Los patrones de diseño son descripciones de objetos y clases que colaboran los cuales son ajustados para resolver un problema general de diseño en un contexto particular.

Los patrones de diseño se pueden considerar microarquitecturas que contribuyen a la arquitectura general del sistema. En otras palabras son una forma de desarrollar y empaquetar componentes de software reutilizables.

La documentación de la arquitectura de un sistema se puede ser hacer claramente usando patrones de diseño como un vocabulario compartido para estructuras de diseño comunes. Por tanto los patrones de diseño constituyen los diseños básicos de construcción de una arquitectura de diseño, permitiendo a los ingenieros realizar el diseño a un nivel mas alto, reduciendo así su complejidad aparente. En lugar de pensar en términos de clases individuales y sus comportamientos, ahora es posible pensar en términos de clases que colaboran, con sus relaciones y responsabilidades.

3.2.1 PATRONES DE DISEÑO Y FRAMEWORKS

Los frameworks están estrechamente relacionados con los patrones de diseño. Un framework de software orientado a objetos esta constituido por una colección de clases relacionadas que pueden ser especializadas o instanciadas, para implementar una aplicación. Un framework es una arquitectura de software reutilizable que provee la estructura y comportamiento genérico para una familia de aplicaciones de software, junto con un contexto que especifica su colaboración y uso dentro de un dominio dado.

A diferencia de una aplicación completa un framework carece de la funcionalidad necesaria específica a la aplicación. Un framework puede considerarse como una estructura prefabricada, o plantilla, de una aplicación funcional, en la cual un numero de piezas en lugares específicos- llamados puntos de enchufe- o no se implementan o se les da una implementación sustituible. Para obtener una aplicación completa de un framework, uno tiene que proveer las piezas faltantes, usualmente implementando un numero de funciones call-backs (esto es, funciones que son invocadas por el framework) para llenar los puntos de enchufe. En un contexto orientado a objetos, esta característica es lograda por ligado dinámico: Una operación puede ser definida en una clase de librería pero implementada en una subclase en el código específico a la aplicación. Un desarrollador puede así ajustar el framework a una aplicación en particular por mediante el subclasado y composición de instancias de clases del framework.

Un framework es así diferente de una librería de clases clásica en que el flujo de control es usualmente bidireccional entre la aplicación y el framework. El framework esta a cargo del manejo del grueso de la aplicación, y el programador de la aplicación solo provee varios trozos y piezas. Esto es similar a programar alguna aplicación orientada a eventos, cuando el programador de aplicaciones usualmente no tiene control sobre la lógica principal de control del código.

Los patrones de diseño pueden ser usados para documentar las colaboraciones entre clases en un framework. Recíprocamente, un framework puede usar varios patrones de diseño, algunos de ellos de propósito general, algunos de ellos específicos al dominio. Los patrones de diseño y los frameworks están así estrechamente relacionados, pero no operan al mismo nivel de abstracción. Un framework esta hecho de software, mientras que los patrones de diseño representan conocimiento, información, y experiencia acerca del software. En este aspecto, los frameworks son de naturaleza física, mientras que los patrones son de naturaleza lógica. Los frameworks son la realización física de una o más soluciones de patrones de software; los patrones son las instrucciones sobre como implementar aquellas soluciones.

3.2.2 PATRONES DE DISEÑO EN SMALLTALK-MVC

MVC consiste de tres clases de objetos. El Modelo es el objeto aplicación, la Vista es la presentación en pantalla, y el Controlador define como la IGU reacciona a la entrada del usuario. Antes de MVC los diseños de la IGU tendían a mezclar esos objetos. La arquitectura MVC los separa para incrementar la flexibilidad y el reuso.

Patrón de diseño Observador

MVC separa vistas y modelos estableciendo un protocolo de suscripción /notificación entre ellos. Una vista debe asegurarse de que su apariencia refleja el estado del modelo. Siempre que los datos del modelo cambian, el modelo notifica a las vistas que dependen de este. En respuesta, cada vista tiene una oportunidad para actualizarse ella misma.

He aquí como trabaja: la clase Objeto, en la raíz de la jerarquía de clases, mantiene una colección de dependientes, permitiendo a cualquier objeto registrarse el mismo como dependiente de otro objeto. Objeto también implementa un método `changed`, así cualquier objeto puede enviar el mensaje `changed` así mismo. Lo que el método `changed` hace es enviar un mensaje `update` a cada uno de los dependientes del objeto. Así cuando un objeto envía un mensaje `changed` al mismo, sus dependientes reciben un mensaje `update`: automáticamente. Los dependientes deben implementar el método `update` correspondiente, en el cual ellos mismos se re-despliegan o toman otras acciones de actualización.

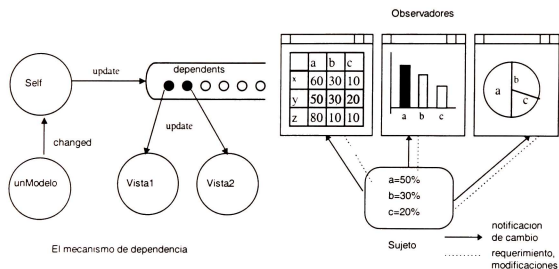


fig. 3.1

La aproximación anterior permite asignar múltiples vistas a un modelo para proveer diferentes presentaciones. También se pueden crear nuevas vistas para un modelo sin tener que re-escribirlo.

El modelo se comunica con sus vistas cuando sus valores cambian y las vistas se comunican con el modelo para acceder a aquellos valores.

El patrón de diseño observador describe el diseño más general donde: se separan objetos para que los cambios que afectan a uno puedan afectar cualquier número de otros sin requerir que el objeto cambiado conozca detalles de los otros. Los objetos principales en este patrón son sujeto y observador, ver figura 3.2. Un sujeto puede tener cualquier número de observadores dependientes, ver figuras 3.1 y 3.2.

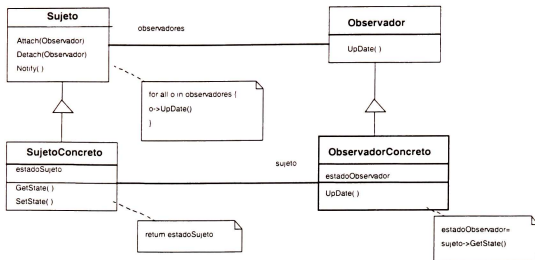


Fig. 3.2

Patrón de diseño Compuesto

Otra característica de MVC es que las vistas se pueden anidar. MVC soporta vistas anidadas con la clase CompositeView, una subclase de DependentComposite. Este es el equivalente compuesto de una vista. Los objetos CompositeView actúan como los objetos View; una vista compuesta puede ser usada siempre que una vista pueda ser usada, pero esta también contiene y maneja vistas anidadas.

El patrón de diseño Compuesto describe el diseño más general donde: se agrupan objetos y se trata el grupo como un objeto individual. Este permite crear una jerarquía de clases en la cual algunas subclases definen objetos compuestos que ensamblan los primitivos en objetos más complejos. La clave del patrón de diseño Compuesto es una clase abstracta que representa tanto a primitivos como a sus contenedores, ver figura 3.3.

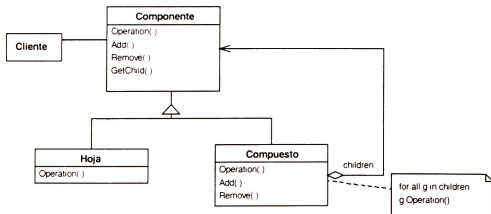


Fig. 3.3

Patrón de diseño estrategia

MVC permite cambiar la forma como una vista responde a la entrada del usuario sin cambiar su presentación visual. MVC encapsula el mecanismo de respuesta en un objeto controlador. Hay una jerarquía de controladores, que hacen fácil crear un controlador nuevo como una variación de uno existente.

Una vista usa una instancia de una subclase controlador para implementar una estrategia de respuesta en particular; para implementar una estrategia diferente, simplemente se reemplaza la instancia con otra instancia de una subclase de controlador diferente. Es aun posible cambiar el controlador en tiempo de ejecución para permitir que la vista cambie la forma en que esta responde a la entrada de usuario. Por ejemplo una vista puede ser deshabilitada para que esta no acepte entrada simplemente dándole un controlador que ignore los eventos de entrada.

La relación Vista-Controlador es un ejemplo del patrón de diseño estrategia. Una estrategia es un objeto que representa un algoritmo. Esto es útil cuando se quiere reemplazar el algoritmo ya sea estáticamente o dinámicamente, cuando se tienen muchas variantes del algoritmo, o cuando el algoritmo tiene estructuras complejas que se quieren encapsular, ver figura 3.4.

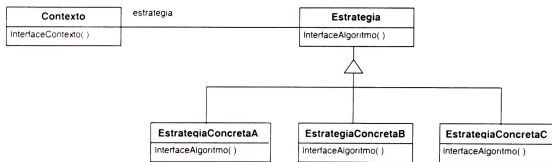


Fig. 3.4

Patrón de diseño adapter

Una clase es más reutilizable cuando se minimizan las suposiciones que otras clases deben hacer para usarla. Construyendo la adaptación a la interfaz de una clase, se elimina la suposición de que otras clases vean la misma interfaz. Por ejemplo cuando se espera que una vista sea usada con diferentes modelos, o todos los modelos deben hablar el lenguaje de la vista o la vista debe ser capaz de adaptarse ella misma a diferentes mensajes. Hacer que los modelos se adapten viola el principio de que ningún modelo debe ser dependiente de sus vistas.

Un adaptor provee un conjunto standard de mensajes de acceso para vistas y controladores (value y value:) mientras permanece muy flexible en cuanto a comunicación con sus modelos. Un adaptor es como un traductor universal por que puede ser entrenado para hablar el lenguaje de cualquier modelo.

La clase abstracta ValueModel, definida para vistas que muestran un solo valor, provee el mecanismo de obtención de valor (value) y almacenamiento (value:). Esta tiene dos subclases, ValueHolder y PluggableAdaptor.

Un ValueHolder traduce (para un objeto simple, guardado en su variable de instancia value, que no se comporta normalmente como modelo) los mensajes estándar value y value: que recibe en la devolución del objeto que guarda y su reemplazo (con notificación a sus dependientes del cambio).

Los adaptadores que se enchufan (pluggable adapters) son comunes en ObjectWorks /SmallTalk.

Un PluggableAdaptor traduce (para un objeto más complicado que tiene un vocabulario diferente que la vista, y posiblemente una semántica diferente) los mensajes standard value y value: que recibe (de las vistas y controladores) en acciones arbitrarias (definidas por bloques) que sirven para obtener y almacenar el valor deseado.

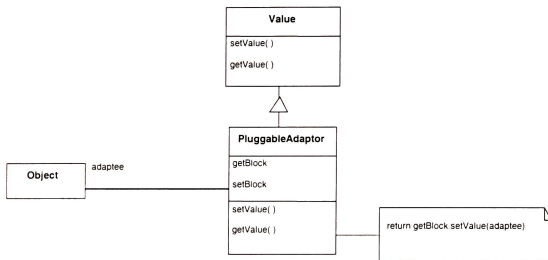


Fig. 3.5

Los escritores de aplicaciones accesan el valor con nombres de dominio específicos al dominio como width y width:, pero ellos no deberían tener que subclasar ValueModel para traducir dichos nombres específicos a la aplicación a la interfaz de ValueModel.

En su lugar PluggableAdaptor permite pasar nombres de selector (como width y width:) directamente para el caso en que el modelo solo usa palabras diferentes para significar la misma cosa que value y value:. Este convierte aquellos selectores en bloques correspondientes automáticamente, ver figura 3.5.

El patrón de diseño Adaptor describe el diseño más general donde: se convierte la interfaz de una clase en otra interfaz que los clientes esperan. Adaptor permite que trabajen juntas clases que de otra manera no podrían hacerlo a causa de sus interfaces incompatibles. Dicho de otra forma la adaptación a una interfaz nos permite incorporar nuestra clase en sistemas existentes que pueden esperar interfaces diferentes a la interfaz de la clase, ver figura 3.6.

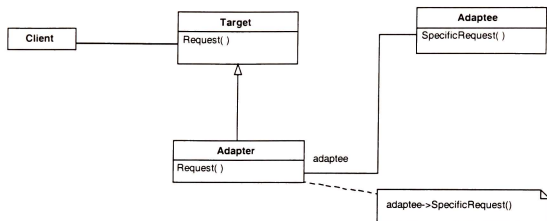


Fig. 3.6

CAPITULO 4

VISUALWORKS

Nuestro editor de flujogramas, se implementó usando como herramienta de desarrollo VisualWorks de ParcPlace Systems. En este capítulo presentamos una descripción de las características de VisualWorks que son necesarias para entender nuestro trabajo.

4.1 CARACTERÍSTICAS DE VISUALWORKS

VisualWorks es un ambiente totalmente orientado a objetos para construir aplicaciones, utilizando el lenguaje SmallTalk de ParcPlace.

VisualWorks permite el desarrollo rápido de aplicaciones (Rapid Application Development o RAD) por medio de herramientas visuales.

Con VisualWorks se pueden construir interfaces gráficas de usuario rápidamente para aplicaciones nuevas o ya existentes.

Usando VisualWorks se crea la IGU de manera interactiva por medio de herramientas que se utilizan moviendo el ratón entre ellas y pulsando botones del mismo (drag-and-drop). Dichas herramientas permiten incorporar controles en la IGU de una aplicación.

VisualWorks además cuenta con mecanismos para reutilizar interfaces y el código de aplicaciones. También tiene portabilidad sobre plataformas UNIX, MicroSoft Windows, OS/2 y Macintosh.

4.2 ESTRUCTURA DE APLICACIONES EN VISUALWORKS

Una aplicación de VisualWorks se divide conceptualmente en dos partes:

- El modelo de información que maneja el almacenamiento de datos y el procesamiento.

- La interfaz de usuario, que maneja entrada y salida

Esta separación de tareas tiene como resultado los niveles correspondientes de objetos:

- Objetos modelos (o simplemente modelos), los cuales definen y manipulan estructuras de datos.

- Objetos de interfaz de usuario (objetos IU), los cuales presentan los datos de los modelos y permiten a los usuarios interactuar con estos datos. Los objetos IU son los objetos que conforman una pantalla.

Los objetos IU y los modelos están altamente interconectados. Por sí, cada objeto IU solo provee características visuales y una respuesta visual a la entrada del teclado y el ratón.

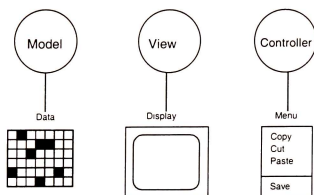


Fig. 4.1 Modelo vista y controlador con sus contrapartes Usuales del mundo real.

De lo dicho de los objetos IU se desprende que:

La interfaz de usuario a su vez se divide en un componente para desplegar salida a la pantalla (conocido como vista) y otro componente que habilita al usuario a interactuar con, o controlar, la aplicación (un controlador). El controlador define la manera como la interfaz de usuario reacciona a las entradas de usuario.

Dicho método de descomposición de una aplicación ha sido usado y refinado desde la concepción de SmallTalk. Este se conoce como arquitectura Model-View-Controller (MVC) ver fig 4.1.

Cada objeto IU consiste de una vista acoplada con un controlador y objetos de soporte asociados.

Una aplicación típica es adicionalmente estratificada para distinguir diferentes clases de modelos. Entre aquellos están los modelos de dominio y los modelos de aplicación.

4.2.1 Modelos de Dominio

Los modelos de dominio simulan el comportamiento y estado de objetos en el mundo real en el dominio de la aplicación.

4.2.2 Modelos de Aplicación

Los modelos de aplicación proveen un nivel de información y servicios entre los objetos IU y los modelos de dominio. Entre otras cosas, un modelo de aplicación define el comportamiento específico a la aplicación de los controles individuales en la interfaz por ejemplo a través de:

Establecer conexiones entre los controles y los datos que representan.

Controlando como interactúan los controles entre sí.

Un modelo de aplicación es responsable de crear y manejar una interfaz de usuario en tiempo de ejecución.

Cada modelo de aplicación define el comportamiento específico a la aplicación para una sola ventana y los controles en esta. Consecuentemente una aplicación multi-ventana tiene varios modelos de aplicación interconectados.

4.3 Modelo de aspecto

Un modelo de aspecto contiene información de un solo aspecto y provee el modelo de comportamiento de un solo componente.

La relación entre el modelo de aplicación y un modelo de aspecto es que un modelo de aplicación contiene uno o más modelos de aspectos.

4.4 Especificaciones de interfaz de usuario

Una especificación de interfaz de usuario es una descripción simbólica y jerárquica de las características y apariencias de una interfaz de usuario.

Cada tipo de componente tiene una clase de especificación correspondiente.

ComponentSpec es el antepasado común abstracto de todas esas clases a las que hereda comportamiento para la persistencia de la interfaz en forma de código fuente o archivo de texto.

Una FullSpec describe una interfaz de usuario completa (en su parte component), incluyendo la ventana (en su parte window) en la cual será desplegada esta así como todos los componentes incluidos en esta. Las FullSpecs son generadas típicamente por el pintor. Los métodos de especificación instalados en clases de aplicación regresan FullSpecs en su forma literal.

Las especificaciones pueden ser generadas dinámicamente por el modelo, pero son típicamente el resultado de la interacción directa del usuario con el pintor de la interfaz.

4.5 Pintor y UIBuilder

Pintor es la herramienta central de VisualWorks. Esta permite editar, durante la fase de diseño, una especificación visualmente a partir de un conjunto estándar de componentes flexibles, provistos por la paleta. Las especificaciones jerárquicas resultantes se almacenan en el método de clase windowSpec de una subclase de ApplicationModel. Las especificaciones son usadas por un UIBuilder (que es creado y usado cuando se abre la interfaz) para generar una ventana operacional.

La liga entre la interfaz en tiempo de diseño y la interfaz en tiempo de ejecución es un `UIBuilder`. Este es responsable de crear objetos IU reales a partir de especificaciones literales, y conectar las dependencias entre el modelo de aplicación y los componentes de la interfaz.

Un builder guarda los objetos IU que crea. De aquí que una aplicación pueda enviar mensajes al builder para acceder desde un programa a un control o la ventana misma.

Un recurso es un objeto usado por el builder para construir la interfaz de acuerdo a las especificaciones.

La fuente del builder es un `ApplicationModel` que provee los recursos necesarios a un builder para construir una interfaz de usuario de acuerdo a las especificaciones. Se guarda en la variable de instancia de `UIBuilder` llamada `source`.

Un builder usa las siguientes variables para el cache de recursos:

- `bindings` para el cache de modelos de aspecto requeridos por componentes activos.
- `labels` para el cache de etiquetas de texto
- `visuals` para el cache de componentes visuales.

4.6 Componentes visuales y vistas

La vista es responsable del despliegue de aspectos del modelo.

Una ventana es un contenedor para un componente visual la vista. La clase abstracta que define el comportamiento de vistas y otros objetos desplegados se llama `VisualComponent`.

Un componente visual pasivo recibe peticiones del contenedor pero no envía peticiones a este. Un componente visual activo recibe y también envía peticiones, así que necesita llevar un seguimiento de su contenedor. La clase abstracta `VisualPart` provee esta habilidad de comunicarse con el contenedor, por tanto es el antepasado de todos los componentes visuales, incluyendo `View` y sus subclases.

Un widget es una parte visual (`visual part`) responsable de la representación visual de un componente.

Como subclase directa de `DependentPart` una vista tiene un modelo y como subclase de `VisualPart` tiene un contenedor.

Una ventana esta preparada para comunicarse con un solo componente visual. Para colocar múltiples componentes en la misma ventana se necesita algún objeto intermediario para contener los subcomponentes.

`CompositePart`, ver fig 4.2, contiene una colección de otros componentes visuales. Para cada uno de sus subcomponentes, este es un contenedor como lo seria una ventana. `CompositePart` tiene dos subclases:

DependentComposite para situaciones en que el objeto compuesto mismo necesita ser un dependiente del modelo. Este es el equivalente compuesto de DependentPart.

CompositeView, una subclase de DependentComposite, el cual tiene un controlador que se extiende a todos sus subcomponentes. Este es el equivalente compuesto de vista.

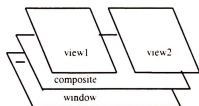


Fig. 4.2 Una CompositePart mantiene una colección de otros componentes visuales

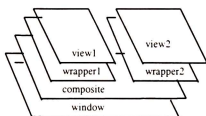


Fig. 4.3 Cada componente hoja está contenido en un wrapper

La clase Wrapper agrega funcionalidad genérica tal como marco a un componente. Esta clase tiene un componente y reenvía mensajes seleccionados hacia y del componente, ver fig 4.3.

Un SpecWrapper es un wrapper que contiene un widget, decoración para el widget, una copia de un objeto WidgetState y ComponentSpec.

SpecWrapper es usado por UIPainter y UIBuilder para unir una especificación (UISpecificación) con el componente específico que la representa visualmente en un lienzo que está siendo pintado, o con el componente específico reflejando la especificación en una interface activa.

4.7 Sublienzos

Un sublienzo es una `CompositeView` intensificada una cuyo interior es adicionalmente descrito por un `FullSpec`. Un sublienzo permite a un lienzo completo ser importado como un compuestio; promoviendo así el reuso de arreglos visuales.

La subaplicación es el modelo de aplicación que maneja el sublienzo. La aplicación madre es el modelo de aplicación que provee la subaplicación como modelo de aspecto.

`SubCanvasSpec` describe un sublienzo y tiene tres propiedades: `clientKey`, `majorKey`, y `minorKey`. El mensaje `clientKey` es enviado a la aplicación madre para adquirir la subaplicación que maneja el sublienzo. La subclase de `ApplicationModel` `majorKey` construye y ejecuta el componente del sublienzo. El método de clase `minorKey` regresa una especificación completa describiendo la interfaz de usuario del sublienzo.

4.8 Control

La posesión de la entrada de usuario es conocida usualmente como control. Cada ventana y cada componente activo dentro de una ventana tienen un controlador asociado.

4.8.1 El Flujo de Control

El sistema operativo anfitrión pasa el control a un manejador de control cuando una ventana de `SmallTalk` es activada. El manejador de control pasa el control a la ventana que contiene el cursor. Esta ventana toma el control si el botón `<window>` es presionado; de otra manera ofrece el control al controlador de su componente. El componente puede tomar el control o puede pasarlo a sus propios componentes, y así sucesivamente hasta que todos los controladores en la cadena de herencia han rechazado el control, o hasta que uno de ellos encuentra apropiado aceptar el control.

4.8.2 La secuencia de control

Un controlador con menú checa para ver si el botón `<operate>` en el ratón es presionado durante su ciclo de actividad. Cuando la prueba tiene éxito, este envía un mensaje `startUp` a su menú. Esta es solo una actividad con la que un controlador puede ser equipado para repetir durante su ciclo de actividad.

Cada `StandardSystemcontroller`, el cual está asociado con una ventana en lugar de una vista, es almacenado en una colección mantenida por una instancia de `ScheduledControllers`. La variable global `ScheduledControllers` mantiene una instancia de `ControlManager`. Este manejador de control es responsable de pasar el control al controlador de la ventana activa.

El controlador de la ventana pregunta a la ventana cual de sus subcomponentes quiere el control, si alguno (`subViewWantingControl`). En respuesta a este mensaje, la ventana requiere `objectWantingControl` de su componente. El componente, si este es un `composite`, reenvía el mensaje a cada uno de sus subcomponentes, y así sucesivamente. Cada componente a nivel hoja, después de recibir este mensaje, pregunta a su controlador `isControlWanted`. Al primer controlador en responder `true` se le envía un mensaje `startUp` por el manejador de control, comenzando su secuencia básica de control.

La secuencia de control básica consiste en tres pasos, ver fig 4.4, que se muestran a continuación:

Inicializa el control
Seguir mientras las condiciones para mantener el control se cumplan
terminar el control

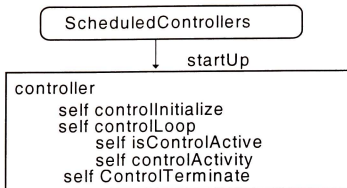


Fig. 4.4

La inicialización es realizada en un método llamado `controlInitialize`. Por defecto un controlador de un componente visual no hace nada en respuesta a este mensaje. Se puede reimplementar el método en una clase controlador para realizar una acción especial cuando el controlador inicia. El método de terminación de control se llama `controlTerminate`.

En su método `controlLoop` un controlador verifica primero que la condición para mantener el control (`isControlActive`) es aun cierta. Si es así, envía `controlActivity` a si mismo, después de lo cual repite la prueba `isControlActive`. En el método `controlActivity` es donde el controlador interroga a su sensor para saber si ha ocurrido un evento de entrada de un tipo particular. Cuando la prueba `isControlActive` falla, el control regresa a `ScheduledControllers`, el cual empieza a sondear a sus controladores planificados para encontrar un nuevo receptor de control.

CAPITULO 5

INTERFAZ GRÁFICA DE LIDA

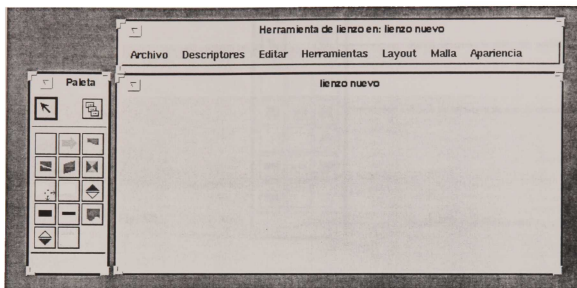


Fig. 5.1 Paleta, herramienta del lienzo y lienzo

Esta tesis presenta un editor de flujogramas flexible con el cual es posible:

Superponer una cuadrícula al lienzo para alinear los nodos del diagrama. Agrupar varios nodos del diagrama para copiarlos, cortarlos, pegarlos, moverlos y guardarlos como si fueran un solo nodo. La operación de pegado se puede realizar también entre lienzo diferentes. Aumentar el número de nodos en el diagrama mientras haya memoria disponible. Cambiar la imagen predeterminada por otra que sea más descriptiva de la información almacenada en dicha relación. Usar pixmaps en lugar de dibujos vectoriales para disponer de un número mayor de fuentes de imágenes.

Otra ventaja de nuestro editor es que es fácilmente extensible y integrable a otros sistemas de programación visual. Estas ventajas son posibles gracias a su arquitectura. En este capítulo mostramos su capacidad de integración y flexibilidad. Las cuales probamos integrándolo a al Lenguaje Iconográfico para el desarrollo de Aplicaciones (LIDA), en el cual se desarrollan programas visuales que definen y realizan consultas a bases de datos.

En el resto de este capítulo mostramos lo que ahora es posible hacer en LIDA con nuestro editor. En los siguientes dos capítulos presentamos el diseño e implementación de nuestro editor.

Básicamente nuestro editor se compone de una paleta, un lienzo y la herramienta del lienzo, ver figura 5.1.

5.1 La Paleta

La paleta (fig. 5.2) es donde se encuentran los tipos de pseudoventanas disponibles para pintar.

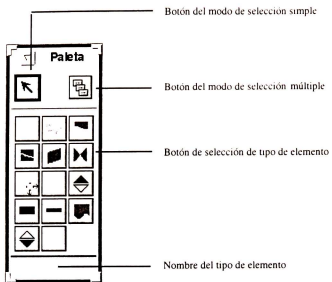


Fig. 5.2

La paleta puede estar en uno de dos modos de selección: simple o múltiple. El primer ítem en la paleta es el botón de selección simple. Cuando está activo el modo correspondiente (o cuando es presionado dicho botón), permite seleccionar algún tipo de pseudoventana en la paleta y pintar una sola copia de esta en el lienzo. Después de que se pinta la pseudoventana en el lienzo el modo de selección simple permanece activo.

El segundo ítem es el botón de selección múltiple. Cuando se activa el modo de selección múltiple, dicho botón desactiva el modo de selección simple. Entonces, en el lienzo, permite pintar múltiples copias del tipo de pseudoventana seleccionada subsecuentemente en la paleta.

Seleccionando otra pseudoventana o pulsando el botón de selección simple se desactiva el modo de selección múltiple.

La forma básica de usar la paleta es

- 1.- Pulse el botón de selección simple o múltiple.
- 2.- Pulse el botón de selección de tipo de pseudoventana.
- 3.- Mover el cursor al lienzo y posicionarlo donde se desee, entonces pulsar el botón izquierdo del ratón para soltar la pseudoventana en el lienzo

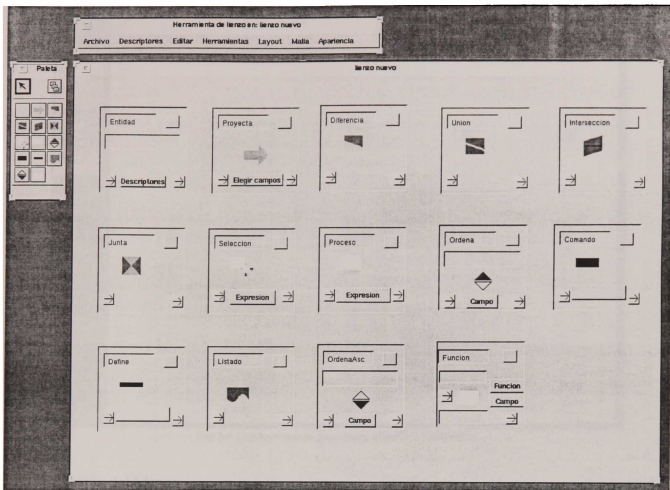


Fig. 5.3 Seudoventanas existentes en LIDA

Las figuras 5.3 y 5.4 muestran las seudoventanas de LIDA en su forma normal y su forma iconizada. El funcionamiento de dichas seudoventanas se explicará en otras secciones de este capítulo.

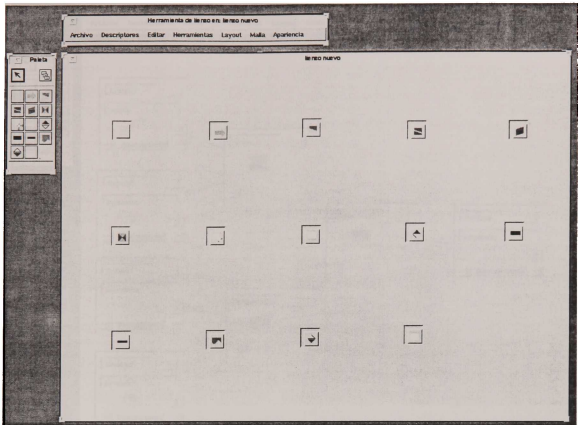


Fig 5.4 Seudoventanas existentes en LIDA iconizadas

5.2 Operación del Lienzo

Si hay una seudoventana seleccionada en la paleta entonces cada vez que el cursor del ratón entra al lienzo aparece una seudoventana, la cual contiene el pixmap del tipo de elemento elegido, que sigue el movimiento del cursor. Cuando se llega a la posición donde se desea colocarla se presiona el botón izquierdo del ratón. Entonces la seudoventana cesa de seguir el movimiento del cursor del ratón. Si en lugar de apretar el botón izquierdo se sigue moviendo el cursor hasta que salga del lienzo entonces la seudoventana desaparece en el momento en que este sale. Si se quiere mover la seudoventana de la última posición donde fue colocada se selecciona esta (presionando el botón izquierdo del ratón cuando el cursor de este se encuentre en una zona de la misma que no contenga ningún control) y se mueve el cursor del ratón mientras se mantiene presionado su botón izquierdo hasta llegar a la nueva posición donde se suelta el botón. Si existe(n) línea(s) entre la seudoventana en movimiento y otra(s) dicha(s) línea(s) se redibuja(n) siguiendo el movimiento de dicha ventana ver figuras 5.5 y 5.6.

5.3 Dibujo de líneas entre seudoventanas

Si se desea dibujar una línea entre dos seudoventanas se tiene que presionar el botón que tiene una flecha horizontal (como etiqueta gráfica) y esta en el lado derecho de la seudoventana, la cual representa al elemento que provee información a otro, después se presiona el botón que tiene una flecha horizontal pero que esta del lado izquierdo de la seudoventana, que representa al elemento que recibe datos. Entonces aparece el segmento de recta (que resultaría de recortar del segmento de recta que va del centro de una seudoventana al centro de la otra las dos partes que estarían dentro de las seudoventanas) entre ambas seudoventanas. Adicionalmente se checa que no se sobrepase el número de elementos que pueden recibir información del elemento que la provee.

5.4 Menú del Lienzo

Existe un menú de aparición súbita que aparece cuando se presiona el botón central del ratón. Dicho menú tiene las opciones siguientes:

- Archivo
 - Guarda
 - Guarda Subdiagrama
 - Lee
 - Nuevo
- Editar
 - Copiar
 - Cortar
 - Desconectar
 - Pegar
 - Expandir
- Acomodar

Para guardar un diagrama o un subdiagrama (cuyos elementos han sido elegidos previamente) se elige dicha opción del menú entonces aparece una caja de diálogos donde se puede dar el nombre del archivo o especificar la extensión (mediante *.lda) realizar esto ultimo ocasiona la aparición de otra caja de dialogo con una lista de nombres de archivos existentes; si se elige uno de estos aparece otra caja de diálogos que nos pide confirmar si se desea reemplazar el diagrama del archivo elegido con el que esta pintado en el lienzo. Para leer un diagrama de disco el proceso es análogo.

Si se desea abandonar la edición del diagrama actual y borrarlo del lienzo sin salvarlo se elige nuevo en el menú.

Las opciones de edición permiten copiar, cortar, y pegar elementos o grupos de elementos que han sido seleccionados previamente, dentro del mismo lienzo o entre lienzo distintos.

Para copiar primero se seleccionan los elementos que serán almacenados en buffer y luego se elige la opción copiar en el submenú de editar.

Para cortar primero se seleccionan los elementos que serán almacenados en un buffer y luego se elige la opción cortar en el submenú de editar. Entonces si un elemento esta seleccionado se eliminan las líneas entre este y cualquier otro elemento al que este enlazado.

Para pegar se requiere que primero se haya realizado una operación de copiado o cortado. Dicha operación almacena los elementos a los que se aplico la operación en un buffer, después se elige la opción pegar en el submenú editar y se elige un lienzo entre los que existan (puede ser el mismo) los elementos almacenados en el buffer aparecen en la misma posición que tenían en el lienzo original.

Acomodar permite ajustar la posición de la esquina superior izquierda de la pseudoventana a la de la esquina superior izquierda de la celda más próxima en la malla.

5.5 Operación de la herramienta del lienzo

Si se elige Descriptores aparece una caja de dialogo, ver fig. 5.7, que contiene una tabla (la tabla de descriptores) que muestra parte de la información de un descriptor como son el nombre del archivo de datos, el numero de registros y el numero de campos. No se muestran los campos por que el numero de estos varia de un descriptor a otro. Hay además, en la caja de dialogo, cuatro botones que tienen los textos siguientes:

Agrega Registro

Agrega Campos

Crea Clase

Entrada

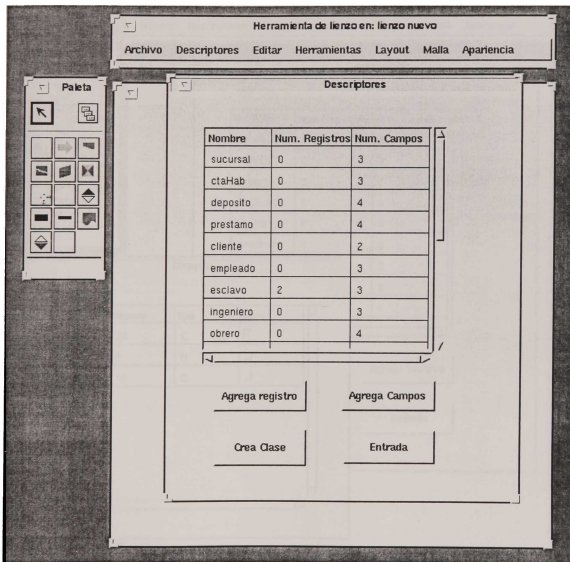


Fig. 5.7 Caja de dialogo que aparece cuando se escoge descriptores en la herramienta del lienzo

Agrega Registro sirve para agregar otro renglón a la tabla de descriptores opera en combinación con Agrega Campos para definir un descriptor el cual se salva en el archivo de descriptores cuando se cierra la caja de dialogo.

Agrega Campos muestra una caja de dialogo, ver fig. 5.8, que contiene una tabla (la tabla de campos, generada dinámicamente de acuerdo al numero de campos) donde se pueden definir o mostrar las características de los campos del descriptor que corresponden al renglón elegido actualmente en la tabla de descriptores.

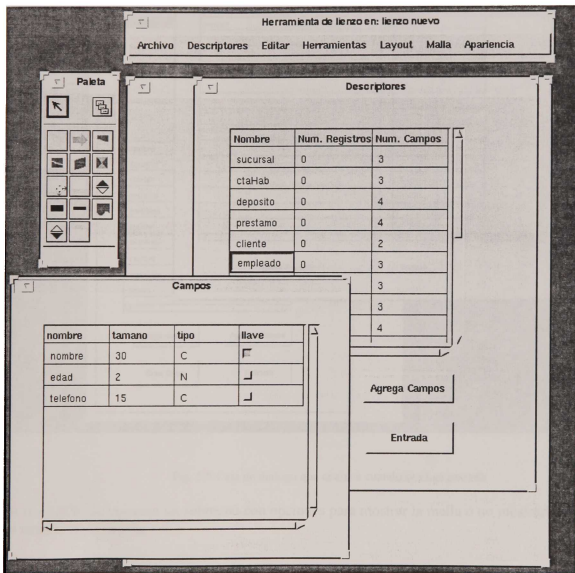


Fig. 5.8 Caja de dialogo que aparece cuando se elige agrega campos

Crea Clase crea en caso que no exista una clase, en SmallTalk, para representar el tipo de registro que describe el descriptor. Dicha clase es necesaria para crear la forma de entrada correspondiente a dicho tipo de registro.

Entrada muestra una caja de dialogo, ver fig. 5.9, que contiene una tabla (la tabla de captura) cuyos renglones sirven para capturar los registros cuyo tipo es descrito por el descriptor elegido en la tabla de descriptores.

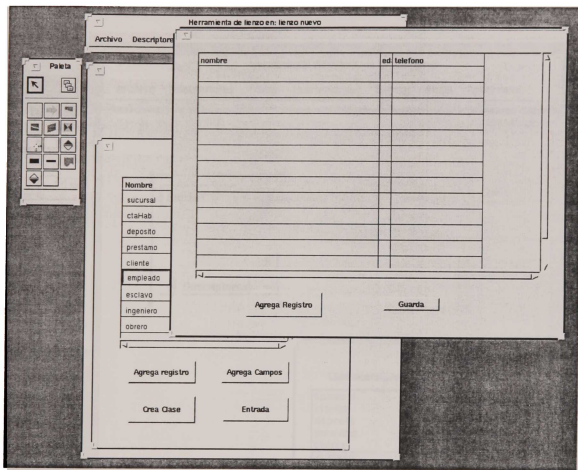


Fig. 5.9 Caja de dialogo que aparece cuando se elige entrada

Si se elige malla aparece un submenu con opciones para mostrar la malla o no mostrarla y para cambiar el tamaño de la misma.

5.6 Capturando información asociada con los elementos del diagrama

Esta sección describe la IGU requerida por la parte lógica de algunos iconos.

5.6.1 Relación

En el caso de una relación la información que se captura es el nombre de la misma para esto se presiona el botón que tiene el texto descriptores después de lo cual aparece una caja de diálogo que contiene una caja de lista, ver fig. 5.10, de donde se pueden elegir entre los descriptores existentes; después de elegir se presiona el botón con el texto OK. La caja de dialogo desaparece y en la pseudoventana aparece una etiqueta, si no la había ya, que muestra la ultima selección realizada.

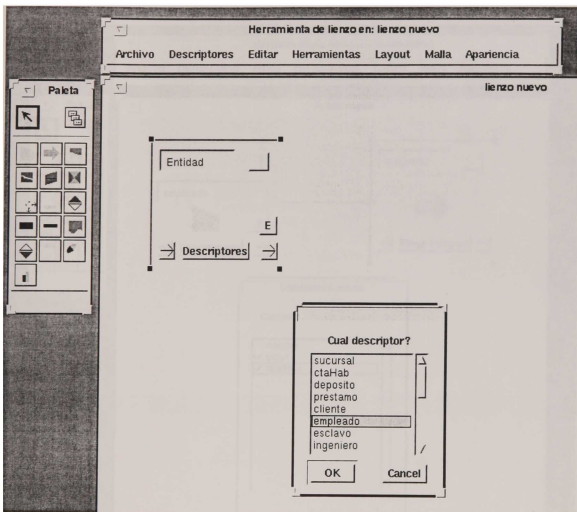


Fig. 5.10 Eligiendo un descriptor de una lista de descriptores

5.6.2 Proyección

Para una proyección la pseudoventana que la representa debe estar unida a otra que le provea de un conjunto de campos esta ultima si no es una relación requiere a su vez quien le proporcione dichos campos. Lo anterior implica una búsqueda recursiva de la(s) relación(s) a partir de las cuales se obtiene dicho conjunto. En caso de que la búsqueda fracase significa que algunas de las operaciones que producen relaciones usadas en la obtención de los campos, para la proyección, no tiene su numero de operandos completo. En caso de éxito se calcula (de acuerdo al álgebra relacional) los campos que resultarían de aplicar las operaciones de las que depende la proyección. Después se despliega una caja de dialogo, ver fig. 5.11, que contiene una lista de los campos calculados, de donde se pueden elegir varios los cuales quedan marcados y si se presiona el botón OK son los que se mostraran marcados cuando aparezca de nuevo la caja de dialogo.

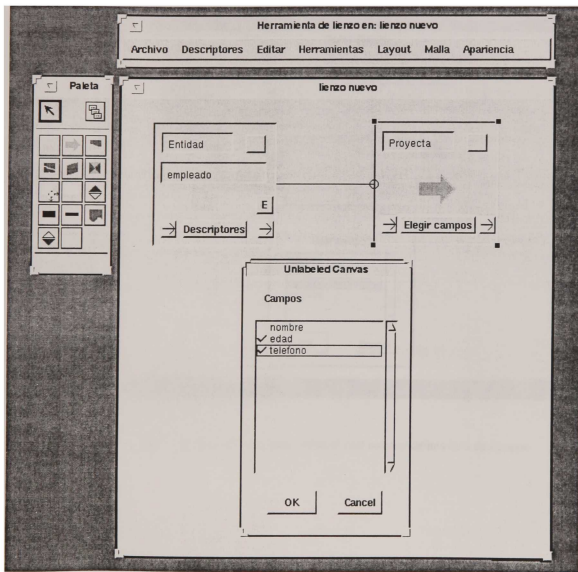


Fig. 5.11 Eligiendo los campos de una lista de campos

5.6.3 Diferencia, Intersección, Junta Natural y Unión

Para la unión, la intersección, junta natural y la diferencia no hace falta capturar información.

5.6.4 Ordenación Ascendente y Descendente

El caso de las ordenaciones es similar al de la proyección a excepción de que solo se puede seleccionar un campo de los disponibles y dicho campo se despliega en una etiqueta de la pseudoventana después de que desaparece la caja de diálogo, ver fig. 5.12, donde se eligió el campo.

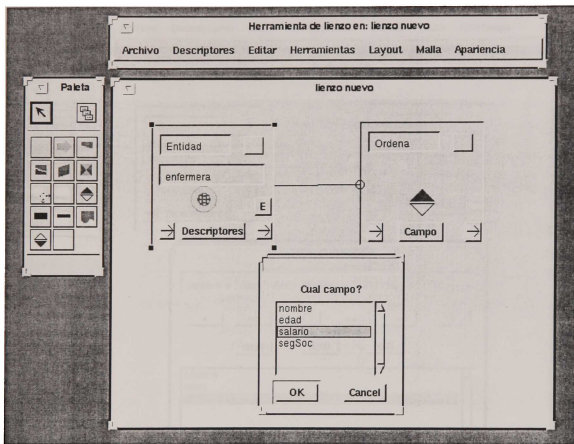


Fig. 5.12 Eliendo un campo sobre el cual ordenar de una lista de campos

5.6.5 Selección

Con la selección se tiene que construir una expresión booleana para ello se necesita capturar información que provee el usuario así como la que se puede obtener a partir del diagrama. Para esto se presiona el botón con la etiqueta expresión. Después de lo cual aparece una caja de diálogo, ver fig. 5.13, donde hay botones que permiten elegir operadores lógicos y relacionales, y una lista de los campos obtenidos en la búsqueda (igual a la realizada en el caso de la proyección). La expresión que se va construyendo se muestra en un campo donde el usuario puede insertar sus datos

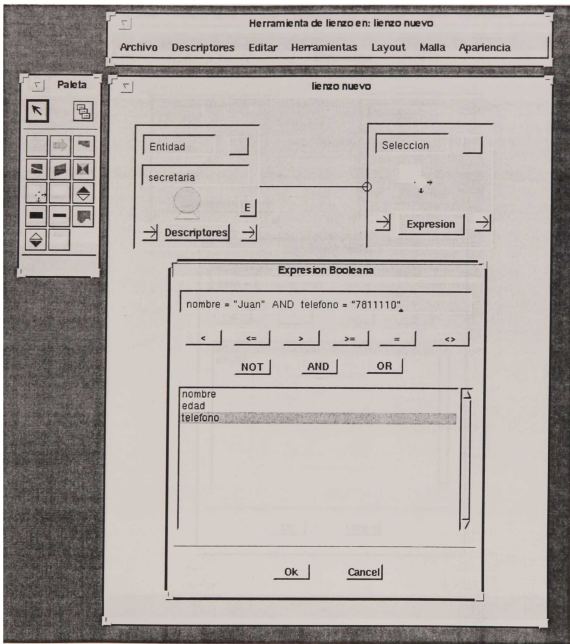


Fig. 5.13 Construyendo una expresión booleana por medio de una caja de dialogo

5.6.6 Proceso

Para el proceso lo que cambia respecto a la selección es que se construye una expresión aritmética por lo tanto la lista de campos contiene solo campos numéricos y los botones deben ser solo para operadores aritméticos, ver fig. 5.14.

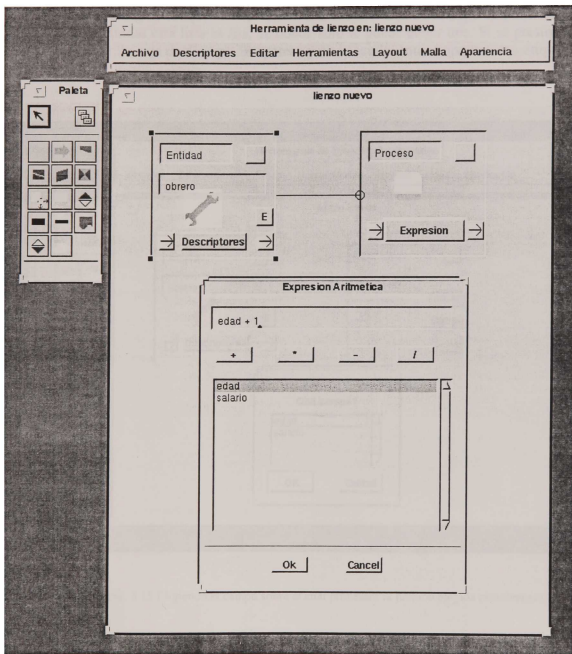


Fig. 5.14 Formando una expresión aritmética por medio de una caja de dialogo

5.6.7 Función

El caso de la función es análogo al de la ordenación lo que cambia es que hay un botón más con el texto función , ver fig. 5.15, que si se presiona aparece una caja de dialogo con una lista de funciones, como son predefinidas esta lista es fija, de donde solo se puede elegir una. Si se presiona el botón OK, después de elegir, la caja de dialogo desaparece y en la pseudoventana aparece una etiqueta que muestra el nombre de la función elegida.

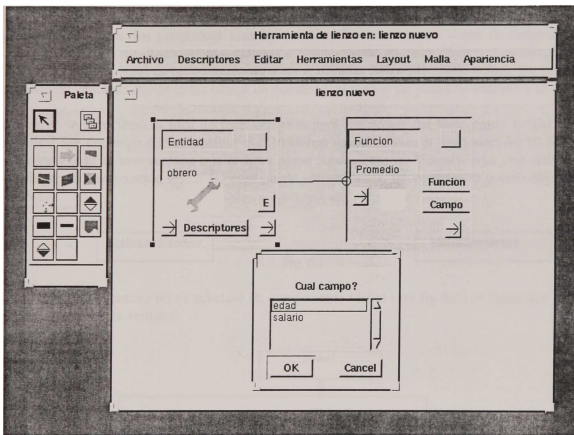


Fig. 5.15 Eligiendo el campo sobre el cual procesara la función elegida previamente

CAPITULO 6

SEUDOVENTANAS

Un elemento importante de las funciones de nuestro editor es su organización en base a pseudoventanas las cuales mencionamos solo superficialmente en el capítulo anterior. En este capítulo describimos su diseño e implementación.

Nuestras pseudoventanas fueron motivadas por el manejo de los bloques visuales de GemStone de Servio Corporation el cual es un Sistema Gestor de Base de Datos Orientado a Objetos que permite programación visual mediante herramientas de construcción de flujogramas [15]. Pero a diferencia de LIDA lo que fluye por las líneas de flujo de datos son campos en lugar de relaciones. Esto por que el propósito básico de los programas visuales (en GemStone) es tomar datos de campos, actuar en los datos, y enviar los resultados a otros campos o almacenarlos en otros objetos. Los bloques visuales son la representación visual de las operaciones a ser realizadas en un programa visual. Dicha representación consiste en ventanas donde todas tienen un nombre y al menos un punto de entrada y uno de salida.

Como ya se mencionó la idea de usar ventanas para los nodos del flujograma surgió de los bloques visuales. Sin embargo no fue posible usar ventanas como se vera a continuación. El flujograma debe estar contenido en una ventana que le sirva como fondo. Pero en VisualWorks una ventana solo puede desplegar un componente visual (el cual puede ser una vista, un compuesto u otro objeto visual) , ver fig. 6.1,

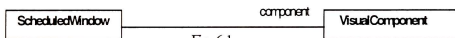


Fig. 6.1

y como la clase ventana no es subclase de componente visual (ver fig. 6.2) se sigue que una ventana no puede contener otra ventana.

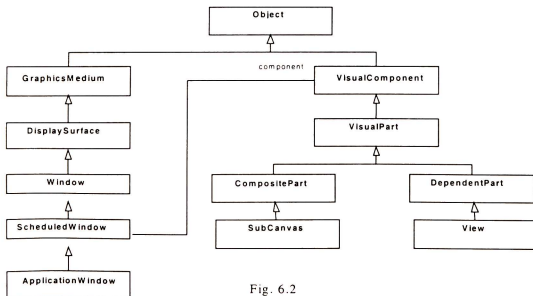


Fig. 6.2

6.1 Características de nuestras seudoventanas

Los iconos de nuestro editor de flujogramas son iconos generalizados: tienen una parte física (la imagen) representada por un pixmap y una parte lógica (el significado) la cual puede tener asociada una interfaz gráfica de usuario para facilitar su manejo.

Denominamos a las seudoventanas así porque son componentes visuales (sublienzo) que deben desplegarse dentro de una ventana, que sirve de fondo para el flujograma, y simular aspectos siguientes de comportamiento de una ventana: movilidad, iconización y respuesta de sus controles a las entradas de usuario.

Las seudoventanas cuentan al menos con los controles siguientes: una etiqueta de texto para el título de la ventana (donde va el nombre del tipo de elemento), un botón para iconizar, una etiqueta gráfica (la parte física) y dos botones uno para activar el modo de enlace y otro para desactivarlo, ver fig. 6.3. Las seudoventanas que poseen controles adicionales son las que tienen una IGU para la parte lógica.

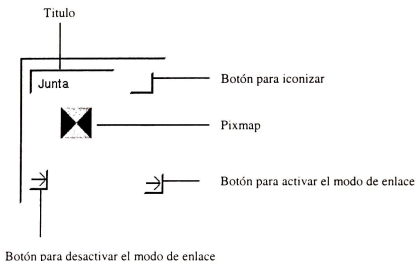


Fig. 6.3 Mínimo de controles en una seudoventana

Si el usuario no necesita la IGU asociada con la parte lógica puede presionar el botón de minimizar entonces la seudoventana es sustituida por otra que solo contiene un botón el cual tiene como etiqueta gráfica el pixmap (reducido) de la primera. La seudoventana sustituta es del tamaño del botón que contiene por tanto es más chica que la seudoventana que reemplaza. De aquí que si existe(n) línea(s) entre la seudoventana iconizada y otra(s) seudoventana(s) dicha(s) línea(s) se redibuja(n). Lo anterior puede proporcionar un diagrama donde lo esencial de la parte física de los iconos sea lo que destaque y el usuario no se sienta abrumado por los detalles pudiendo concentrarse en los aspectos generales de la consulta.

6.2 Clases usadas en la implementación de las pseudoventanas

Cada tipo de pseudoventana usa dos clases descendientes de `ApplicationModel`. Una para la presentación normal de la pseudoventana y otra para la iconizada. La primera es subclase directa de `AtomoVent` y la segunda de `AtomoVentIco`. `AtomoVent` y `AtomoVentIco` son ambas subclases directas de `ApplicationModel`, ver fig 6.4.

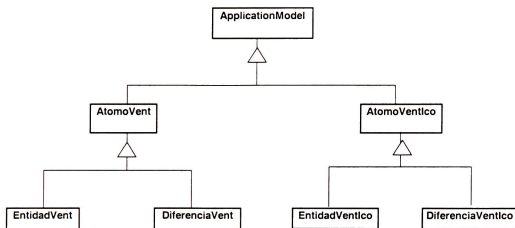


Fig. 6.4 Diagrama de clases de las pseudoventanas

Una instancia de `AtomoVent` almacena el índice del elemento, de la representación interna del flujograma, al que representa gráficamente.

El mecanismo de `VisualWorks` más simple para lograr dicho comportamiento es el de reuso de aplicaciones. Pues dicho mecanismo permite desarrollar la interfaz de usuario de una aplicación en un lienzo y luego integrarlo como sublienzo en otro lienzo.

Cuando una instancia de `AtomoVent` necesita localizar al elemento que representa visualmente busca un índice que coincida con el suyo en la colección de elementos referenciada por la instancia del diagrama referenciada en el pintor.

Si se enlazan dos pseudoventanas la pseudoventana que hace el papel de fuente almacena una referencia al elemento que representa visualmente en la variable de instancia del pintor llamada `operando` y la pseudoventana que hace el papel de destino puede acceder al elemento que referencia la fuente por medio de dicha variable.

6.3 Estructura de las pseudoventanas en el lienzo

El builder del pintor (un `UIBuilder`) tiene una ventana (instancia de `ScheduledWindow`) cuyo componente es un `GridWrapper` en cual contiene la vista del Pintor (una `UIPintorView`) por ser instancia de una subclase de `CompositePart` mantiene una colección (llamada `components`) de wrappers (instancias de `SpecWrapper`) donde cada uno de estos contiene un `BorderedWrapper` (el cual dibuja el borde de la pseudoventana) el cual contiene a su vez un `WidgetStateWrapper` (usado para unir una `CompositePart` con un `WidgetState`) este a su vez contiene un sublienzo (instancia de `SubCanvas`).

Un sublienzo puede contener la IGU de una pseudoventana en la colección de wrappers (de nuevo instancias de SpecWrapper) que contiene, ver fig. 6.5.

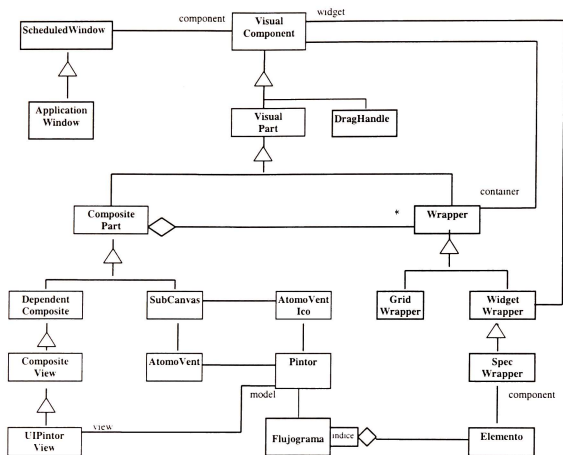


Fig 6.5 Diagrama de clases de las pseudoventanas en el lienzo

Lo anterior se aprecia mejor en el diagrama de instancias (obtenido a partir del diagrama de clases anterior, ver figuras 6.5, 6.6 y 6.7) de un lienzo que contiene una pseudoventana seleccionada (una pseudoventana seleccionada tiene cuatro cuadraditos negros, llamados asas, en cada esquina) y una sin seleccionar. Como se puede ver además de los dos SpecWrappers para las dos pseudoventanas hay cuatro DragHandles (las asas). La prolongación de la rama marcada con asterisco, que corresponde a una pseudoventana de tipo entidad, se muestra en un diagrama contiguo.

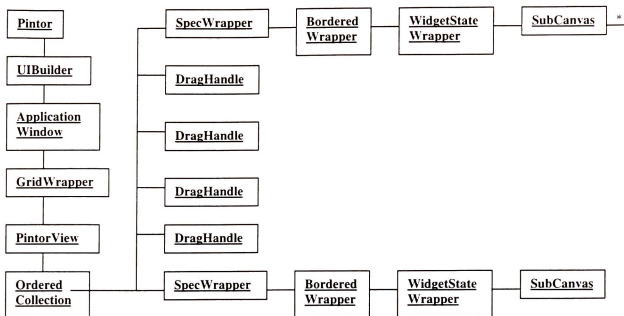


Fig. 6.6 Diagrama de instancias de dos pseudoventanas en el lienzo

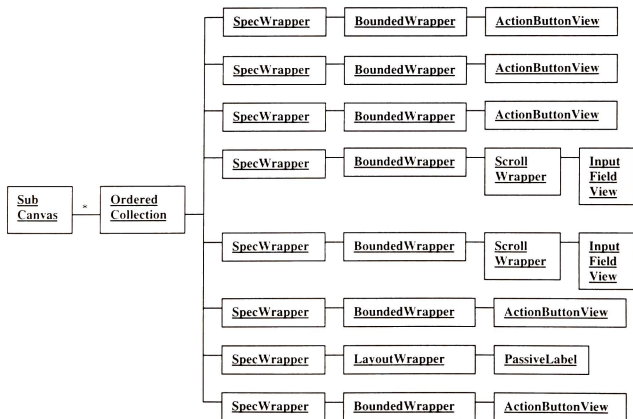


Fig. 6.7 Continuación de la rama del diagrama anterior marcada con asterisco

La seudoventana correspondiente al tipo de elemento, del álgebra relacional, entidad junto con los nombres de las clases de los widgets que la componen se muestra a continuación.

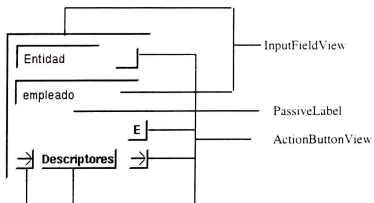


Fig. 6.8 Seudoventana y los nombres de las clases de sus controles

Como se puede ver en la figura 6.8 dicha seudoventana esta constituida por: dos InputFieldViews, una PassiveLabel y cinco ActionBarViews.

El SubCanvas que contiene la IGU de la seudoventana referencia a la colección ordenada siguiente:

```

OrderedCollection(
  a SpecWrapper on: a BoundedWrapper on: a ActionBarView
  a SpecWrapper on: a BoundedWrapper on: a ActionBarView
  a SpecWrapper on: a BoundedWrapper on: a ActionBarView
  a SpecWrapper on: a BoundedWrapper on: a ScrollWrapper on: a InputFieldView
  a SpecWrapper on: a BoundedWrapper on: a ScrollWrapper on: a InputFieldView
  a SpecWrapper on: a BoundedWrapper on: a ActionBarView
  a SpecWrapper on: a LayoutWrapper on: a PassiveLabel
  a SpecWrapper on: a BoundedWrapper on: a ActionBarView )
    
```

6.4 Estructura de la especificación de una seudoventana

La especificación de una seudoventana es un FullSpec. La parte component del FullSpec se usa para describir la IGU de la seudoventana. Como se puede ver a partir de la forma literal de la especificación de la IGU de la seudoventana del tipo entidad la estructura de dicha especificación es semejante a la del sublienzo, que contiene la IGU. Pero en este caso a la clase de una vista corresponde una clase de una especificación. Así en lugar de una ActionBarView se tiene una ActionBarSpec, en lugar de una InputFieldView se tiene una InputFieldSpec y en lugar de una PassiveLabel se tiene una LabelSpec.


```

#(#FullSpec
  #window:
  #(#WindowSpec)
  #component:
  #(#SpecCollection
    #collection: #(
      #(#ActionButtonSpec)
      #(#LabelSpec)
      #(#ActionButtonSpec)
      #(#InputFieldSpec)
      #(#InputFieldSpec)
      #(#ActionButtonSpec)
      #(#ActionButtonSpec)
      #(#ActionButtonSpec)
    )
  )
)

```

Forma literal de la especificación de una pseudoventana de tipo entidad

6.5 Iconización de pseudoventanas

El mecanismo de VisualWorks de reuso de aplicaciones permite desarrollar la interfaz de usuario de una aplicación en un lienzo y luego integrarlo como sublienzo en otro lienzo, ver fig. 6.9.

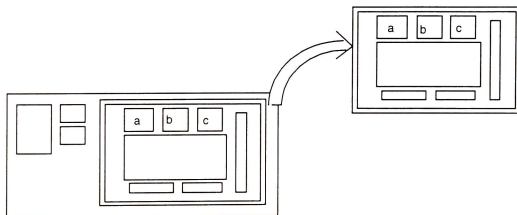


Fig. 6.9 Un nuevo lienzo se construye insertando uno previo y referenciando modelos de aplicación y modelos de dominio existentes

Además el mecanismo de reuso de aplicaciones permite cambiar, en tiempo de ejecución, la especificación de la IGU actual en el sublienzo por otra.

Por tanto para iconizar una pseudoventana cuando se presiona su botón de iconización una instancia de una subclase de `AtomoVent` hace lo siguiente:

Obtiene la especificación de la pseudoventana iconizada a partir de una subclase de AtomoVentIco. Y crea una instancia de dicha subclase donde almacena su referencia al pintor, su índice y su referencia a la instancia de la clase SubCanvas que usa la especificación de la pseudoventana normal. Después envía el mensaje `client:spec:` a la instancia de clase SubCanvas para sustituir la especificación de la pseudoventana normal por la especificación de la pseudoventana iconizada, ver fig 6.10.

Para desiconizar una pseudoventana cuando se presiona su botón de restauración una instancia de una subclase de AtomoVentIco hace lo siguiente:

Obtiene la especificación de la pseudoventana normal a partir de una subclase de AtomoVent. Y crea una instancia de dicha subclase donde almacena su referencia al pintor, su índice y su referencia a la instancia de la clase SubCanvas que usa la especificación de la pseudoventana iconizada. Después envía el mensaje `client:spec:` a la instancia de clase SubCanvas para sustituir la especificación de la pseudoventana iconizada por la especificación de la pseudoventana normal ver figura 6.10.

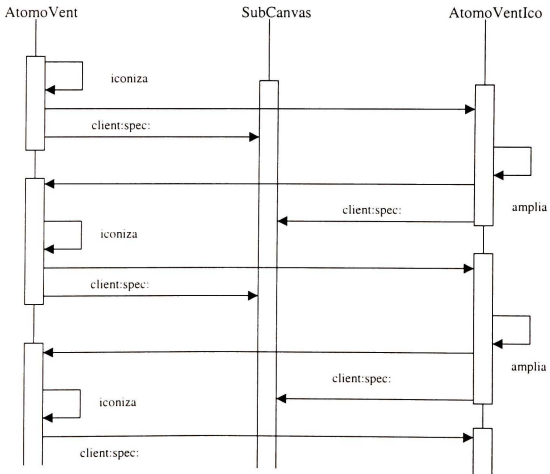


Fig. 6.10 Diagrama de interacción

CAPITULO 7

IMPLEMENTACION DE NUESTRO EDITOR DE FLUJOGRAMAS

La implementación de nuestro editor de flujogramas se basa en la modificación del editor gráfico de VisualWorks. El diagrama siguiente, ver fig. 7.1, muestra la jerarquía de clases de VisualWorks más usadas en el editor de flujogramas.

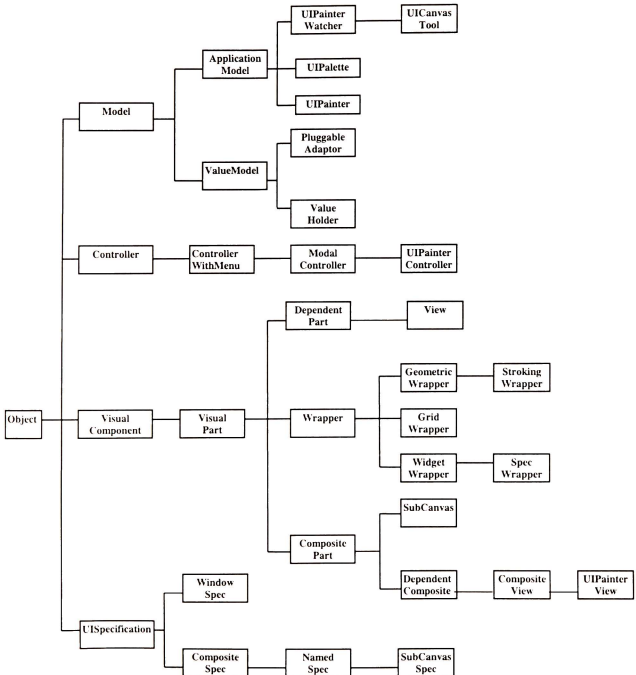


Fig. 7.1

Debido a que nos basamos en el código del del editor gráfico de VisualWorks se modificaron las clases principales que componen dicho editor gráfico. De aquí que las clases UIPalette, UIPainter, y UIPainterController sean sustituidas por Paleta, Pintor y PintorController respectivamente.

7.1 Pintor

Pintor es la clase más importante, crea la ventana (el lienzo) donde se dibuja y edita el flujograma. La ventana de la Paleta y la de la Herramienta del Lienzo son satélites de esta.



Fig. 7.2 Arquitectura MVC en el editor de flujogramas

7.2 PintorController

PintorController es el controlador del pintor, ver fig. 7.2, y es en esta clase donde muchas de las aplicaciones ofrecidas por el pintor son implementadas. También ofrece un menú de aparición súbita para el lienzo.

PintorController es subclase de ModalController la cual a su vez es subclase de ControllerWithMenu. Una instancia de ModalController redirecciona eventos relacionados con el ratón a un objeto ControlModo, almacenado en su variable currentModo, el cual puede entonces realizar alguna acción. Cuando no delega el control procede a su actividad normal de los botones select y operate.

La relación entre las clases PintorController y ControlModo es un ejemplo del patrón de diseño estrategia. En este caso PintorController es el contexto que delega peticiones de sus clientes a la instancia de ControlModo. La clase abstracta ControlModo es la estrategia, en el patrón de diseño estrategia, y es donde se declara la interfaz común a todos los modos.

7.3 Clases de control

Las clases SelectModo, SelectionDragModo y DragPlacementModo, derivadas de ControlModo, son las estrategias concretas (en el patrón de diseño estrategia) utilizadas por las instancias de PintorController, ver fig. 7.3.

Cada subclase concreta de ControlModo sirve para implementar una estrategia particular de respuesta. Se puede cambiar a una estrategia diferente reemplazando una instancia de una subclase de ControlModo por una instancia de otra subclase de ControlModo.

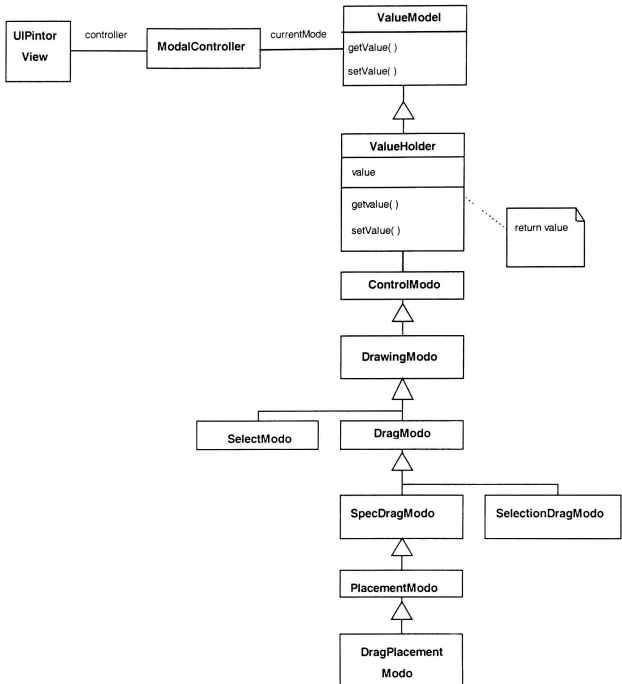


Fig. 7.3 Diagrama de clases del control del lienzo

7.3.1 SelectModo

SelectModo es el modo de control usado para seleccionar entidades en el lienzo.

El comportamiento dinámico de esta clase es el siguiente:

Si alguna entidad fue tocada por la pulsación inicial del botón del ratón esta es seleccionada.

Si ninguna entidad fue tocada y el botón ya no esta presionado entonces se remueven todas las selecciones.

Si se mueve el ratón con el botón presionado entonces

Si la entidad fue tocada por la pulsación original entonces se realizara el arrastre de dicha entidad.

Si ninguna entidad fue tocada entonces se realizara una selección de todas aquellas entidades que estén en el área definida por el movimiento del ratón mientras se presiona el botón.

La actividad dragObject:StartingAt:inController: se expande en el diagrama de estado de la clase SelectionDragModo.

El diagrama de estado de SelectModo se muestra en la fig. 7.4

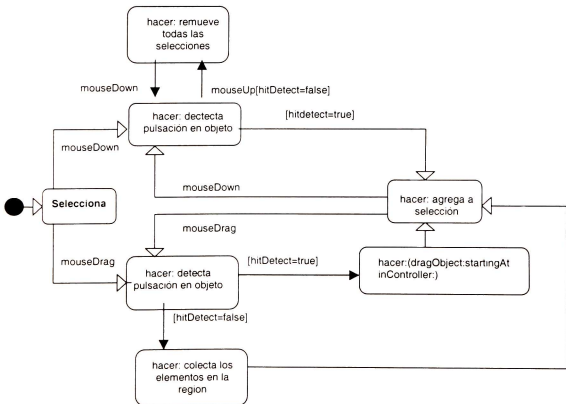


Fig. 7.4

7.3.2 SelectionDragModo

SelectionDragModo es el modo de arrastre usado para repositionar una pseudoventana en el lienzo cuando la pseudoventana completa esta siendo arrastrada.

El comportamiento dinámico de esta clase es el siguiente: mientras se mueva el ratón y se presiona el botón si el desplazamiento en x es mayor que el desplazamiento en y entonces se mueve horizontalmente la entidad seleccionada. Si el desplazamiento en y es mayor que el desplazamiento en x entonces se mueve verticalmente la entidad seleccionada.

El diagrama de estado de SelectionDragModo se muestra en la fig. 7.5.

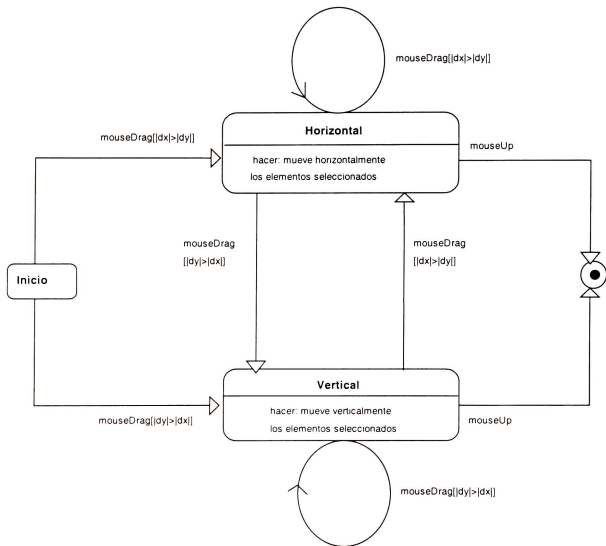


Fig. 7.5

7.3.3 DragPlacementModo

DragPlacementModo es la clase de modo de colocación (PlacementModo) con el cual la nueva pseudoventana aparece cuando el cursor del ratón entra en el lienzo, se arrastra con todos los botones del ratón arriba.

Cuando el cursor del ratón entra en el lienzo se agrega el subcanvas (que el objeto UIBuilder creo a partir de la especificación de la IGU de la pseudoventana, correspondiente al tipo de elemento elegido en la paleta) a la vista del pintor. Lo anterior ocasiona que se dibuje la IGU de la pseudoventana en el lienzo. Mientras se mueva el cursor del ratón dentro del lienzo el dibujo de la IGU de la pseudoventana se mueve (PintorView move:). Si el cursor del ratón sale del lienzo se remueve el subcanvas de la vista del pintor lo que provoca que el dibujo de la IGU de la pseudoventana desaparezca (PintorView removeComponent:).

Si se presiona el botón <select> del ratón y el modo de la paleta no es de repetición de selección se cambia el modo de control en la paleta a SelectModo.

Si se presiono el botón y el modo de la paleta es de repetición de selección entonces se agrega el subcanvas a la vista del pintor. Si después se mueve el ratón o el cursor de este sale del lienzo se realizan las mismas acciones que se mencionaron antes para dichos eventos.

El diagrama de estado de DragPlacementModo se muestra en la fig. 7.6.

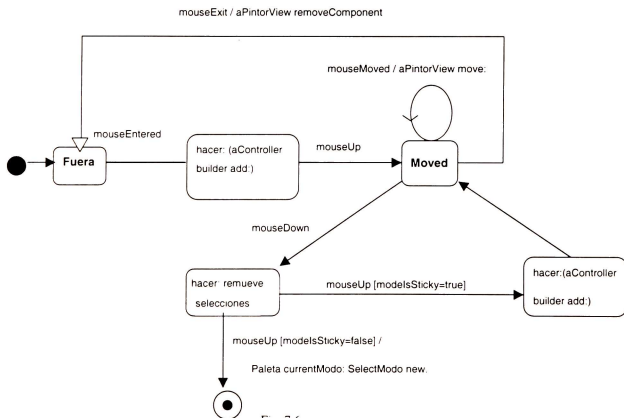


Fig. 7.6

7.4 Diseño de la Paleta

La discusión que sigue se basa en la fig. 7.7. A cada tipo de pseudoventana que puede ser pintada en el lienzo corresponde un botón que sirve para elegirlo en la paleta y una clase de especificación que describe su interfaz de usuario.

Cada clase de especificación es subclase de SubCanvasSpec e implementa los mensajes: specGenerationBlock, paletteIcon y componentName. El mensaje specGenerationBlock genera una instancia de dicha subclase. El mensaje paletteIcon regresa la imagen que representa la especificación en la paleta. El mensaje componentName regresa el nombre del tipo de elemento.

La variable de clase ActiveSpecsList referencia una colección de nombres simbólicos de clases de especificación y la variable de instancia activeSpecs referencia una colección de clases de especificación.

Cuando se inicializa la clase Paleta se agregan los nombres simbólicos de las clases de especificación a ActiveSpecsList. Después cuando se inicializa una instancia de la clase Paleta se usan los nombres almacenados en ActiveSpecsList para obtener las clases de especificación y agregarlas a activeSpecs.

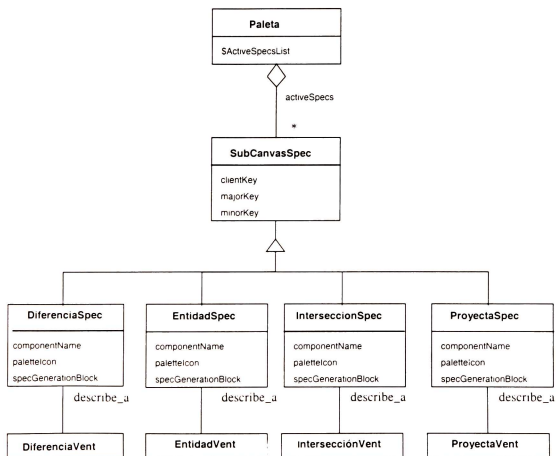


Fig. 7.7 Relación de la clase Paleta con las clases de las especificaciones de las pseudoventanas

Los cambios finales a la interfaz de la paleta antes de su apertura se hacen con el método `postBuildWith`: el cual invoca al método `populatedSpec`: para crear los botones de selección de tipo de pseudoventana. La especificación de uno de esos botones se genera dinámicamente pero debe de ser adaptada de acuerdo al tipo de pseudoventana que le corresponde. La adaptación se hace antes de que el builder de la paleta agregue la especificación a la interfaz. La adaptación consiste en asignarle al botón una imagen y un `PluggableAdaptor` en un `ValueHolder` como modelo, cuando se arma su especificación. En el patrón de diseño adapter el botón es el cliente, el `PluggableAdaptor` el adapter y el adptee es el `ValueHolder` (`CurrentMode`) cuyo value se establece a una instancia de una subclase de `ControlModo` (en este caso `DragPlacementModo`) en dicha instancia se almacena el bloque que regrese el método `specGenerationBlock` de la especificación de la pseudoventana a la que corresponde el botón, ver figuras 7.8 y 7.9.

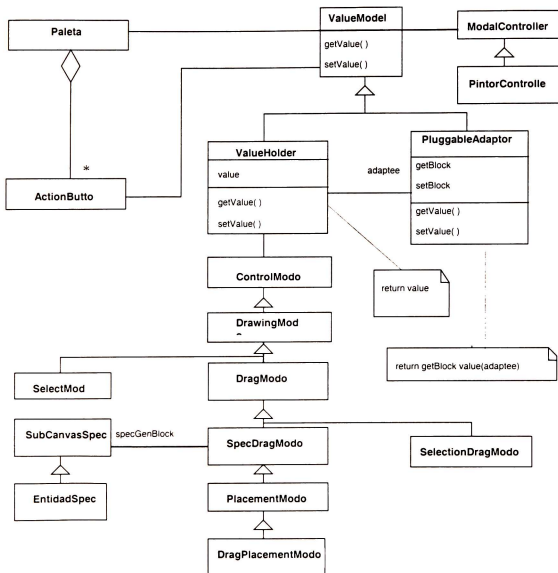


Fig. 7.8

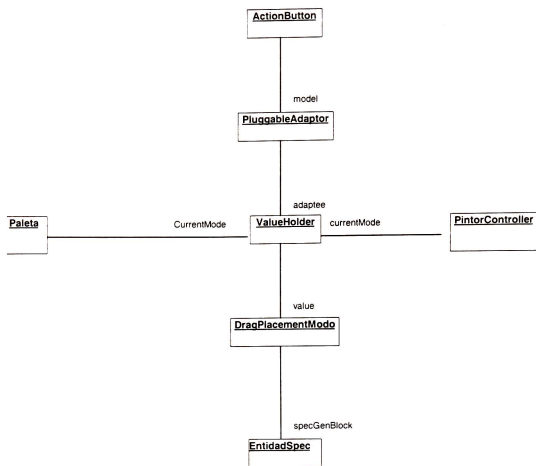


Fig. 7.9 Diagrama de instancias del editor de flujogramas

En la parte superior de la paleta están los botones de selección simple y múltiple.

La Paleta puede estar en uno de dos modos de selección: simple o múltiple. Cada modo se activa con el botón correspondiente.

En el modo simple para pintar el mismo tipo de pseudoventana varias veces se tiene que elegir dicho tipo en la paleta cada vez y en el múltiple solo la primera vez.

El comportamiento de la paleta es el siguiente

Cuando inicia su ejecución el modo de selección es simple y el de control de esta es una instancia de SelectModo.

Si se presiona un botón de selección de tipo de pseudoventana entonces se crea una instancia de la clase DragPlacementModo y se envía el mensaje value: a la variable de clase CurrentMode con la instancia creada como argumento.

Si cuando se presiono el botón

a) no se presiona la tecla shift y el modo de control no es SelectModo

entonces se activa el modo de selección simple

b) se presiona la tecla shift entonces se activa el modo de selección múltiple.

Si se presiona nuevamente un botón de selección de tipo de pseudoventana entonces se crea una instancia de la clase SelectModo y se envía el mensaje value: a la variable de clase CurrentMode con la instancia creada como argumento.

Si se presiona el botón de selección simple se crea una instancia de la clase SelectModo, se envía el mensaje value: a la variable de clase CurrentMode con la instancia creada como argumento y se activa el modo de selección simple.

Si se presiona el botón de selección múltiple el modo actual de selección se inactiva y el otro se activa.

El diagrama de estado de paleta se muestra en la fig. 7.10.

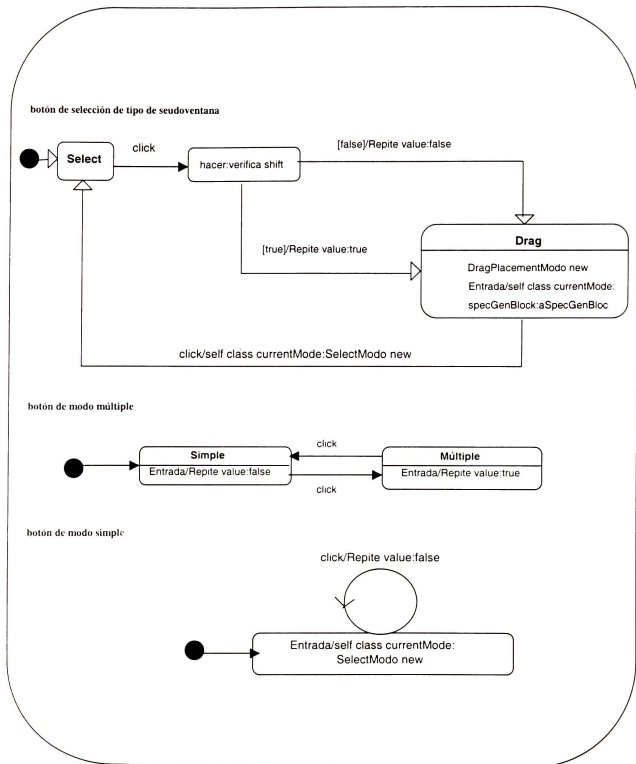


Fig. 7.10

7.5 La secuencia de control en el editor de flujogramas

A partir de que el cursor del ratón entra en el lienzo la secuencia de control es la siguiente:

El controlador de la ventana (instancia de `ApplicationStandardSystemController`) pregunta a dicha ventana (instancia de `ScheduledWindow`) cual de sus componentes quiere control, mediante el mensaje `subViewWantingControl`. En respuesta a este mensaje, la ventana del pintor requiere `objectWantingControl` de su componente, una instancia de `GridWrapper`. Dicho `GridWrapper` reenvía el mensaje a su único componente, una instancia de `PintorView`. El objeto de la clase `PintorView` después de recibir este mensaje pregunta a su controlador, una instancia de `PintorController`, `isControlWanted`. Dado que este controlador responde true se le envía el mensaje `startUp`, lo cual inicia su secuencia de control, ver fig. 7.11.

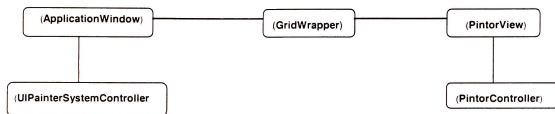


fig. 7.11

Esta instancia de `PintorController` delega a su vez el control al modo de control el cual es `DragPlacementModo`. El modo de control genera una instancia de una subclase de `SubCanvasSpec`. De esta última instancia se usa el nombre simbólico almacenado en su variable `clientKey` como argumento del mensaje `perform` que se envía al pintor para que ejecute el método que crea la instancia de la subclase de `AtomoVent` correspondiente a esta especificación. El pintor importa la interfaz de usuario de la instancia creada al sublienzo.

7.6 La secuencia de control en la respuesta de los controles a la entrada del usuario

Si en el lienzo se ha dibujado únicamente una pseudoventana del tipo entidad y esta fue seleccionada entonces la vista del pintor (instancia de `PintorView`) contiene los cinco componentes siguientes: un `SpecWrapper` y cuatro asas (ver fig. 6.7).

Si en la situación anterior se presiona el botón izquierdo del ratón y el modo de control es `SelectModo` entonces lo primero que se hace es enviar el mensaje `subViewWantingControl` a la instancia de `PintorView`. La instancia de `PintorView` invoca el método `componentWantingControl` en el cual envía el mensaje `objectWantingControl` a cada uno de sus componentes. Cuando el componente `SpecWrapper` recibe el mensaje lo reenvía a su componente, un `BorderedWrapper`, el cual lo reenvía a su componente, un `WidgetStateWrapper`, el cual lo reenvía a su componente un `SubCanvas` el cual invoca el método `componentWantingControl` en el cual envía el mensaje `objectWantingControl` a cada uno de sus subcomponentes (en este caso wrappers para cinco botones, dos campos de entrada y una etiqueta pasiva ver fig. 6.7).

Cuando un componente que es un `SpecWrapper` que envuelve un botón recibe el mensaje lo reenvía a su componente un `BoundedWrapper`, el cual lo reenvía a su componente, una `ActionButtonView`, el cual envía el mensaje `isControlWanted` a su controlador, un `TriggerButtonController`. Dicho controlador toma el control si comprueba que dicha vista (instancia de `ActionButtonView`) contiene el cursor del ratón.

CONCLUSIONES Y TRABAJO FUTURO

8.1 Conclusiones

Esta tesis presentó un editor de flujogramas flexible que permite: Mover el cursor de edición un número arbitrario de unidades. Colocar por primera vez o reubicar iconos en el lienzo con solo arrastrar y soltar. Aumentar el número de nodos en el diagrama mientras haya todavía memoria disponible en el sistema. Superponer una cuadrícula al lienzo para alinear los nodos del diagrama. Agrupar varios nodos del diagrama para copiarlos, cortarlos, pegarlos, moverlos y guardarlos como si fueran un solo nodo.

Nuestro editor fue diseñado con UML y patrones de diseño, e implementado con VisualWorks por que permite desarrollar aplicaciones basadas en la arquitectura MVC de Smalltalk la cual hace un uso intensivo de dichos patrones. La aplicación de dichas herramientas hacen a nuestro editor fácilmente integrable y extensible. Para mostrar la integrabilidad de nuestro editor lo integramos con LIDA. Anteriormente LIDA no podía:

Mover el cursor de edición un número arbitrario de unidades. Colocar por primera vez o reubicar iconos en el lienzo con solo arrastrar y soltar. Aumentar el número de nodos en el diagrama mientras haya todavía memoria disponible en el sistema. Superponer una cuadrícula al lienzo para alinear los nodos del diagrama. Agrupar varios nodos del diagrama para copiarlos, cortarlos, pegarlos, moverlos y guardarlos como si fueran un solo nodo.

Nuestro editor se integro con LIDA, y debido a esto es más flexible ahora como se muestra en el capítulo 5.

8.2 Trabajo Futuro

La misma estructura hace nuestro sistema extensible. Por ejemplo, sería conveniente integrarlas funciones de edición de iconos. Pues se recordará que:

Un sistema para desarrollar programas en un lenguaje visual consiste de un Diccionario de Iconos, un Diccionario de Operadores, una BDAS y conceptualmente dos editores. Uno de los editores se utiliza para definir y editar los iconos del lenguaje visual. Este proceso implícitamente crea y actualiza el Diccionario de Iconos, el Diccionario de Operadores, y la BDAS. La definición/ edición de iconos especifica/ modifica los atributos de los iconos tales como forma, color, tamaño, etcétera. La función del otro editor es editar los valores de los atributos de los iconos, y manipular los iconos disponibles para especificar un programa visual. A este editor nos referiremos como editor de flujogramas, o distribuciones espaciales de iconos interconectados donde los iconos representan operaciones sobre datos y las interconexiones el flujo de los mismos.

El primer editor mencionado podría ser el editor de un Lenguaje de Definición de Iconos o un subsistema sencillo de edición de iconos como el de la figura 8.1. En dicho subsistema para describir la parte física solo se usará un pixmap. Dicho pixmap se puede crear en cualquier editor gráfico que permita la adquisición de pixmaps, mediante un scanner, y su edición.

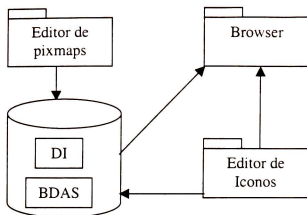


Fig. 8.1 Arquitectura del subsistema simple de edición de iconos

La parte lógica se definirá en un editor de iconos el cual permitirá al usuario recuperar de disco la parte física y la lógica de los iconos mediante un browser. En el editor de iconos el usuario podrá capturar los nombres, los tipos y los valores por default de los atributos de los iconos. Además se podrán capturar las acciones semánticas de los iconos las cuales se almacenaran como cadenas en la BDAS ver figura 8.2

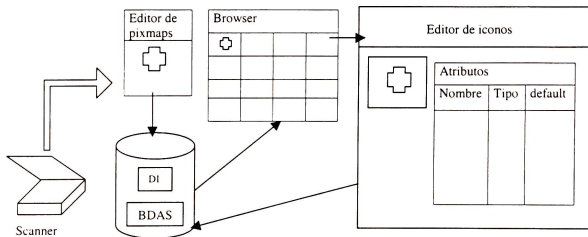


Fig. 8.2 Subsistema simple de edición de iconos

El estilo del segundo editor se podría hacer más similar a los que son prácticamente un estándar de facto para el desarrollo rápido de aplicaciones, como Visualworks, Visual Basic, Delphi, Visual Age, etcétera, usando hojas de propiedades en lugar de pseudoventanas. Por tanto el editor de flujogramas constaría de los módulos siguientes: la paleta, el lienzo, los controladores, la hoja de propiedades, y el traductor, ver figura 8.3.

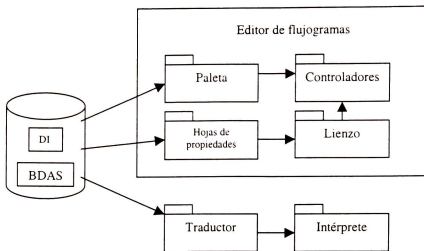


Fig. 8.3 Arquitectura del editor de flujogramas incluyendo hojas de propiedades en lugar de pseudoventanas

La paleta es donde se eligen los iconos que aparecerán en el diagrama que constituye un programa del LFD (flujograma) y es el lienzo donde se editan y ejecutan estos flujogramas. En lugar de las pseudoventanas se tendrían hojas de propiedades para establecer los valores de los atributos de la parte lógica del icono que este seleccionado en el flujograma editado en el lienzo.

Si se integra el subsistema de edición de iconos sencillo con la codificación mencionada a nuestro editor de flujogramas este funcionaría como se muestra a continuación:

Del Diccionario de Iconos se tomarían los atributos de las partes lógicas de los iconos para crear las hojas de propiedades y los pixmaps (las partes físicas de los iconos) que aparecerán en la paleta.

La ejecución de un flujograma se inicia a partir de un conjunto de nodos terminales (que son nodos que reciben pero no proporcionan datos) el cual esta ordenado por prioridad. Los terminales de dicho conjunto se ejecutan desde la prioridad más alta hasta la más baja. Cuando se ejecuta uno de dichos terminales los datos fluyen en dos direcciones primero en un sentido y luego en el opuesto. Dicho terminal puede requerir un dato que sea el resultado de una operación la cual para realizarse necesita obtener los datos de sus operandos. Pero si alguno de esos operandos es el resultado de una operación (está a su vez necesita los datos de sus operandos) esto se puede repetir hasta llegar a operandos que no sean resultado de una operación. A partir de entonces el flujo de datos es en sentido contrario pues cada que un nodo (del flujograma cuyo icono asociado representa una operación) dispone de todos sus operandos se genera una cadena (en un lenguaje intermedio) que se pasa a un intérprete el cual produce como resultado de la operación otra cadena que puede ser utilizada como operando por otra operación. La cadena que se da al intérprete la construye un módulo traductor sustituyendo (en la cadena que se obtiene de la BDAS y la cual es la interpretación del icono asociado con el nodo) los operandos formales y los nombres de los atributos de la parte lógica del icono por sus valores en dicho nodo.

De lo dicho en el párrafo anterior se sigue que por cada lenguaje de flujo de datos del que se quieran ejecutar programas se necesitan además del Diccionario de Iconos y la BDAS un traductor y un intérprete. Nuestra arquitectura requiere además que quien defina el lenguaje proporcione funciones para generar las cadenas del lenguaje intermedio que pueda llamar al traductor y un intérprete para dicho lenguaje intermedio.

BIBLIOGRAFÍA

- 1-Sergio V. Chapa Vergara
Programación Automática a Partir de Descriptores de Flujo de Información
Tesis doctoral 1991
CINVESTAV-IPN-México
- 2-Noé Sierra Romero
HEVICOP: Herramienta visual para la construcción de programas
Tesis de maestría 1995
CINVESTAV-IPN-México
- 3-Pedro E. Alday Echavarría
Diseño de base de datos con evex entidad vinculo extendido para Xwindows
Tesis de maestría 1997
CINVESTAV-IPN-México
- 4-Laura Méndez Segundo
DALI: Herramienta para la representación gráfica de información
Tesis de maestría 1998
CINVESTAV-IPN-México
- 5- Ju-Shi Guang
Visualización de una base de datos espacial
Tesis doctoral 1996
CINVESTAV-IPN-México
- 6-N.C. Shu
Visual Programming
Van Nostrand Reinhold Company
New York 1988
- 7-Visual Languages and Visual Programming
Shi-Kuo Chang, Editor
Plenum Press
New York 1990
- 8-J. Rumbaugh, M.Blahá, W. Premerlani, F. Eddy y W. Lorenzen
Modelado y diseño orientado a objetos con aplicaciones
Addison-Wesley Iberoamericana
1993

- 9-E. Gamma, R. Helm, R. Johnson y J. Vlissides
Design Patterns: elements of reusable object oriented software
Addison-Wesley profesional computing series
1995
- 10-Jean-Marc Jézéquel, Michel Train, Christine Mingis
Design Patterns and Contracts
Addison-Wesley
- 11-Martin Fowler, Kendall Scott
UML, gota a gota
Addison-Wesley Longman
México, 1999
- 12-Craig Larman
UML y Patrones introducción al análisis y diseño orientado a objetos
Prentice Hall
México, 1999
- 13-VisualWorks 2.0 Tutorial
ParcPlace Systems
1992
- 14-VisualWorks 2.0 User's Guide
ParcPlace Systems
1992
- 15-Developing Applications in GeODE
Servio Corporation
1993

Los abajo firmantes, integrantes del jurado para el examen de grado que sustentará la Lic. **Roberto Tecla Parra**, declaramos que hemos revisado la tesis titulada:

“Diseño e Implementación de un Sistema para Edición de Flujogramas”

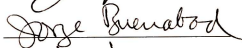
y consideramos que cumple con los requisitos para obtener el grado de Maestro en Ciencias, con especialidad en Ingeniería Eléctrica.

Atentamente,


Dr. Sergio V. Chapa Vergara

Handwritten signature of Sergio V. Chapa Vergara in black ink, written over a horizontal line.

Dr. Jorge Buenabad Chávez

Handwritten signature of Jorge Buenabad Chávez in black ink, written over a horizontal line.

Dr. Manuel González Hernández

Handwritten signature of Manuel González Hernández in black ink, written over a horizontal line.

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL
INSTITUTO POLITÉCNICO NACIONAL

BIBLIOTECA DE INGENIERIA ELECTRICA
FECHA DE DEVOLUCION

El lector está obligado a devolver este libro
antes del vencimiento de préstamo señalada
por el último sello.

17 OCT. 2002

- 7 NOV. 2002

- 9 JUL. 2003

29 JUL. 2003

21 OCT. 2003

DEVOLUCION

