BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

# User Interface Management System

Osvaldo Javier Rosenfeld · Pablo Luis Safar

Director: Lic. Francisco Javier Díaz

Facultad de Ciencias Exactas · U.N.L.P.

1990

# PROLOGO.

La instrumentación de un programa interactivo puede ser dividida en dos componentes básicas: la funcionalidad y la interfase con el usuario. La funcionalidad define qué es lo que el programa puede hacer, y la interfase define cómo los usuarios le dicen al programa qué es lo que debe hacer y cómo el programa le dice a los usuarios qué es lo que hizo.

Esta separación es deseable porque facilita la construcción de buenas interfases y su modificabilidad. Además es factible porque, por ejemplo, la porción de un programa que calcula un valor no tiene por qué estar relacionada con cómo el valor es mostrado al usuario o cómo el usuario pide que ese valor sea calculado.

Una User Interface Management System (UIMS) es una herramienta que permite construir y administrar interfases con el usuario, en forma independiente de la funcionalidad de la aplicación.

Este trabajo tiene como objetivo la creación de una UIMS que otorgue la mayor separación posible entre las dos componentes de un programa nombradas previamente, de modo de facilitar la construcción de interfases con el usuario y dividir las tareas de diseño de una aplicación entre un programador de la funcionalidad y un programador de la interfase con el usuario.

La primera parte contiene una introducción general al mundo de las User Interface Management System y una profundización de los conceptos incluidos en éste prólogo.

La segunda parte presenta un estudio de dos de los casos estudiados sobre el tema: uno de ellos realizado por Pedro Szekely [1] y el otro referido al modelo de interfases de Smalltalk·V.

Finalmente se presenta un nuevo modelo de UIMS, con sus aspectos conceptuales y de implementación, incluyendo un manual para el programador de interfases con el usuario.

# INDICE.

**Capitulo 1**

Introducción ........................................................................1

**Capitulo 2**

Casos estudiados ..................................................................9

**Capitulo 3**

Un nuevo modelo de UIMS: Genius ...............................17

**Capitulo 4**

Manual del programador de la interfase .........................24

**Capitulo 5**

Implementación ....................................................................35

**Bibliografia** ..........................................................................83

BIBLIOTECA
... INFORMÁTICA
U.N.L.P

# 1 - INTRODUCCION.

La creación de buenas interfases con el usuario (UI) es una tarea muy difícil. No hay guias o técnicas que garanticen que el software sea fácil de usar y en general las interfases provistas son poco "amigables" para los usuarios. Por lo tanto, las interfases frecuentemente tienen que ser prototipadas y repetidamente modificadas, adaptándolas a distintos tipos de usuarios, formatos de salida,etc.

Además, la UI de una aplicación consume en general una gran parte del código total. Hay estudios que indican esta porción comprende aproximadamente entre el 30% y el 90% del código. Desafortunadamente se puede dar el caso de que las interfases sean muy simples para un usuario final pero, al mismo tiempo, sean muy difíciles de construir para el diseñador de la UI. Por lo tanto, sería muy interesante desarrollar herramientas que ayuden al diseño e implementación de UI. Afortunadamente las interfases proveen una gran oportunidad para reusar código, ya que aunque diferentes programas realicen distintas tareas, sus interfases pueden ser muy similares. Las técnicas convencionales de implementeción de interfases no permiten un grado de reusabilidad importante. Esto es un motivo más para el desarrollo de las herramientas mencionadas. El uso de estas herramientas resulta en mejores interfases porque:

· Los diseños pueden ser rápidamente prototipados e implementados, posiblemente antes que el código de la aplicación sea escrito.

· Es más fácil incorporar cambios luego de que el usuario testee la interfase porque la interfase es fácilmente modificable.

· Una aplicación puede tener muchas interfases alternativas.

· Si bien es costoso el desarrollo de estas herramientas, una vez creadas, no se requerirá mucho esfuerzo para la creación de las UI.

· El código de la interfase estará mejor estructurado, más modular, porque ha sido separado de la aplicación. Esto permite al diseñador modificar la interfase sin afectar a la aplicación y modificar la aplicación sin modificar la interfase.

· Las dependencias de los dispositivos están aisladas en estas herramientas, por lo cual una aplicación es fácilmente portable a distintos ambientes. Estas herramientas

para diseño de interfases se agrupan en dos grandes formas: User Interface Tool Kits y User Interface Management Systems (UIMS).
Una User Interface Tool Kit es una biblioteca de técnicas de interacción, donde una técnica de interacción es una forma de uso de un dispositivo físico de entrada (mouse, teclado, etc) para ingresar un determinado tipo de valor (comandos, números, etc). Ejemplos de estas técnicas son menúes, scroll bars, on-screen "ligth-buttons". Usando una User Interface Tool Kit el programador es responsable de invocar y organizar las técnicas de interacción. Por otro lado, una UIMS es una herramienta que ayuda al programador a crear y manejar muchos aspectos de la interfase. Además de las características de las tool kits, las UIMS generalmente contienen una componente de control de diálogo, que manipula la secuencia de eventos y las técnicas de interacción y pueden tener también una componente de análisis, que ayuda al estudio y evaluación de la interfase una vez que ha sido creada.

## 1.1 Qué es una UIMS?

Hay varias definiciones de una UIMS en términos de sus capacidades como herramienta.

· Es una herramienta usada por el Administrador de Interfases con el Usuario para construir interfases con el usuario, de la misma forma que un Administrador de Bases de Datos usa el DBMS para administrar los datos almacenados.

· Una UIMS contiene herramientas y técnicas especiales para construir y administrar interfases con el usuario.

· Una UIMS proporciona al diseñador una manera de especificar la interfase en un lenguaje de alto nivel. La UIMS traslada esta especificación dentro de la interfase, administrando los detalles de la pantalla, sus entradas y salidas asociadas y también la interacción con el resto del programa.

Una UIMS es usada en el diseño de la interfase con el usuario y en la administración de la interacción con el usuario en el dominio de la aplicación. Esto significa que hay dos categorias de personas que usan la UIMS: los diseñadores de software y los usuarios finales. Por lo tanto, las UIMS's pueden ser descriptas desde dos puntos de vista diferentes. Un diseñador de software ve a la UIMS como una herramienta

que provee soporte para la definición del diálogo usuario-aplicación, impone un control externo sobre la aplicación, provee soporte para la presentación de salidas de la aplicación e incluye una componente interactiva ayudando a la interacción entre la aplicación y el usuario final. Desde el punto de vista del diseñador, una UIMS debería proveer una interfase con el usuario con las siguientes características:

· Consistencia.
· Capacidad de aceptar todo tipo de usuarios (desde principiantes a expertos).
· Capacidad para el manejo de errores y su recuperación.

Cuando los diseñadores de software usan la UIMS, los resultados son mejores debido a que:

· Hay mayor consistencia entre la interfase y la aplicación relacionada.
· Se facilitan los cambios en la interfase diseñada cuando es necesario.
· Se posibilita el desarrollo y uso de componentes de software reusable.
· Se aislan las aplicaciones de las complejidades del ambiente.
· Se facilita el aprendizaje y el uso de la aplicación.

Para el usuario final, la primera meta de una UIMS es apoyar el uso fácil y efectivo de la aplicación. Aunque los usuarios finales no necesitan estar enterados de la existencia de la UIMS entre ellos y la aplicación, estas son algunas ventajas que les da la UIMS:

· Interfases similares en las distintas aplicaciones.
· Múltiples niveles de ayuda.
· Apoyo para el aprendizaje sobre la aplicación.

En particular, una UIMS deberá:

· Manejar el mouse y otros dispositivos de entrada.
· Validar entradas del usuario.
· Manejar errores del usuario.
· Procesar cortes de operaciones por parte del usuario.
· Proveer un feedback apropiado para mostrar que las entradas fueron recibidas.
· Proveer helps y prompts.

- Permitir que el usuario se habitúe a la interfase.
- Modificar la pantalla cuando se cambian los datos de la aplicación.
- Notificar a la aplicación cuando el usuario modifica los datos.
- Aislar la aplicación de la ventana o funciones de manejo de pantalla.
- Automáticamente evaluar mejoras a la interfase o por lo menos proveer información para permitir al diseñador evaluar la interfase.

## 1.2 Primera aproximación a la estructura de una UIMS.

Es una gran ventaja separar la interfase con el usuario y su administración, de la aplicación. Las razones son muchas, incluyendo mantenibilidad, facilidad de diseño, reusabilidad, etc. El modelo Seeheim muestra las componentes que "deben aparecer en una UIMS". La componente de presentación es responsable de la presentación externa de la interfase con el usuario. La componente de control de diálogo define la estructura del diálogo entre el usuario y la aplicación, y la interfase con la aplicación es la representación de la aplicación desde el punto de



Figura 1. El modelo Seeheim.

vista de la interfase con el usuario.

Este modelo parece ser un bosquejo perfecto de una UIMS. Los principales problemas del modelo Seeheim, el cual fue dirigido a separar todos los aspectos de la interfase con el usuario de la aplicación, son el feedback semántico y la performance. Por ejemplo, cuando un usuario selecciona una acción que provoca un cambio en alguna componente de la aplicación y ese cambio tiene que ser comunicado al usuario por medio de la interfase, se produce un feedback semántico. El problema es que este tipo de feedback tiene que ser virtualmente inmediato, lo cual es probablemente imposible si hay que pasar tokens a través de los tres niveles de abstracción. Debido al énfasis en la separación entre la interfase y la aplicación, la comunicación entre ellas puede resultar lenta.

Hasta ahora hemos hablado de UIMS's soportando el estilo conversacional, pero la idea que está creciendo rápidamente es el estilo de "manipulación directa". En manipulación directa, el usuario se comunica con objetos individuales de interés más que con el sistema como un todo, no existiendo la idea de secuencialidad tipica de un modelo de diálogo. Esto implica para las UIMS's que la sintaxis deberá estar en términos de objetos individuales. Una acción sobre objetos en el dominio de la aplicación casi siempre tiene sus consecuencias. Permitiendo la representación en pantalla de esos objetos, en gran parte se agrega la ilusión de que la representación del objeto es el objeto mismo. Se pueden hacer las siguientes observaciones acerca de UIMS's que soportan manipulación directa:

- La sintaxis deberá estar expresada en términos de objetos individuales. Deberá también ser minimizada, usando acciones físicas tales como indicaciones y arrastres sobre la pantalla, dando así preferencia a efectos mas intuitivos por sobre formalizaciones que obliguen a usar un nuevo lenguaje.
- El feedback, especialmente el semántico, es sumamente importante y necesita ser especialmente considerado.
- Es importante la flexibilidad en la componente de presentación, particularmente la aptitud para diseñar técnicas específicas de interacción y combinar esas técnicas dentro de dispositivos abstractos.

El tema de sintaxis basadas en objetos individuales tiene una fuerte relación con la programación orientada a objetos. Muchas UIMS's han sido recientemente

implementadas usando técnicas orientadas a objetos. Hay también muchos sistemas que usan técnicas orientadas a objetos para manipulación directa que pueden ser vistos como tool kits gráficos más que como verdaderas UIMS's.

## 1.3 Aspectos de Desarrollo.

En muchas UIMS's el diseñador especifica la interfase usando un lenguaje de propósito especial. Este lenguaje puede tomar muchas formas, incluyendo árboles de menúes, context-free grammars, state transition networks, lenguajes declarativos, lenguajes de eventos, lenguajes orientados a objetos, etc. Con muchos de estos sistemas el lenguaje es usado para especificar la sintaxis de la interfase, por ejemplo las secuencias legales de acciones de entrada y salida. Otras UIMS's permiten definir interfases colocando objetos sobre la pantalla y usando un mouse. Esto se debe a la observación de que la presentación visual es muy importante en interfases gráficas, y una herramienta gráfica parece ser la forma mas apropiada de especificar su apariencia. Otra ventaja de esta técnica es que usualmente es mucho mas fácil de usar por el diseñador y hasta por no programadores. Una nueva clase de UIMS's intenta crear la interfase directamente desde una especificación de los procedimientos semánticos de la aplicación, y luego permite al diseñador modificar la interfase para mejorarla.

Hay varias áreas dentro del dominio de las UIMS's que merecen nuestra atención. Dos de ellas son: "técnicas de especificación de diálogo" y "relación entre la UIMS y la aplicación".

Una de las mayores ventajas de usar una UIMS es la facilidad de diseño, mantenimiento y cambios en la interfase durante su desarrollo. La separación entre la interfase y la aplicación implica la necesidad de métodos para especificar varios aspectos de la interfase en una forma declarativa y de alto nivel. Dos métodos de especificación para el aspecto sintáctico de la interfase son "transition networks" y "grammars".

Transition networks es la notación para especificación de diálogo mas antigua. Cada nodo en la red corresponde a un cierto estado del diálogo, y los arcos entre los nodos definen interacciones, que pueden ser de entradas del usuario o de salidas por parte del sistema. Algunos arcos pueden ser no-terminales, lo cual significa

que tienen asociadas redes que son invocadas si el arco es alcanzado. Las transition network tienen un serio problema: muchas interfases tienen un gran número de estados, lo cual implica una red muy grande. Para solucionar este problema pueden usarse subnetworks, (como por ejemplo los arcos no-terminales).
La especificación grammar es equivalente en poder expresivo a las transition network, pero suelen ser más dificiles de leer y entender.
Una técnica de especificación nueva es el "modelo de eventos", que se puede describir de la siguiente manera:
El modelo de eventos ve a la interfase como una colección de eventos y manejadores de eventos. Un evento es generado en cada momento que el usuario interactúa con un dispositivo de entrada. Estos eventos son procesados por el manipulador de eventos asociado con la entrada envuelta en la interacción. La colección de eventos procesados por el manipulador de eventos puede ser vista como un estado. El conjunto de manipuladores de eventos activos (apto para recibir eventos) en un determinado momento define las acciones legales del usuario en ese punto del diálogo. Una diferencia importante con los métodos anteriores es que ellos incorporan un ordenamiento explícito de eventos de I/O, lo cual significa que la especificación expresa explícitamente qué secuencia de eventos de I/O constituye un diálogo válido entre el usuario y la aplicación. El modelo de eventos, en cambio, resalta un ordenamiento implícito, donde la especificación define el conjunto de eventos de I/O, sin mencionar un ordenamiento especifico. Las restricciones de ordenamiento están implícitas en la UIMS, que interpreta la especificación.

## 1.4 Algunos problemas de las UIMS's.

Las UIMS's han ganado aceptación en el mundo de la investigación y los negocios. Pero a pesar de las ventajas que indudablemente ofrecen, hay algunos hechos que hacen que las UIMS's no sean muy usadas. Vamos a describirlos brevemente:

**Facilidad de uso.**

Uno de los puntos que determinan cuán fáciles de usar son las UIMS's para el diseñador de la interfase, es la habilidad para programar que ellas requieren. Una UIMS que necesita programación es probablemente mas dificil de usar que una que no la necesita. Otra posición diferente en este punto es que la meta no debería ser

eliminar la programación, sino proveer al diseñador de la interfase, quien no necesariamente debe ser el diseñador de la aplicación, un conjunto apropiado de herramientas. Estas herramientas pueden llegar a requerir alguna habilidad para programar. Hay UIMS's donde las construcciones de programación necesarias son simples operadores aritméticos, saltos condicionales y llamadas a funciones, de tal modo que aun los no programadores puedan aprender a usar este tipo de UIMS's.

**Portabilidad.**

Muchas de las UIMS's que existen actualmente están hechas para un sistema operativo, computadora o sistema gráfico particular. Esto se debe a que no hay interfases de alto nivel apropiadas que puedan resguardar a la UIMS de tener que manipular directamente varios dispositivos de entrada como mouse y teclado y crear directamente figuras sobre la pantalla. Para hacer a las UIMS's más portables se deben crear modelos abstractos tanto para entradas como para salidas [6].

**Mantenibilidad.**

Cuando un programa grande es construido con una UIMS, muchas de las estructuras de control del programa son puestas dentro de la UIMS, la cual, para el programador de la aplicación, es una caja negra. Esto hace del mantenimiento del sistema un gran problema. Con el desarrollo de las UIMS's fueron creadas nuevas formas de expresión de programas, por lo tanto, los problemas de manejo de software deben ser redireccionados dentro del contexto de las UIMS's.

# 2.CASOS ESTUDIADOS.

## 2.1 - Pedro Szekely: Separación entre la interfase con el usuario y la funcionalidad de la aplicación

Los artículos leidos coinciden en recomendar la separación entre la interfase con el usuario y la funcionalidad de la aplicación, de modo que se favorezca la reusabilidad y se produzcan interfases de alta calidad.

El diseñador de la UIMS debe crear una separación entre la aplicación y la interfase para que su desarrollo sea independiente.

La primera aproximación a esta separación fue mantener una división estricta de responsabilidades entre la UI y la aplicación: la aplicación hacia el trabajo específico y la interfase se comunicaba con el usuario. Sin embargo, la interfase generalmente asume alguna responsabilidad sobre las funciones de la aplicación. Por ejemplo, una buena interfase no debería permitirle al usuario que invoque una operación que no puede ser ejecutada con éxito. Al mismo tiempo, si la interfase tiene demasiado conocimiento sobre la aplicación, la separación entre estas dos componentes no es clara. Por esto, las preguntas que se tratan de responder son tales como dónde y de qué manera dibujar la linea separatoria entre ambas componentes y cómo comunicarlas, o qué tipo de control se necesita para coordinar la ejecución y la comunicación.

P. Szekely [1] propone la posibilidad de particionar la implementación de un programa interactivo en dos componentes: funcionalidad e interfase con el usuario. La funcionalidad define qué puede hacer el programa y la UI define cómo el usuario le dice al programa qué hacer y cómo el programa le dice al usuario lo que hizo.

La comunicación entre el usuario y el programa requiere que los participantes compartan un entendimiento de cómo el mundo trabaja y un lenguaje en el cual codificar la información que se desea comunicar. Compartir un entendimiento del mundo no significa que el usuario y el programa tengan idénticos modelos del mundo. El modelo del mundo del programa (modelo del diseñador) es definido por el diseñador del programa. El usuario adquiere su modelo (modelo del usuario)

interactuando con el programa, leyendo acerca de él, hablando con otros usuarios, etc.

Los dos modelos no necesitan ser idénticos y normalmente no lo son, pero la existencia de incompatibilidades entre el modelo del diseñador y del usuario provocan problemas en la comunicación. Cuando el programa es pensado como entidades capaces de comunicarse, el programa está compuesto de dos componentes: conceptual y de comunicación.

La componente conceptual se refiere al aspecto del mundo que el programa entiende. Es usualmente descripta en términos de objetos y operaciones provistas por el programa. Los objetos corresponden a entidades del mundo real y las operaciones corresponden a acciones que pueden ser realizadas sobre los objetos para transformarlos o combinarlos con otros objetos.

La componente de comunicación se refiere al lenguaje usado para comunicar los conceptos definidos en el modelo conceptual.

Es importante distinguir entre la semántica y la sintaxis del lenguaje de comunicación.

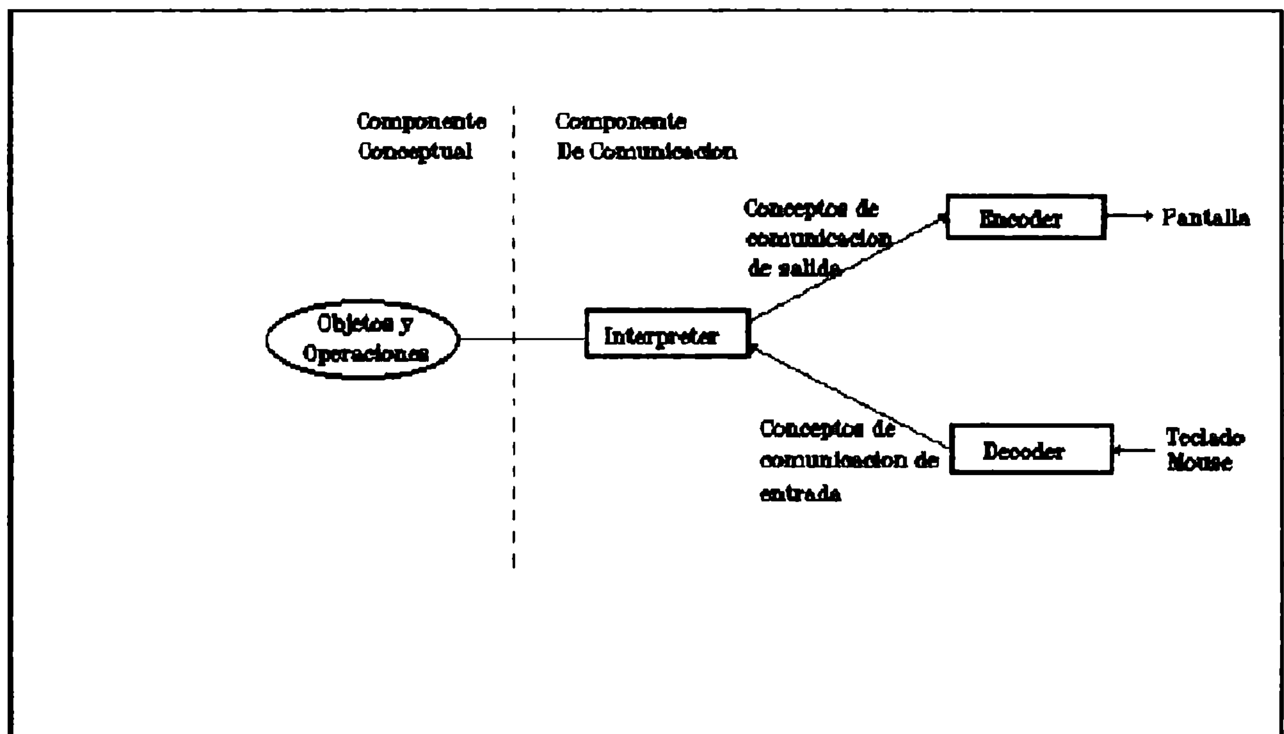


Figura 2.1 - Modelo propuesto por Pedro Szekely

La semántica del lenguaje de comunicación no deberia confundirse con la componente conceptual del programa. La semántica no está definida por los objetos y operaciones provistas por el programa. Los objetos y operaciones definen el modelo del mundo acerca del cual el usuario y el programa se comunican, mientras que la semántica define la información acerca de objetos y operaciones que el usuario y el programa pueden comunicar. Estas piezas de información son llamados conceptos de comunicación.

La sintaxis define cómo codificar los conceptos de comunicación para la transmisión entre el usuario y el programa.

La componente conceptual del programa consiste en la definición de objetos y operaciones. La componente de comunicación consiste de tres partes. El "decoder" mapea entradas de los dispositivos en conceptos de comunicación de entradas. El "interpreter" responde a los conceptos de comunicación de entrada usando el conocimiento acerca del programa contenido en la componente conceptual y produce conceptos de comunicación de salida. El "encoder" mapea conceptos de comunicación de salida en imágenes y sonido.

El limite entre la componente conceptual y de comunicación depende del conocimiento acerca de los objetos y operaciones que el interpreter necesita para interpretar los conceptos de comu·nicación. Separar la interfase de la funcionalidad del programa es dificultoso porque distintos lenguajes de comunicación soportan distintos conjuntos de conceptos de comunicación y su interpretación requiere distintos conocimientos acerca de los objetos y operaciones definidas en la componente conceptual. En consecuencia, es imposible determinar el conocimiento acerca de objetos y operaciones que deberá ser definido en la componente conceptual, antes que la semántica del lenguaje de comunicación haya sido definida. De todos modos, la sintaxis del lenguaje de comunicación no afecta los limites entre las componentes conceptual y de comunicación.

A partir de estos conceptos P. Szekely avanza en el diseño de una UIMS con una buena separación entre funcionalidad e interfase con el usuario.

### Conceptos de Comunicación de Salida.

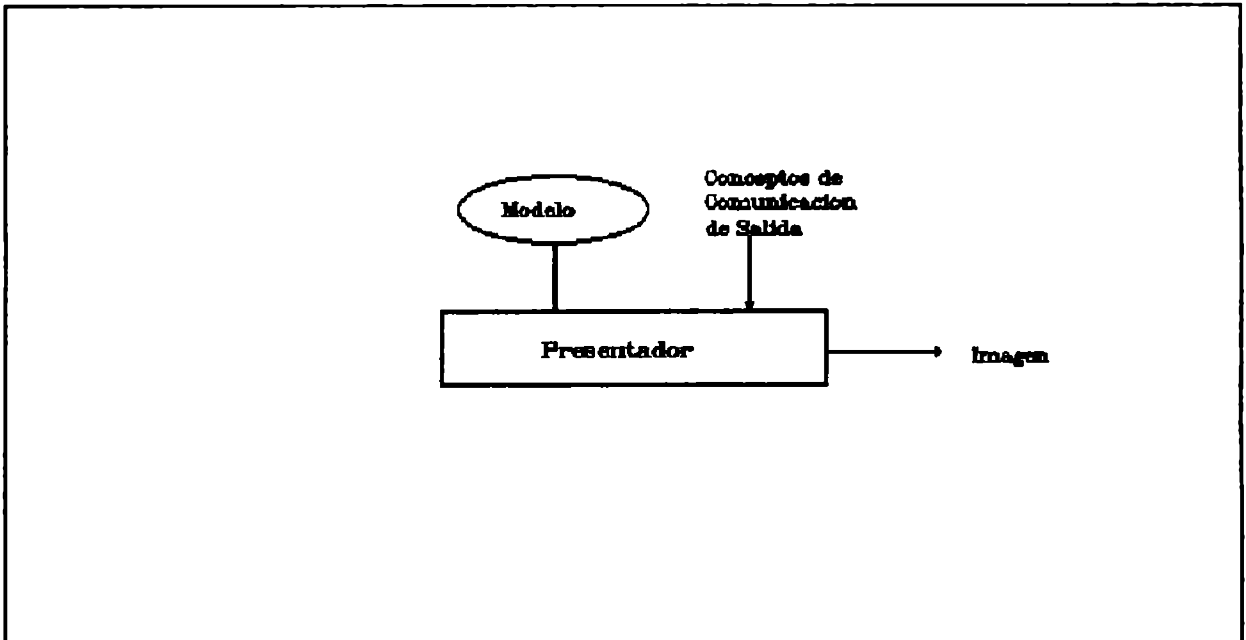


Figura 2.1. Modelo del presentador

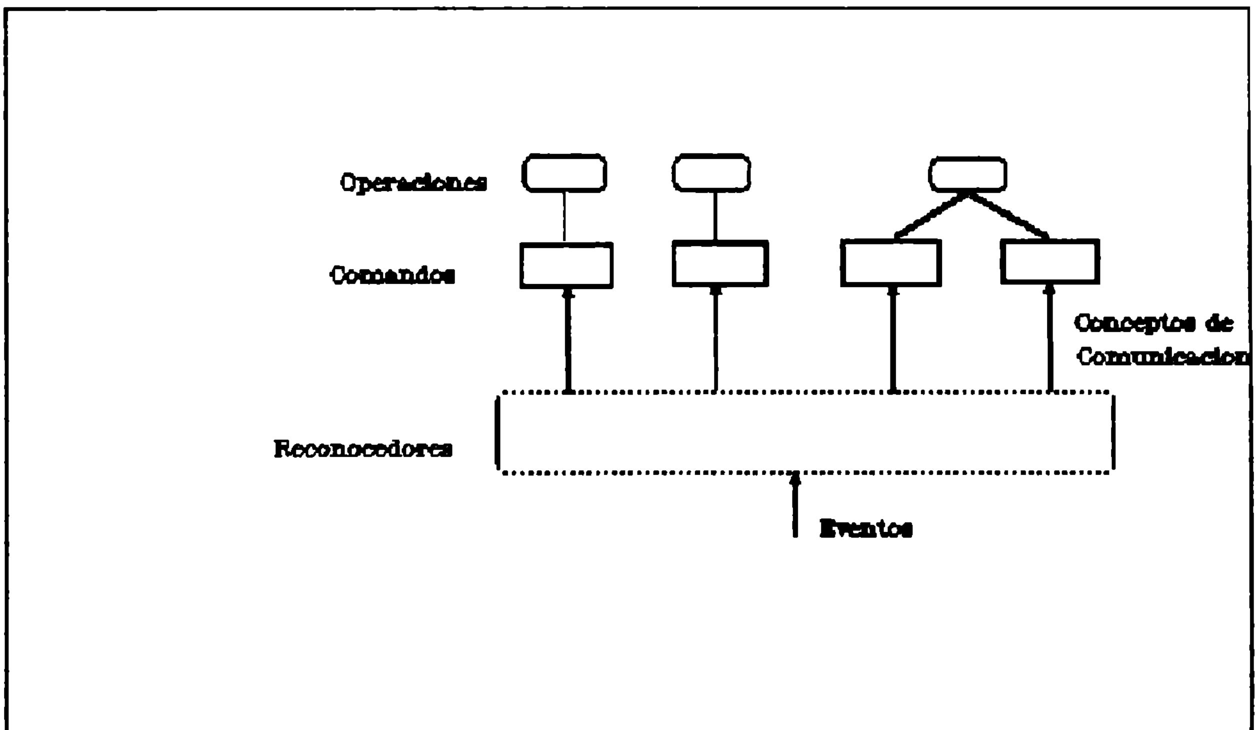


Figura 2.2 Comandos y Reconocedores

La presentación de un concepto es una codificación de información referida al concepto en términos de imágenes y sonidos. Nuestro intéres está centrado fundamentalmente en las imágenes.
Las presentaciones codifican información en atributos de forma, tamaño, posición, orientación y color de sus imágenes constituyentes. Esta información no es más que un conjunto de conceptos de comunicación comunicados por el programa.
Los conceptos de comunicación de salida podrían ser divididos en:

· Conceptos de comunicación de contenidos: comunican información sobre los contenidos de objetos y los atributos de objetos y operaciones.

· Conceptos de comunicación de cambios: comunican información sobre los cambios producidos por la ejecución de operaciones.

La explicación de los conceptos anteriores estará hecha en términos de presentadores, que son una abstracción del cuerpo de software que produce una presentación.

*Presentadores*

Una presentación de un concepto de un programa, sea éste un objeto o una operación, es una representación de sí mismo que puede ser emitida sobre un dispositivo de salida. Estas representaciones son construidas por entidades denominadas presentadores.
Como se ve en la figura 2.1 un presentador está co·nectado a un concepto, denominado "modelo del presentador". El presentador usa conceptos de comunicación sobre el modelo para producir una representación de él, denominada imagen.
El comportamiento de un presentador se define por un conjunto de reglas que especifican la imagen asociada con el modelo del presentador. Estas reglas deben hacer ciertas suposiciones sobre las propiedades del modelo, y en consecuencia el presentador no puede ser usado por conceptos que no satisfacen estas suposiciones. Los conceptos que pueden usar el presentador, es decir, el conjunto de conceptos que pueden ser modelo para el presentador, se denomina "dominio del presentador".

Los conceptos de comunicación de cambios informan al usuario sobre los cambios producidos por la ejecución de operaciones. Hay dos formas en las que la noción de un cambio puede incorporarse dentro del modelo de comunicación usado en esta tesis:

· *Modelo de Renovacion continua:* el presentador asociado a un concepto actúa como si estuviese continuamente consultando el estado de los conceptos que definen la presentación y regenerando la imagen que los codifica. Este modelo es una abstracción del método usado por muchos dispositivos de display para modificar la pantalla consultando repetidamente una representación de la pantalla almacenada en memoria.

· *Modelo de mensajes:* se usan otros conceptos de comunicación para comunicar cambios de interés al usuario. El presentador espera que su modelo le informe sobre cambios que se produzcan en el para poder comunicarlos al usuario.

**Conceptos de comunicacion de entrada.**

La comunicación de usuarios a programas se centra alrededor de la invocación a operaciones. Los usuarios deben comunicar muchos conceptos de comunicación para invocar una operación, y estos conceptos de comunicación son codificados en las manipulaciones de los dispositivos de entrada. Podriamos clasificar los conceptos de comunicación de entradas y el conocimiento sobre los programas necesarios para interpretar estos conceptos, en términos de propiedades de "comandos" y "reconocedores", que son una abstracción del cuerpo de software que interpreta entradas. La figura 2.2 muestra el rol de los comandos y reconocedores en el control del flujo de información de usuarios a programas. La información entra al programa en forma de eventos, que son producidos cuando un usuario manipula dispositivos de entrada. Se asume que los cambios en estos dispositivos se codifican como entidades denominadas eventos, y que los eventos se ubican sobre una cola de la que los reconocedores pueden tomarlos.

Los reconocedores son la parte del programa que acepta eventos e identifica en ellos los conceptos de comunicación provistos por el usuario. Cuando un reconocedor identifica un concepto de comunicación lo envía a un comando para que sea interpretado. Por lo tanto, los comandos son la porción del programa que interpreta

conceptos de comunicación.

Un comando interpreta los conceptos de comunicación relevantes a la invocación de una operación. Como cada comando puede ser diseñado para interpretar conceptos de comunicación en una forma particular, la definición de una nueva forma para interactuar con una operación obliga a la definición de un nuevo comando. Es decir, más de un comando puede estar asociado con una operación, proveyendo más de una forma de invocar a la operación en el mismo programa.

La dependencia entre un comando y una operación esta dada por el conocimiento que el primero debe tener acerca del otro para realizar la interpretación. Supongamos que un comando recibe un concepto de comunicación que contiene el valor de una entrada. Para hacer la interpretación , el comando necesita saber si el valor de entrada es correcto. El conocimiento de si un valor de entrada es correcto es un ejemplo del conocimiento acerca de la operación necesario para interpretar conceptos de comunicación. Diferentes comandos pueden interpretar el mismo concepto de comunicación a su manera. Por ejemplo, un comando podría validar las entrada tan pronto como ellas fueran dadas, y otro comando podría validarlas a partir de un pedido del usuario. Un comando podría automáticamente ofrecer valores alternativos en caso de error, y otro podría producir que la computadora haga sonar una alarma. No importa cómo el comando interprete un concepto de comunicación, todos necesitan cierto conocimiento acerca de la operación. Este es el conocimiento que define las dependencias entre la funcionalidad y la porción de entrada de la interfase.

## 2.2 - Modelo de Interfase de Smalltalk.

El modelo de interfase con el usuario de Smalltalk-V está basado en el paradigma MPD (Model-Pane-Dispatcher), similar al MVC (Model-View-Controller) de Smalltalk-80. Smalltalk provee interfases gráficas interactivas a través de ventanas. En una ventana intervienen tres componentes:

- *Model:* es el objeto que se va a mostrar y/o modificar.
- *Pane:* es el objeto que determina cómo el modelo va a ser mostrado.
- *Dispatcher:* es el objeto que maneja las interacciones del teclado y mouse.

De esa forma se deduce que son posibles muchas visiones de un mismo objeto, es decir, puede haber múltiples ventanas para el mismo model. El procesamiento podríamos pensarlo dividido en dos actividades:

· Procesamiento de entrada, que es responsabilidad del dispatcher.

· Procesamiento de salida, que es responsabilidad del pane.

En el modelo MPD cada pane conoce explícitamente quién es su modelo y quién es su dispatcher. Cada dispatcher conoce explícitamente cuál es su pane. Pero no hay conexión explícita del modelo hacia sus panes y dispatcher. Esto permite aislar la funcionalidad de una ventana de los aspectos del modelo que ella muestra. Cuando algún aspecto del modelo cambia, éste informa a los panes interesados en ese aspecto que ha cambiado, y esos panes reaccionan pidiéndole al modelo la información necesaria para volver a mostrarla. La clase Pane tiene dos subclases: TopPane, que coordina a todos los subpanes de la ventana, y SubPane. La clase SubPane tiene tres subclases: GraphPane, ListPane, que muestra una lista de strings y permite seleccionar un item, y TextPane, que permite editar el texto que contiene. Cada pane tiene un único dispatcher asociado. El tipo de pane determina el tipo de dispatcher asociado. Una instancia de un dispatcher actúa como un mensajero que recolecta las entradas desde el teclado y mouse, enviando mensajes a su pane para tomar las acciones acordes con los eventos de entrada. Smalltalk permite la creación de interfases con el usuario con estas herramientas, pero también permite ampliar las herramientas que provee definiendo nuevos tipos de panes y nuevos tipos de dispatchers. Para estudiar el modelo MPD, hemos desarrollado una pequeña aplicación y luego una interfase para ella. En el diseño hemos observado que Smalltalk provee código reusable para construir interfases y una aceptable separación entre la funcionalidad de la aplicación y la interfase con el usuario. Hemos podido desarrollar una aplicación especificando la interfase en un alto nivel, sin desarrollar nuevas componentes de software para interfases.

Sin embargo, la separación de responsabilidades entre la interfase con el usuario y la aplicación no es del todo clara. Si a partir de la aplicación desarrollada quisiéramos modificar aspectos de cómo se muestran los objetos, podríamos encontrarnos con el problema de que el cambio en la interfase con el usuario involucraría modificar algún aspecto de la funcionalidad de la aplicación. Por lo tanto, queda resaltada la

idea original de proveer una herramienta que otorgue mayores facilidades sobre este punto.

# 3. Un nuevo modelo de UIMS : Genius.

El modelo de UIMS que aqui se presenta, toma como base conceptual fundamental para su implementación el trabajo de P.Szekely, visto en el capítulo anterior. Esta no es una elección antojadiza. Se debe a que consideramos que aquél provee un buen nivel de separaciün entre una aplicación y su interfase.

Brevemente, haremos una nueva referencia a estos conceptos básicos para poder utilizarlos como punto de partida para la especificación de esta nueva herramienta creada.

Un reconocedor es un objeto de la interfase capaz de recibir y procesar eventos exteriores provenientes del teclado o del mouse. Su función consiste meramente en recibir el evento y traducirlo en un concepto de comunicación de entrada.

El concepto de comunicación de entrada producido por el reconocedor es recibido por un comando asociado, que interpreta el significado de ese concepto y lleva a cabo sus tareas de recolección de datos y validación para comunicarse con una operación de la aplicación.

Finalmente, el tercer elemento básico de la UIMS propuesta por Szekely es el presentador, que se encarga de informar al usuario de los cambios producidos en el mundo de los objetos de la aplicación.

Una diferencia importante y que se puede considerar como básica entre el modelo de P. Szekely y el modelo que propone Genius es que en el primero el diseño de una nueva interfase para una aplicación implica la definición y codificación de nuevas clases de objetos, como subclases de las clases predefinidas provistas por el modelo, de acuerdo a las necesidades de la interfase que se está generando. Esto trae aparejado que la generación de una interfase requiere, generalmente, la producción de mucho código adicional. Por su parte el modelo de Genius propone la generación de una interfase con el usuario a partir de la creación de instancias de objetos de las clases ya provistas por la UIMS, sin necesidad de crear otras distintas, y por lo tanto facilitando notablemente el trabajo del programador de la interfase.

Otra diferencia conceptual con respecto al modelo de P. Szekely es la inclusión

de un nuevo objeto, el "scheduler", que actuará como gerenciador de la interfase, el cual será descripto más adelante.

## 3.1 - El modelo.

La figura 3.1 muestra el esquema de la estructura básica de Genius y las dependencias entre los cuatro elementos fundamentales que lo componen: reconocedor, presentador, comando y scheduler.

### 3.1.1 - Reconocedores.

Como ya se expresó en varias oportunidades, los reconocedores están capacitados para recibir eventos desde el teclado o mouse. Durante la ejecución de una aplicación, la pantalla del usuario está cubierta por la presencia de reconocedores de objetos accesibles, de modo tal que cualquier evento externo será recibido de la forma apropiada por alguno de ellos. Obviamente, sólo uno de los reconocedores del conjunto es el que recibe la señal, aquel que corrientemente está vinculado a la posición física de la pantalla apuntada por el cursor del mouse.

Esto nos lleva a la idea de que hay sólo un reconocedor activo en cada momento y que la activación o desactivación de cada uno de ellos está determinada por los cambios en la posición del cursor.

Un reconocedor es la llave para la invocación a una operación de la aplicación accesible por el usuario (y que por lo tanto está presente de alguna manera en la interfase). El usuario manifiesta su intención de invocar a una operación moviendo el cursor a través de los distintos objetos disponibles por su presencia en la pantalla y provoca, en forma absolutamente transparente para él, la activación y desactivación de los reconocedores.

### 3.1.2 - Comandos.

La asociación entre un reconocedor y una operación no es directa. Existe un elemento, el comando, que se interpone entre ambos y cuyo papel es fundamental en la UIMS Genius.

Todo reconocedor tiene asociado un comando y todo comando tiene asociada una

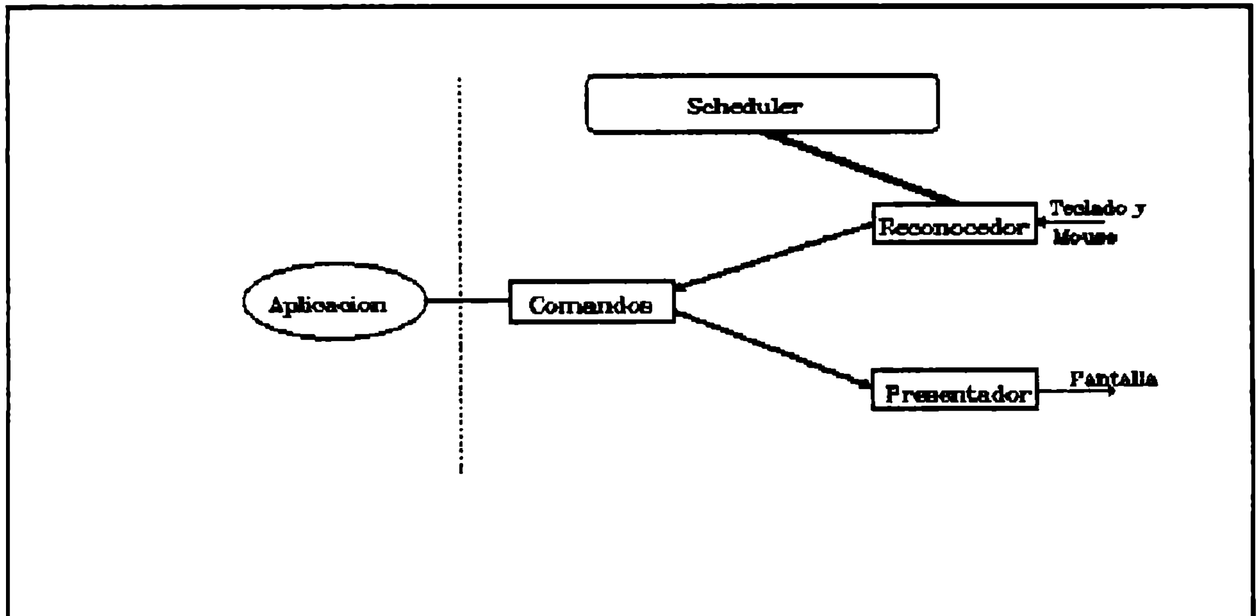


Figura 3.1. El modelo de Genius

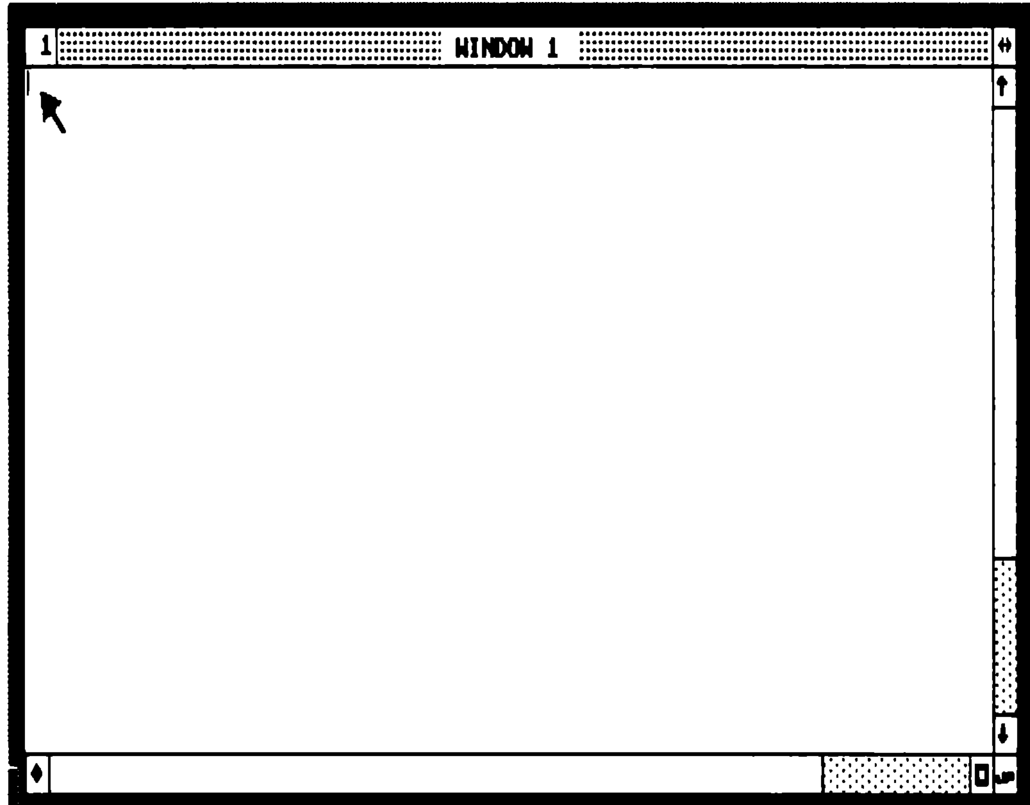


Figura 3.2. Ventana de Genius.

operación de la aplicación. Cuando el usuario invoca a una operación a través de un evento sobre un reconocedor presente en la pantalla, internamente es activado el comando correspondiente.

El papel de un comando consiste en recolectar y validar los datos necesarios para que la operación invocada se ejecute con normalidad, liberando a ésta de estas engorrosas tareas y proveyendo una buena separación entre funcionalidad de la aplicación y la recolección de entradas. Esto permite la modificación de, por ejemplo, el rango de datos válidos para una operación o la forma en que se pedirán las entradas sin necesidad de realizar cambios en la operación misma. Es decir, se pueden modificar fácilmente aspectos del diálogo usuario-aplicación, sin afectar la funcionalidad de la aplicación.

Adicionalmente, la UIMS da la posibilidad de habilitar o inhabilitar un comando, como se verá más adelante. Esta última situación consiste en inhibir la invocación de una operación. Las herramientas que utiliza Genius para realizar la recolección de datos de entrada serán descriptas posteriormente.

### 3.1.3 - Presentadores.

Una vez que el comando ha completado su tarea, la operación invocada se lleva a cabo, y puede producir cambios sobre aspectos de los objetos que están siendo mostrados sobre la pantalla. Cuando esto ocurre la operación informa al comando asociado "algo ha cambiado en algún objeto de la aplicación". Cada comando contiene una lista de presentadores, objetos que muestran al usuario visiones de elementos de la aplicación. Cuando el comando recibe el mensaje desde la operación, envia a cada uno de sus presentadores la orden "muestre el cambio producido en la aplicación".

### 3.1.4 - Scheduler.

Como se ha expresado anteriormente, el Scheduler actúa como gerenciador de la interfase, dando el control a uno u otro reconocedor existente sobre la pantalla (a aquél que contiene el cursor). Cuando el cursor deja un reconocedor, éste se desactiva y el scheduler decide cuál será activado en ese momento. Como vemos, la tarea

del scheduler es muy simple, pero de fundamental importancia en la UIMS Genius, basada en la manipulación directa de objetos sobre la pantalla.

## 3.2 - Ventanas.

Las ventanas son las componentes fundamentales de la interacción entre el usuario y la aplicación. Ellas son capaces de mostrar los resultados de las operaciones invocadas así como de recibir las especificaciones de entradas necesarias para su ejecución.

La figura 3.2 muestra una ventana de Genius. La parte central, el "área de trabajo", es la zona sobre la que se realiza la presentación de objetos. Una ventana está formada por un número de páginas determinado por el programador de la interfase, las cuales se muestran sobre el área de trabajo. El resto de las zonas accesibles otorgan a la ventana capacidades funcionales inherentes a su manejo, sin relación alguna con la aplicación. Así se tiene:

- Una zona superior que contiene el label de la ventana y el número de página que está siendo visualizada.
- Dos zonas para realizar scroll horizontal y vertical sobre la página visible.
- Una zona que permite mover la ventana dentro de la pantalla.
- Una zona que permite modificar el tamaño de la ventana.
- Dos zonas que permiten modificar el número de página visible.
- Una zona que permite reinicializar el estado del scroll.
- Una zona que permite cambiar aspectos de visualización dentro de la ventana, tales como tipo y tamaño de las letras y tipo y grosor de las líneas utilizadas para mostrar gráficos.

Una característica importante de Genius es que posibilita crear interfases con el usuario con múltiples ventanas, las cuales pueden mostrar objetos diferentes pero de acuerdo a la configuración dada a cada una de ellas. Por ejemplo, dadas dos ventanas se podría graficar una misma figura geométrica con lineas gruesas y continuas sobre una y con lineas finas y punteadas sobre la otra, como de observa en la figura 3.3.

Independientemente del número de ventanas existentes, sólo una se encuentra activa

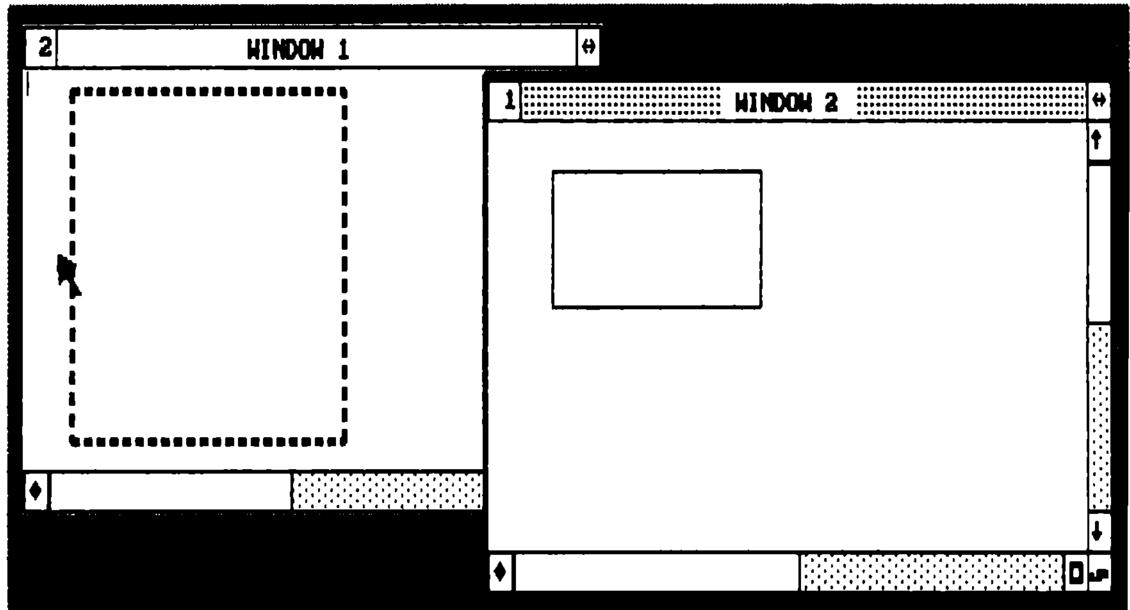


Figura 3.3. Configuración de Ventanas.

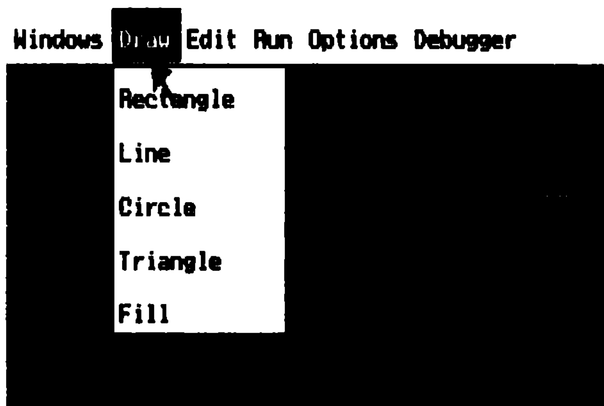


Figura 3.4. Menú.

por vez y será aquella que contenga el cursor. Obviamente, Genius permite al usuario de la aplicación modificar la ventana activa simplemente moviéndose a aquella sobre la que desea trabajar.
Las ventanas del modelo Genius permiten al usuario la escritura de texto sobre sus áreas de trabajo. Al tener la capacidad de configurar el tipo y tamaño de letras a utilizar, y además hacerlo cada una en forma independiente, podemos concluir que se provee de numerosas formas de diseño de texto. Sin embargo, conviene tener en claro que Genius no proporciona un editor de texto tal como los comúnmente conocidos. Sólo presenta una serie de herramientas básicas que podrían transformarse, en un paso posterior de programación, en un editor completo. Por el momento, en consecuencia, sólo debemos conformarnos con la capacidad de escritura, pero con la seguridad de que es posible el desarrollo del editor sin grandes esfuerzos adicionales. En el capitulo siguiente se hace referencia a la forma en que se ha instrumentado esta funcionalidad, para que sirva como referencia para la tarea posterior.

## 3.3 - Menúes.

Genius otorga al programador de la interfase la posibilidad de crear menúes. Cada uno de ellos puede tener asociado un submenú desde el cual es posible invocar a una operación definida previamente. Los menúes aparecerán en la pantalla ocupando la franja superior horizontal, con los items conformantes desplegados de izquierda a derecha.
Los submenúes se despliegan en forma vertical partiendo desde el ítem de menú que los aglutina. Sería redundante aclarar que la invocación de una operación desde un submenú se realiza presionando el botón del mouse cuando el cursor se encuentra sobre la opción deseada. Si una operación ha sido inhabilitada, es decir, no puede invocarse, el menú no responderá a una intención de invocarla que demuestre el usuario.
La figura 3.4 muestra un menú sobre la pantalla y el submenú correspondiente a uno de sus ítems.

## 3.4 - Iconos.

Un icono es un signo gráfico representativo de una operación disponible para el usuario de la aplicación. No es más que una nueva forma de invocación provista por Genius.

Los iconos pueden existir en cantidades ilimitadas y pueden también aparecer en cualquier sector de la pantalla. Inclusive se permite asociar más de un icono a la misma operación. Las acciones que se desprenden de una invocación a una operación desde un icono son idénticas a las que se producen al invocarla desde un menú o una ventana.

La figura 3.5 muestra la presencia de iconos sobre la pantalla.

## 3.5 - Recolección de entradas.

Genius provee dos herramientas fundamentales para la recolección de entradas requeridas por una operación :

a) a través de cajas de diálogo

b) a través del ingreso de puntos sobre una ventana.

### a) Cajas de diálogo.

Una caja de diálogo es una ventana abierta sobre la pantalla con la única finalidad de recibir las entradas requeridas por el comando asociado a una operación. Una vez que cumplió su tarea, la caja de diálogo desaparece. Desde el momento en que cada comando (o cada operación) requiere entradas distintas, es posible concluir que cada uno de ellos esta ligado a una caja de diálogo distinta. La figura 3.6 muestra una caja de diálogo típica de Genius.

Como se observa en la figura, hay dos formas de especificar valores de entrada. La primera, a través de la selección de una opción en un conjunto, en donde es posible también definir valores a tomar por default. La segunda, a través del tipeado del valor de entrada deseado, sobre una linea de edición de texto.

En ningún momento será posible no especificar las entradas, ni ingresar valores de un tipo incompatible con el esperado.

Una vez que se han especificado todos los valores pedidos, el usuario deberá

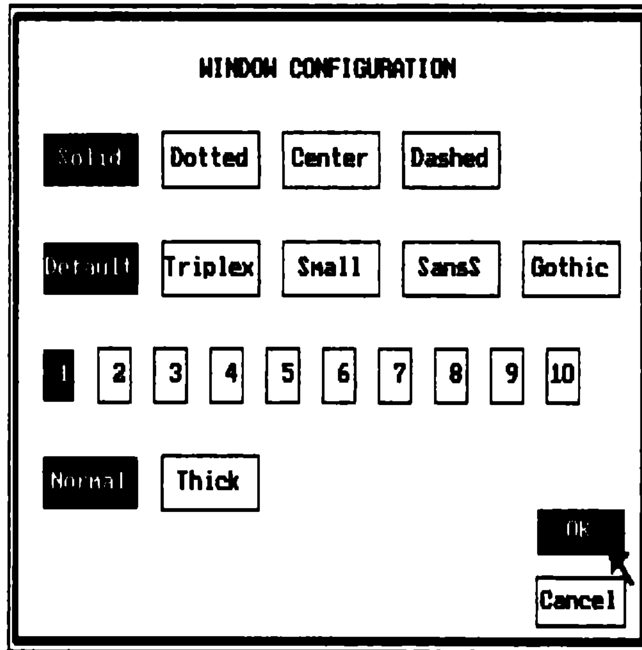


Figura 3.6. Cajas de Diálogo.

confirmar o rechazar la invocación a la operación a través de las opciones 'OK' y 'Cancel', que aparecerán en todas las cajas de diálogo que se creen. Si los datos ingresados no son correctos, o el comando que los recibe ha realizado una validación con resultados negativos, aparecerá un mensaje de error que debe ser definido por el programador.

**b) Entrada de puntos sobre la pantalla.**

Hay operaciones que requieren para su ejecución el ingreso de coordenadas de algún punto de la pantalla. Como clásico ejemplo de este esquema, podríamos nombrar la coordenada superior izquierda y la inferior derecha necesarias para el dibujo de un rectángulo. La invocación a una operación de este tipo produce un pedido de entradas de esta forma y una validación semejante a la que se hace desde una caja de diálogo.

# 4. MANUAL DEL PROGRAMADOR DE LA INTERFASE.

El presente capítulo está destinado al programador de la interfase, que es el usuario final de la UIMS Genius. En él se brinda un completo manual de programación de interfases utilizando esta herramienta. La tarea del programador de una interfase consiste en la creación de instancias de algunas de las clases provistas por la UIMS. Estas clases son: *Recognizer, Command, Presenter, Window, Menu, Icon, Dialog Box* y otras que juegan el rol de accesorios para las anteriores.

## 4.1 - Creación de reconocedores.

La forma general de crear un reconocedor es la siguiente:

```
aRecognizer:=New(RecognizerPtr,Init(aFrame,aCommand)) ;
```

donde:

*aRecognizer* es la nueva instancia creada y debe ser declarada del tipo RecognizerPtr.

*aFrame* es una instancia de la clase Rectangle que indica la zona de la pantalla asociada a aRecognizer.

*aCommand* es una instancia de la clase Command y es el comando asociado a aRecognizer.

## 4.2 - Creación de Comandos.

La forma general de crear un comando es la siguiente :

```
aCommand:=New(CommandPtr,Init(anAssociator)) ;
```

donde:

*aCommand* es la nueva instancia creada y debe ser declarada del tipo CommandPtr.

*anAssociator* es un procedimiento creado por el programador y que será descripto más adelante.

Ya hemos dicho que los comandos se encargan de requerir las entradas para una operación. Las entradas en la UIMS son instancias de una clase especial denominada *Input*. Por lo tanto, para asociar una entrada a un comando es necesaria la generación de objetos de esta última clase. La forma general para la creación de una entrada es la siguiente:

```
anInput:=New(InputPtr,Init(aDefaultValue, aType,aPredicate, aChoiceSet, aMsgError));
```

donde:

*anInput* es la nueva instancia creada y debe ser declarada del tipo InputPtr.

*aDefaultValue* es una instancia de la clase Value que indica el valor que por defecto tomará la entrada.

*aType* indica el tipo de valor que tomará la entrada y es del tipo InputType, que se describirá más adelante.

*aPredicate* es una función que recibe como parámetro una instancia de la clase Value, y devuelve un valor booleano que indica si esa instancia representa un valor de entrada correcto para la operación.Si el programador no desea realizar este tipo de validación el valor de aPredicate deberá ser "NilPredicate".

*aChoiceSet* es una lista de valores posibles que puede tomar la entrada, en la que cada valor es una instancia de la clase Value. Si el programador no desea especificar un conjunto restringido de valores posibles para la entrada, el valor de aChoiceSet deberá ser "Nil".

*aMsgError* es un String que contiene el mensaje de error a mostrar en caso de que el usuario especifique un valor incorrecto para la entrada.

Los parámetros anteriores utilizan instancias de la clase Value, que representan un valor. La forma general de crear un valor para una entrada es la siguiente :

```
aValue:=New(ValuePtr.SetValue( Val : aSize)) .
```

donde:

*aValue* es la nueva instancia creada y debe ser declarada del tipo ValuePtr.

*Val* es una variable que contiene un valor y que debe ser declarada del tipo del valor que contiene.

*aSize* es el tamaño en bytes de la variable Val.

El siguiente ejemplo ilustra la creación completa de una entrada para un comando:

```
Var
   MsgError : String;
   V1,V2,V3 : Integer;
   Value1,Value2,Value3,DefaultValue1 : ValuePtr;
   anInput1 : InputPtr;
   aChoiceSet : ValueListPtr;
     .
     .
     .
Function aPredicate(aValue : ValuePtr) : Boolean;
Var I : Integer;
Begin
   aValue^.GetValue(I);
   aPredicate:=(I<=3) And (I>=1);
End;
   .
   .
   .
 aCommand:=New(CommandPtr,Init(anAssociator));
 V1:=1;
```

```
Value1:=New(ValuePtr,SetValue(V1,SizeOf(V1)));
V2:=2;
Value2:=New(ValuePtr,SetValue(V2,SizeOf(V2)));
V3:=3;
Value3:=New(ValuePtr,SetValue(V3,SizeOf(V3)));
DefaultValue:=New(ValuePtr,SetValue(V2,SizeOf(V2)));
aChoiceSet:=New(ValueListPtr,Init);
aChoiceSet^.AddLast(Value1);
aChoiceSet^.AddLast(Value2);
aChoiceSet^.AddLast(Value3);
MsgError:='El valor ingresado es incorrecto';
anInput := New (InputPtr,Init (DefaultValue,IntegerType,aPredicate,aChoiceSet,MsgEr
ror));
aCommand^.AddInput(anInput);
```

*anInput* es una entrada para el comando *aCommand*.La asociación entre anInput y aCommand se hace a través del método "addInput".

Previamente se crearon las instancias de Value que representan los posibles valores que puede tomar la entrada anInput,en este caso los enteros 1, 2 y 3. Así, fue generado el ChoiceSet "aChoiceSet" asignándole dichos valores a través del método "addLast".

El valor default de anInput será 2.

La función *aPredicate* validará que el valor de anInput esté entre 1 y 3.

Las formas de ingresar los valores para una entrada son las siguientes:

a) Cajas de diálogo.

b) Puntos sobre el área de trabajo de una ventana.

La forma general para la creación de una caja de diálogo es la siguiente:

```
aDialogueBox:=New(DialogueBoxPtr,Init(aTitle,aFrame))
```

donde:

*aDialogueBox* es la nueva instancia creada y debe ser declarada del tipo DialogueBoxPtr.

*aTitle* es el título asignado a la caja de diálogo.

*aFrame* es una instancia de la clase Rectangle que indica la zona de la pantalla asociada a la caja de diálogo.

El siguiente ejemplo ilustra la creación completa de una caja de diálogo utilizada por el comando "aCommand" para obtener el valor de la entrada anInput.

```
Var
   anInputCommand : InputCommandPtr;
   aPoint1,aPoint2 : PointPtr;
   aFrame : RectanglePtr;
   aDialogBox : DialogBoxPtr;
   .
   .
   .
   aPoint1:=New(PointPtr,Init(GetMaxX div 2-200,GetMaxY div 2-100));
   aPoint2:=New(PointPtr,Init(GetMaxX div 2+200,GetMaxY div 2+100));
   aFrame:=New(RectanglePtr,Init(aPoint1,aPoint2));
   DB:=New(DialogueBoxPtr,Init('Ejemplo',aFrame));
   anInputCommand:=New(InputCommandPtr,Init(anInput));

aFrame:=New(RectanglePtr,Init(New(PointPtr,Init(180,100)),New(PointPtr,Init(220,120))));

   aDialogBox^.AddOption(anInputCommand,aFrame,'UNO',Value1);
aFrame:=New(RectanglePtr,Init(New(PointPtr,Init(180,140)),New(PointPtr,Init(220,160))));

   aDialogBox^.AddOption(anInputCommand,aFrame,'DOS',Value2);

aFrame:=New(RectanglePtr,Init(New(PointPtr,Init(180,180)),New(PointPtr,Init(220,200))));
```

```
aDialogBox^.AddOption(anInputCommand,aFrame,'TRES',Value3);
aCommand^.SetBox(aDialogBox);
```

*anInputCommand* es una instancia de la clase InputCommand y debe ser declarada del tipo InputCommandPtr.
*aDialogBox* es una caja de diálogo asignada al comando "aCommand" a través del método SetBox.
El método *AddOption* asigna a "aDialogBox" un posible valor de la entrada con una presentación asociada a él.
Dentro de la caja de diálogo "aDialogBox" podríamos permitir el ingreso de un valor de entrada por edición de texto.El siguiente ejemplo muestra la especificación de una entrada de este tipo:

```
aFrame:=New(RectanglePtr,Init(New(PointPtr,Init(280,220)),New(PointPtr,Init(380,230)))
);
aDialogBox^.AddInputText(anInput,aFrame,'Valor de entrada: ');
```

La asociación entre una entrada por texto y una caja de diálogo se realiza a través del método "AddInputText".
La forma de especificar entradas a partir de puntos sobre el área de trabajo de una ventana es la siguiente:

**Comm^.AddPointInput** , donde *Comm* es un comando.

Es posible que un programador desee habilitar o inhabilitar la invocación a una operación en forma dinámica. Por ejemplo:

```
aCommand^.DisableCommand
```

inhabilita la invocación de la operación asociada a "aCommand". En forma análoga,

```
aCommand^.EnableCommand
```

habilita la invocación de la operación asociada a "aCommand".

La asociación de un presentador a un comando se realiza a través del método "AddPresenter".Por ejemplo:

```
aCommand^.AddPresenter(aWindow^.GetPresenter)
```

asocia a "aCommand" el presentador correspondiente al área de trabajo de *aWindow*. Así, si la operación asociada a "aCommand" produce algún cambio que se debe comunicar al usuario de la interfase, deberá enviar el mensaje:

```
aCommand^.Changed
```

con el fin de que "aCommand" ordene a los presentadores asociados que remuestren su contenido. En el ejemplo anterior,el envío de tal mensaje provocará que sea remostrada el área de trabajo de aWindow.

## 4.3 - Creación de Menúes.

La forma general de crear un menú es la siguiente :

```
aMenu:=New(MenuPtr,Init) ,
```

donde:

*aMenu* es la nueva instancia creada y debe ser declarada del tipo MenuPtr.

El siguiente ejemplo muestra la creación de un menu:

```
Var
```

```
aMenu : MenuPtr;
.
.
.
aMenu:=New(MenuPtr,Init);
aMenu^.AddSubMenu('Windows');
aMenu^.AddSubMenu('Draw');
aMenu^.AddSubMenu('Edit');
aMenu^.AddSubMenu('Disk');
aMenu^.AddItem('Windows','Open  Window  1',aCommand1);
aMenu^.AddItem('Windows','Open  Window  2',aCommand2);
aMenu^.AddItem('Windows','Close  Window  1',aCommand3);
aMenu^.AddItem('Windows','Close  Window  2',aCommand4);
```

El método "AddSubmenu" agrega un ítem al menú "aMenu". El método "AddItem" agrega una opción a un Item de "aMenú" con un comando asociado a ser invocado cuando la opción sea seleccionada. En el ejemplo, *aMenu* tendrá cuatro ítems:

*Windows, Draw, Edit y Disk.*

El item "Windows" tendrá cuatro opciones :
*OpenWindow1*, con *aCommand1* como su comando asociado,
*OpenWindow2*, con *aCommand2* como su comando asociado,
*CloseWindow1*, con *aCommand3* como su comando asociado, y
*CloseWindow2*, con *aCommand4* como su comando asociado.

## 4.4 - Creación de Ventanas.

La forma general de crear una ventana es la siguiente :

```
aWindow:=New(WindowPtr.Init(aTitle.aFrame.aCommand.aPages)) ,
```

donde:

*aWindow* es la nueva instancia creada y debe ser declarada del tipo WindowPtr.
*aTitle* es el label de la ventana, del tipo String.
*aFrame* es una instancia de la clase Rectangle que indica la zona de la pantalla que ocupará la ventana.
*aCommand* es una instancia de la clase Command que indica el comando asociado al área de trabajo de la ventana "aWindow".
*aPages* es el número máximo de páginas que tendrá la ventana.
Como ejemplo, podemos crear la siguiente ventana:

```
Var   aWindow : WindowPtr;
   .
   .
   .
aFrame:=New(RectanglePtr.Init(New(PointPtr.Init(50.30)).New(PointPtr.Init(GetMaxX-50.GetMaxY-30))));
aWindow:=New(WindowPtr.Init('WINDOW  TITLE'.aFrame.aCommand.2));
```

Para que una ventana aparezca sobre la pantalla, deberá recibir el siguiente mensaje:

```
aWindow^.OpenWindow
```

Análogamente, para cerrar una ventana se deberá enviar el siguiente mensaje:

```
aWindow^.CloseWindow.
```

Para escribir un carácter sobre una ventana, simplemente se le deberá enviar el

mensaje:

**aWindow^.WriteChar(aChar)** ,donde *aChar* es del tipo Char.

Para escribir una línea sobre una ventana, simplemente se le deberá enviar el mensaje:

**aWindow^.WriteLine(aLine)** ,donde *aLine* es del tipo String.

Finalmente, el método "SetChangedArea" informa a la UIMS la porción del área de trabajo que debe ser remostrada al ser enviado el mensaje "Changed" al comando asociado.Por ejemplo:

```
Graph.Rectangle(X1,Y1,X2,Y2);
ActiveWindow^.SetChangedArea(X1,Y1,X2,Y2);
aCommand^.Changed;
```

dibujará un rectángulo en la ventana activa.

## 4.5 - Creación de Iconos.

La forma general de crear un icono es la siguiente :

```
anIcon:=New(IconPtr,Init(aRecno,aCommand)) .
```

donde:
*anIcon* es la nueva instancia creada y debe ser declarada del tipo IconPtr.
*aRecno* es el número de registro dentro del archivo "Iconos.Dat" que contiene la figura del icono.
*aCommand* es una instancia de la clase Command que contiene el comando asociado al icono.

Para crear un nuevo registro en el archivo antes nombrado, se debe crear la imagen e incorporarla al archivo usando las rutinas del Turbo Pascal Database ToolBox.El registro es de la forma:

```
IconRecord=Record
   Status : LongInt;
   X : Integer;
   Y :Integer;
   Bytes : Array[1..1306] of Bytes;
End;
```

donde *X* e *Y* indican las coordenadas de la pantalla donde debe aparecer el icono, y *Bytes* contiene la imagen previamente creada y tomada con la operación GetImage.

# 5. IMPLEMENTACION.

En este capitulo, serán desarrollados los aspectos más importantes de la implementación del modelo de Genius, la cual fue llevada a cabo en el paradigma de programación orientada a objetos. Se describirá entonces la jerarquía de clases que componen la implementación y cuáles son los métodos correspondientes a cada una de ellas.

## 5.1 - Las Clases de Genius

Primero vamos a enumerar la jerarquia de las clases fundamentales en la implementación de Genius, las cuales se corresponden con los cuatro componentes básicos descriptos en el capítulo anterior, que son:
Además de estas clases, Genius posee otras que le permiten llevar a cabo tareas adicionales, tal como:
En la siguiente sección se hará una descripción más detallada de cada una de estas clases.

## 5.2 Enciclopedia de Clases

**Recognizer**
Como se ha expresado en varias oportunidades a lo largo de este texto, los reconocedores tienen como función principal capturar eventos provenientes del teclado o del mouse y transmitirlos hacia los comandos. Por lo tanto esa es la función implementada en la clase Recognizer y en cada una de sus subclases, variando solamente entre ellas el tipo de objetos dentro de la UIMS con los cuales están relacionados. Vamos a encontrar los reconocedores propiamente dichos, los cuales se comunican directamente con comandos de la UIMS, y otros reconocedores que se comunican, no con comandos, sino con otros objetos encargados de realizar tareas especiales dentro de la UIMS, tal como administraciones dentro de las ventanas y de los menúes.

*Variables de instancia:*

**Active:** Contiene true cuando el reconocedor está activo. En caso contrario contiene false. Frame: Contiene el rectángulo de la pantalla dentro del cual el reconocedor procesa eventos de entrada. aCommand: Contiene la instancia de la clase Command a la cual el reconocedor comunica los eventos de entrada significativos.

*Métodos de instancia:*

**Init(aRectangle,aCommand)**

Inicializa una nueva instancia de esta clase con aRectangle como zona de reconocimiento y aCommand como comando asociado.

**Done**

Libera la memoria asociada al receptor cuando este es destruido.

**SetActive**

Setea la variable de instancia Active del receptor en true.

**ChangeFrame(NewFrame)**

Cambia la variable de instancia frame del receptor a NewFrame.

**ActiveRec**

Retorna true si el receptor es el reconocedor activo. En caso contrario retorna false.

**Command**

Retorna la variable de instancia aCommand del objeto receptor.

**Activate**

Pone como activo al receptor y realiza todo el procesamiento de eventos de entrada mientras este reconocedor tenga el cursor dentro de su frame. Este es el loop principal de procesamiento de eventos de entrada para un reconocedor.

**Deactivate**

Pone en false la variable de instancia Active del objeto receptor y por lo tanto lo desactiva.

**ProcessInput**

Procesa los eventos de entrada mientras el receptor se encuentra activo.

**ProcessLastInput(aCharacter)**

Procesa el útimo evento de entrada antes que el receptor sea desactivado.

**ProcessFunctionKey(aCharacter)**

Procesa un evento de entrada cuando éste se trata de una selección o de la presión de botón del mouse.

**ProcessInputKey(aCharacter)**

Procesa un evento de entrada cuando éste se trata de una entrada del caracteres desde teclado.

**hasCurser**

Retorna true si el receptor tiene el cursor dentro de su frame. En caso contrario retorna false.

**isWindowRecognizer**

Retorna true si el objeto receptor es una instancia de la clase WindowRecognizer. En caso contrario retorna false.

**isMenuRecognizer**

Retorna true si el objeto receptor es una instancia de la clase MenuRecognizeer. En caso contrario retorna false.

**GetFrame**

Retorna la variable de instancia Frame del receptor.

## IconRecognizer:

La función de esta clase es reconocer eventos cuando el cursor del mouse se encuentra dentro de alguno de los iconos que pueden estar dispersos en la pantalla. Un objeto de esta clase, como los de cualquier reconocedor, es capaz de administrar los movimientos del cursor dentro de su zona de influencia y además de aceptar la invocación a la operación que está representada por la figura del icono.

*Variables de instancia:*

**Active:** (de la clase Recognizer)

**Frame:** (de la clase Recognizer)

*Métodos de instancia:*

**Activate:**

Activa al receptor y procesa eventos de entrada mientras el cursor de mouse se encuentra dentro de su frame.

**ProcessFunctionKey:**

Procesa un evento de entrada cuando este se trata de una selección o de la presión de un botón del mouse.

### WindowRecognizer:

Este tipo de reconocedor tiene un comportamiento muy simple ya que su única función es, cuando es activado, invocar al objeto encargado de gerenciar el procesamiento dentro de una ventana, es decir, una instancia de la clase WindowCommand.

*Variables de instancia:*

**Active:** (de la clase Recognizer)

**Frame:** (de la clase Recognizer)

**WindowCommand:** Contiene una instancia de la clase WindowCommand, encargada de gerenciar la actividad dentro de una ventana.

*Métodos de instancia:*

**Init(aFrame,aWindowCommand)**

Inicializa una nueva instancia de esta clase con aFrame como frame y aWindowCommand como el objeto al cual el reconocedor comunica los eventos de entrada significativos.

**Done**

Libera la memoria asignada a una instancia de esta clase cuando es liberada.

**Activate**

Pone como activo al reconocedor receptor e inmediatemente invoca a la instancia de la clase WindowCommand que tiene asociada.

**GetWindowCommand**

Retorna la instancia de la clase WindowCommand asociada al receptor.

### SubWindowRecognizer:

Bajo este tipo de reconocedor se agrupan los distintos reconocedores correspondientes a las distintas partes de la ventana descriptas previamente. Estos reconocedores tienen la particularidad de que no comunican los eventos que reciben a los comandos, sino que lo hacen al objeto encargado de administrar el control de la ventana (una instancia de la clase WindowCommand), para que se realice la acción seleccionada. Estas acciones pueden ser mover una ventana, redimensionarla, realizar scrolls en distintas direcciones y otras ya mencionadas con anterioridad.

*Variables de instancia:*

**Active**: (de la clase Recognizer)

**Frame**: (de la clase Recognizer)

*Metodos de instancia:*

**Init(aFrame)**

Inicializa una nueva instancia de sta clase con aFrame como frame.

**Done**

Libera la memoria asociada a una instancia de esta clase cuando esta es liberada.

**ProcessFunctionKey(aCharacter)**

Procesa un evento de entrada cuando éste se trata de una selección o de la presión de un botón del mouse.

**SetFrame(aFrame)**

Setea el frame del receptor con aFrame.

### WorkAreaRecognizer:

Estos reconocedores corresponden a la zona de trabajo dentro de una ventana. Su función es administrar el movimiento del cursor dentro de esta área de trabajo y aceptar el tipeo de caracteres por parte del usuario. Cabe mencionar que la especificación de entradas que se puede realizar dentro de esta área es función de los objetos de la clase PointInputRecognizer, que será descripta luego.

*Variables de instancia:*

**Active**: (de la clase Recognizer)

**Frame**: (de la clase Recognizer)

*Métodos de instancia:*

**Init(aFrame,aCommand)**

Inicializa una nueva instancia de esta clase con aFrame como frame y aCommand como comando.

**ProcessFunctionKey**

Procesa un evento cuando éste se trata de una selección o de la presión de un botón del mouse.

**ProcessInputKey**

Procesa un evento de entrada cuando éste se trata de un carácter tipeado por el usuario, el en caso de ingreso de texto.

## PointRecognizer:

Un objeto de esta clase se encarga de reconocer eventos de movimiento de cursor y presión de botones del mouse cuando se esta realizando un movimiento o un redimensionamiento de una ventana.

*Variables de instancia:*

**Active:** (de la clase Recognizer)

**Frame:** (de la clase Recognizer)

*Métodos de instancia:*

**Activate**

Procesa un movimiento del cursor cuando se está realizando una operación de movimiento o redimensionamiento de una ventana.

**ProcessFunctionKey(aCharacter)**

Procesa una selección o la presión de un botón del mouse cuando se está realizando una operación de movimiento o redimensionamiento de una ventana.

**ScreenRecognizer:**

Un objeto de esta clase reconoce los movimientos del cursor sobre el fondo de la pantalla. Este tipo de reconocedor es el que, por defecto, toma el control cuando ningún otro objeto dentro de la pantalla puede ser activado. Obviamente las únicas acciones que pueden ser realizadas en esta zona, y por lo tanto las únicas que este reconocedor permite realizar, son movientos del cursor del mouse.

*Variables de instancia:*

**Active:** (de la clase Recognizer)

**Frame:** (de la clase Recognizer)

*Métodos de instancia:*

**Activate**

Procesa un movimiento del cursor del mouse y devuelve el control al scheduler.

**MenuRecognizer:**

El accionar de estos reconocedores es muy simple ya que un objeto de esta clase, al ingresar el cursor dentro de su zona de influencia y por ende al activarse, inmediatemente comunica este evento al objeto encargado de administrar el control dentro del menú, es decir una instancia de la clase MenuCommand.

*Variables de instancia:*

**Active:** (de la clase Recognizer)

**Frame:** (de la clase Recognizer)

**aMenuCommand:** Contiene la instancia de la clase MenuCommand asociada con el receptor.

*Métodos de instancia:*

**Init(aMenuComm)**

Incializa al receptor y le asigna aMenuComm como la instancia de la clase MenuCommand asociada.

**Done**

Libera la memoria asociada al receptor cuando éste es destruido.

**Activate**

Pone como activo al receptor e inmediatemente invoca a la instancia de la clase MenuCommand asociada.

**GetMenuCommand**

Retorna la instancia de la clase MenuCommand asociada con el receptor.

### SubMenuRecognizer:

Estos reconocedores tienen un funcionamineto similar a los anteriores pero a nivel de submenúes, activando al objeto administrador de los submenúés, una instancia de la clase SubMenuCommand, cada vez que él es activado.

*Variables de instancia:*

**Active:** (de la clase Recognizer)

**Frame:** (de la clase Recognizer)

**aSubMenuCommand:** Contiene una instancia de la claseSubMenuCommand asociada con el receptor.

*Métodos de instancia:*

**Init(aFrame,aSubMenuComm)**

Inicializa al receptor y le asigna aFrame como frame y aSubMenuComm como la instancia de la clase SubMenuCommand asociada.

**Done**

Libera la memoria asociada al receptor cuando éste es liberado.

**Activate**

Pone como activo al receptor e inmediatemente invoca a la instancia de la clase SubMenuCommand asociada.

**SubMenuCommand**

Retorna la instancia de la clase SubMenuCommand asociada con el receptor.

### DefaultSubMenuRecognizer:

La tarea de estos reconocdores es muy simple, ya que sólo permiten realizar movimientos del cursor dentro de la zona del menú que no tiene submenúes asociada.

*Variables de instancia:*

**Active:** (de la clase Recognizer)

**Frame:** (de la clase Recognizer)

*Métodos de instancia:*

**Init**

Inicializa al receptor cuando éste es creado.

**Activate**

Procesa un evento de entrada e inmediatemente retorna el control a la instancia de la clase MenuCommand que lo invocó.

## InputRecognizer:

*Variables de instancia:*

**Active:** (de la clase Recognizer)

**Frame:** (de la clase Recognizer)

**aInputCommand:**Contiene la instancia de la clase InputCommand asociada con este reconocedor

**Value:**Contiene el corriente valor correspondiente a la entrada asociada con este reconocedor.

*Métodos de instancia:*

**Init(aFrame,aInputCommand,aValue)**

Inicializa al receptor y le asigna aFrame como frame, aInputCommand como la instancia de la clase InputCommand asociada y aValue como el valor de la entrada correspondiente al receptor.

**Done**

Libera la memoria asociada al receptor cuando éste es destruido.

**GetValue**

Retorna el contenido de la variable de instancia Value del receptor.

**ProcessFunctionKey(aCharacter)**

Procesa un evento de entrada cuando éste se trata de la presión de un botón del mouse.

### DefaultInputRecognizer:

*Variables de instancia:*

**Active:** (de la clase Recognizer)

**Frame:** (de la clase Recognizer)

**aDefaultInputCommand:** Contiene la instancia de la clase DefaultInputCommand asociada con este reconocedor.

*Métodos de instancia:*

**Init(aFrame,aDefaultInputCommand,aValue)**

Inicializa al receptor con aFrame como su frame, aDefaultInputCommand como la instancia de la clase DefaultInputCommand asociada y aValue como el valor asociado a este reconocedor.

**Done**

Libera la memoria asignada al receptor cuando éste es destruido.

**GetDefaultCommand**

Retorna la instancia de la clase DefaultInputCommand asociada con el receptor.

**ProcessFunctionKey**

Procesa un evento de entrada cuando éste se trata de la presión de un botón del mouse.

### InputTextRecognizer:

Estos reconocedores tiene una sobrecarga de tareas con respecto a los ya descriptos, ya que además de las cuestiones relacionadas con movimientos del cursor deben reconocer eventos desde el teclado, tal como el tipeo de caracteres correspondiente a la especificación de entradas en forma de texto.

*Variables de instancia:*

**Active:** (de la clase Recognizer)

*Frame:* (de la clase Recognizer)

**aInputCommand:** Contiene la instancia de la clase InputCommand asociada con los objetos de esta clase.

*Métodos de instancia:*

**Init(aFrame,aInputCommand)**

Inicializa al receptor asignándole aFrame como su frame y aInputCommand como la instancia de la clase InputCommand asociada.

**Done**

Libera la memoria asignada al receptor, cuando éste es liberado.

**GetCommand**

Retorna la instancia de la clase InputCommand asociada con el receptor.

**ProcessFunctionKey(aCharacter)**

Procesa un evento de entrada cuando éste se trata de la presión de un botón del mouse.

**ProcessInputKey(aCharacter)**

Procesa un evento de entrada cuando éste se trata de un carácter ingresado por el usuario en una entrada de texto.

## ErrorRecognizer:

*Variables de instancia:*

**Active:** (de la clase Recognizer)

**Frame:** (de la clase Recognizer)

**ErrorCommand:** Contiene la instancia de la clase ErrorCommand asociada con este reconocedor.

*Métodos de instancia:*

**Init(aFrame,aErrorCommand)**

Inicializa al receptor y le asigna aFrame como su frame y aErrorCommand como la instancia de la clase ErrorCommand asociada.

**Done**

Libera la memoria asignada al receptor, cuando éste es liberado.

**GetErrorCommand**

Retorna la instancia de la clase ErrorCommand asociada al receptor.

**ProcessFunctionKey(aCharacter)**

Procesa un evento de entrada cuando éste se trata de la presión de un botón del mouse.

### DefaultDialogBoxRecognizer:

La terea de estos reconocedores es muy simple, ya que sólo se encargan de reconocer eventos de movimientos del cursor cuando éste se encuentra sobre el fondo de una Dialog Box o fuera de ella.

*Variables de instancia:*

**Active:** (de la clase Recognizer)

*Frame:* (de la clase Recognizer)

*Métodos de instancia:*

**Init**

Inicializa al receptor cuando éste es creado.

**Activate**

Procesa un evento de movimiento del cursor cuando éste se encuentra en el fondo de una dialog box

**ProcessFunctionKey(aCharacter)**

Procesa un evento de entrada cuando éste se trata de la presión de un boton del mouse.

### HelpRecognizer:

*Variables de instancia:*

**Active:** (de la clase Recognizer)

**Frame:** (de la clase Recognizer)

**Value:** Contiene el valor del ítem del help correspondiente a este reconocedor.

*Métodos de instancia:*

**Init(aFrame,aValue)**

Inicializa al receptor y le asigna aFrame como su frame y aValue como el valor correspondiente al ítem del help asociado.

**Done**

Libera la memoria asignada al receptor, cuando éste el destruido.

**GetValue**

Retorna el valor de la variable de instancia Value del receptor.

**ProcessFunctionKey(aCharacter)**

Procesa un evento de entrada cuando éste se trata de la presión de un botón del mouse.

## DefaultHelpRecognizer:

Tal como en los objetos de la clase DefaultDialogBoxRecognizer, estos objetos reconocen eventos de movimientos del cursor del mouse cuando éste se encuentra sobre el fondo de una ventana de help o fuera de ella.

*Variables de instancia:*

**Active:** (de la clase Recognizer)

**Frame:** (de la clase Recognizer)

*Métodos de instancia:*

**Init**

Inicializa al receptor cuando éste es creado.

**Activate**

Procesa un evento de movimiento del cursor cuando éste se encuentra en el fondo de una ventana de help.

**ProcessFunctionKey(aCharacter)**

Procesa un evento de entrada cuando éste se trata de la presión de un botón del mouse.

## Presenter

Ya se ha mencionado anteriormente que la función de los presentadores es mostrar en la pantalla visiones de objetos de la aplicación. Debido a que esas visiones dependen del tipo de objeto que se desee presentar, la clase Presenter es una clase abstracta pero que provee soporte para todas sus subclases.

*Variables de instancia:*

**Frame**: Contiene la zona rectangular de la pantalla en la cual el presentador mostrará su imagen asociada.

*Métodos de instancia:*

**Init(aFrame)**

Inicializa al objeto receptor asignando aFrame como el frame correspondiente.

**Done**

Libera la memoria asignada al receptor, cuando éste es destruido.

**ChangeFrame(NewFrame)**

Cambia el frame del receptor por el nuevo frame NewFrame.

**Display**

Muestra la imagen asociada al receptor. Este es el método principal de un objeto de la clase Presenter y de todas sus subclases.

**GetFrame**

Retorna el frame correspondiente al receptor.

**SetFrame(X1,Y1,X2,Y2)**

Cambia las coordenadas del frame del receptor por las nuevas X1,Y1,X2 e Y2.

**HighLight**

Hace titilar la imagen presentada en pantalla por el recptor.

**IsIconPresenter**

Retorna true si el receptor es una instancia de la clase IconPresenter. En caso contrario retorna false.

### IconPresenter

Los objetos de esta clase tienen como función presentar la figura de los iconos existentes en la pantalla.

*Variables de instancia:*

**Frame**: (de la clase Presenter)

**RecNo**: Contiene el número del registro donde se encuentra la imagen asociada al presentador dentro del archivo ICONOS.DAT.

*Métodos de instancia:*

**Init(aFrame,aRecno)**

Inicializa al receptor asignándole aFrame como su frame y aRecno como el número del registro donde se encuentra la imagen dentro del archivo ICONOS.DAT

Display Mustra en la pantalla la imagen correspondiente al receptor.

### WorkAreaPresenter

Esta clase es quizás una de los presentadores más importantes de Genius, ya que se encarga de presentar los objetos existentes en el área de trabajo de la ventana. Hay una interacción constante entre este presentador y los objetos de la aplicación que se desean mostrar o remostrar, ya que el WorkAreaPresenter consulta con a modelo para conocer cuáles porciones del área de trabajo tienen que presentarse nuevamente y cuáles no.

*Variables de instancia:*

**Frame**: (de la clase Presenter)

**VisualFrame**: Contiene el frame que indica la porción de la página que está siendo visualizada.

**ChangedArea**: Contiene el frame que indica la zona de la página que fue recientemente modificada.

**CurrentPage**: Contiene el número de página que está siendo visualizada.

**Pages**: Contiene el total de páginas que tiene la ventana.

**LineStyle**: Contiene el estilo de líneas corriente.

WithStyle: Contiene el estilo de grosor de líneas corriente.

*FontStyle:* Contiene el estilo de letras corriente.

**FontSize**: Contiene el tamaño de letras corriente.

*Métodos de instancia:*

**Init(aFrame,aPages)**

Inicializa al receptor asignándole aFrame como su frame y aPages como el total de páginas de la ventana.

**Display**

Muestra la porción del área que tiene que ser remostrada, por orden del comando

correspondiente.

**SetChangedArea(aFrame)**

Setea el valor de la zona cambiada del área de trabajo con aFrame.

**SetVisualFrame(aFrame)**

Setea el valor de la zona visible del área de trabajo con aFrame.

**GetVisualFrame**

Retorna el frame correspondiente a la zona visible del área de trabajo.

**SetCurrentPage(aPage)**

Setea el valor de la página corriente con aPage.

**GetCurrentPage**

Retorna la página del receptor que está siendo visualizada.

**GetPages**

Retorna el número de paginas asignado al receptor en su creación.

**SetLineStyle(Style)**

Setea el estilo de líneas de la ventana con Style.

**GetLineStyle**

Retorna el corriente estilo de líneas de la ventana.

**SetWidthStyle(Style)**

Setea el estilo de ancho de líneas de la ventana con Style.

**GetWidthStyle**

Retorna el corriente estilo de ancho de líneas de la ventana.

**SetFontStyle(Style)**

Setea el estilo de letras de la ventana con Style.

**GetFontStyle**

Retorna el corriente estilo de letras de la ventana.

**SetFontSize(Style)**

Setea el valor correspondiente al tamaño de las letras de la ventana.

**GetFontSize**

Retorna el corriente tamaño de letras de la ventana.

BIBLIOTECA
FAC. DE INFORMATICA
U.N.L.P.

**LabelPresenter**

Este presentador muestra en la pantalla la zona de label de una ventana, la cuál contiene el nombre de la misma y el número de la página que está siendo visualizada.

*Variables de instancia:*

**Frame:** (de la clase Presenter)

**Titulo:** Contiene el label de la ventana.

*Métodos de instancia:*

**Init(aTitulo,aFrame)**

Inicializa al receptor asignandole aTitulo como su label y aFrame como su frame.

**Display**

Muestra la zona de label de la ventana así como el número de página corriente.

**DisplayPageNumber**

Muestra la zona del label correspondiente al número de página corriente.

**Hide**

Realiza un ocultamiento de la zona de label de la ventana cuando la ventana se desactiva.

**UpArrowPresenter**

Este presentador muestra la zona de la ventana que permite moverse hacia las páginas superiores dentro de una ventana.

*Variables de instancia:*

**Frame:** (de la clase Presenter).

*Métodos de instancia:*

**Init(aFrame)**

Inicializa al receptor asignándole aFrame como su frame.

*Display*

Muestra la imagen correspondiente a la zona de "página hacia arriba" de la ventana.

**DownArrowPresenter**

Este presentador muestra la zona de la ventana que permite moverse hacia las páginas inferiores dentro de una ventana.

*Variables de instancia:*

**Frame**: (de la clase Presenter).

*Métodos de instancia:*

**Init(aFrame)**

Inicializa al receptor asignándole aFrame como su frame.

**Display**

Muestra la imagen correspondiente a la zona de "página hacia abajo" de la ventana.

### ConfigPresenter

Este presentador muestra la aona de la ventana que permite abrir una "dialog box" para configurar los aspectos de presentación de una ventana.

*Variables de instancia:*

**Frame**: (de la clase Presenter).

*Métodos de instancia:*

**Init(aFrame)**

Inicializa al receptor asignándole aFrame como su frame.

**Display**

Muestra la imagen correspondiente a la zona de configuración de los aspectos de visualización de la ventana.

### ResetScrollPresenter

Este presentador muestra la zona de la ventana que permite reinicializar el estado del scroll.

*Variables de instancia:*

**Frame**: (de la clase Presenter).

*Métodos de instancia:*

**Init(aFrame)**

Inicializa al receptor asignándole aFrame como su frame.

**Display**

Muestra la imagen correspondiente a la zona de reinicialización del estado del scroll de la ventana.

### BarPresenter

Este presentador muestra las zonas de la ventana que indican las porciones del área de trabajo que no estan siendo visualizadas.

*Variables de instancia:*

**Frame:** (de la clase Presenter).

*Métodos de instancia:*

**Init(aFrame)**

Inicializa al receptor asignándole aFrame como su frame.

**Display**

Muestra la imagen correspondiente a las zonas del área de trabajo no visibles dentro de la ventana.

### ScrollDefaultPresenter

Este presentador muestra las zonas de la ventana que indican la porción del área de trabajo que se está visualizando.

*Variables de instancia:*

**Frame:** (de la clase Presenter).

*Métodos de instancia:*

**Init(aFrame)**

Inicializa al receptor asignándole aFrame como su frame.

**Display**

Muestra la imagen correspondiente a la zona visible del area de trabajo dentro de la ventana

### MovePresenter

Este presentador muestra la zona de la ventana que permite realizar movimientos de la ventana dentro de la pantalla.

*Variables de instancia:*

**Frame:** (de la clase Presenter).

*Métodos de instancia:*

**Init(aFrame)**

Inicializa al receptor asignándole aFrame como su frame.

**Display**

Muestra la imagen correspondiente a la zona de movimiento de la ventana.

### ReframePresenter

Este presentador muestra la zona de la ventana que permite redimensionar la misma.

*Variables de instancia:*

**Frame:** (de la clase Presenter).

*Métodos de instancia:*

**Init(aFrame)**

Inicializa al receptor asignándole aFrame como su frame.

**Display**

Muestra la imagen correspondiente a la zona de redimensionamiento de la ventana.

### ItemPresenter

Los objetos de esta clase tienen la función de mostrar en video inverso o en video normal los items pertenecientes a un submenú, de acuerdo a si el cursor del mouse está o no apuntándolos.

*Variables de instancia:*

**Frame:** (de la clase Presenter).

**Titulo:** Contiene el item que se debe presentar.

Métodos de instancia:

**Init(aString,aFrame)**

Inicializa al receptor asignándole aFrame como su frame y aString como el título correspondiente al ítem.

**Display**

Remuestra el ítem receptor cuando es apuntado por el cursor del mouse.

### DefaultItemPresenter

Los objetos de esta clase muestran la porción de los menúes que no tienen ítems asociados.

*Variables de instancia:*

**Frame**: (de la clase Presenter)

*Métodos de instacia:*

**Init**

Incializa al receptor cuando éste es creado

**Display**

Muestra la porción de los menúes que no tienen ítems asociados.

### InputPresenter

Estos reconocedores procesan eventos de entrada cuando el cursor del mouse se encuentra dentro de un ítem que indica uno de los posibles valores de una entrada en una dialog box.

*Variables de instancia:*

**Frame**: (de la clase Presenter)

**Titulo**: Contiene el título correspondiente a la entrada asociada con el presentador.

*Métodos de instacia:*

**Init(aString,aFrame)**

Inicializa al receptor asignándole aFrame como su frame y aString como el título correspondiente a la entrada asociada.

**Display**

Muestra en video inverso el ítem correspondiente a un posible valor que puede tomar una entrada, cuando éste es selccionado. Si el receptor está seleccionado realiza

la operación inversa.

**Show**

Muestra el item correspondiente a un posible valor que puede tomar una entrada dentro de una dialog box.

### InputTextPresenter

Estos presentadores muestran el texto que está siendo tipeado por el usuario cada vez que está relizando una entrada de texto.

*Variables de instancia:*

**Frame:** (de la clase Presenter)

**Titulo:** Contiene el titulo correspondiente a la entrada asociada con el presentador.

**Down:** Contiene la zona de la pantalla por debajo del texto que está siendo ingresado.

*Métodos de instacia:*

**Init(aString,aFrame)**

Inicializa al receptor asignándole aFrame como su frame y aString como el titulo correspondiente a la entrada asociada.

**DisplayText(aText)**

Remuestra el texto que está siendo ingresado por el usuario.

**Show**

Muestra el título correspondiente a la entrada relacionada con el receptor.

ShowTextCursor Muestra el cursor de texto en el lugar correspondiente.

### ScreenPresenter

Los objetos de esta clase realizan la presentación del fondo de la pantalla. Variables de instancia:

**Frame:** (de la clase Presenter)

*Métodos de instacia:*

**Display**

Muestra todo el fondo de la pantalla.

**DefaultPresenter**

Los objetos de esta clase tiene la función de presentar en la pantalla los fondos de las dialog boxes y de las ventanas de help.

*Variables de instancia:*

**Frame**: (de la clase Presenter)

*Métodos de instacia:*

**Init**

Inicializa al receptor cuando éste es creado.

**Display**

Muestra los fondos de las ventanas de help y de las dialog boxes con el color correspondiente.

**Command**

La clase Command implementa fielmente el concepto de comando explicado en las secciones anteriores. Por lo tanto una instancia de esta clase, luego de ser activada por un reconocedor, se encarga de realizar ciertas tareas antes de invocar a la operación corrrespondiente. Tales tareas son recolectar las entradas necesarias para la operación, validar las entradas ingresadas y además informar al usuario los posibles errores cometidos durante el ingreso de los datos. Por otro lado, cuando el comando es informado de algún cambio en la aplicación, avisa a los presentadores correspondientes que deben realizar alguna presentación. Para realizar la tarea de recolección de entradas, el comando cuenta con dos listas de entradas que deberá especificar el usuario. Una de ellas contiene las entradas que serán ingresadas por medio de dialog boxes, y la otra las entradas que representarán puntos dentro del área de trabajo. Una vez que las entradas fueron especificadas el comando debe validarlas. La UIMS Genius provee al programador de la interfase dos formas de realizar la validación de los datos ingresados. Una de ellas es especificado el conjunto de valores válidos para una entrada determinada, que puede encontrarse en la implementación de Genius como "Choice Set". La otra de ellas es especificando una función encargada de realizar la validación, que será encontrada en la implementación como "Predicate". Si el comando encuentra que los valores ingresados para las entradas no son correctos, puede presenatar el mensaje de error correspondiente a la entrada incorecta. Esto será explicado con mas detalle en la sección correspondiente a "Error Box". Una vez que todas las entradas fueron correctamente especificadas, el comando invoca a la operación. Obviamente, la ejecución de ésta puede provocar cambios sobre los objetos que están siendo presentados, caso en el cual la aplicación debe informar al comando del cambio producido. El comando no sabe "que" ha cambiado, pero si conoce, a traves del mensaje antes mencionado, que "algo" ha cambiado. Por lo tanto el comando recorre la lista de presentadores que posee y los hace presentar en la pantalla los aspectos del modelo que han cambiado, con lo cual finaliza la tarea del comando.

*Variables de instancia:*

**Enable:** Contiene True si el comando esta disponible o habilitado para ser invocado.

En caso contrario contiene False.

**Presenters:** Contiene la lista de presentadores asociados con el comando. Si el comando no tiene presentadores, contiene Nil.

**Associator:** Contiene el nombre del procedimiento asociador escrito por el programador y que asocia un comando con una operación de la aplicación.

**Inputs :** Contiene la lista de entradas requeridas por el comando cuando es invocado. Si el comando no tiene entradas contiene Nil.

**PointInputs :** Contiene la lista de las entradas requeridas por el comando cuando es invocado, en la que cada entrada es un punto de la pantalla que será ingresado desde una ventana. Si el comando no requiere entradas del tipo Punto, contiene Nil.

**Box:** Contiene una instancia de la clase Dialogue-Box, que es utilizada por el comando para que el usuario ingrese los valores de las entradas a una operación.

**ErrorBox :** Contiene una instancia de la clase Error-Box, que es utilizada por el comando para informar al usuario que los valores de las entradas ingresadas no han sido correctos.

**InputIterator :** Es un iterador utilizado por el comando para recorrer la lista de entradas (Inputs). Si no hay entradas, contiene Nil.

**PointInputIterator:** Es un iterador utilizado por el comando para recorrer la lista de entradas de punto (PointInputs). Si no existen tales entradas, contiene Nil.

*Métodos de instancia:*

**Init(anAssociator)**

Inicializa el receptor asignando anAssociator a la variable de instancia Associator.

**Done**

Libera la memora asignada al receptor cuando éste es destruido.

**Able**

Retorna True si el comando esta habilitado y False en caso contrario.

**EnableCommand**

Habilita al comando para que pueda ser invocado.

**DisableCommand**

Inhabilita al comando para que no pueda ser invocado.

**GetInputValue(var Valor)**

Retorna en Valor el valor de una entrada especifica del comando.

**GetPointInputValue(Var X,Y)**

Retorna en X e Y las coordenadas de una entrada del tipo punto específica.

**SetPointInput(X,Y)**

Asigna a X e Y los valores de las coordenadas correspondientes de una entrada del tipo punto ingresada por el usuario

**GetBox**

Retorna el dialogue-box del comando receptor.

**GetErrorBox**

Retorna el dialogue-box del comando receptor.

**SetErrorBox(aErrorBox)**

Asigna aErrorBox a la variable de instancia ErrorBox del comando receptor.

**RemoveErrorBox**

Libera el ErrorBox del comando receptor, asignándole Nil.

**SetBox(aBox)**

Asigna aBox a la variable de instancia Box del comando receptor.

**RemoveBox**

Libera el Box del comando receptor, asignándole Nil.

**AddPresenter(aPresenter)**

Agrega aPresenter a la lista de presenters del comando receptor.

**RemovePresenter(aPresenter)**

Remueve aPresenter de la lista de presenters del comando receptor.

**AddInput(anInput)**

Agrega anInput a la lista de inputs del comando receptor.

**AddPointInput**

Agrega una entrada del tipo punto en la lista PointInputs del comando receptor.

**Run**

Inicializa las listas de entradas a la operación. Invoca al método Run del Dialogue-Box que posee el receptor y, si los valores de entrada ingresados son

válidos, invoca al asociador apuntado por Associator para ejecutar la operación de la aplicación.

**Validate**

Retorna True si todas las inputs del comando receptor contienen valores válidos para la operación. En caso contrario al método que muestra el mensaje de error asociado, retornado False.

**RunWith(aCharacter)**

Asigna aCharacter como valor de una entrada del comando receptor.Es utilizado para las entradas de caracteres desde el teclado.

**Changed**

Envia a todos los presentadores de la lista de presentadores, excepto a los presentadores de iconos, la orden de mostrarse.

**DisplayIcons**

Envia la orden de mostrarse a todos los presentadores de iconos de la lista de presentadores.

**IconIntersection(aFrame)**

Retorna True si la intersección entre aFrame y el frame de algún presentador de iconos de la lista de presentadores no es nula. Retorna False en caso contrario.

**HighLightIcons**

Envia la orden de resaltado a los presentadores de iconos de su lista de presentadores.

## WindowCommand

Esta clase se encarga de gerenciar el procesamiento cuando el cursor se encuentra dentro de la ventana. Actúa como administradora de la activación y de desactivación de los reconocedores existente dentro de la ventana y ordena la presentaciones correspondientes a los scrolls, movimientos y redimensionamientos de la ventana, etc.

*Variables de instancia:*

**TextCursor:** Contiene una instancia de la clase TextCursor. Es el cursor de texto de la ventana asociada.

**WorkAreaRecognizer:** Contiene una instancia de la clase WorkAreaRecognizer, utilizada por la ventana asociada.

**LabelRecognizer:** Contiene una instancia de la clase SubWindowRecognizer, y es el reconocedor de la zona del label de la ventana asociada.

**MoveRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona de Move de la ventana asociada.

**ReframeRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona de Reframe de la ventana asociada.

**PageUpRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona de PageUp de la ventana asociada.

**PageDownRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona de PageDown de la ventana asociada.

**ScrollUpBarRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona de Scroll hacia arriba de la ventana asociada.

**ScrollDefaultVerticalRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona del Scroll vertical que muestra el area de la página visible de la ventana asociada.

**ScrollDownBarRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona de Scroll hacia abajo de la ventana asociada.

**PageDownRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona de cambio de página hacia abajo de la ventana asociada.

**ScrollLeftBarRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona de Scroll hacia la izquierda de la ventana asociada.

**ScrollDefaultHorizontalRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona del Scroll horizontal que muestra el area de la página visible de la ventana asociada.

**ScrollRightBarRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona de Scroll hacia la derecha de la ventana asociada.

**ConfigRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona de configuración de la ventana asociada.

**ResetScrollRecognizer:** Contiene una instancia de la Clase SubWindowRecognizer, y es el reconocedor de la zona de reseteo del scroll de la ventana asociada.

**WorkAreaPresenter:** Contiene una instancia de la clase WorkAreaPresenter, utilizada por la ventana asociada.

**LabelPresenter** :Contiene una instancia de la clase LabelPresenter, y es el presentador de la zona del label de la ventana asociada.

**MovePresenter:** Contiene una instancia de la Clase MovePresenter, y es el presentador de la zona de Move de la ventana asociada.

**ReframePresenter:** Contiene una instancia de la Clase ReframePresenter, y es el presentador de la zona de Reframe de la ventana asociada.

**PageUpPresenter:** Contiene una instancia de la Clase UpArrowPresenter, y es el presentador de la zona de PageUp de la ventana asociada.

**PageDownPresenter:** Contiene una instancia de la Clase DownArrowPresenter, y es el presentador de la zona de PageDown de la ventana asociada.

**ScrollUpBarPresenter:** Contiene una instancia de la Clase BarPresenter, y es el presentador de la zona de Scroll hacia arriba de la ventana asociada.

**ScrollDefaultVerticalPresenter:** Contiene una instancia de la Clase ScrollDefaultPresenter, y es el presentador de la zona del Scroll vertical que muestra el area de la página visible de la ventana asociada.

**ScrollDownBarPresenter:** Contiene una instancia de la Clase BarPresenter, y es

el presentador de la zona de Scroll hacia abajo de la ventana asociada.

**ScrollLeftBarPresenter:** Contiene una instancia de la Clase BarPresenter, y es el presentador de la zona de Scroll hacia la izquierda de la ventana asociada.

**ScrollDefaultHorizontalPresenter:** Contiene una instancia de la Clase ScrollDefaultPresenter, y es el presentador de la zona del Scroll horizontal que muestra el area de la página visible de la ventana asociada.

**ScrollRightBarPresenter:** Contiene una instancia de la Clase BarPresenter, y es el presentador de la zona de Scroll hacia la derecha de la ventana asociada.

**ConfigPresenter:** Contiene una instancia de la Clase ConfigPresenter, y es el presentador de la zona de configuración de la ventana asociada.

**ResetScrollPresenter:** Contiene una instancia de la Clase ResetScrollPresenter, y es el presentador de la zona de reseteo del scroll de la ventana asociada.

*Metodos de instancia:*

**Init(aLabel,aFrame,aCommand,aPages)**

Inicializa el comando de la ventana(instancia de WindowCommand) receptora, tomando aLabel como el nombre de la ventana, aFrame como el frame de la ventana, aCommand como el comando asociado al WorkAreaRecognizer, y aPages como la cantidad máxima de paginas que tendrá la ventana. Crea tambien las instancias de clases asociadas a sus variables de instancia.

Done

Libera la memoria asignada al receptor cuando éste es destruido.

**GetWorkAreaPresenter**

Retorna el WorkAreaPresenter del receptor.

**GetWorkAreaRecognizer**

Retorna el WorkAreaRecognizer del receptor.

**Run**

Activa la ventana receptora. Actúa como gerenciador de todos los reconocedores existentes en ella. Es el loop principal de procesamiento de eventos cuando el cursor se encuentra dentro de la ventana. Display Muestra la ventana completa.

**Hide**

Oculta la ventana receptora.

**SearchActiveRecognizer**

Retorna el reconocedor activo de todos los que componen la ventana.

**PerformAction**

Ordena la ejecución de la acción asociada al reconocedor activo de la ventana.

**DoNothing**

No realiza ninguna acción.

**Move**

Mueve la ventana hacia otro lugar de la pantalla.

**Reframe**

Mueve la ventana receptora hacia otro lugar de la pantalla cambiando su tamaño y el de todas sus componentes.

**ScrollDown**

Realiza el scroll hacia abajo de la ventana receptora.

**ScrollUp**

Realiza el scroll hacia arriba de la ventana receptora.

**ScrollLeft**

Realiza el Scroll hacia la izquierda de la ventana receptora.

**ScrollRight**

Realiza el scroll haci la derecha de la ventana receptora.

**ResetScroll**

Reinicializa los scroll de la ventana receptora.

**ResetFrames(X1,Y1,X2,Y2)**

Cambia los frames de la ventana receptora según los nuevos limites de la ventana, dados por las coordenadas pasadas como parámetros.

**ShowBox(DownBox,X1,Y1,X2,Y2)**

Muestra un rectángulo con linea punteada, cuando el usuario está cambiando la ubicación de una ventana.

**HideBox(DownBox,X1,Y1,X2,Y2)**

Oculta el rectángulo punteado generado por el método ShowBox.

**PageUp**

Produce la visualización de la página siguiente a la corriente.

**PageDown**

Produce la visualización de la página anterior a la corriente.

**WriteChar(aChar)**

Imprime un carácter sobre el área de trabajo de la ventana.

**WriteLine(aLine)**

Imprime una línea sobre el área de trabajo de la ventana.

**SetChangedArea(X1,Y1,X2,Y2)**

Setea un nuevo rectángulo como la changed-área del presentador del área de trabajo de la ventana receptora.

**GetTextCursor**

Retorna el cursor de texto correspondiente a la ventana receptora.

**MoveTextCursor**

Mueve el cursor de texto a la posición indicada por el cursor.

**Config**

Permite el seteo de tipo y tamaño de letra, tipo y grosor de línea, por parte del usuario, para la ventana receptora.

## MenuCommand

Esta clase se encarga de gerenciar el procesamiento cuando el cursor se encuentra dentro de un menú, administrando la activación y desactivación de los reconocedores y ordenando las presentaciones de los ítems correspondientes.

*Variables de instancia:*

**Items** : Contiene la lista de los items de menú del receptor.

**ActiveSubmenu** : Contiene una referencia al submenú del receptor que está activo corrientemente.

**DefaultItem** : Contiene una instancia de la clase DefaultSubmenuRecognizer, un reconocedor para la zona del menú que no tiene ítems asignados.

**DefaultPresenter** : Contiene una instancia de la clase DefaultItemPresenter,

un preentador para la zona del menu que no tiene ítems asignados.

*Métodos de instancia:*

**Init**

Inicializa el objeto receptor.

**Done**

Libera la memoria asignada al receptor cuando éste es destruido.

**FrameOf(aString)**

Retorna el frame del ítem del menú receptor que tiene label aString.

**Display**

Muestra sobre la pantalla el menú completo.

**Run**

Activa el menu receptor. Actúa como gerenciador de todos los ítems existentes en él. Es el loop principal de procesamiento de eventos cuando el cursor se encuentra dentro del menú.

**Add(aRec,aPres)**

Agrega un ítem de menú al menú receptor.

**SearchActiveItem(aRec,aPres)**

Retorna en aRec y aPres el reconocedor y el presentador del ítem de menu corrientemente activo

•

**SubMenuCommand**

Esta clase se encarga de gerenciar el procesamiento cuando el cursor se encuentra dentro de un submenú, administrando la activación y desactivación de los reconocedores y ordenando las presentaciones de los ítems correspondientes.

*Variables de instancia:*

**Items:** Contiene la lista de los ítems del submenú receptor.

**Down:** Contiene un puntero a una zona de memoria en la que se guarda la imagen de la pantalla que se encuentra por debajo del submenu receptor. Es de uso interno de la UIMS.

*Métodos de instancia:*

**Init**

Inicializa el submenu receptor.

**Done**

Libera la memoria asignada al receptor cuando éste es destruido.

**Display**

Muestra sobre la pantalla el submenú completo.

**Hide**

Oculta el submenu receptor.

**Run**

Activa el submenu receptor. Actúa como gerenciador de todos los ítems existentes en él. Es el loop principal de procesamiento de eventos cuando el cursor se encuentra dentro del submenú.

**Add(aRec,aPres)**

Agrega un ítem de submenú al submenú receptor.

**SearchActiveItem(aRec,aPres)**

Retorna en aRec y aPres el reconocedor y el presentador del ítem de submenu corrientemente activo.

### InputCommand

Esta clase realiza el gerenciamiento de los posibles items que puede seleccionar el usuario para setear un valor para una entrada determinada.

*Variables de instancia:*

**Items:** Contiene una instancia de la clase InputCommandList, una lista de presentadores y valores posibles asociados a una Input.

**Input:** Contiene una instancia de la clase Input, a la que está asociada la inputCommand, para poder darle el valor de la entrada elegido por el usuario.

**ActivePresenter:** Contiene una referencia al presentador de la lista apuntada por items que está corrientemente activo.

*Métodos de instancia:*

**Init(aInput)**

Inicializa el receptor, asignando aInput a la variable de instancia Input.

**Done**

Libera la memoria asignada al receptor cuando éste es destruido.

**DisplayDefault(aValue)**

Ordena al presentador de aValue que se muestre en video inverso. aValue es el valor default de la input asociada.

**GetInput**

Retorna la Input del receptor.

**ClearActivePresenter**

Pone en Nil el ActivePresenter. Run(aValue) Hace un display del presentador activo y setea el valor de la Input asociada con aValue.

**SearchPresenter(auxValue)**

Retorna el presentador del valor auxValue.

**AddPresenter(aPresenter,aValue)**

Agrega una ocurrencia a la lista de items, con un presentador aPresenter y un valor aValue.

## DefaultInputCommand

*Variables de instancia:*

**OKPresenter**: Contiene una instancia de InputPresenter, un presentador para el OK de un dialogue-Box.

CancelPresenter: Contiene una instancia de InputPresenter, un presentador para el CANCEL de un Dialogue-Box.

**Items**: ( de la clase InputCommand).

**Input**: ( de la clase InputCommand).

**ActivePresenter**: (de la clase InputCommand).

*Métodos de instancia:*

**Init(aOKPres,aCancelPres)**

Inicializa el receptor,asignando aOKPres a la variable de instancia OKPresenter y aCancelPres a CancelPresenter.

**Done**

Libera la zona de memoria asignada al receptor,cuando éste es destruido.

**Run(aValue)**

Si aValue es 'OK' setea la instancia de OK del D.Box del comando activo. En caso contrario setea el CANCEL del mismo objeto.

**GetOKPresenter**

Retorna el OKPresenter del receptor.

**GetCancelPresenter**

Retorna el CancelPresenter del receptor.

## InputTextCommand

Esta clase administra la entrada de texto por parte del usuario cuando se está especificando una entrada de este tipo, y ademas setea el valor correspondiente.

*Variables de Instancia:*

**Text:** Contiene el valor de la entrada por texto para el comando asociado.

**Input:** Contiene una referencia a la Input cuyo valor el objeto receptor obtendrá del usuario.

**Presenter:** Contiene el presentador de la entrada que se desea obtener.

*Métodos de instancia:*

**Init(aInput,aPresenter)**

Inicializa el objeto receptor, asignando aInput a Input y aPresenter al Presenter.

**Done**

Libera la zona de memoria del receptor, cuando éste es destruido.

**DisplayDefault**

Muestar el valor default dado para la Input que desea obtener el receptor.

**Run(aCharacter)**

Procesa el carácter recibido por el receptor,y en caso de ser correcto, cambia el valor de la input asociada poniendo como nuevo valor el nuevo contenido de text.

**GetPresenter**

Retorna el presentador del objeto receptor.

**ClearText**

Pone el valor default de la entrada en text.

## ErrorCommand

Realiza la administración del procesamiento cuando se encuentra abierta una ventana de error.

*Variables de instancia:*

**OKPresenter:** Contiene un presentador para el OK de la caja que muestra el mensaje de error de un comando.

**Items:** ( de la clase InputCommand).

**Input:** ( de la clase InputCommand).

**ActivePresenter:** (de la clase InputCommand).

*Métodos de instancia:*

**Init(aOKPres)**

Inicializa el objeto receptor, asignando aOKPres a la variable de instancia OKPresenter.

**Done**

Libera la zona de memoria del receptor cuando éste es destruido.

**Run**

Setea el OK del error box activo.

**GetOKPresenter**

Retorna el OKPresenter del objeto reeptor.

## Help

Realiza la administración del procesamiento cuando se encuentra abierta una ventana de help.

*Variables de instancia:*

**Frame** : Contiene el frame sobre el que se van a mostrar los textos del help.

**Down** : Contiene un puntero a una zona de memoria en la que se guarda la imagen de la pantalla que esta por debajo del frame del help. Es de uso interno de la UIMS

•

**DefaultRec** : Contiene un reconocedor para la pantalla del help.

**DefaultPres** : Contiene un presentador para la pantalla del help.

**Title** : Contiene el título del mensaje de help corriente.

**Text** : Contiene un arreglo de diez líneas con el texto del help corriente.

**HelpItems** : Contiene una lista de los ítems del help que son accesibles desde el ítem corriente.

**ActiveHelp** : contiene un número entero que referencia a un ítem del help, que es el que está corrientemente activo.

*Métodos de instancia:*

**Init**

Inicializa el objeo receptor.

**Done**

Libera la zona de memoria asignada al receptor cuando es destruido.

**Run(HelpNumber)**

Invoca al método que muestra el ítem del help referenciado por el número HelpNumber.

**OpenBox**

Abre la pantalla para mostrar los mensajes de help.

**Display**

Muestra el mensaje de help corrientemente activo.

**AddHelp(X1,Y1,HelpNumber,Title)**

Agrega un mensaje de help , asociándole el número HelpNumber y el título title.

**Hide**

Oculta el mensaje de help corrientemente activo.

**RunHelp**

Activa la pantalla de help. Actúa como gerenciador de todos los reconocedores existentes en él. Es el loop principal de procesamiento de eventos cuando el cursor se encuentra dentro de la pantalla de help.

**SearchActiveHelp(aRec)**

Retorna el help asociado al reconocedor aRec.

**SetActiveHelp(aValue)**

Setea como help activo al asociado con el número aValue.

**ScreenHelp**

Activa el help del screen.

**MenuHelp**

Activa el help del menu.

**LabelHelp**

Activa el help del label de una ventana.

**DialogueBoxHelp**

Activa el help del dialogueBox.

**WorkAreaHelp**

Activa el help del area de trabajo de una ventana.

**MoveHelp**

Activa el help del move de una ventana.

**ReframeHelp**

Activa el help del reframe de una ventana.

**ScrollHelp**

Activa el help del scroll de una ventana.

**PageHelp**

Activa el help del paginado de una ventana.

**ConfigHelp**

Activa el help de la configuración de una ventana.

**IconHelp**

Activa el help de los iconos.

**ResetScrollHelp**

Activa el help del reseteo de los scroll de una ventana.

## DialogueBox

Esta clase proporciona uno de los principales medios de ingreso de datos en la

UIMS Genius. Estas entradas pueden ser especificadas por medio de selección de posibles valores o por medio de ingreso de datos por teclado.

*Variables de instancia:*

**Titulo**: Contiene un string con el titulo de la caja de diálogo.

**InputItems:** Contiene una lista de inputRecognizers, cada uno de ellos asociado con una de las posibles selecciones que se pueden hacer en una caja de diálogo.

**InputTextItems:** Contiene una lista de inputTextRecognizers, cada uno de ellos asociado con una de las entradas por texto que se pueden hacer en una caja d diálogo.

**Down:** Contiene un puntero a una zona de memoria en la que se guarda la imagen de la parte de la pantalla ubicada por debajo del dialogueBox. Es de uso interno de la UIMS.

**OK**: Contiene True si se dio el OK en la caja de diálogo para que se invoque a una operación.Es False en caso contrario.

**Cancel**: Contiene True si se eligió cancelar la invocación a una operación por la que se estaban ingresando datos de entrada. Es False en caso contrario.

**DefaultRec**: Contiene un reconocedor para el fondo de la caja de diálogo.

**DefaultPres**: Contiene un presentador para el fondo de la caja de diáilogo.

**OkRec**: Contiene una instancia de DefaultInputRecognizer, un reconocedor para la opción de OK.

**CancelRec**:Contiene una instancia de DefaultInputRecognizer, un reconocedor para la opción de Cancel.

*Métodos de instancia:*

**Init(aTitulo,aFrame)**

Inicializa el objeto receptor, asignando aTitulo al Titulo y aFrame al Frame.

**Done**

Libera la zona de memoria asignada al receptor cuando éste es destruido.

**AddOption(aCommand,aFrame,aPresentation,aValue)**

Agrega un nuevo posible valor de entrada para las Input de la caja de diálogo en la lista InputItems.

**AddInputText(aInput,aFrame,aValue)**

Agrega una nueva InputText en la lista InputTextItems, para que el usuario ingrese un valor de entrada por texto.

**Display**

Muestra la caja de diálogo.

**Hide**

Oculta la caja de diálogo.

**Run**

Activa la caja de diálogo. Actúa como gerenciador de todos los reconocedores existentes en él. Es el loop principal de procesamiento de eventos cuando el cursor se encuentra dentro de la caja de diálogo.

**GetOK**

Retorna un valor booleano con el valor de la variable de instancia OK.

**GetCancel**

Retorna un valor booleano con el valor de la variable de instancia Cancel.

**SetOK**

Pone en True el valor de OK.

**SetCancel**

Pone en True el valor de Cancel.

**GetOKRecognizer**

Retorna el valor de OKRec.

**GetCancelRecognizer**

Retorna el valor de CancelRec.

**SetOKInput(aRec)**

Asigna aRec a OKRec.

**SetCancelInput(aRec)**

Asigna aRec a CancelRec.

**SearchActiveInput(aRec)**

Retorna en aRec el reconocedor activo de toda la lista de InputItems, incluyendo el OKRec y el CancelRec.

**SearchActiveInputText(aRec)**

Retorna en aRec el reconocedor activo de toda la lista de inputTextItems.

### Error Box

Esta clase tiene la función de informar al usuario los errores cometidos durante el ingreso de datos.

*Variables de instacia:*

**Msg**: Contiene el mensaje a mostrar.

**Down**: Contiene un puntero a la posición de memoria en donde se encuentra la parte de la pantalla que está por debajo de la ventana de error.

**OK**: Contiene true si el usuario seleccionó OK. En caso contrario contiene false.

**Frame**: Contiene el frame correspondiente a la ventana en donde se mostrará el mensaje de error.

**DefaultRec**: Contiene el reconocedor que administrará los movimientos del cursor sobre el fondo de la ventana de error.

**DefaultPres**: Contiene el presentador del fondo de la ventana de error.

**OKRec**: Contiene el reconocedor correspondiente al ítem de OK dentro de la ventana de error.

*Métodos de instancia:*

**Init(aMsg)**

Inicializa al receptor con aMsg como mensaje de error.

**Done**

Libera la memoria asignada al receptor cuando éste es liberado.

**Display**

Muestra al receptor sobre la pantalla.

**Hide**

Oculta al receptor.

**Run**

Administra los movimientos del cursor y selecciones mientras se encuentra activa la ventana de error.

**GetOK**

Retorna el valor de la variable se instancia OK del receptor.

**SetOK**

Pone en true el valor de la variable de instancia OK del receptor.

**GetOKRecognizer**

Retorna el reconocedor del ítem de OK del receptor.

**SearchActiveRecognizer(aRecognizer)**

Retorna el reconocedor activo dentro de la ventana de error.

## Input

La clase Input es de fundamental importancia en la tarea de ingreso de datos por parte del usuario. Toda entrada que debe ser especificada por el usuario tiene asociada una instancia de la clase Input. Dentro de esta clase se realiza la tarea de validación de las entradas y de información de los posibles errores al usuario.

*Variables de instancia:*

**Value** : Contiene el valor correspondiente de la entrada.

**DefaultValue**: Contiene el valor que tomará por defecto la entrada cuando sea inicializada.

**MsgError**: Contiene el mensaje que se mostrará al usuario cuando éste ingrese un valor no permitido.

**InputType**: Contiene el tipo predefinido al cual pertenecen los valores correspondiente a la entrada. Genius permite pasar como parámetro las siguientes constantes: IntegerType, LongIntType, RealType, StringType y CharType.

**Predicate**: Contiene la función especificada por el programador de la interfase utilizada para validar los datos ingresados por el usuario.

**ChoiceSet**: Contiene el conjunto de valores válidos para la entrada.

*Métodos de instancia:*

**Init(aDefaultValue,aType,aPredicate,aChoiceSet,aMsgError)**

Inicializa al receptor asignando en sus variables de instancia los valores correspondientes a los parámetros.

**Done**

Libera la memoria asignada al receptor cuando este es destruído.

**InitValue**

Reinicializa el valor de la entrada con el valor default.

**GetValue**

Retorna el valor corriente del receptor.

**SetValue(aValue)**

Setea el valor del receptor con aValue.

**PutValue(aText)**

Setea el valor de la entrada ingresado en forma de string.

**GetStrValue**

Retorna el valor de la entrada en forma de string.

**GetDefaultValue**

Retorna el valor default del receptor.

**GetType**

Retorna el tipo de los datos correspondientes al receptor.

**Validate**

Realiza la validación del dato asignado al receptor. Retorna true si la validación resulta correcta o false en caso contrario.

**Warning**

Crea una instancia de la clase ErrorBox y ordena su presentación, con el objeto de informar al usuario el ingreso de un dato inválido.

## Value

Esta clase es usada junto con la clase Input, y tiene como función contener valores posibles que puede tomar una entrada.

*Variables de Instancia:*

**Buffer**: contiene los bytes correspondientes al valor almacenado.

**Size**: contiene la cantidad de bytes que contiene Buffer.

*Métodos de instancia:*

**SetValue(aValue,aSize)**

Inicializa al receptor con los bytes correspondientes a aValue y Size correspondiente a aSize.

Done Libera la memoria asignada al receptor cuando éste es creado.

**Equal(aValue)**

Retorna True si el receptor es igual a aValue. En caso contrario retorna false.

**ChangeValue(aValue,aSize)**

Cambia el contenido del receptor por los nuevos valores recibidos.

**GetValue**

Retorna el valor almacenado por el receptor

.

## Menu

Esta clase provee una forma simple de crear menúes para la interfase y de agregarle items y submenúes.

*Variables de instancia:*

**aMenuCommand:** Contiene una instancia de la clase MenuCommand.

*Métodos de instancia:*

**Init**

Inicializa el objeto receptor.

Done

Libera la zona de memoria asignada al receptor cuando éste es destruido.

**ScheduleMenu**

Abre el menu receptor y lo agrega en el Scheduler.

**AddItem(aString1,aStrng2,aCommand)**

Agrega el ítem aString2 al Submenu cuyo ítem de menú es aString1.

**AddSubmenú(aString)**

Agrega el ítem de menú aString al menú receptor.

**SubMenuOf(aString)**

Retorna el submenú correspondiente al ítem de menú aString.

## Icon

Esta clase provee una forma simple de incorporar iconos a la interfase.

*Variables de instancia:*

No posee.

*Métodos de instancia:*

**Init(Recno,aCommand)**

Inicializa el icono receptor, leyendo del archivo correspondiente la figura del icono con número de registro Recno y asociándole el comando aCommand.

### Window

Esta clase provee una forma simple de abrir ventanas, comunicarse con ellas durante su utilización y cerrarlas cuando se considere necesario.

*Variables de instancia:*

**WindowRec:** Posee una instancia de la clase WindowRecognizer.

*Métodos de instancia:*

**Init(aLabel,aFrame,aCommand,aPages)**

Inicializa la ventana receptora, que tendrá como label aLabel, como frame aFrame, como comando aCommand y como máximo de páginas a usar aPages.

**OpenWindow**

Abre la ventana receptora y la agrega en el Scheduler.

**CloseWindow**

Cierra la ventana receptora.

**Done**

Libera la zona de memoria asignada al receptor cuando éste es destruido.

**GetPresenter**

Retorna el presentador de la ventana receptora.

### Scheduler

Esta clase implementa en concepto de scheduler explicado en secciones anteriores, cuya función fundamental es gerenciar la activación y desactivación de todos los reconocedores existentes en la pantalla.

*Variables de instancia:*

**Recognizers:** Contiene una lista de todos los reconocedores de la interfase, entre los

que realiza su tarea de scheduling.

**DefaultRecognizer:** Contiene un reconocedor del fondo de la pantalla, aquél que se activa si no existe otro reconocedor sobre la zona de la pantalla apuntada por el cursor.

**Running:** Contiene True mientras el Scheduler esté activo, es decir, mientras la UIMS esté trabajando. Cuando se poone en False,la UIMS se desactiva.

*Métodos de instancia:*

**Init**

Inicializa el scheduler receptor.

**Done**

Libera la zona de memoria asignada al receptor cuando éste es destruido.

**Quit**

Pone Running en False.

**InitDefault(aScreenRec)**

Inicializa el reconocedor del fondo de la pantalla.

**Run**

Activa el Scheduler. Es el loop principal de la UIMS.

**Add(aRecognizer)**

Agrega un recognizer a la lista Recognizers.

**SearchActiveRecognizer**

Retorna el recognizer activo de la lista Recognizers.

**Remove(aRecognizer)**

Remueve un reconocedor de la lista Recognizers.

**RedrawScreen**

Redibuja la pantalla cuando se cierra o se mueve una ventana.

BIBLIOTECA
FAC. DE INFORMÁTICA
U.N.L.P.

# Bibliografía.

[1] Pedro Szekely. Separating the User Interface from the Functionality of Application Programs. CMU-CS-88-101. 1988.

[2] Gerhard Fischer. Human-Computer Interaction Software: Lessons Learned, Challenged Ahead. IEEE Software, January 1989.

[3] Rex Hartson. User Interface Management Control and Communication. IEEE Software, January 1989.

[4] William Hurley and John Sibert. Modeling User Interface-Application Interactions. IEEE Software, January 1989.

[5] Brad Myers. User Interface Tools: Introduction and Survey. IEEE Software, January 1989.

[6] Jonas Lowgren. History, State and Future of UIMS's.

[7] Brad Myers. Tools For Creating User Interfaces: An Introduction and Survey. CMU-CS-88-107. January 1988.

[8] Wilf LaLonde & John Pugh. Pluggable Tiling Windows. JOOP September 1989.

[9] Smalltalk-V. Tutorial and Programming Handbook. Digitalk Inc.

[10] M.Green. A Survey of Three Dialogue Models. ACM Transactions on Graphics.

[11] Daniel Mazzuca. Sistema Manejador de Interfases con Usuarios. ESLAI 1988.