30
2ej.

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES ACATLAN

"ANALISIS Y REFERENCIA SOBRE LA PROGRAMACION ORIENTADA A OBJETOS EN EL LENGUAJE SMALLTALK/V Y ALGUNAS APLICACIONES"

T E S I S

QUE PARA OBTENER EL TITULO DE:

LICENCIADO EN MATEMATICAS APLICADAS Y COMPUTACION

PRESENTA:

BRENDA ELISA SOLIS PEREZ

ASESOR DE TESIS:

ING. EMILIANO LLANO DIAZ

NAUCALPAN, EDO. DE MEXICO

Marzo de 1994

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

VNIVERSIDAD NACIONAL AVFNOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES "ACATLAN"
DIVISION DE MATEMATICAS E INGENIERIA
PROGRAMA DE ACTUARIA Y M.A.C.

SRITA. BRENDA ELISA SOLIS PEREZ
Alumna de la carrera de Matemáticas
Aplicadas y Computación.
Presente.

De acuerdo a su solicitud presentada con fecha 26 de enero de 1993, me complace notificarle que esta Jefatura tuvo a bien asignarle el siguiente tema de tesis: "ANALISIS Y REFERENCIA SOBRE LA PROGRAMACION ORIENTADA A OBJETOS EN EL LENGUAJE SMALLTALK/V Y ALGUNAS APLICACIONES", el cual se desarrollará como sigue:

I.- Tecnología orientada a objetos.
II.- Utilización de objetos en Smalltalk
III.- Polimorfismo dentro de Smalltalk
IV.- Algunas aplicaciones de Smalltalk/v
V.- El papel de Smalltalk hoy en día
Conclusiones
Bibliografía

Asimismo fue designado como Asesor de Tesis el Ing. Emiliano LLano Díaz, profesor de esta Escuela.

Ruego a usted tomar nota que en cumplimiento de lo especificado en la Ley de Profesiones, deberá prestar servicio social durante un tiempo mínimo de seis meses como requisito básico para sustentar examen profesional, así como de la disposición de la Coordinación de la Administración Escolar en el sentido de que se imprima en lugar visible de los ejemplares de la tesis el título del trabajo realizado. Esta comunicación deberá imprimirse en el interior de la tesis.

Atentamente
"POR MI RAZA HABLARA EL ESPIRITU"
Acatlán, Edo. de Méx., a 22 de febrero de 1994.

JEFATURA DEL PROGRAMA DE ACTUARIA Y M.A.C.

ACT. [illegible] RIVERA BECERRA
Jefe del Programa de Actuaría y M.A.C.

AGRADECIMIENTOS

A Dios,
a quien doy gracias por Todo.

A mis Padres,
pues la obtención de ésta meta, se debe principalmente a los principios que me inculcaron desde siempre.

A mi Familia y Amigos,
a quienes agradezco siempre de alguna forma estar conmigo en todo.

Al Ing. Emiliano Llano Díaz,
por quien siento una gran admiración como Profesor, y por haber asesorado éste trabajo.

Al Mtro. en C. Act. Carlos Cadena Sandoval,
muy en especial porque sin su ayuda no hubiese sido posible la realización de éste trabajo.

Al matrimonio de la Act. Luz Ma. Lavín y el Ing. Oscar Meza,
a quienes agradezco haberme apoyado y orientado verdaderamente, para poder llegar a éste momento.

Al Fis. Manuel Valadez Rodríguez,
por contribuir con su apoyo a la elaboración de éste estudio.

Y finalmente deseo agradecer a todos aquellas personas que de alguna forma ayudaron para la realización de éste trabajo,

GRACIAS.

INDICE

INTRODUCCION

Actualmente es fácil encontrar una automatización de los procesos de cualquier empresa o institución en donde se maneje información. Cada vez más, éstas organizaciones dependen del flujo de datos en el computador, para casi cualquier operación o movimiento que se realice en ellas.

Muchas veces, el software que se maneja, resulta obsoleto en los procesos que cada día se actualizan de acuerdo al crecimiento de dicha organización. El problema real, surge al tratar de dar mantenimiento a los sistemas con los que se cuenta, pues la dependencia entre módulos, implica cambios en toda la estructura del programa.

Todos estos problemas, nos llevan a pensar en nuevas técnicas de programación para poder reducir al máximo nuestro tiempo invertido en la actualización del software.

Así pues, la programación orientada a objetos se desarrolla como una respuesta coherente a la solución de éstos problemas en el software, que significan un gasto inútil de recursos económicos y de tiempo.

En éste trabajo de investigación, se pretende mostrar a uno de los lenguajes pioneros en la orientación a objetos, como una propuesta para el desarrollo de software: Smalltalk/V. El análisis y la referencia a él, irán a la par a lo largo de éste trabajo, dentro de los aspectos más importantes del lenguaje.

En el primer capítulo "TECNOLOGIA ORIENTADA A OBJETOS", se pretende mostrar tanto el surgimiento de la programación orientada a objetos y en especial de Smalltalk en sus primeras versiones, así como la teoría general de ésta programación. Posteriormente dentro del mismo capítulo, se da una introducción a los conceptos básicos de la orientación a objetos en Smalltalk/V.

En el segundo capítulo "UTILIZACION DE OBJETOS EN SMALLTALK" se verá la forma en que los objetos actúan, a través de mensajes y su creación en las clases en Smalltalk, y particularmente en Smalltalk/V.

Posteriormente, en el capítulo III "POLIMORFISMO DENTRO DE SMALLTALK", se definen los tres principales componentes que conforman a un lenguaje orientado a objetos: la herencia, la encapsulación y el polimorfismo. Estos conceptos serán analizados desde sus significados, y hasta su aplicación práctica en el lenguaje.

En el capítulo IV "ALGUNAS APLICACIONES DE SMALLTALK/V", mostraremos los conceptos básicos de gráficos en Smalltalk/V para posteriormente utilizarlos en los problemas a los que se aplica el lenguaje. Cabe mencionar que en éste capítulo se aplicarán los conceptos estudiados en los capítulos anteriores, a ejemplos prácticos.

Finalmente y para saber que impacto tiene la programación orientada a objetos en el mercado,

se toca el tema del papel que juega Smalltalk actualmente en el mundo de la computación, en el capítulo V "EL PAPEL DE SMALLTALK HOY EN DIA", en relación con los lenguajes más populares orientados a objetos.

CAPITULO I

"TECNOLOGIA ORIENTADA A OBJETOS"

1.1 Surgimiento de Smalltalk

1.2 Ambiente de Objetos

1.3 Introducción a la programación orientada a objetos en Smalltalk/V en contraste con los lenguajes convencionales

CAPITULO I. TECNOLOGIA ORIENTADA A OBJETOS.

1.1. Surgimiento de SmallTalk

Dentro del desarrollo de Software que se ha tenido en el campo computacional, podemos referirnos a los lenguajes de programación como una herramienta fundamental para la solución de problemas de computación a través de ciertos procesos especificados por algún algoritmo, pero no podemos decir que de una tarea específica sea posible su realización factible en cualquier lenguaje de programación. Un lenguaje de programación que resuelva todas las situaciones, debe contemplar todas las aplicaciones existentes y absolutamente todo el equipo computacional disponible hasta la actualidad.

El investigador Knuth en el año de 1974 pensó que para 1984 ya se habría desarrollado un lenguaje (que tendría por nombre UTOPIA 84) que adoptara metodologías sistemáticas de diseño de programación con una fuerte confiabilidad, legibilidad y modificación de programas.

Dado que todavía no existe tal "Lenguaje Perfecto", es importante saber que lenguaje se utilizará para cierta aplicación.

La clasificación más común de los lenguajes de programación se basa en las distintas generaciones por las que han desfilado los diferentes lenguajes con sus propios propósitos y basados en las tecnologías del momento. Por ejemplo, en la Primera Generación (principios de 1960) las estructuras de flujo de control están basadas en la instrucción GOTO y las estructuras de datos son consideradas primitivas, basadas en variables de punto flotante, caracteres y valores lógicos con representaciones relativamente sencillas. Los lenguajes más populares de esta generación son FORTRAN (FORmula TRANslation) para aplicaciones numéricas y los lenguajes intérpretes, entre los que figura BASIC (Beginner's All-purpose Simbolic Instruction Code) para la programación por pasos.

En la Segunda Generación los tipos de estructuras son considerados como generalidades de la Primera Generación; la segunda generación se caracteriza por los arreglos dinámicos y los tipos de datos definidos. Su mayor contribución está en su eficiente manejo de memoria dinámica y la introducción a las estructuras de bloque dentro de los programas. Podemos encontrar aquí a lenguajes como ALGOL-60 con sus estructuras de control, que eliminan en gran parte a la instrucción GOTO.

En lo que a la Tercera Generación se refiere, podemos decir que PASCAL es el modelo principal de este tipo de lenguajes estructurados con simplicidad y eficiencia en el manejo de datos, nombres y estructuras de control. Aquí se originan ciertas disciplinas de programación haciendo modular a cada segmento de código y definiendo los tipos de datos de usuario, de acuerdo a sus necesidades para una programación estructurada. En la Tercera Generación, también se tienen nuevas estructuras de control como la estructura CASE.

La Cuarta Generación saca partido de la Tercera Generación y además utiliza la programación concurrente y la comunicación entre tareas (task). Ejemplo de ésta generación es ADA, en donde el lenguaje provee varias herramientas para poder desarrollar aplicaciones más sofisticadas.

Muchos autores consideran que a partir de 1990 y hasta la fecha se esta viviendo la Quinta

Generación en el diseño de los lenguajes de programación. Conforme la tecnología de la arquitectura de las computadoras avanza, también avanzará el diseño e implementación del Software, a la par. Generalmente los lenguajes de Quinta Generación tienen una "Orientación" a una técnica definida, es decir, se organizan alrededor de ciertos procesos. Por ejemplo, podemos hablar de tres tipos de técnicas de programación: Programación "Orientada" a Funciones, Programación "Orientada" a la Lógica y la Programación "Orientada" a Objetos.

La Programación Orientada a Funciones o Funcional provee un alto nivel en las estructuras de control, reduciendo los operadores a utilizar. Se caracteriza por el método de recursión con respecto a las iteraciones, y por su manejo de almacenamiento dinámico.

Lenguajes de Orientación a Funciones, como LISP por ejemplo, proveen ciertas abstracciones por medio de procesos con información simbólica reduciendo las posibilidades de error.

Los lenguajes Orientados a la Lógica como PROLOG se basan en predicados y dominios que hacen a los programas Autodocumentables. La lógica y el control, son técnicas fundamentales para desarrollar éste tipo de software enfocado a la Inteligencia Artificial, en donde la precisión en programación se hace mucho más clara en el código.

Actualmente la Orientación a Objetos es considerada por varios autores como la mejor técnica de programación, debido a su flexibilidad, autodocumentación y reutilización de código en programación, trabajando con Objetos (paralelamente al mundo real en donde todo es un Objeto). Así pues, en la Programación Orientada a Objetos el computador se adapta al problema, y ya no se trata de adaptar entonces el problema al computador.

El primer lenguaje Orientado a Objetos fue SIMULA 1 desarrollado en el año de 1962. Su historia comienza en 1940 cuando Kristen Nygaard y Ole-Johan Dahl trabajaban para un Instituto de Investigación. Nygaard se dedicó a la investigación operacional, mientras que Dhal se dedicaba al diseño, programación e implementación de cierto lenguaje. Los problemas a los que se enfrentaron fueron los de inconsistencia, poca flexibilidad y poco poder en el lenguaje, por lo que en 1960 Nygaard se trasladó al Norwegian Computing Center (NCC), para desarrollar una investigación en torno a los procesos de información y es cuando se crea SIMULA 1 que tomó muchos de sus rasgos de ALGOL 60, que era el primer lenguaje estructural que contaba con procedimientos para poder simular la concurrencia en las computadoras seriales, que toman y ejecutan una instrucción a la vez.

En 1967 Dhal y Nygaard decidieron trabajar en el perfeccionamiento de SIMULA 1, resultando en SIMULA 67 con adiciones como: contar con nociones de encapsulación[1] y objetos, técnicas de acceso a los atributos de los objetos, reconocer que las clases de objetos tienen varias propiedades en común (propiedad actualmente conocida como herencia) y técnicas de acción que habilitaron la concurrencia en el área de objetos activos.

SIMULA era un lenguaje que trataba de imitar lo que sucedía en el mundo real, trabajando con entidades reales como los objetos, y llegando a ser un lenguaje de propósito general ofreciendo capacidades de simulación como una aplicación de sus conceptos básicos. Las aplicaciones que se hicieron con SIMULA se enfocaron a la simulación de procesos físicos e industriales.

---

[1] Propiedad que permite aislar los datos de un objeto.

SIMULA es el germen de todos los lenguajes actuales más comerciales orientados a la programación con objetos, que tienen aproximadamente 7 años en el mercado de software.

Pero la popularidad de SIMULA llego a su fin a principios de 1970, con el desarrollo de nuevos compiladores para las mainframes de IBM y las UNIVAC. Posteriormente siguieron surgiendo más lenguajes portables para estaciones de trabajo Sun, Apollo, Hewlett Packard, Macintosh y Atari entre otras.

Debido a que en 1970, la tecnología se basaba en una integración a gran escala y en tiempos compartidos en una sala de terminales, los usuarios de las computadoras de aquella época eran solamente personas técnicas, es decir, los lenguajes existentes eran diseñados por y para especialistas en el área de aplicaciones, tales como las científicas y comerciales.

Alan Kay, co-creador del lenguaje puro orientado a objetos Smalltalk, era un estudiante graduado de la Universidad de Utah que se dio cuenta del poder de desarrollo que una computadora personal podría proveer a cualquier usuario (y ya no solo a los especialistas), utilizando un lenguaje orientado a la simulación y a graficación. Su idea fue propuesta a la Corporación XEROX PARC (Palo Alto Research Center) de California, quien le diera el nombre de DYNABOOK al proyecto. Se pretendía crear una computadora que contara con una pantalla plana parecida a la de una calculadora, sensible al tacto con los dedos de las manos.

De hecho, Kristen Nygaard tuvo gran admiración hacia Kay sobre Dynabook, expresando su asombro sobre la poderosa idea de usar una interfase gráfica por medio de ventanas o "windows".

El Centro de Investigación Xerox Palo Alto apoyó la investigación, en el año de 1971, para la creación de la computadora Dynabook. En ese mismo año Dan Ingalls escribió el evaluador "SmallTalk" y un año más tarde, Alan Kay que tenía experiencia en el diseño con el lenguaje FLEX (el cual toma las bases de "objetos" de SIMULA), lo modifica para crear "Smalltalk-72" como un lenguaje exclusivo para Dynabook, que combinaba aspectos de SIMULA como las clases y los objetos. Smalltalk, entonces sería considerado como un lenguaje revolucionario de su época.

En 1973 se crea "Interim Dynabook" para escritorio y es entonces implementado Smalltalk-72 para investigaciones y experimentos acerca de las computadoras personales, ya no en exclusividad con los expertos en computadoras.

El lenguaje Smalltalk ha sufrido varias revisiones en su funcionalidad. Dentro de estas versiones se incluyen las siguientes: En 1974 Smalltalk-74, en 1976 Smalltalk-76, en 1978 Smalltalk-78. En el año 1980 la Corporación Xerox define Smalltalk-80 concediendo licencias de operación a Apple, Tektronix, Hewlett Packard y DEC, y un año más tarde Apple aporta Smalltalk-80 para el prototipo LISA. En el año 1981, el grupo XEROX PARC cambia su nombre al de SCG (Software Concepts Group).

Dentro de las aportaciones consideradas las más importantes para la Tecnología Orientada a Objetos[2] en Smalltalk a partir del año 1983, se mencionan la fundación de DIGITALK, la presentación del libro Azul Smalltalk-80 por Adele Goldberg (presidente de ParcPlace Systems dedicados a desarrollar

[2] Según Revista Patrocinada por Borland, "Happy 25th Anniversary Objects!", Publicaciones SIGS, 1992, Pp.7.

software orientado a objetos), en 1984 Digitalk maneja Smalltalk/V para DOS, en 1987 ParcPlace maneja Smalltalk para Macintosh, mientras que Digitalk maneja Smalltalk/V para Macintosh y Smalltalk/V286 para procesadores 80286.

Para poder introducirnos a la utilización de Objetos en Smalltalk dentro de este trabajo de investigación, el software en que se basa la Tesis es el intérprete SMALLTALK/V para DOS R.1.2 de Digitalk Inc. 1986, el cual requiere una Computadora Personal IBM, XT, AT o compatible con 512 Kilobytes de memoria RAM como mínimo, dos lectores de disco flexible (Drives) o una unidad de disco duro y una de disco flexible, monitor a color o monocromático, un controlador de gráficos como CGA, EGA, Hércules, VGA o un AT&T, y un Sistema Operativo PC/DOS o MS/DOS versiones posteriores a la 2.0. Opcionalmente se puede tener una expansión de 640 Kilobytes de memoria RAM, un Mouse (compatible con Microsoft) y un coprocesador de punto flotante 8087 sobre XT o 80287 sobre AT.

1.2. Ambiente de Objetos

En los ambientes de objetos existen tres características fundamentales que permiten crear programas orientados a objetos: la encapsulación, la herencia y el polimorfismo.

La "herencia" es la propiedad que permite a cierto objeto recibir las caracteristicas de otro objeto. El mayor beneficio que se obtiene de la herencia es la reutilización de código, es decir, un objeto hereda comportamiento de otro objeto sin necesidad de escribir más código y además puede añadir a él nuevas características. El comportamiento de un objeto y sus atributos pueden ser elementos privados ó "encapsulados", a los que tendremos acceso solo mediante la invocación al objeto. Por otro lado, los diferentes objetos pueden responder a la misma acción de diferente manera, es decir, pueden ser "polimorficos". Por ejemplo la acción "escribe" puede tener una respuesta diferente para una persona, para una impresora o para una máquina de escribir.

Para poder hablar de un "Ambiente de Objetos" en general, debemos comenzar por saber las diferencias principales de los lenguajes de programación orientada a objetos dentro los ambientes puros e híbridos.

Los lenguajes de programación orientados a objetos "puros" están desarrollados con tecnología orientada a objetos desde su diseño, es decir, poseen características propias del lenguaje como la herencia, encapsulación y polimorfismo, mencionadas anteriormente, que a diferencia de los lenguajes híbridos estos rasgos son adicionales a la estructura original del lenguaje. Esto lo vemos claramente en el lenguaje C por ejemplo, el cual es un lenguaje tradicional y donde se añade una orientación a objetos en el producto C++ haciéndolo un lenguaje de programación orientado a objetos "híbrido".

Existen algunos productos de software que proveen la simulación de un ambiente de lenguaje

puro[3], basados en ligas dinámicas (o dynamic binding, que eliminan cierta estabilidad requerida por las librerías para poder adaptarse fácilmente a programas sofisticados) y ventanas de trabajo.

En lenguajes híbridos como C++ o Pascal 5.5, existe un precompilador que interpreta el código orientado a objetos y se genera entonces un archivo que puede ser compilado de manera estándar; se presentan además librerías de clases definidas por el lenguaje, para ser usadas en los programas que se codifiquen, y el ambiente de desarrollo en el que se trabaja es análogo al lenguaje original.

Por otra parte, la mayoría de los lenguajes "puros" (como Smalltalk, SIMULA, Eiffel[4] o Actor) utilizan interfases gráficas de usuario o GUI (Graphical User Interface), con controles mediante ventanas, colores y gráficos para mayor facilitad de manejo para los usuarios, minimizando su complejidad y permitiendo una "programación rápida". Otra de las ventajas de los ambientes puros es que algunos de ellos pueden ejecutar módulos independientes del ambiente. Actor por ejemplo, basado en Microsoft Windows 3.0 puede ejecutar programas enteros o módulos, que no sean propiamente del ambiente, como aplicaciones de Windows o rutinas de C.

Además la herencia, encapsulación y polimorfismo, optimizan sus funciones en un ambiente puro, en donde todo es considerado como un objeto (incluso el propio lenguaje de programación).

Pese a todas estas ventajas de los ambientes puros, los ambientes de objetos híbridos poseen una mayor velocidad en ejecución, debido a la no utilización del ligado dinámico[5] (dynamic binding), donde se consume más tiempo. Este problema se minimiza en Smalltalk, en donde es imposible generar aplicaciones que sean independientes del ambiente. Por otra parte, se tiene también una desventaja de los ambientes puros en el costo de la memoria, pues un ambiente completo puede consumir muchos kilobytes en memoria RAM de operación (e incluso varios megas).

La programación en un ambiente híbrido es semejante a programar en un lenguaje tradicional, por lo que resulta muy fácil el salto de un lenguaje tradicional a uno en ambiente de objetos de tipo híbrido, en donde la programación sigue siendo de alguna manera estructurada y toma algunos de los principios procedurales. Mientras que en los ambientes puros (donde se programa únicamente con técnicas orientadas a objetos) se requiere de un tiempo y esfuerzo mayor, pues es un ambiente nuevo de programación para el usuario que ha programado gran parte de su vida en lenguajes tradicionales basados en funciones y procedimientos, pero el esfuerzo bien vale la pena debido a la flexibilidad que se afirma, podemos encontrar.

Muy a menudo los programas orientados a objetos se describen como "Simulaciones", puesto que imitan el comportamiento del mundo real en donde todo lo que existe son objetos que operan de cierta manera en particular.

---

[3] Por ejemplo los ADD-ONS basados en Windows, OS/2, MS Windows NT, X-Windows o NewWave.

[4] Eiffel fué la siguiente versión de SIMULA en 1985, que pulía sus imperfecciones.

[5] Mecanismo basado en los objetos para ligar dinámicamente los métodos heredados en el tiempo de corrida de un programa.

Hoy en día, debemos considerar seriamente el paradigma de la programación orientada a objetos, la cual nos brinda diferentes aspectos conceptuales y organizacionales, para poder construir programas robustos, fáciles de leer, fáciles de modificar y libres de errores.

El principal elemento de construcción en la orientación a objetos es el propio objeto, el cual trata de comportarse como un ente real, es decir, se emplea para describir un fenómeno. Un objeto se define por su clase como la unión de datos (atributos) públicos o privados y "métodos" o servicios que utilizan estos datos y procesan la información a través de una secuencia de actividades a realizar cuando un mensaje (descrito más adelante) es recibido por un objeto. La "clase" es dada a definirse como una plantilla del objeto, que no lo crea por ella misma, sino que "puede" ser usada como un tipo de dato para crear un objeto. El objeto será entonces la implementación de una clase en particular y se le conoce también como "instancia de clase". Estas variables de instancia (objetos que existen como instancia o caso particular de la clase, pero que no están contenidas dentro de la clase) contienen copias privadas de las variables que existen en la clase. En el diagrama 1.1 vemos un claro ejemplo de lo que sería una clase con sus atributos y métodos, y la creación de un objeto, instancia de la clase.

Una metaclase, es una clase cuya instancia particular es por sí misma una clase, mientras que una subclase se define como la construcción de una clase tomando ciertas propiedades de otra clase y en algunos casos modificando algunas de éstas características.

Algunas veces, dentro de la terminología de la programación orientada a objetos, los métodos o servicios se denominan también como "funciones miembro", pues en cierto modo son funciones que pertenecen al objeto. Los métodos definidos dentro de una clase, también pueden ser públicos o privados.

Muchos de los sistemas que operan en las diferentes empresas o instituciones, al término de cierto tiempo, resultan obsoletos debido a los cambios que van sufriendo dichos establecimientos. Por ejemplo, en ciertas rutinas se reinventan las instrucciones en las diferentes aplicaciones, en lugar de reutilizar código. Generalmente en los sistemas que se tienen, la dependencia entre módulos y datos es muy grande, y resulta bastante difícil su mantenimiento y actualización, pues un pequeño cambio en el sistema afectaría a toda la estructura de información. Este problema lo podemos resolver aplicando la orientación a objetos, que reutiliza código a través de su conce to llamado "herencia".

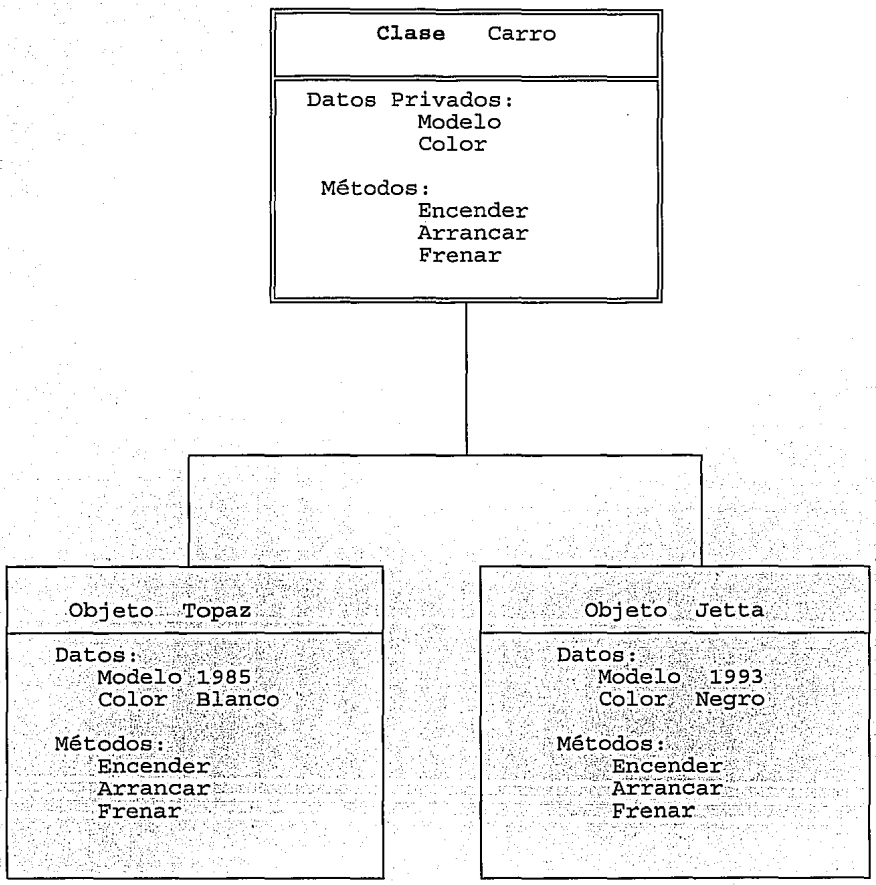


Diagrama 1.1 Instancias de Clase

Un objeto llamado "Fauna" por ejemplo, puede heredar comportamiento a un objeto llamado "Animal_Salvaje" que pasará a ser su hijo, y a su vez "Tigre" puede heredar características del objeto anterior. Aquí se aplica el concepto de herencia, dándose una "jerarquía" entre objetos al heredar características en forma descendente. Como podemos observar, la reutilización de código se va dando

a través de recibir comportamiento (atributos y servicios) entre objetos.

En éste sencillo ejemplo, podemos pensar en una modificación de código bastante fácil. Si se requiere crear un nuevo objeto llamado "León", bastará con heredar comportamiento del objeto "Animal_salvaje" y añadir las nuevas características deseadas; o si quisiéramos reemplazar el objeto "Tigre" por el nuevo objeto "Búfalo", podríamos quitar ese módulo "Tigre" y poner el otro. Así pues, podremos ir creando nuestra propia biblioteca de clases de la cual se podrá reutilizar código.

También existe la "herencia múltiple", que es la habilidad que un objeto tiene de heredar comportamiento de varios padres.

Por otro lado, la programación orientada a objetos permite descomponer el programa general en ciertos módulos independientes, que podemos llamar objetos. Esta característica, como decíamos, nos permite una fácil modificación de ciertas partes del sistema, sin necesidad de alterar parte o todo el sistema general. Muchas veces en un programa hecho en un lenguaje procedural, necesitamos tener una variable global para ser utilizada solamente por ciertos procedimientos, pero en el tiempo de ejecución podemos caer en la alteración incorrecta de ese dato, debido a que no se tiene una restricción con respecto a el uso de dicho valor por los procedimientos. En la programación con objetos, la encapsulación nos permite enlazar los datos haciéndolos privados, es decir, solamente los métodos definidos dentro del objeto tendrán acceso a los elementos que fueron declarados como privados (ver diagrama 1.1), y así en caso de tener algún error de programación, nos resultará más fácil una depuración, pues bastará con verificar los objetos a los que tiene acceso tal o cual dato. Dentro de la clase tenemos métodos y atributos, que pueden accederse unicamente mediante la invocación al objeto, por medio de mensajes.

El mensaje es un servicio (que puede ser público o privado) que contiene información y que nos permite la manipulación y comunicación entre objetos, y es el que influye en la selección del método adecuado dentro del objeto al que se le envió dicho mensaje. El mensaje se divide en: Mensaje Selector (Message Selector) que indica la acción a realizar y Mensaje Argumento (Message Argument) que son los argumentos que contienen datos en particular. El Receptor de Mensajes (Receiver Object), donde el objeto recibe el mensaje que se manda, pertenece a la clase que contiene la descripción del método que procesará como respuesta a ese mensaje, y no es necesario indicarle al objeto como seleccionar algún método; mientras que el Dirigente de Mensajes (Message Sender), que contiene el mensaje que se enviará, pertenece a un objeto que requiere cierta manipulación. También podemos tener mensajes sin argumentos llamados Mensajes Unarios (Unary Message).

El Sistema Operativo que utilizamos, por ejemplo, es un objeto grande que a su vez contiene otros objetos (propiedad permitida en la programación orientada a objetos)[6], en donde mando un mensaje y obtengo alguna respuesta sin importarme como se hace el proceso de administrar recursos, ejecutar un comando, etc. Los objetos de un programa no deben saber entre ellos como se implementa un comportamiento, o de donde se obtiene un atributo solicitado por medio del mensaje enviado, para poder lograr un alto grado de modularidad.

El "diccionario de métodos", es otro de los términos importantes dentro de la programación con

---

[6] Conferencia sobre "Programación Orientada a Objetos" impartida por Ing. Rubén Romero y Ing. Emiliano Llano en ENEP Acatlán, Noviembre de 1992.

objetos, y se define a sí mismo como un conjunto de asociaciones entre los mensajes selectors y los métodos incluidos en cada descripción de clase.

Independientemente de todos los demás, un objeto debe saber dar respuesta al mensaje que se le envíe, por lo que podemos tener entonces dos objetos diferentes entre sí y enviarles el mismo mensaje; cada objeto sabrá en que forma procesar el mensaje, independientemente del otro objeto al que se envió el mismo mensaje. Este "Polimorfismo" debe soportarse en cualquier lenguaje orientado a objetos, dado que el mismo lenguaje es considerado como un objeto mayor.

Si tenemos la siguiente expresión:

8*5

el objeto será el número 8, el mensaje que envío es "*" (multiplicación) con parámetro 5. La respuesta a ese mensaje será el número 40. Ahora bien si mandamos al objeto "TESIS " el mensaje +"MAC", obtendremos la palabra "TESIS MAC". De este modo, es sencillo pensar que cada objeto tiene cierta respuesta para cada tipo de mensaje.

Algunos lenguajes como Smalltalk o Eiffel, se basan en la abstracción de datos para eliminar la "tipificación". Esto es, que no debemos preocuparnos por el tipo de dato que se envíe como mensaje a cierto objeto, pues debe suponerse que él sabrá como procesar dicho mensaje.

Pero a todo ésto, para poder crear la aplicación que necesitamos en forma satisfactoria, es necesario tener dentro de nuestro programa, un flujo de datos y una lógica matemática que nos guíe por el camino adecuado de operación.

1.3. Introducción a la Programación Orientada a Objetos en SmallTalk/V en contraste con los lenguajes convencionales

La mayoría de los autores que se han consultado para ésta investigación, consideran a Smalltalk/V como uno de los lenguajes más potentes en la Programación Orientada a Objetos. Y se le atribuye este poder debido a sus características de interacción con el usuario en un ambiente puro.

El hecho de contar con rasgos puramente orientados a objetos, nos refleja una forma revolucionaria de crear software, lo cual significa una nueva dimensión en la cual se organiza el software de forma altamente reutilizable.

Smalltalk/V se considera como una avanzada implementación de lo que es Smalltalk, por lo que todo lo referido a continuación son algunas de las cualidades básicas que provee el lenguaje.

Todas las versiones de Smalltalk proveen una GUI con gran variedad de textos y gráficas. Esta última versión, creada para los ambientes de PC (Computadoras Personales), se soporta en algunos de

los ambientes de Inteligencia Artificial para estaciones de trabajo. Smalltalk pretende ser utilizado tanto para pequeñas evaluaciones, como para grandes aplicaciones (por ejemplo la construcción del propio "Ambiente Smalltalk/V").

El lenguaje orientado a objetos Smalltalk, es considerado como todo un "Ambiente", debido a que el usuario tiene un acceso directo sobre el lenguaje y manipulación sobre ciertos componentes del sistema (por ejemplo, puede modificar las ayudas que el sistema proporciona), sin necesidad de permanecer solamente en una opción predefinida por el mismo, con todas las restricciones que esto implica; esto es, Smalltalk utiliza el paradigma de "interfase de usuario" a través de ventanas de forma amigable.

Las ventanas no son otra cosa sino divisiones en forma de cuadro o rectángulo (según el formato que se defina) que despliegan en pantalla cierta información. Ver la figura 1.1

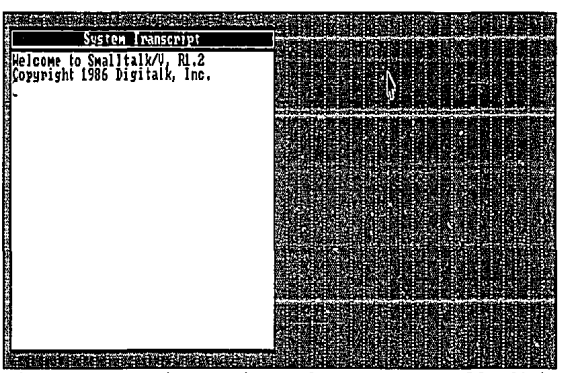


Figura 1.1 Primera vista al entrar a Smalltalk/V

La mayoría de las interacciones con el ambiente Smalltalk/V se dan a través de la ventana estándar principal de salidas: System Transcript, que se muestra en la figura anterior y que sería similar a los archivos de salida que crea C++ o Pascal. En Smalltalk, así como en los diferentes sistemas comerciales de actualidad que utilizan interfases de usuario, con el uso de ventanas podemos desplazar dicha ventana por la pantalla al lugar que se desee (como en el Sistema Operativo Windows por ejemplo). La manipulación de las funciones de entrada en ambientes de altas resoluciones con gráficos, puede hacerse a través de algún dispositivo como el teclado, o más fácilmente con el mouse (ratón) que es un aparato que apunta la dirección del cursor[7] y que puede moverse en la pantalla a través de una superficie de apoyo. Si el usuario llegara a seleccionar una ventana no deseada, podría remendar su error, volviendo a apretar el mismo botón en el mouse, sin repercusión alguna.

Alan Kay, no solamente creó a Smalltalk como un ambiente, sino como todo un "ambiente

[7] El Cursor se define como un punto en la pantalla, en donde se indica que estamos escribiendo algo o situando una posición.

integrado", en donde podemos entrelazar actividades con el mínimo esfuerzo sin perder información. Smalltalk con su integración de ventanas con gráficos y textos, surge como un ejemplo a seguir para aquellos lenguajes convencionales que quisieran integrarse a los futuros ambientes de programación que se desarrollen, pues se afirma que Smalltalk es un lenguaje pionero en las interfases gráficas de usuario.

Este paradigma de "interfase de usuario" esta basado en el traslape de ventanas (colocación de una ventana sobre otra, según las necesidades de cada usuario para optimizar el uso de todo espacio en pantalla), en donde varias tareas pueden ser vistas simultáneamente a través del despliegue en pantalla, delimitadas por una ventana. De esta forma podemos cambiarnos a alguna tarea específica o regresar a la actividad anterior. Ejemplos de algunas de éstas actividades son: crear gráficas, simulaciones, depurar programas y editar código, entre otras.

Smalltalk es un lenguaje usado para varias aplicaciones con un singular estilo de programación que promueve la creación y pruebas sobre prototipos (que pueden ser cambiados sin repercusiones posteriores). Estos prototipos no son sino metodologías en base a pruebas y errores para la creación de sistemas.

El código que se genera tiende a ser poco, debido a que el software ya nos proporciona utilerías existentes. Las salidas de un programa se manejan a través del traslape de ventanas y las entradas por medio de menús seleccionados con el mouse (ó teclado), lo que nos permite una programación "rápida". Un menú es un objeto que contiene una lista de opciones referentes a cierta ventana en particular. El manejo de textos, símbolos, información numérica y gráficas se advierten a la primera sesión que se tenga con Smalltalk.

De hecho se afirma que Smalltalk es un sistema orientado a Menús. Existen diferentes tipos de menús, y el tipo dependerá del lugar en donde se encuentre el cursor a la hora de seleccionar. Se pueden dar los siguientes casos de menús:

1.- Cuando el cursor se encuentre fuera de toda ventana, el menú que aparece será el del Sistema (System Menu), que es el principal y con el que más estaremos trabajando en el sistema (ver figura 1.2). Los principales comandos que contiene sirven para crear nuevas ventanas, salvar nuestro trabajo y salir del ambiente, entre otros.

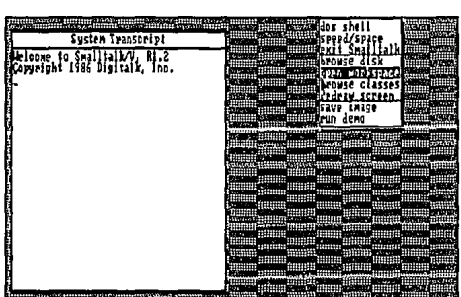


Figura 1.2 System Menu

2.- Cuando el cursor se encuentre sobre la etiqueta de una ventana marcada en video inverso, el menú que aparecerá será el de Ventana (Window Menu), el cual variará de acuerdo a cada ventana (ver figura 1.3)

La ventana activa, como su nombre lo indica es aquella que se encuentra solícita y se distingue por tener su etiqueta en video inverso y encontrarse sobre las otras ventanas.

3.- Cuando el cursor se encuentre dentro de la ventana en el área de trabajo (ó "pane") aparecerá el Menú Panel (Pane Menu) específico para cada panel que da cierta manipulación sobre ésta área de trabajo. En

la figura 1.4 se muestra por ejemplo, un Menú Panel del editor de Smalltalk.

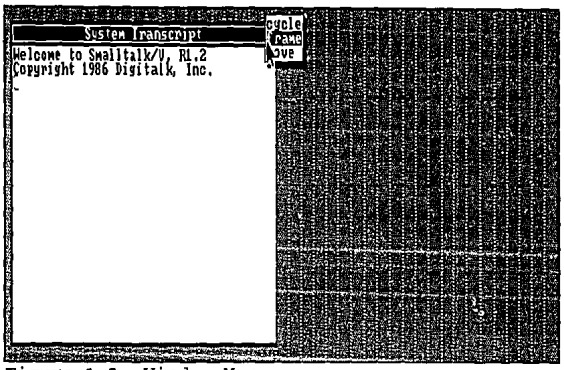


Figura 1.3 Window Menu

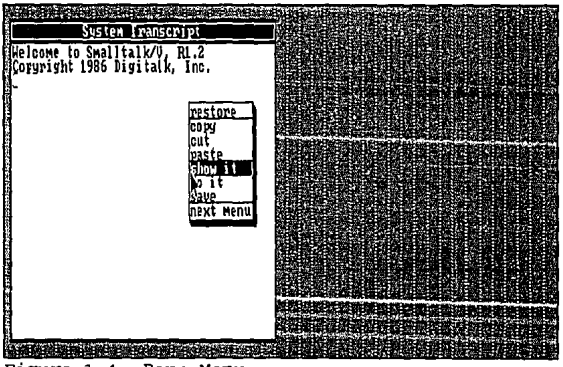


Figura 1.4 Pane Menu

En Smalltalk/V cada ventana y panel tienen su propio menú de ventana y panel, dependiendo de la aplicación de cada ventana.

Para poder introducirnos al lenguaje, lo más recomendable es comenzar por familiarizarse con el manejo de ventanas y menús. Para ello, es conveniente iniciar por correr las demostraciones que nos plantea el menú principal (ver figuras 1.2 y 1.5, respectivamente).

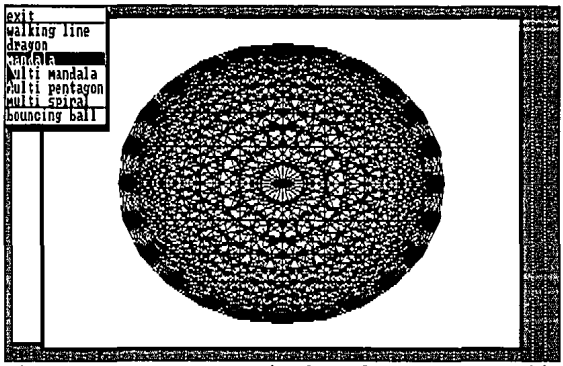


Figura 1.5 Corriendo la demostración seleccionada

Se dice que Smalltalk es un lenguaje "Interactivo", es decir, cuando un comando se ejecuta, nos enfrentamos directamente a un "diálogo" con una ventana. Este diálogo (como se menciona anteriormente) consiste en la ejecución de expresiones que envuelven las ideas que nos proponen la solución a lo requerido, es decir, el usuario indica que es lo que quiere hacer y el sistema reacciona proponiendo las diferentes maneras de hacerlo.

El editor de textos o también llamado "Panel de Texto" (Text Pane) de Smalltalk/V siempre trabajará con una copia del texto, la cual puede ser un archivo, cadenas de caracteres o pedazo de código. Esto depende en gran medida del tipo de text pane en el que estemos trabajando. En el text pane de la ventana "Class Hierarchy Browser" por ejemplo, lo que se edita es código de Smalltalk, a diferencia del text pane del System Transcript o del Workspace en donde se editan cadenas de caracteres. Un archivo, cualquiera que sea, puede ser editado, por ejemplo en la ventana del browse disk[8] que permite invocar archivos del disco.

La mayoría de los lenguajes de propósito general carecen de consistencia, cualidad que a Smalltalk caracteriza. El editor de textos no requiere al usuario de permanecer mucho tiempo en el teclado debido a su sencillez y objetividad en el uso de ventanas. Al ejecutar cualquier comando, no se solicita una confirmación al usuario, por lo que es importante saber elegir cuidadosamente una opción; ésta acción implica mandar un mensaje a un objeto. Existen dos formas de mandar un mensaje: seleccionando una opción en el menú o directamente por medio de una instrucción en Smalltalk, pues recordemos que todo en Smalltalk es un objeto, incluso el mismo lenguaje.

En el editor, un comando puede tener solamente un parámetro; por ejemplo podemos seleccionar un pedazo de texto que queremos borrar (selección activa) que pasa a ser el parámetro del comando "cut" (cortar, borrar). Si quisiéramos copiar ese pedazo de texto en otro lado, podríamos inmediatamente seleccionar la opción "paste" (pegar) e indicar donde queremos colocarlo.

---

[8] Todas estas ventanas serán explicadas más adelante.

Como ya habremos notado, una ventana está compuesta por un título en su parte superior (etiqueta), por una hoja, cuadro o también llamado "panel" dentro de ella, y por una o varias sub-ventanas de menú, con sus opciones. Estas sub-ventanas de menú tienen a su vez, opciones para activar otra ventana y cerrar o recobrar una ventana anterior, entre las principales. La opción que se desee utilizar en cualquier ventana o sub-ventana, debe ser seleccionada con las flechas del teclado (ó con el mouse) en el menú de barras de video inverso. Al activar una ventana seleccionando una opción, también podemos encontrarnos con texto y gráficas en un mismo recuadro (ver figura anterior).

Puede darse el caso en el que queramos ejecutar algún comando o expresión que no se encuentre en el menú activo (un menú activo es aquel que fue seleccionado). El problema puede resolverse fácilmente colocando el cursor en una parte de nuestro texto, en donde ya no haya caracteres, y escribir lo que necesitemos. Por ejemplo, podemos escribir la expresión:

```
8 max: 9
```

y marcarla en video inverso a base del cursor de texto en forma de viga y el mouse, y posteriormente seleccionar del menú la opción "do it" (Ver la figura 1.6) que nos dará por resultado: 9 que significa que el máximo número entre 8 y 9 es el 9.

El comando "do it" permite realizar la operación marcada pero no nos muestra el resultado, mientras que "show it" si nos mostrará en pantalla el producto.

El texto marcado a evaluar puede ser, ya sea una expresión o un comando de otro menú no activo. Por ejemplo podemos estar en el panel y escribir el comando:

```
Demo run
```

que pertenece al System Menu (la opción ahí se llama run Demo); activando el Pane Menu y seleccionando "do it" se presentarán las demostraciones listas para correr.

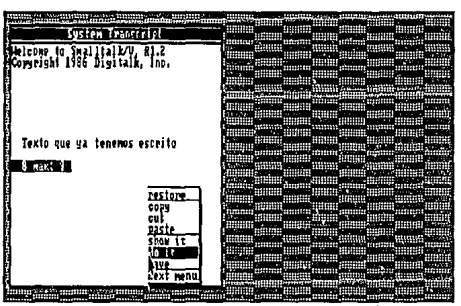


Figura 1.6 Evaluando una expresión

Si quisiéramos, podríamos evaluar aserciones múltiples, ya que Smalltalk/V nos permite la evaluación de varias expresiones separadas por un punto (.):

```
8 + 5.  2 * 8.
```

sin tomar en cuenta los espacio o tabulaciones que se hayan dado. La utilización de variables, es un punto de gran ayuda que nos proporciona el sistema. En Smalltalk/V al igual que en algunos lenguajes como C, el uso de Mayúsculas y minúsculas no es indistinto, pues existen palabras reservadas por el lenguaje con cierta peculiaridad que debe ser respetada.

Para poder manejar las variables temporales es necesario su declaración entre barras verticales (|) al principio del bloque que deseamos evaluar, pero que a diferencia de C++ o Pascal, no es necesario indicar de que tipo son:

```
| variableTemporal |
variableTemporal := 1 * 3.
```

si necesitáramos más variables, bastaría con declararlas entre las mismas barras (|) pero separadas cada una por espacios; y para saber el contenido de cierta variable, bastará con indicarlo anteponiendo el símbolo ^

```
^variableTemporal.
```

que tendrá por resultado 3, claro está, usando el remarcado de texto y el comando "show it".

En el ejemplo siguiente se muestra un bloque de código más elaborado sobre ejecuciones condicionales:

```
| bandera |
bandera := Prompter prompt: 'Desea responder?'
                                        default: 'no puedo'.
bandera = 'no puedo'
        ifTrue: [^0]
        ifFalse: [^1].
```

en donde bandera es una variable temporal que por default guarda el valor de 'no puedo' retornando el valor de 0, pero de responder otra cosa, el valor de retorno será 1. Y como podemos notar, nos auxiliamos del uso de prompters que son pequeñas ventanas ó paneles que heredan características directamente del "Text Pane", pero con la peculiaridad de que solamente cuentan con el menú del text pane y con una línea para editar; el tamaño del panel del promter dependerá del tamaño de la cadena que se indica en el prompt (en éste ejemplo sería la cadena: 'Desea responder?'). Por lo general se asignan a una variable para la entrada de cadenas de caracteres, dandose por omisión alguna respuesta. Para aceptar esta respuesta por omisión, lo único que hay que hacer es teclear <Enter> y el prompter se cerrará.

Como podemos ver con todo lo anterior, las asignaciones a variables en Smalltalk se hacen a través de un := al igual que en el lenguaje Pascal. De hecho, podemos comparar algunas de las instrucciones más usadas de Smalltalk[9] con algún lenguaje convencional como Pascal, en lo siguiente:

1.- Asignaciones a variables escalares. Tienen la misma sintaxis tanto en Smalltalk como en Pascal. Ejemplo:

```
e := 3 * e
```

2.- Series de expresiones. Las siguientes expresiones:

```
a := 'cadena de caracteres';
b := y;
c := 8
```

de asignación de cadenas de caracteres, variables y números, de Pascal, son iguales a una asignación en Smalltalk, excepto en que Smalltalk utiliza un punto (.) al final de cada expresión (en una serie de ellas); y en Pascal se utiliza un punto y coma (;) para separar instrucciones. En ambos lenguajes no es necesario ponerlo en la última instrucción.

3.- Llamada a una función con un argumento. En la siguiente instrucción en Pascal:

---

[9] Manual de Smalltalk/V, "Tutorial and Programming Handbook", Xerox Corporation, 1986, Pp.6-10.

```
y := size(var)
```

el valor de retorno de la función size (tamaño) con el argumento var es asignado a la variable y. Esto mismo es posible en Smalltalk de la siguiente manera:

```
y := var size
```

donde llamar a una función equivale a enviar un mensaje, por lo que se envía el mensaje size al contenido de la variable var.

4.- Llamadas a funciones con 2 argumentos. En Pascal podemos tener el número de argumentos requeridos separados por una coma (,):

```
a := max(arg1,arg2)
```

pero como en Smalltalk todo es un objeto tomaremos como base que el mensaje max se envía al primer argumento con el contenido del segundo argumento y el resultado se asigna a la variable a:

```
a := arg1 max: arg2
```

antecediendo el primer argumento al mensaje max:.

5.- Llamadas a funciones con 3 argumentos. En Smalltalk cuando existen 3 argumentos, el mensaje aparece antecediendo a los argumentos 2 y 3, y precediendo al primero:

```
a := arg1 max: arg2 and: arg3
```

en donde el nombre del mensaje es max:and:.

6.- Acceso a los arreglos. El acceso y manipulación de arreglos en Smalltalk, a diferencia de pascal, utiliza nombres de mensajes:

```
a := b at: i.
b at: i + 1 put: x.
b at: i + 1 put: (b at: i)
```

mientras que Pascal que utiliza paréntesis cuadrados y asignaciones:

```
a := b[i];
b[i+1] := x;
b[i+1] := b[i]
```

En realidad Pascal necesita para la utilización de un arreglo una declaración como ésta:

```
var
    r : array[1..13] of integer;
```

mientras que Smalltalk debe especificar (como caso único) la magnitud del arreglo para la variable con ese valor:

```
f := Array new: 13
```

pues para la utilización de las demás variables basta con mencionarlas temporales como se indica anteriormente.

7.- Ejecuciones condicionales

a) Instrucción if. En el siguiente ejemplo en Pascal:

```
if a = b then
  b := ord(c);
```

si la condición propuesta llega a cumplirse, se realizará la asignación. En Smalltalk un mecanismo parecido actúa sobre esta condición:

```
a = b
    ifTrue: [b := c asciiValue].
```

en donde "asciiValue" es equivalente a la función ord() de Pascal que retorna el valor ASCII de la variable.

b) Instrucción if-else. Algo parecido en ambos lenguajes se da con la condición "else". En Pascal:

```
if funcion(variable) then
        x := sum(variable,m)
else
        reset(variable)
```

En Smalltalk:

```
variable funcion
        ifTrue: [x := variable+m]
        ifFalse: [variable reset]
```

donde se da una expresión en caso de resultar falsa la comparación.

c) Instrucción de ciclo while. Actúan de forma similar en ambos lenguajes. Por ejemplo en Pascal:

```
while x < y do begin
        x := x+1;
        z := next(var)
end;
```

y en Smalltalk:

```
[x < y]
        whileTrue: [
                x := x+1.
                z := var next ].
```

donde mientras la condición resulte verdadera se retomará el ciclo.

c) Instrucción de ciclo for. Nuevamente como en las instrucciones anteriores, existe similitud. Veamos en Pascal:

```
for i:= 1 to 100 do
        m := m + a[i]
```

y su equivalente en Smalltalk:

```
1 to: 100 do: [ :i |
        m := m + (a at: i)
```

con lo que se cumplirá el ciclo de la variable i tomando valores de 1 a 100.

8.- Retornando el resultado de una función. En Pascal podemos asignar el resultado de una función a una variable. En Smalltalk se asignará el resultado de la evaluación de un método a una variable. Véase el siguiente ejemplo:

```
^respuesta
```

el símbolo ^ se antepone a la variable que guardará el resultado del método que retornará algún valor.

Es importante auxiliarnos de las instrucciones anteriores, pero no recomendable tratar de seguir la línea de programación de los lenguajes tradicionales, pues debemos tratar de sacar partido de las diferentes opciones y programación con ventanas que nos ofrece Smalltalk/V, basándonos en las clases y métodos que deseemos utilizar, modificar o llegar a crear.

Por otra parte, como se menciona anteriormente, es posible trabajar con múltiples ventanas a la vez. Dentro del menú de la ventana "System Menu" tenemos la opción de "open workspace" que nos indica la apertura de una ventana con un espacio de trabajo. Estando dentro de esta ventana podemos mandar el resultado de alguna evaluación a la ventana de "System Transcript". Por ejemplo en la figura

1.7 se manda el resultado de 5 * 7 a la ventana Transcript, mostrándolo por medio de los caracteres ///[10], donde en la primera expresión el objeto printOn se manda a imprimir con el argumento 5 * 7 de una forma más general, y en la segunda se manda el mensaje show (con el argumento ///) a la ventana Transcript. Si deseáramos mandar como argumento una cadena de caracteres, se tendría que encerrar entre comillas simples ('), pues las comillas dobles (") encierran a los comentarios.

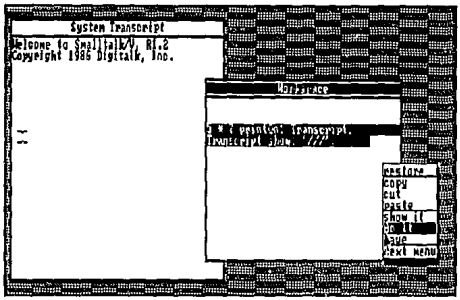


Figura 1.7 Ventanas Múltiples

Podemos también combinar expresiones, mandando como mensaje a "cr" utilizando el objeto Transcript, con lo cual se indica un retorno de carro (ó podemos mandar una tabulación con "tab"):

```
Transcript show: 'Palabra'.
Transcript cr.
```

ó utilizar una sola vez la palabra Transcript en una sola línea, mandando el resultado de ese objeto, a otro mensaje separando los mensajes con un punto y coma (;):

```
Transcript show: 'Palabra'; cr; show: ' Impresa'; tab.
```

lo cual equivale a:

```
Transcript show: 'Palabra'.
Transcript cr.
Transcript show: ' Impresa'.
Transcript tab.
```

Una de las principales herramientas que se nos presentan en Smalltalk para interactuar con el ambiente, es el "Intérprete" del lenguaje, el cual verifica la sintaxis línea a línea, marcando los errores; si no existe error, inmediatamente la ejecuta. El ambiente de programación Smalltalk "incluye un intérprete y un compilador", y aunque el lenguaje es considerado como un intérprete, muchos autores osan en llamarlo "lenguaje incrementalmente compilado". En nuestras primeras etapas de desarrollo, trabajaremos más con el intérprete al evaluar expresiones.

Cuando se compilan partes de nuestro programa, éstas llegan a ser parte del ambiente. Sin embargo, cuando el código es sometido por el intérprete, no llega a ser parte del ambiente, sino que las instrucciones son ejecutadas y posteriormente descartadas. En Smalltalk/V, eventualmente el código fuente que se desea interpretar es subordinado al compilador. El compilador produce un código objeto listo para ejecutarse, temporalmente alojado en memoria; cuando este código se ejecuta, la memoria libera su estadía y solamente aparece el código fuente en la ventana de edición.

---

[10] cada una de las líneas con las expresiones nos indican las dos formas diferentes de mandar a imprimir a otra ventana.

Pero ¿Qué es lo que entendemos cuando decimos que el código llega a ser parte del ambiente? y ¿A que se nos referimos con un "Lenguaje incrementalmente compilado"?

La respuesta está en que a diferencia de otros compiladores como C++ por ejemplo, en Smalltalk el ambiente incluye un compilador que se va incrementando al ir haciendo parte suya a los programas fuente que son compilados. Es decir, el código ejecutable llega a ser parte del ambiente corriendo en el propio contexto del ambiente de programación. En C++ todo lo que se requiere para correr un programa está apartado del lenguaje; el editor, compilador y ligador son programas por separado que no llegan a ser parte del archivo ejecutable que fue creado (separado también) cuando se compiló.

Una de las principales restricciones a las que estamos sujetos en los lenguajes compiladores, es la de tener que someter un programa (ó módulo) entero al compilador, pues no es posible analizar o cuestionar solamente cierto pedazo de código, que debe ser compilado y ligado entero para poder examinarlo posteriormente. La ventaja que se tiene en Smalltalk, es que el ambiente de programación está siempre presente en el compilador, pues podemos examinar aún el más pequeño pedazo de código (evaluar alguna expresión) con el "intérprete", o compilar una pequeña parte de código que llegará a ser parte del ambiente de programación (incluyendo en éste ambiente a las librerías, métodos y clases).

Los programas o pedazos de ellos que llegan a ser compilados cambian, como mencionamos anteriormente, el ambiente de programación; pero hablando en términos de software éste código ya ejecutable llega a ser parte del IMAGE SYSTEM de Smalltalk/V (que conjunta al código y los datos que forman parte del ambiente) y una vez incrementado éste archivo, podremos invocar el código para evaluar expresiones con el intérprete, si así se desea, o en su caso evaluar antes de compilar el programa.

Las modificaciones al propio ambiente, son posibles en Smalltalk/V debido a su interactividad con el archivo Image que puede ser cambiado a los intereses propios de cada usuario. De hecho Smalltalk/V fue escrito en Smalltalk, y podemos si deseamos aún modificar más el sistema. Existen 100 clases y 2 000 métodos con los que contamos para usar o modificar. Estas clases y métodos manipulan datos, números, fechas, horas, menús, archivos, ventanas, etc.

Para poder incrementar el ambiente es necesario salvar lo deseado, antes de salir a DOS. Al salir se nos presentan 3 opciones que son las siguientes:

1.- forget Image, que nos indica salir sin salvar
2.- Continue, nos permite continuar en Smalltalk/V sin salir
3.- save Image, salva nuestros programas en el Image System y sale a DOS. Si se quisiera entrar de nuevo a Smalltalk/V, el sistema comenzaría con los cambios que salvamos

o podemos salvar a través del System Menu.

Otra de las bondades entre las que nos proporciona el lenguaje, se encuentra el llamado "browser" (mirar, observar, ojear), que no es otra cosa sino que un objeto que funge como unidad básica en la interfase de usuario. El browser o browser de clases se presenta como una ventana que puede mostrarnos que métodos están definidos en una clase, y que clases están disponibles para trabajar, alfabéticamente o por jerarquías. Dentro del browser tenemos la ventaja de poder manipular los métodos y atributos de la clase.

El formato del browser es el siguiente: Se despliega una ventana con las clases disponibles, al seleccionar alguna de esas clases se muestra una ventana con sus métodos y otra con sus atributos, además de que se provee un pequeña ventana de editor en donde podemos modificar el código de un método, para posteriormente interpretarlo y verificar si se tiene algún error.

En sí a través del browser podemos hacer uso de la Herencia. Al indicar la clase de la que se pretende heredar comportamiento, debemos seleccionar los atributos que queremos heredar en la nueva clase, indicando también que nombre tendrá. Dentro del editor del browser, podremos después, añadir métodos a la nueva clase, e inclusive borrar o modificar la clase de la cual se está heredando. Otras de las operaciones que permite el browser de clases, son las siguientes: añadir o borrar atributos de una clase, y modificar métodos.

Dentro del System Menu además del browser de clases (Browser Classes) existe otro tipo de browser que es el "browse disk" que de una manera sencilla nos mostrará el contenido de nuestro disco, a través de varios paneles (List Pane). Primero al seleccionar dicha opción, se presenta un prompter que nos pregunta que unidad de disco queremos seleccionar. Por default aparecerá la unidad en la que está instalado el Smalltalk/V; posteriormente aparece la ventana del browse disk en donde la etiqueta es el volumen de la unidad. En ella aparecen 3 particiones o paneles como se muestra en la figura 1.8. La de la izquierda superior es una ventana en donde se muestran los directorios en forma de árbol y por jerarquías. Al seleccionar algún directorio, aparecerá en la ventana derecha superior el contenido de dicho directorio. Y al seleccionar en ésta ventana algún archivo, se llenará la ventana de centro inferior con la información que contiene dicho archivo, y la cual se podrá modificar desde ahí con el editor de textos. Si no se llegará a seleccionar ningún archivo de la ventana derecha superior, el contenido de la ventana de centro inferior serán los mismos archivos de ese directorio, pero indicándose sus bytes, fecha y hora, como se muestran en el MS/DOS al introducir el comando "dir".

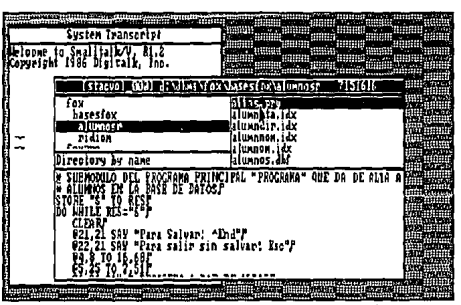


Figura 1.8 Formato del Browse Disk

La depuración de programas se hace de una manera sencilla, en donde el sistema indica el error, y al modificarlo no será necesario volver a correr todo el programa desde el principio, sino a partir del pedazo de código donde se halló el error. Esto se logra a través de la función "restart" que es de gran ayuda sobre todo cuando programas grandes corren bien hasta que se topan con un error. Los mensajes de error aparecen en puntos en los que la ejecución se interrumpe por un descuido del programador, en donde se notifica que tipo de problema surgió. Si un error es supuesto como irrecuperable, el mensaje que aparece es el siguiente:

self error: 'error Whatever'.

Uno de los mensajes con los que es posible se tope frecuentemente el programador, es el de:

"Message not understood"

lo cual significa que un mensaje no fue entendido por Smalltalk, pues un método es enviado como mensaje a un objeto que no tiene definida ni en su clase y ni en su superclase al método que recibirá a dicho mensaje. En el tiempo de ejecución de un programa, los objetos comienzan a procesar los mensajes, y es por lo tanto que la detección de ese tipo de errores se da al correr el programa y no en el tiempo de compilación.

Si quisiéramos dibujar la traza "mandala" que se encuentra en las opciones de Run Demo, tendríamos que manejar ciertos parámetros que se discutirán posteriormente. Tecleando las siguientes instrucciones, podemos verificar un error que se da al tratar de evaluar el código marcándolo con video inverso y seleccionando la opción "do it" (ver figura 1.9). Por ahora lo que nos interesa es ejemplificar los errores al tiempo de compilación y los errores al tiempo de ejecución.

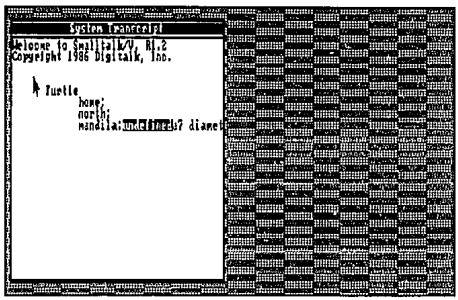


Figura 1.9 Error de Compilación

El error de compilación detectado es "undefined" (No definido) que nos indica que no está definida la variable "a7" y por lo tanto el parámetro debe ser numérico (y lo escrito es a7) y quedar como 7 nada más. Al corregir esto y evaluarlo se nos presenta otro error, pero ahora en tiempo de ejecución (cuando ya no existen más errores de compilación). Ver figura 1.10. El mensaje enviado debía ser "mandala" en lugar de "mandila", por lo que se indica un mensaje no entendido en una ventana llamada "WalkBack Window". El resultado de corregir los errores tecleados se observa en la figura 1.11 con la traza "mandala" con 7 vértices.

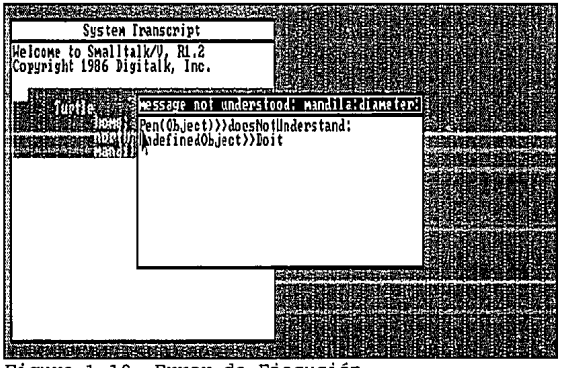


Figura 1.10 Error de Ejecución

Debido a que un objeto tiene la capacidad de mandar mensajes a otro objeto y recordar sus propios procesos, se dice que Smalltalk es un sistema que incluye un estado dinámico de sus propios

procesos. Este proceso dinámico es una extensión de las ideas con las actividades de interacción humana, es decir, del mundo real. Cuando el programador desarrolla ciertas aplicaciones, usualmente lo hace teniendo en mente ciertos objetos. Basados en dichos objetos podemos programar de manera consistente y desarrollar las aplicaciones necesarias. Los objetos pueden ser las entradas o salidas en éste proceso dinámico.

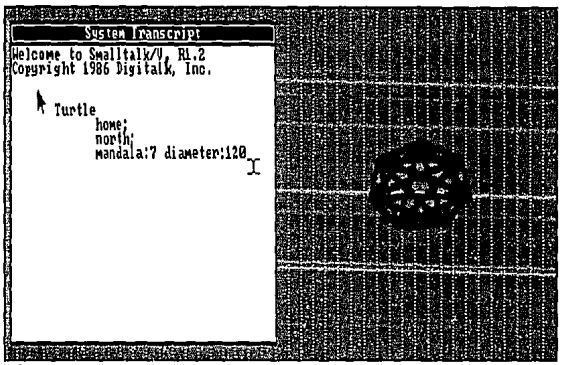


Figura 1.11 Traza MANDALA con 7 vértices y diametro 120

Con todo lo expuesto anteriormente, podemos darnos cuenta de la infinidad de características que hacen diferente a Smalltalk/V de los lenguajes tradicionales, refiriéndonos a lenguajes tradicionales como aquellos que se usan comunmente en desarrollos prácticos como BASIC, Fortran, Pascal y C, entre otros.

En la programación procedural construimos programas que no nos garantizan un programa robusto y libre de errores. La importancia de la programación orientada a objetos radica en su modo de uso, facilidad de depuración y lectura, y su sencillez en las modificaciones a los programas.

En lenguajes tradicionales como Pascal o C, los procedimientos (ó en el caso de C, funciones) son la construcción básica de los programas, partiendo de una vista que va de lo general a lo particular (subrutinas). Se plantea un problema general y nosotros trataremos de darle una posible solución partiéndolo en pedazos lo más concretos posibles, generándose así cierto nivel de abstracción en la descomposición del planteamiento inicial. En la orientación a objetos, ésta descomposición parte del concepto de jerarquías en donde se ordena al conocimiento en una estructura común. Cabe mencionar que Smalltalk fue el primer lenguaje que manejó los términos de clase y herencia, aunque Smalltalk/V aún no soporta la herencia múltiple.

La técnica de arriba hacia abajo, permite a los objetos actuar uno después de otro. La estructura entre la programación tradicional y la programación orientada a objetos es diferente. En la programación con objetos, como hemos estado viendo, la estructura fundamental es el propio objeto que se rige de ciertos métodos. De hecho los desarrolladores de Smalltalk en un principio decidieron llamar a los objetos "task" (tarea), pero posteriormente el nombre se cambió al de "object" (objeto).

Muchas veces en la programación procedural se utilizan variables globales que envuelven a todos los procedimientos, y las cuales pueden ser afectadas de forma no intencional. En la programación orientada a objetos no se tiene ese problema, pues existe un ocultamiento y abstracción de los datos. Este ocultamiento de datos puede lograrse en un lenguaje procedural, pero resulta difícil el análisis de la lógica que se seguirá, y la acción de forzar a salirse al lenguaje de sus definiciones propias, por lo que es más factible su desarrollo en un lenguaje que ya lo prevé, como todos los lenguajes orientados a objetos, y entre ellos Smalltalk/V.

Entre las ideas que distinguen a los procedimientos de los objetos, existe la de que éstos últimos tienen cierto ciclo de vida, es decir, que pueden ser creados y destruidos sin afectar a todo el programa. Mientras que las ataduras entre procedimientos se da en el paso inevitable de parámetros, por lo que resulta difícil la modificación y mantenimiento de los programas. En la programación orientada a objetos con Smalltalk/V las conexiones entre los componentes del sistema se reducen de forma extraordinaria al aplicar sus propios conceptos.

Podemos pensar en que como la programación orientada a objetos es la técnica más sobresaliente por su innovación de conceptos, resulta fácil un favorecimiento a Smalltalk en contraste con los lenguajes convencionales. Pero podemos hacer también esta comparación con los lenguajes híbridos de orientación a objetos basándonos en todo lo expuesto anteriormente, y además podemos añadir concretamente otras particularidades, que a continuación se mencionan.

La ventaja de que Smalltalk/V sea un ambiente, es que los usuarios pueden definir nuevos tipos tomando las características de otras clases y modificar el mismo ambiente (como se explica anteriormente).

Se dice que en un lenguaje que no cuenta con una verificación de "Tipo de dato declarado", se tienen más errores de lo normal. En lenguajes donde existe una declaración de tipos como Pascal y C, los errores que pueden aparecer son los de:

"type mismatches"

en donde los tipos de datos no concuerdan. En Smalltalk, BASIC y LISP por ejemplo, no es necesaria una declaración. En particular en Smalltalk/V un programa no rompe su secuencia por ese tipo de errores; el error que se reporta es el de "Message not understood" quien cuenta con una ventana (window notify) que nos ayuda a reparar rápidamente el problema, sin necesidad de "adivinar" en donde estuvo el descuido.

En Smalltalk/V no es necesario seguir el ciclo tradicional de editar-compilar-depurar, pues los errores se encuentran anticipadamente a la hora de examinar el programa. Para no tener el problema de no documentar un programa con las declaraciones de tipos de datos, es recomendable escoger el nombre adecuado a cada objeto que se utilice dependiendo de su función.

Por último, creemos que la mayor diferencia entre los lenguajes convencionales y los objetos en Smalltalk, radica en la manera de pensar de los usuarios con respecto a la forma de programar. Esa es la cuestión.

CAPITULO II

"UTILIZACION DE OBJETOS EN SMALLTALK"

2.1 Los mensajes

2.2 Los tipos de variables y expresiones

2.3 Estructuras de control

2.4 Qué son las clases y su forma de utilización

2.5 El uso de métodos

CAPITULO II. UTILIZACION DE OBJETOS EN SMALLTALK.

2.1. Los Mensajes

A partir de éste capítulo comenzaremos con la utilización de objetos en el lenguaje Smalltalk, y específicamente en el sistema Smalltalk/V para ejemplificar el ambiente de programación orientada a objetos.

Como sabemos, los objetos son la construcción básica en el lenguaje; representan un componente en el sistema Smalltalk/V, y se definen como datos por sí mismos. Un objeto en sí puede tomar cualquier forma, desde una cadena de caracteres, hasta un arreglo, por ejemplo:

```
'Este es un objeto tipo cadena'.
#('Objeto1' $d 3 ('cuatro' 5)).
```

Como podemos notar en la última expresión, el arreglo es un objeto que a su vez puede contener otros objetos. Como arreglos dentro de arreglos.

Pero a pesar de ser los objetos el concepto fundamental en el ambiente, ellos no pueden actuar por sí mismos, es decir, es necesario establecer una comunicación entre ellos, por medio de los mensajes.

Un mensaje es una petición al objeto de ejecutar cierta operación, de éste modo el mensaje especifica la operación deseada, pero no como ejecutarla.

Todo proceso en Smalltalk envuelve enviar mensajes a los objetos; además, los mensajes se utilizan para crear instancias de clase (objetos). Los mensajes son entonces la interacción del lenguaje, que nos permite expresar nuestros requerimientos hacia los objetos. El conjunto de mensajes a los que el objeto puede responder, se les conoce como "interfase del objeto" con el resto del sistema. Y así entonces, la única manera de interactuar con el objeto, es a través de su interfase.

Fundamentalmente el mensaje se divide en dos partes: el objeto que recibe el Mensaje (Receiver Object) y el Mensaje Selector (Message Selector). A su vez el mensaje selector contiene cero o más argumentos (Message Argument).

En la siguiente expresión:

```
#('1' '2' '3' '4') at: 4  ==>>  '4'.
```

el mensaje selector es at: con un argumento número entero 4, y el objeto receptor es el arreglo de cadenas. El resultado de la expresión es el objeto '4', pues un mensaje siempre retornará un objeto simple.

Nota: Las flechas ==>>, como denotación particular en éste trabajo, nos indicarán el resultado después de evaluar las expresiones.

Generalmente los mensajes con cero argumentos se conocen como Mensajes Unarios (Unary Messages).

Podemos también tener el caso de mensajes unarios compuestos, es decir, más de un mensaje (sin argumentos) se envía a un objeto receptor:

```
ObjetoReceiver Mensaje1 Mensaje2
```

y las evaluaciones siempre se ejecutarán de izquierda a derecha.

Al enviar un mensaje a un objeto, se ejecutan las siguientes funciones:
1. Identificar el objeto al que se le envía el mensaje (objeto receptor)
2. Identificar los argumentos contenidos en el mensaje (message argument)
3. Y determinar el método que concuerde con el mensaje selector, dentro de los métodos definidos en el objeto receptor.

Ahora bien, puede darse el caso en el que mandemos un mensaje incorrecto a un objeto, como por ejemplo:

```
#('1' '2' '3' '4') * 3
```

en donde se manda el operador * a un arreglo. En éste caso, la respuesta será una ventana WalkBack que nos indica un mensaje no entendido:

Con esto, podemos darnos cuenta de que un objeto siempre sabe de que manera responder al tipo de mensaje que se le envía.

Una forma de conocer los mensajes permitidos para cierto objeto, es a través del Browser de Clases, observando el panel de métodos.

Los mensajes con uno o más argumentos son nombrados como Mensajes de Palabras Clave (Keyword Messages):

```
'Indicando la letra A' at: 20.
```

Dentro de los mensajes Keyword existen los mensajes que trabajan con dos o más cadenas que reciben sus propios argumentos:

```
'MAk' at: 3 put: $C.
```

En ésta expresión podemos cambiar los elementos de la cadena, con at: put:, el cual es un mensaje que se divide en dos palabras clave.

Algunos objetos en Smalltalk, no permiten el uso de dos o más claves en un mensaje keyword, pues como hemos dicho antes, no cuentan con una definición de ello en sus métodos. Sin embargo, este problema puede resolverse con el uso de paréntesis. Por ejemplo, podemos tener la siguiente expresión:

```
pen goto: (ColeccionDefinida at:1)
```

en donde se mandan dos mensajes keyword a la variable pen que anteriormente fue inicializada como tipo Pen[1], la cual no tiene definido algún método que acepte un mensaje con dos claves (en éste caso goto: at:), por lo que se utilizaron los paréntesis para separar mensajes (el uso del paréntesis permite evaluar primero la expresión que se encierre en ellos), de lo contrario el sistema podría generar un dialogo de error en donde se interpretarían dos mensajes sin entender:

```
"goto:at:" not understood.
```

pues el intérprete evaluaría la expresión de izquierda a derecha y pensaría que se envían dos claves no definidas.

En Smalltalk no existe precedencia de operadores, es decir, se evalúan las expresiones estrictamente de izquierda a derecha. Para alterar el orden de evaluación, como vemos, podremos hacer uso de los paréntesis ().

Otro tipo de mensajes en Smalltalk son los Aritméticos (Messages Arithmetic), que se basan en operaciones aritméticas entre números. Como en cualquier lenguaje tenemos operadores de suma (+), resta (-), multiplicación (*) y división (/). Por ejemplo:

```
10 // 3  ==>>  3
```

muestra una expresión donde // indica una división entera. En éste caso el Objeto Receptor es el número 10, // es el Mensaje Selector y el número 3 es el argumento. De ésta evaluación resulta el objeto número entero 3 (cociente), pues la división 10//3 se trunca a un número entero. Si se desea saber el residuo, se utiliza el operador \\, que igualmente regresa un valor entero.

Al evaluar una operación racional (/), el resultado siempre se almacena en una fracción (un número numerador y un número denominador) puesto que no existe redondeo en la aritmética de fracciones en Smalltalk, que se expresa de forma exacta. Y así, al evaluar cualquier fracción tendremos como resultado otra fracción simplificada.

---

[1] **Clase que se deriva de la superclase Object, y que será explicada más adelante.**

```
10 / 20 ==>> 1 / 2.
```

En el caso de tener un número fraccionario ya simplificado como 1/3, el resultado será el mismo número fraccionario:

```
1 / 3 ==>> 1 / 3.
```

Nota: En caso de que la computadora con la que se trabaje cuente con un Coprocesador Matemático, se pueden evaluar números puntos flotantes en expresiones aritméticas (V.gr: $36 + 4.5\ e^3$).

Los mensajes aritméticos se clasifican dentro de los mensajes binarios, pues éstos son aún más generales.

Un Mensaje Binario (Binary Message) se aplica con un operador sobre dos argumentos. Estos argumentos pueden ser números, cadenas o arreglos. El operador siempre debe ser una mensaje selector que se encuentre definido en los mensajes permitidos para la activación del objeto, además de no ser número, ni letra. Ejemplo de éste operador es la coma (,) que concatena argumentos cadenas y arreglos:

```
'Tesis',' MAC' ==>> 'Tesis MAC'
#(2 4),#(6 8) ==>> (2 4 6 8).
```

El orden de precedencia para evaluar los mensajes, se da en la siguiente tabla de jerarquías:

Orden de Precedencia	Tipo de Mensaje
>	
.	Unarios
.	Binarios
.	Keyword
<	

Como siempre, los paréntesis podrán alterar el orden de precedencia entre los mensajes.

También existe la posibilidad de tener varios mensajes dentro de un mensaje, en donde se respeta el orden de precedencia:

```
'cadena' size + #(1 2 3 4) size factorial between: 'hola' size and: 30
```

el mensaje anterior contiene los siguientes 6 mensajes: size, +, size, factorial, between: and: y size. Para ésta expresión el orden de evaluación es el siguiente:

```
'cadena' size ==>> 6
#(1 2 3 4) size ==>> 4
#(1 2 3 4) size factorial ==>> 4 factorial ==>> 24
'cadena' size + #(1 2 3 4) size factorial ==>> 6 + 24 ==>> 30
'hola' size ==>> 4
30 between: 4 and: 30 ==>> True.
```

de donde obtenemos el valor verdadero (true) por encontrarse el número 30 entre el rango [4..30], respetándose así el orden de precedencia entre mensajes.

Los objetos referidos como verdadero (true) o falso (false) son llamados objetos booleanos, los cuales representan valores "si" o "no" como una respuesta. Los objetos booleanos responderán siempre a mensajes que computen proposiciones lógicas.

2.2. Los Tipos de Variables y Expresiones

Una expresión es una secuencia de caracteres que describen a un objeto. En Smalltalk/V existen tres tipos de expresiones: expresiones literales, expresiones de mensajes y expresiones con nombres de variables:

1) Las expresiones literales se refieren a manifestaciones por medio de letras, números y símbolos. Por ejemplo:

```
'expresión literal'
```

donde se determina una expresión de letras, o bien podemos referenciar una expresión de símbolos y números:

```
#(1 2).
```

2) Las expresiones de mensajes se refieren a utilizar un mensaje para exponer una acción al objeto receptor:

```
Transcript show: 'Palabra'.
```

El valor de una expresión de mensaje siempre se determina por el método que será invocado, y que se encuentra en la clase del objeto receptor.

3) Finalmente las expresiones con nombres de variables referencian a un objeto por medio del nombre de una variable. Por ejemplo:

```
variable := 'objeto cadena de caracteres'.
```

Una variable en Smalltalk siempre contiene un objeto, y su nombre se utiliza para referenciar a ese objeto en particular. Un nombre de variable es un identificador definido en una secuencia de letras y dígitos, comenzando siempre con una letra. En Smalltalk/V no es necesario truncar su longitud a 8 caracteres, como en la mayoría de los lenguajes de programación.

El objeto contenido por la variable, puede cambiar dependiendo de las asignaciones que se hagan a dicha variable.

Las variables pueden ser tanto privadas, como compartidas. Las variables privadas son accesadas solamente por un simple bloque de código, mientras que las compartidas se accesan en todo el ambiente.

Existen tres tipos de variables: variables de instancia, variables temporales y variables globales. Las dos primeras, son ejemplos de variables privadas, mientras que las globales son compartidas. Las variables de instancia son un tipo especial de variable, que será explicada más adelante en éste capítulo.

Por ahora, comencemos a analizar las variables globales. Estas variables se pueden accesar por múltiples objetos en todo el ambiente, esto quiere decir, que son variables utilizables en todo el sistema, y no solamente en un programa o pedazo de él.

Los nombres de las variables globales siempre comienzan con una letra mayúscula y van seguidas de otras letras y/o números.

Para poder hacer parte permanente del ambiente a un objeto, es necesario la declaración de una variable global. Las variables globales se almacenan en un diccionario especial de Smalltalk[2], para poder ser accesadas por las instancias de todas las clases que Smalltalk contiene.

Smalltalk provee una gran cantidad de variables globales predefinidas para su uso en cualquier aplicación -exactamente 140-, lo cual le otorga un gran poder al lenguaje sobre otros en su rama.

Cuando tratemos de evaluar una variable que no existe en el ambiente, la respuesta que obtendremos será un menú con dos opciones:

Una de ellas nos pregunta si se desea definir a la variable como global, y la otra, como caso contrario supone a la variable no definida en el sistema, por lo que se asume un error y se despliega un mensaje que indica el equívoco.

Undefined.

---

[2] **Que será analizado posteriormente en éste mismo capítulo.**

Con éste mensaje Smalltalk nos protege de creaciones accidentales de variables globales.

Una de las variables globales definidas por el lenguaje, más utilizadas en Smalltalk/V, es la variable global "Turtle" (tortuga), quien es una instancia de la clase "Pen" (pluma).

Todas las versiones de Smalltalk, proveen una clase Pen para dibujar imágenes gráficas. Un uso práctico de Pen se refleja en el trazo de líneas entre puntos. Pen usa variables para recordar su posición (arriba o abajo) y su color.

Cuando una pluma se encuentra en posición "up" (arriba), podemos movernos a cualquier lado de la pantalla sin marcarla. En caso contrario, si se encuentra en posición "down" (abajo), podremos marcar un trazo a otra posición desde el punto en que se encuentre, hasta el indicado.

Para poder dibujar una línea, la pluma necesita conocer el lugar en donde comenzará y la dirección. La dirección contiene el punto final a donde se trazará la línea, mediante el mensaje direction:. El ángulo de una pluma se expresa en grados (el este a 0° y el norte a 270°).

Nosotros podemos crear un nuevo objeto Pen con el mensaje new,

```
objetoPen := Pen new
```

donde se inicializa un nuevo objeto de tipo Pen en la posición 'down' al centro de la pantalla, por lo que es importante posicionar el Pen arriba, si no se desea hacer ningún trazo.

```
objetoPen up.
```

Podemos también cambiar el color de una pluma, de acuerdo a los siguientes mensajes: black (negro), gray (gris) y white (blanco):

```
objetoPen gray.
```

Una pluma cuenta con un punto que indica su posición en la pantalla y con un número que indica la dirección hacia la que se mueve. Una pluma siempre entenderá los mensajes que cambien su posición o dirección.

Entre otros, podemos enviar a cualquier pluma los siguientes mensajes:
- home, que centra en la pantalla la pluma
- north, direcciona la pluma a 270 grados
- go: unaDistancia, que mueve al objeto receptor tipo pluma hacia la longitud señalada en el objeto argumento del mensaje (distancia).
- goto: unPunto, mueve la pluma hacia el punto indicado.

La idea de éste tipo de pluma, fue tomada del lenguaje LOGO que provee una variable llamada "turtle" para el dibujo de líneas en la pantalla. De hecho, es por ello que Smalltalk crea la variable global tipo pen, llamada igualmente "Turtle".

El protocolo para el soporte de plumas en Smalltalk/V, es parecido al que provee LOGO. Este consiste de comandos para ordenar a la pluma la distancia, posición, movimiento, etc.

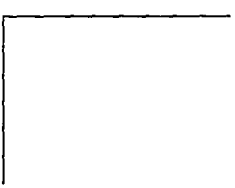
En Smalltalk/V, la variable global Turtle es una pluma que a través de su movimiento, dibuja en pantalla su trazo. Turtle contiene a un objeto. De hecho, todas las variables globales contienen un objeto simple:

```
Turtle ==>> a Pen.
```

Para ejemplificar el uso de las plumas, podemos evaluar una serie de expresiones del objeto Turtle como sigue:

```
"Se inicializa Turtle a color negro"
        Turtle black.
"Se inicializa Turtle al centro"
        Turtle home.
"Se direccionan los movimientos de Turtle"
        Turtle go: 100.
        Turtle turn: 90.
        Turtle go: 100.
        Turtle turn: 90.
```

En esta serie de expresiones mandamos diferentes mensajes al objeto Turtle para dibujar dos trazos.



Estas expresiones también pueden escribirse por medio de "expresiones en cascada":

```
Turtle
   black;
   home;
   go: 100;
   turn: 90;
   go: 100;
   turn: 90.
```

donde se evalúan múltiples mensajes a un mismo objeto, denotando cada instrucción por medio de un punto y coma (;). Recordemos que en el capítulo anterior utilizamos de la misma forma al objeto Transcript.

Con éste tipo de expresiones, se da cierta estructura a la programación en Smalltalk, para mayor claridad al programador.

Otra forma de expresar la tarea anterior, se hace a base de ciclos (loops) simples. En ellos se iteran los mensaje enviados al objeto. Estos ciclos simples se basan en un objeto número entero (objeto receptor), al que se le manda un mensaje selector como el timesRepeat: para repetir una tarea, un determinado número de veces.

```
Turtle black;
        home.
2 timesRepeat: [Turtle go: 100; turn: 90]
```

Aquí el mensaje selector toma como argumento a un bloque de código que indica las expresiones a ejecutar.

Hablemos ahora de un tipo de variables comúnmente utilizadas por el programador de Smalltalk: las variables temporales. Las variables temporales son variables que necesitan ser declaradas entre barras verticales (|) y comenzar siempre con una letra minúscula. Dichas variables son creadas para actividades específicas, y están disponibles solamente en el tiempo de duración de la actividad, por lo que se consideran de uso auxiliar en los programas.

Al igual que las variables globales, las variables temporales no necesitan declararse con un tipo específico; una variable temporal puede envolver cualquier tipo de objeto.

Dentro de las variables temporales tenemos tres tipos de ellas: variables temporales en pedazos de código (program temporaries), variables temporales contenidas en bloques (block arguments) y variables temporales de métodos (method temporaries).

Por ahora, lo que nos interesa, es hablar de las variables temporales en pedazos de código, pues por la estructura de éste trabajo de investigación, las variables temporales contenidas en bloques y las variables temporales de métodos, serán explicadas en los siguientes subcapítulos de "Estructuras de control" y "El uso de métodos", respectivamente.

Las variables temporales en pedazos de código, se utilizan en forma auxiliar en un programa, y posteriormente a su ejecución, son destruidas. Esto es que cuando hacemos uso de las variables temporales, éstas se crean para un uso determinado, pero posteriormente son descartadas de la memoria, pues como su nombre lo indica, son de uso "temporal" o provisorio.

Por ejemplo, podemos calcular el factorial de un número determinado, con variables temporales:

```
| var nf |
var := 4.
nf := var factorial.
^nf
```

en donde se declaran las variables temporales var y nf, y se hace una asignación de expresiones (Expressions Assignment):

```
var := 4.
nf := var factorial.
```

En la asignación de expresiones, también podemos tener asignación del resultado de un mensaje

a una variable, como se muestra anteriormente.

```
nf := var factorial.
```

El resultado del mensaje es un objeto simple. Finalmente regresamos el valor calculado con una expresión de retorno (return expression) ^nf.

Para poder ver el contenido de una variable temporal, aún cuando se pida el retorno de su valor por medio del símbolo ^, es necesario evaluar el código con la opción "show it" en donde el mensaje "printOn" (imprimir) se envía automáticamente a la variable que se desea retornar; y el argumento en éste caso, será el contenido de la variable que se evaluó en el proceso. Con la opción "do it" el intérprete procesará lo deseado, pero no nos mostrará el resultado en pantalla.

Por otra parte, con respecto al tipo de chequeo de las variables en Smalltalk, éste se lleva a cabo en forma dinámica, como ya hemos mencionado en el capítulo anterior.

En los compiladores de lenguajes tradicionales como C, Ada o Pascal, la forma de chequeo de las variable utilizadas, es el Estático (Static). En el tipo de chequeo estático, el compilador verifica que los argumentos pasados entre las funciones coincidan con el tipo de valor con que fue declarado, pues de lo contrario se producirá un error. En cambio en la forma dinámica no se indica el tipo de variable que se utiliza, pues durante el tiempo de ejecución, una variable puede cambiar de tipo tanto como lo vaya requiriendo la lógica del programa; y es a la hora del paso de mensajes entre objetos, cuando se cuestiona el tipo de dato ó clase a la que pertenece. Por ejemplo, inicialmente podemos asignar a una variable el valor de una cadena, y posteriormente cambiarlo al de un arreglo o un objeto que maneje ventanas:

```
variable := 'cadena'.
            ...
variable := #('cadena' 9 10 @ 20).
            ...
variable := defaultNew (Window, 'Mensaje dentro de ella').
```

según vayamos requiriendo.

Posiblemente, podemos enfrentar algunos problemas, al no saber específicamente que tipo de objeto manejará cierta variable, por lo que es importante saber elegir cuidadosamente y objetivamente el nombre de la variable, y saber así a que objeto haremos referencia.

2.3. Estructuras de Control

El sistema Smalltalk/V incluye objetos y mensajes que implementan el control de estructuras estándar que podemos encontrar en la mayoría de los lenguajes de programación, pero con ciertas peculiaridades.

En general las estructuras de control se definen a sí mismas como la agrupación de instrucciones que evalúan expresiones condicionales y ejecutan cierta acción basadas en dicha evaluación.

Como ya hemos visto en el capítulo anterior, para poder operar ejecuciones condicionales, es necesario la utilización de las estructuras de control, que nos permiten la manipulación directa de ciertas instrucciones en el lenguaje.

Smalltalk se basa en la comparación de objetos, por medio de mensajes para las expresiones condicionales. Estos se implementan como "mensajes binarios". Los operadores relacionales que trabajan como mensajes binarios son los siguientes:

Operador	Significado
<	menor que
>	mayor que
<=	mayor o igual
>=	menor o igual
=	igual que
==	equivalente a
~=	diferente de

sin existir un orden de precedencia.

Con respecto a ésta tabla, existe una diferencia entre la igualdad (=) y la equivalencia (==), que consiste en que la igualdad retorna valor verdadero para aquellos objetos que contienen los mismos elementos, mientras que la equivalencia compara que dos objetos sean el mismo objeto. Ejemplifiquemos esto:

```
| objeto1 objeto2 |
objeto1 := #(1 2)
objeto2 := #(1 2)
objeto1 = objeto2  ==>> true
```

pero

```
| objeto1 objeto2 |
objeto1 := #(1 2)
objeto2 := #(1 2)
objeto1 == objeto2  ==>> false.
```

Un ejemplo de una equivalencia verdadera es el siguiente:

```
| objeto1 objeto2 |
objeto1 := #(1 2).
objeto2 := objeto1.
objeto1 == objeto2  ==>> true.
```

Volviendo a las expresiones condicionales, en Smalltalk existen mensajes que cuestionan el estado

de un objeto (testing objects), y que retornan un valor de verdadero o falso, según sea su caso. Por ejemplo:

```
('cadena' at: (#(1 2 3) at:1)) isLowerCase  ==>> true
8 odd  ==>> false
```

en la primera expresión, la subexpresión:

```
('cadena' at: (#(1 2 3) at:1))
```

tendrá por resultado el caracter $c. De donde el mensaje isLowerCase cuestiona a un objeto de tipo caracter, verificando si es una letra minúscula. En la segunda expresión se evalúa a un objeto número entero sobre su condición de ser par o impar (odd).

Este tipo de mensajes, al igual que los operadores binarios, se utilizan para realizar ejecuciones condicionales. Como por ejemplo:

```
$e isVowel
        ifTrue: ['ejecuta bloque verdadero']
        ifFalse: ['ejecuta bloque falso'].
```

Conjuntando todo lo que hemos visto, hagamos un ejemplo que utilice variables temporales, cadenas y objetos testing:

```
"Programa que convierte primera letra
 del arreglo de nombres, a mayúscula"
| arreglo nombre primeraletra i |
"arreglo contiene el arreglo a evaluar"
arreglo := #('brenda' 'elisa' 'Solis' 'perez').
"la variable i es un contador indice al arreglo"
i := 0.
arreglo size timesRepeat: [
        i := i+1.
        nombre := arreglo at: i.
        primeraletra := nombre at: 1.
        nombre at: 1
                put:
                        (primeraletra isLowerCase
                                ifTrue: [primeraletra asUpperCase]
                                ifFalse: [primeraletra := primeraletra]).
        ].
^arreglo.
```

con lo que evaluando se tendrá por resultado convertir las primeras letras de cada elemento del arreglo de nombres, en mayúsculas:

```
#('Brenda' 'Elisa' 'Solis' 'Perez').
```

En los ejemplos anteriores tenemos solamente evaluaciones simples, por medio de operadores binarios u objetos testing. En Smalltalk, al igual que en muchos otros lenguajes, se cuenta también con

evaluaciones compuestas en base a expresiones booleanas. Para hacer esto podemos auxiliarnos de los mensajes and: (mensaje 'y') y or: (mensaje 'o'). Su estructura general es la siguiente:

(expresion1 and: [expresion2])
(expresion1 or: [expresion2])

donde el objeto receptor es la expresión1 y el argumento del mensaje selector (or: o and:) es la expresión2 encerrada entre corchetes.

Como podemos notar, las estructuras de control en Smalltalk se basan en "bloques de instrucciones" (program block) que realizan ciertas operaciones. Estos bloques de instrucciones contienen piezas de código ejecutable, y se delimitan por corchetes.

[...]

En Smalltalk, un bloque de instrucciones se utiliza como el argumento de un mensaje. Este bloque no necesariamente ocupa espacio en memoria como lo hace un objeto; esto es que un bloque de instrucciones no ocupa memoria hasta que no es evaluado, además de que una vez ejecutado, se desaloja de ella.

La ventaja de Smalltalk con respecto a los bloques de instrucciones, es que pueden ser asignados a variables. En los lenguajes tradicionales como C o Pascal los bloques de instrucciones no tienen una equivalencia exacta con Smalltalk. Todos esos lenguajes utilizan el término "bloque" para representar solamente a un segmento de código, mientras que Smalltalk lo utiliza como un dato en particular.

Los bloques de instrucciones pueden manejar argumentos dentro de ellos, que son variables contenidas en bloques (block arguments), las cuales deben ir precedidas por dos puntos (:.) Estos argumentos son como una especie de variables temporales en pedazos de código (program temporaries), pero que no necesitan ser declarados. Su estructura general es la siguiente:

[:argumentoDelBloque | ... instrucciones ...]

en donde las instrucciones o también llamado "texto del programa" (program text) debe ir separado de los argumentos, por una barra vertical (|).

Los bloques de instrucciones, generalmente son utilizados por los iteradores. Los iteradores como su nombre lo indica, iteran sobre ciertas instrucciones, basados en una condición. Existen dos tipos de iteradores: los iteradores simples y los generalizados.

Los iteradores simples como: timesRepeat:, whileTrue:, whileFalse: y to: do:, entre otros, son mensajes de ciclos (looping messages) que se basan en bloques de instrucciones para iterar. Este último mensaje usa un incremento de uno en uno, pero podemos especificar nuestro propio incremento utilizando el mensaje to: by: do:, como por ejemplo:

1/2 to: 7/2 by: 1/2 do: [...].

Los iteradores generalizados son similares a los ciclos (loops) que contienen un objeto que

funciona como un apuntador. Este objeto apunta a un elemento de la colección a la vez, avanzando en cada iteración hacia el siguiente elemento.

Smalltalk provee cuatro diferentes tipos de iteradores generalizados, los cuales trabajan en conjunto con los bloques de instrucciones: do:, select:, reject: y collect:.

El más simple de ellos es el iterador do:, que maneja como argumento a un bloque de instrucciones, pues el mismo es un mensaje. Su estructura general es la siguiente:

```
objeto receptor do: [:argumento| ...instrucciones ...]
```

en donde el objeto receptor puede ser una cadena de caracteres o un arreglo. El método do: es quien maneja los valores del argumento e invoca el código dentro del bloque.

Específicamente éste iterador, va en ciclo a través de una colección (cadena o arreglo) y pasa cada elemento de ella como argumento dentro del bloque. Por ejemplo:

```
"Cuenta letras mayusculas
de un arreglo de caracteres"
| mayusculas cad |
mayusculas := 0.
cad := 'Cadena de Caracteres'.
cad do: [:elem |
                elem isUpperCase
                        ifTrue: [mayusculas := mayusculas + 1 ]
        ].
^mayusculas
```

En los bloques de instrucción de todos los iteradores generalizados, debe aparecer solamente un argumento que manipule la operación.

Este argumento actúa similar a una variable local en los lenguajes procedurales, excepto que aquí el valor del argumento proviene de afuera del bloque; esto es que fuera del bloque se dan los valores que pasarán a procesarse dentro de él, por medio del argumento del bloque. Podemos interpretar entonces a un bloque como una especie de función en cualquier otro lenguaje, pero sin un nombre específico.

Otro ejemplo del iterador do:, pero con arreglos se muestra a continuación:

```
"multiplicacion de cada elemento
 del arreglo, por un numero dado"
| arreglo numero i |
numero := 3.
i := 0.
arreglo := #(3 8 12 1).
arreglo do: [:elem|
                arreglo at: i
                        put:
                                (elem * numero).
                i := i+1.].
^arreglo
```

de donde se tendrá por resultado el arreglo #(9 24 36 3).

El mensaje select: al igual que do: es otro de los iteradores, el cual tiene como argumento a un bloque de instrucciones. Este bloque a su vez recibe un argumento. El resultado que nos retorna éste iterador es un conjunto de miembros para los que la evaluación dentro del bloque resultó verdadera. De hecho select: se considera como uno de los iteradores más poderosos con los que cuenta Smalltalk.

Para entender el funcionamiento de select: a continuación se ilustra un ejemplo muy sencillo de su utilización:

```
#(1 2 3 4 5 6 7) select: [:n |
                          n odd]
```

en donde formará un arreglo de todos lo elementos que resultaron ser impares (odd):

(1 3 5 7).

Otro de los iteradores generalizados es el mensaje reject:, el cual contrariamente a select: trabaja sobre los elementos que resulten falsos de la evaluación. Si a el ejemplo anterior, cambiamos la palabra select por reject, obtendremos por resultado el arreglo (2 4 6).

El iterador collect: por último, es un mensaje que evalúa los elementos de un arreglo o cadena, y regresa por resultado un arreglo que contendrá los valores que resultaron de evaluar cada elemento. Por ejemplo, podemos simplificar el ejemplo del iterador do:, con éste nuevo iterador:

```
"multiplicacion de cada elemento
 del arreglo, por un numero dado"
| arreglo numero |
numero := 3.
arreglo := #(3 8 12 1).
arreglo collect: [:elem|
                          elem * numero ].
```

y si evaluamos lo anterior con 'show it', obtendremos por resultado al arreglo (9 24 36 3).

2.4. Qué son las Clases y su forma de utilización

Generalmente los sistemas orientados a objetos hacen una distinción entre la descripción del objeto, y el objeto por si mismo. Varios objetos con características similares pueden ser creados por una descripción general.

Una clase describe la implementación de un conjunto de objetos que representan los mismos componentes de un sistema. Los objetos individuales descritos por una clase son conocidos como instancias. Todo objeto es una instancia de una clase; por ejemplo el arreglo:

#('arreglo' 'de' 'cadenas')

es una instancia de la clase "Array".

A diferencia de lenguajes de programación orientada a objetos como C++ y Pascal, en donde se hace una declaración de cierta clase para generar un objeto (instancia de la clase), en Smalltalk las clases son por ellas mismas "objetos". Esto es que en Smalltalk todas las clases son instancias de la clase llamada "class" de la librería de clases de Smalltalk.

Podemos entonces, usar a las clases en las expresiones de mensajes, pues como objetos pueden responder a los mensajes. Como por ejemplo, al crear una variable tipo Pen:

```
Pen new
```

la clase Pen se encuentra dentro del mensaje, y actúa como el objeto receptor. Si nosotros evaluamos la palabra Pen, con la opción "show it", aparecerá el mensaje:

```
a Pen
```

que nos indica que todo objeto se puede desplegar a sí mismo.

La solución de desplegarse los objetos por si mismos, nos muestra las ventajas de extensibilidad de Smalltalk. En éste lenguaje, todo objeto podrá responder siempre al mensaje "print" por medio de la opción "show it", retornando una cadena de caracteres.

Como ya se ha dicho, en Smalltalk las variables y los objetos se comportan de forma dinámica, es decir, no necesitan ser declarados con un tipo específico de información. Y es en el tiempo de ejecución de un programa, cuando se cuestiona su clase. La desventaja de los objetos dinámicos, es la cantidad de memoria requerida para retener la información del tipo de dato y la jerarquía entre clases.

Muchas veces, resulta de vital importancia saber la clase a la que pertenece un objeto, por lo que existe un mensaje que nos proporciona dicha función: species (tipo, especie). Podemos así, cuestionar el tipo de dato de cualquier objeto, mediante la opción "show it" la cual nos devolverá la clase a la que pertenece, como se muestra a continuación:

```
Turtle species.      ==>> Pen
(5 @ 5 corner: 15 @ 25) species. ==>> Rectangle
999 species.         ==>> SmallInteger
Transcript species.  ==>> TextEditor
```

De la misma manera, el mensaje class permite al objeto conocer a que clase pertenece. Si en los anteriores ejemplos cambiamos la palabra species por class, el resultado será exactamente el mismo.

Los objetos conocen su especie o clase a la que pertenecen, debido a que retienen información acerca de su tipo, después de que son creados. En los lenguajes donde se requiere una previa declaración del tipo de dato, la especie no está disponible en el tiempo de ejecución de un programa.

Por otra parte, para poder realmente sacar partido a un lenguaje orientado a objetos, es

fundamental el uso de las Librerías de Clase (class libraries) que provee el lenguaje. Las librerías de clase no son otra cosa sino clases definidas que se nos presentan como herramienta para la reutilización de código; ésta es la principal ventaja de programar con las librerías de clase, pues fácilmente podemos heredar código ya desarrollado y depurado de antemano.

En Smalltalk/V es posible modificar dichas librerías, de acuerdo a nuestras propias necesidades. Por ejemplo, puede darse el caso en el que deseemos utilizar cierta librería que no cumple en un 100% con nuestros requerimientos; entonces podremos modificar dicha clase de acuerdo a lo que se desee, para así no tener un consumo innecesario de memoria. Muchas veces en un lenguaje procedural por ejemplo, al correr un programa, el código que no se utilizará de una librería estándar, se liga en el programa ejecutable, consumiéndose así más memoria de la que podríamos ocupar.

Las librerías de clase que provee cualquier lenguaje orientado a objetos, se conocen como "Librerías de Clase Generales" (general class libraries). Las librerías de clase generales también llamadas Librerías de utilidades o herramientas (utility libraries), son las librerías que provee el sistema en adición con las que el usuario crea para su uso en general.

En Smalltalk, se menciona que existe una diferencia entre las clases definidas por el sistema, y las definidas por el usuario. Esta distinción radica simplemente en que las clases definidas por el usuario, fueron creadas por el mismo. Pero la diferencia es relativa. Esto es que, la diferencia solamente la conoce el programador (usuario), pues el sistema integrará en un mismo ambiente a todas las clases (en una misma librería de clases) para su utilización.

Las librerías de clase que nos proporciona específicamente Smalltalk/V son llamadas "Librerías de Clase de Envase" (container class libraries). Para poder operar el sistema, en la creación de aplicaciones, es necesario saber manejar las librerías de clase de envase.

La mayoría de las librarías de clase de envase de Smalltalk/V, se basan en el diseño original que propone Smalltalk-80. En seguida en la figura 2.1 se muestran las librerías de clase de Smalltalk/V y en la figura 2.2 las definidas en una versión anterior (Smalltalk-80).

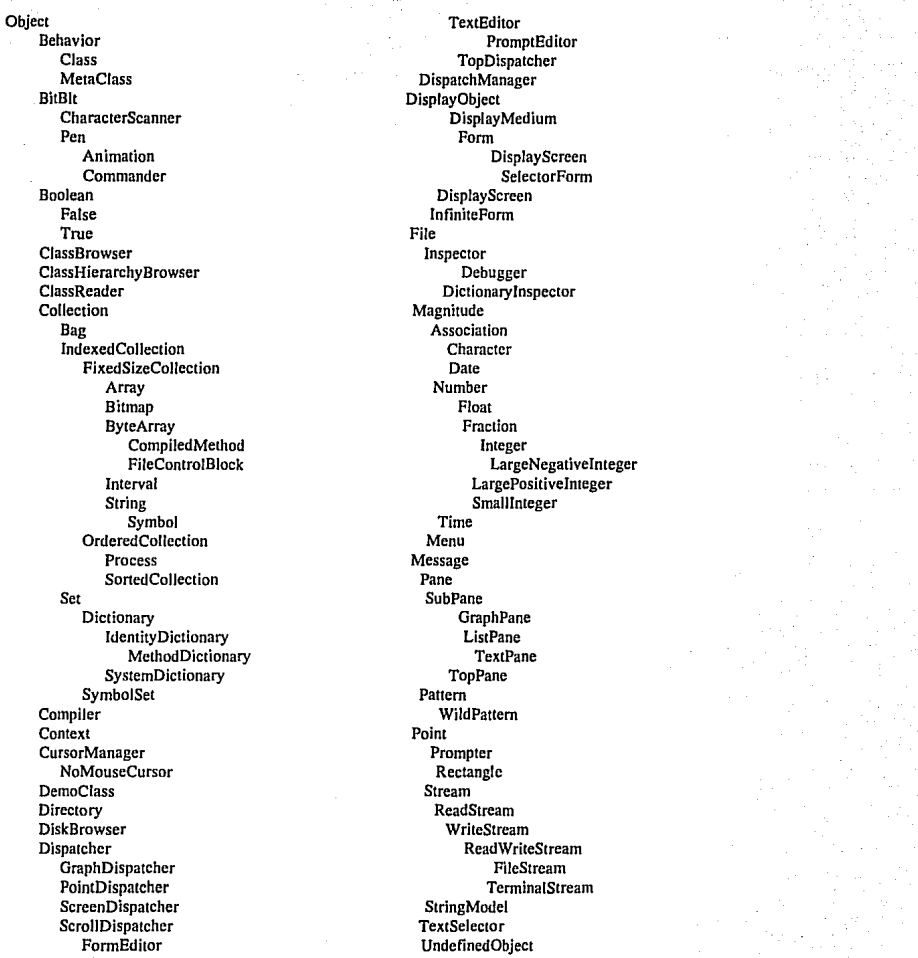


Figura 2.1 Clases de Smalltalk/V

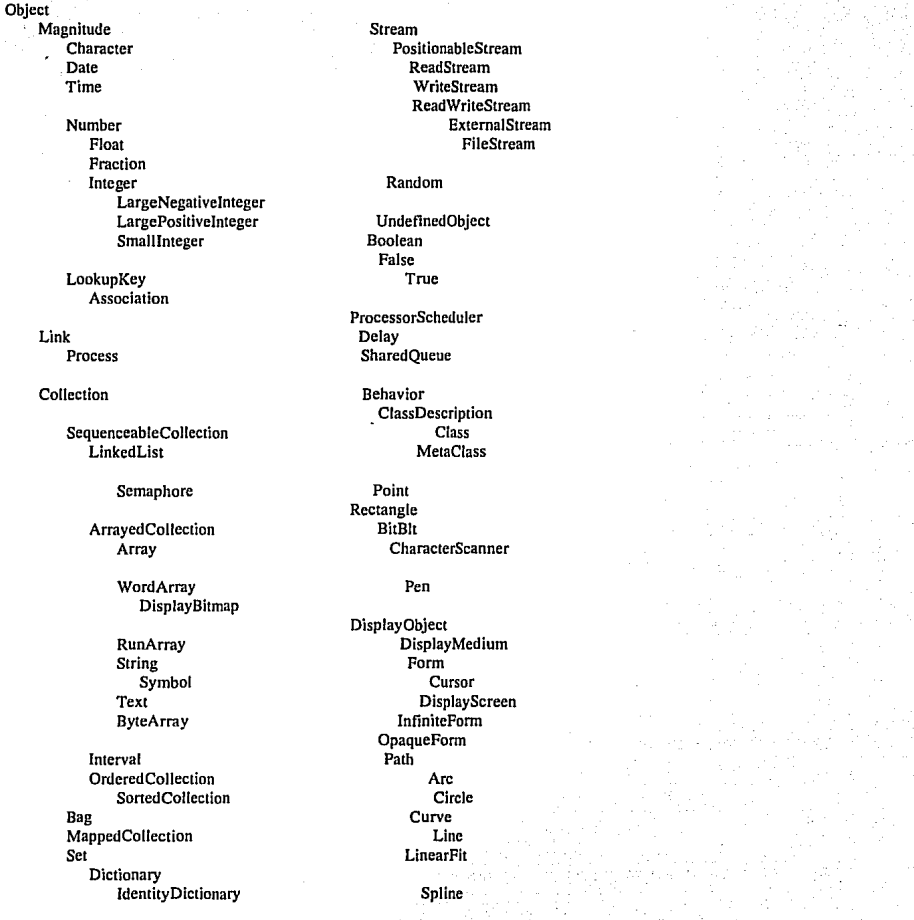


Figura 2.2 Clases de Smalltalk-80

Aunque la variación de clases es poca, las versiones 80 y /V de Smalltalk se consideran en cierta forma incompatibles, pues algunos de los comandos de Smalltalk-80 cambian su estructura en Smalltalk/V; además de que la herencia no proviene de una jerarquización idéntica.

Podemos recalcar que muchos aspectos del lenguaje, son lo mismo para todas sus versiones, sin embargo en particularidades no, por lo que en algunas partes de éste trabajo nos referimos al lenguaje que estamos analizando (Smalltalk/V) en forma particular.

Cabe mencionar que muchos de los lenguajes más populares de orientación a objetos, están influenciados por las librerías de clase de envase de Smalltalk. Por ejemplo, mientras que Turbo C++ y Zortech's C++, adquieren ciertas ideas de él, los sistemas C++ Views (para Microsoft Windows) y Actor, toman directamente la librería de Smalltalk-80.

Para poder entender la función principal de las clases de envase de Smalltalk/V, es necesario analizarlas. A continuación estudiaremos algunas de las clases más importantes y de mayor apoyo para la programación en éste lenguaje.

Comencemos por analizar una de las clases de envase más sencilla de entender de Smalltalk/V: Magnitude.

Magnitude es una de las clases más frecuentemente usada en el lenguaje. De ella se desprenden objetos como caracteres, tiempos, fechas y números.

La mayoría de sus métodos definen comparaciones, conteos y ordenaciones, basados en los operadores aritméticos y relacionales que ya conocemos.

La subclase de Magnitude, Character (caracter), hace referencia a objetos tipo caracter por medio de su respectivo valor ASCII (entre 0 y 255) o a través de una literal. Los métodos de Character definen las comparaciones y ordenaciones posibles en su clase. Contamos con 256 instancias de la clase Character en el sistema, cada uno de ellos se asocia con un código ASCII. Podemos expresar éste tipo de objetos precediendo el caracter por un signo de pesos ($).

Por otro lado, los tiempos y fechas (Time y Date) también dentro de Magnitude, representan datos específicos dentro del calendario y tiempo que conocemos. Podemos por ejemplo obtener el tiempo y día actual,

```
Time now.
Date today.
```

o cambiarlos:

```
'11 October 1993' asDate
```

con los meses en idioma inglés y comenzando con letra mayúscula. Con esto y las variables globales dentro de Smalltalk, podemos por ejemplo crear un día especial:

```
DiaEspecial := Date new.
DiaEspecial := '26 August 1970'.
```

y calcular el día de la semana en que cayó esa fecha:

```
DiaEspecial dayName  ==>>  Saturday
```

o comparar fechas:

```
DiaEspecial > Date today  ==>>  false.
```

Ahora bien, con respecto a la clase que maneja números (Number), Smalltalk soporta 3 tipos de números: enteros (Integer), de punto flotante (Float) y racionales (Fraction). Las operaciones que podemos efectuar con cada uno de ellos, especialmente son de tipo aritmético y funciones numéricas. Entre las funciones numéricas encontramos la exponencial (exp), seno (sin), coseno (cos), tangente (tan), arcoseno (arcSin), truncar un valor (truncated), raíz cuadrada (sqrt), valor absoluto (abs), valor inverso (reciprocal), etc. Podemos también cuestionar el estado positivo o negativo de un número:

```
-3 positive  ==>>  false
-3 negative  ==>>  true.
```

o negarlo:

```
-3 negated  ==>>  3
```

para obtener así su valor contrapuesto en la recta numérica.

Un número de punto flotante (Float) es un número real con 18 dígitos de precisión y se representa entre el rango:

$$- 4.19\ e^{-307} \quad a \quad + 4.19\ e^{-307}.$$

Por otro lado, los números enteros son instancias de la clase Integer definidos como negativos largos(LargeNegativeInteger), positivos largos (LargePositiveInteger) o enteros pequeños(SmallInteger). Frecuentemente se utilizan para conteos o como índices de arreglos y colecciones. Los números enteros pequeños se encuentran en el rango:

$$- 16\ 384 \quad a \quad + 16\ 383$$

mientras que las clases de enteros largos, ya sea positivos o negativos, se representan con 64 Kilobytes de precisión.

Los números racionales se encuentran como instancias de la clase Fraction, en donde se tienen dos variables de instancia representando a números enteros como numerador y denominador. El mensaje que se utiliza para crear un número fraccionario, es el slash (/).

Por otra parte, una de las clases que permite la manipulación de colecciones y conjuntos, es la llamada "Collection" (colección). Una colección representa a un grupo de objetos. Estos objetos son llamados elementos de la colección. La clase Collection se encuentra dentro de la librería de clases de Smalltalk/V. Véase figura 2.1. En ella podemos encontrar formas simples de colecciones con distintas

funciones, como las "Bag" (bolsa) y "Set" (conjunto) por ejemplo.

Al igual que Set, la forma Bag permite la creación de una colección, en donde no se garantiza el orden de sus elementos (unordered collections). Para adherir elementos a cualquier tipo de colección, se utiliza el mismo protocolo, mediante el mensaje add:. Este mensaje se envía a la colección, teniendo como argumento a un tipo de objeto como son los Puntos, Rectángulos, Caracteres, Cadenas, Ventanas, etc.

Por ejemplo, crearemos una bolsa Bag en donde todos sus elementos sean cadenas de caracteres:

```
"declaración de variable temporal"
        | coleccionBag |
"creando la nueva coleccion de Bag"
        coleccionBag := Bag new.
"sumando elementos a la coleccion"
        coleccionBag add: 'elemento1'.
        coleccionBag add: 'elemento1'.
        coleccionBag add: 'elemento2'.
        coleccionBag add: 'elemento3'.
        coleccionBag add: 'elemento4'.
"retornado el valor de la coleccion Bag"
        ^coleccionBag
```

como podemos observar se produce un nuevo objeto Bag y se asigna a la variable coleccionBag para hacer referencia al nuevo objeto. Recordemos que en el ambiente de programación Smalltalk, las clases son objetos, por lo que la variable global "Bag" representa a la clase y al objeto Bag. Dentro del programa anterior, en la subexpresión:

```
Bag new.
```

se envía el mensaje new a la variable global Bag (que ya estaba creada por el sistema). Este mensaje unario pertenece a la lista de métodos de la clase Bag. La salida de éste fragmento de código es la siguiente (evaluando con "show it"):

```
Bag('elemento4' 'elemento3' 'elemento2' 'elemento1' 'elemento1')
```

en donde el orden se determina por un algoritmo implementado por el lenguaje Smalltalk, que ayuda a encontrar los elementos de una colección que eventualmente son distribuidos, para un propósito de búsqueda rápida (Algoritmo de Hash). En nuestro trabajo de investigación no es necesario conocer como funciona dicho algoritmo, pues solo nos abocaremos a lo que al lenguaje y sistema Smalltalk se refiere, y no a su forma de organización interna.

La colección Set es similar a Bag, en el sentido de que no se garantiza el orden de sus elementos. La diferencia entre Bag y Set es que éste último no duplica a los elementos que contiene. Si al ejemplo anterior aplicamos un Set en lugar de Bag (sustituyendo la variable global Bag por Set), obtendremos el siguiente orden:

```
Set('elemento1' 'elemento2' 'elemento3' 'elemento4')
```

y así podemos observar que el orden de los elementos, además de la no duplicidad de un tipo de objeto, difiere de Bag. El orden no se nos garantiza que sea el mismo, aún cuando volviésemos a evaluar la misma expresión, es decir, no existe un orden definido para las colecciones Set y Bag.

Muchas veces es necesario la creación de colecciones ordenadas, en donde se garantice el orden del conjunto, de acuerdo a cierto criterio. Por ejemplo en una colección de puntos para formar cierto trazo definido, es necesario mantener cierto orden en los elementos, pues de lo contrario podríamos obtener una figura diferente a la deseada. Smalltalk/V provee una forma de colección que nos facilita éstas características: OrderedCollection (colección ordenada). Este tipo de colección mantiene los elementos en el orden en el que fueron adheridos, y actúa con técnicas similares a Set y a Bag, en su creación.

La figura 2.3 nos muestra el uso de una colección ordenada (OrderedCollection) para dibujar líneas entre puntos con el objeto Pen, en donde el orden de los elementos, debe ser fundamental para el trazo deseado; aquí la figura debe ser dibujada sin levantar la pluma, ni pasar dos veces por la misma línea.

Ahora bien, si se desea ordenar de acuerdo a cierto criterio de comparación, podemos usar la clase derivada de OrderedCollection, SortedCollection(clasificación de la colección), la cual actúa sobre cadenas y números en base a comparaciones lexicográficas[3] y de menor a mayor cantidad.

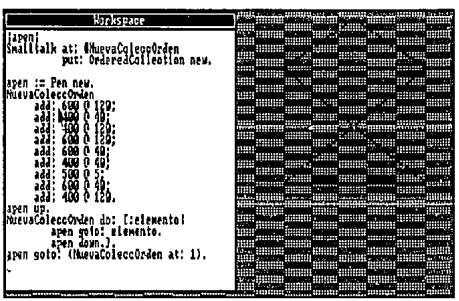


Figura 2.3 El orden de los puntos es fundamental para la solución a éste trazo, derivado de la gráfica llamada "la firma del diablo" en Teoría de Gráficas

También podemos cambiar el criterio de ordenación, basándonos en los llamados bloques de clasificación (sort block), que son pedazos de código del programa que sirven para comparar y ordenar los elementos, y los cuales reciben dos argumentos y retornan el valor de verdadero (true) o falso (false).

Ejemplificando lo anterior en la figura 2.4, podemos tener una colección ordenada en donde se ordena de mayor a menor. Como el valor por omisión dado a ordenar es el de menor a mayor, el bloque de clasificación se puede imaginar como el siguiente:

[:a :b | a <=b].

[3] **En el caso de los caracteres, por orden alfabético.**

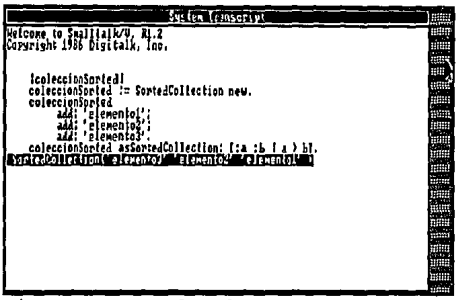


Figura 2.4 Orden de mayor a menor en una colección Sorted

En Smalltalk también contamos con los llamados Diccionarios (Dictionary), que se aplican a bases de datos o tablas de símbolos del compilador. Los diccionarios funcionan a través de asociaciones por medio de claves (key) y valores (value). Las claves son las referencias que tendremos para buscar algún valor de definición. Estas pueden ser cualquier tipo de objeto en Smalltalk, como por ejemplo un Número, Ventana, Cadena de Caracteres, Rectángulo o Punto. Las claves no pueden estar duplicadas - Recordemos que los diccionarios se derivan de la clase Set, donde no deben repetirse dos claves, además de que el orden no se garantiza- pues de lo contrario, si se trata de sumar una nueva clave con código existente, ésta será borrada y reemplazada por el nuevo valor.

Para introducir los datos al diccionario que creará el programador, es necesario utilizar un Mensaje keyword que contiene dos palabras clave (at: put:), cada una con un argumento:

NombreDelDiccionario at: '...clave...' put: '...valor...'.

el mensaje at: se asocia con la clave, mientras que put: con el valor. En el siguiente ejemplo (figura 2.5) podemos ver la creación de un diccionario que contiene una agenda telefónica.

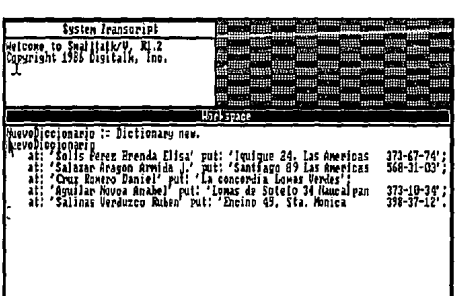


Figura 2.5 Colección Dictionary

Para poder hacer parte permanente del ambiente a un diccionario, es necesario la declaración de una variable global que contenga la definición de un nuevo diccionario, como se muestra en la figura anterior. Ahora bien, podemos evitarnos la pregunta que el sistema nos hace con respecto a si se desea crear una nueva variable global, usando el mensaje at: put:

```
Smalltalk at: #NombreDelDiccionario put: Dictionary new.
```

que define una variable global, en donde 'Smalltalk' es un diccionario del sistema. Como se mencionó en el Capítulo I, Smalltalk/V provee un diccionario conocido como "Diccionario de Métodos" (method dictionaries), que archiva todas las variables globales en el sistema y todas las clases. Este diccionario de métodos se creó como una variable global llamada "Smalltalk", la cual a su vez usa diccionarios para cada clase en el ambiente, que archivan el código objeto de los métodos compilados.

El diccionario Smalltalk contiene una serie de símbolos que representan las claves de un diccionario. Estos símbolos pueden ser considerados como cadenas que están precedidas por el signo "#".

Ahora bien, si anteriormente creamos la variable global NuevoDiccionario, entonces ésta se encontrará en Smalltalk como:

```
#NuevoDiccionario
```

así como las demás variables globales que hayamos creado (además de las que ya tenía definido el sistema).

Al utilizar la técnica de creación de variables globales, el consumo de memoria se va haciendo cada vez más grande. Este problema podemos resolverlo mediante el mensaje removeKey: que se basa en el diccionario especial 'Smalltalk' para remover de la memoria la clave deseada.

```
Smalltalk removeKey: #NuevoDiccionario.
```

De hecho el tiempo de vida de una variable global se termina cuando explícitamente se borra del ambiente.

Por otro lado, para poder desplegar los valores de cualquier diccionario, bastará con marcar el nombre del diccionario y evaluar con la opción "show it", en donde se desplegarán todos los valores contenidos en él. Si se desea buscar específicamente un valor bastará con hacerlo de la siguiente forma:

```
NombreDelDiccionario at: '...clave específica...'
```

Contrario a lo anterior, podemos buscar la clave por medio de un valor específico:

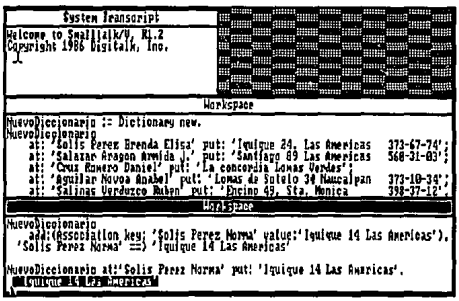
```
NombreDelDiccionario keyAtValue: '...valor específico...'
```

Como hemos visto, los diccionarios son formas de asociaciones, con dos llaves. En ellos también podemos aplicar el mensaje add:, pero creando asociaciones directamente:

```
NombreDelDiccionario add: (Association key:'...clave...' value:'...valor...')
```

Aquí la principal diferencia entre el mensaje add: y at: put: radica en que al evaluar la expresión anterior

la asociación aparece completa, mientras que con at: put: solamente aparecerá el valor. Para ejemplificar esto véase la figura 2.6.



Si se desea, se pueden imprimir todas las claves por medio de:

NombreDelDiccionario keys

o todos los valores:

NombreDelDiccionario values.

Si en el diccionario creado existieran dos valores iguales, al evaluar la expresión anterior, se tendría por resultado una colección Bag (Recordemos que Bag permite la duplicidad de elementos).

Por otra parte, al igual que el diccionario 'Smalltalk', existe otro diccionario dentro de él que provee el sistema: CharacterConstants (caracteres constantes). Este diccionario contiene todas las variables constantes (globales) de caracteres especiales como tabuladores (Tab), espacios (Space), retorno de carro (Cr), etc. Y se le conoce como diccionario tipo pool (alberca), pues está contenido como diccionario dentro del diccionario Smalltalk.

Ahora, con lo visto anteriormente, podemos cuestionar el contenido de CharacterConstants, así como de todas las variables globales existentes (ver figura 2.7).

Si se desea saber el tamaño de un diccionario, bastará con pedirlo de la siguiente forma:

NombreDelDiccionario size.

Como habremos notado, los tipos de colecciones anteriores no tienen un tamaño específico en su estructura, es decir, pueden ser tan grandes o pequeños como queramos, sumándoles elementos de una forma "dinámica". Los arreglos (clase Array) contrariamente a lo anterior, actúan de una forma estática, es decir, un arreglo debe definir su tamaño al momento de ser creado:

arregloNuevo := Array new: 8

en ésta expresión se define como 8 el tamaño del arreglo. De ésta forma, el índice que hace referencia al arreglo tomará valores mayores que 0 y menor o igual que la dimensión del arreglo (en éste caso 8).

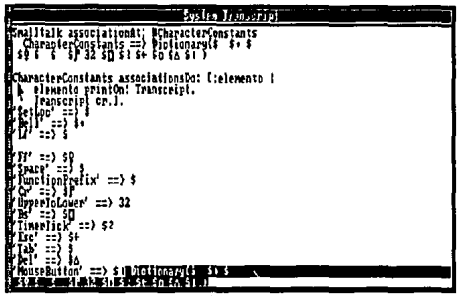


Figura 2.7 La variable global Smalltalk

Las colecciones en Smalltalk no comienzan con el elemento 0, como sucede en el lenguaje C con los arreglos (Array). Al igual que el lenguaje Ada, Smalltalk cuenta con el elemento número uno como inicio de la colección.

Cuando creamos un arreglo con un número definido de elementos, éstos se inicializan automáticamente al objeto "nil" (nulo). El objeto nil es una pseudo-variable que se refiere a un objeto usado como el valor nulo, y es la única instancia de la clase UndefinedObject. De hecho todos los objetos nuevos se asignan a éste objeto.

Volviendo a los arreglos, para introducir elementos al arreglo debemos utilizar el mensaje at: put:, pues debemos contemplar el índice del arreglo para colocar el elemento en la posición deseada, por lo que es lógico pensar que no podemos introducir elementos con el mensaje add:. Por ejemplo, para crear el siguiente arreglo:

('nombre' 'direccion' 'telefono')

una forma de introducir los elementos es la siguiente:

```
|nombreDelArreglo|
nombreDelArreglo := Array new: 3.
nombreDelArreglo at: 2 put: 'direccion'.
nombreDelArreglo at: 3 put: 'telefono'.
nombreDelArreglo at: 1 put: 'nombre'.
```

en donde el orden de entrada de los elementos depende de los requerimientos del programador. Así pues, podemos crear objetos Array mediante el mensaje new: con un argumento entero.

Como ya hemos visto con los mensajes Keyword, los arreglos también pueden ser creados en forma temporal como expresiones a evaluar. Esto se hace mediante el uso del símbolo # antepuesto a los paréntesis que delimitarán al arreglo:

#('elemento1' 10 @ 10 corner: 24 @ 3 'AB' 2 @ 2 1).

en éste caso los elementos del arreglo son una cadena, un rectángulo, otra cadena, un punto y un número entero.

Como se puede observar en la expresión anterior, una de las principales ventajas de los arreglos en Smalltalk, es que soportan polimorfismo, es decir, podemos tener en un mismo arreglo, diferentes tipos de objetos a la vez. Posiblemente ahora nos suene un tanto lejano éste concepto de "polimorfismo", pero más adelante, en el capítulo III, será analizado con más detalle.

Por ahora, es importante entender el uso de las colecciones y saber como aplicarlas a ciertos problemas.

Una herramienta que Smalltalk nos proporciona para construcción de estructuras de datos más complejas, es la de colecciones dentro de colecciones (collections of collections). Quizás no hallamos notado que hemos estado utilizando colecciones dentro de colecciones; por ejemplo en la siguiente expresión:

```
#('cadenas,' 'cadenas' 'y más cadenas')
```

se tiene una colección "arreglo" que contiene colecciones de cadenas (Strings).

Por otra parte, en Smalltalk/V también podemos convertir colecciones definidas de un tipo, a otro tipo diferente, e incluso a su vez, a varios tipos más. Por ejemplo podemos convertir un diccionario (Dictionary) a un conjunto ordenado (OrderedCollection) y a su vez éste, a una colección Set. Esto es posible a través de los siguientes mensajes: asBag, asSet, asDictionary, asSortedCollection, asOrederedCollection, y asArray, que se envían al nombre de la correspondiente colección creada. Ver figura 2.8.

El ambiente Smalltalk como podemos observar, nos proporciona muchas bondades en las herramientas de programación. Por ejemplo, las ocurrencias de un elemento de cualquier colección, pueden contarse mediante el mensaje occurrencesOf: que se aplica sobre un elemento conocido de la colección, para saber cuantas veces aparece en el conjunto.

En la figura 2.9 se muestra el uso de éste mensaje, conjuntamente con el iterador select:, en

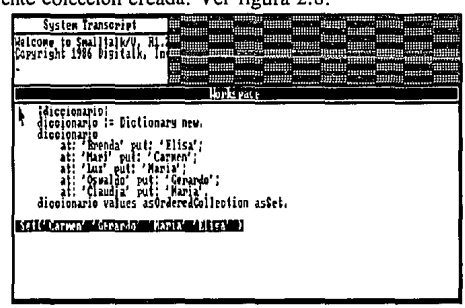


Figura 2.8 Aplicando dos conversiones sobre un diccionario temporal

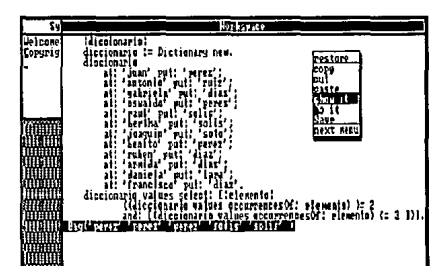


Figura 2.9 Con el código mostrado obtendremos los elementos de un diccionario que se repiten entre 2 y 3 veces, en los valores de un diccionario

donde obtendremos por resultado una colección Bag. Recordemos que select: nos retornará un conjunto de elementos que resulten verdaderos a la evaluación.

Ahora bien, por medio de la ventana del browser de clases, podemos observar las diferentes subclases que se derivan de la clase Collection. Encontramos que la clase String (cadenas) se encuentra al mismo nivel que la clase Array. Es importante saber que String es también una forma de colección, que manipula objetos tipo cadenas de caracteres.

Al mandar el mensaje at: a alguna cadena, con un argumento entero, tendremos por resultado un caracter.

'cadenas' at: 7 ==>> $s

En ésta expresión el índice se refiere al séptimo elemento de la colección String, que tendrá por resultado el caracter $s. Recordemos que el signo de pesos siempre debe anteponerse a cualquier caracter.

La ventana del Browser de Clases puede abrirse por medio del Menú del Sistema (System Menu), con la opción "browse classes" en donde aparecerá un Browser con todas las clases existentes en el ambiente.

Otra forma de invocar a la ventana Browser de Clases, es a través de la evaluación de cierta expresión, cuya estructura general es la siguiente:

ClassHierarchyBrowser new openOn:(Array with: Clase).

Podemos utilizar ésta expresión para observar varias clases. Por ejemplo, podemos seleccionar las clases de magnitud, puntos y rectángulos, y evaluar con la opción "do it" (figura 2.10).

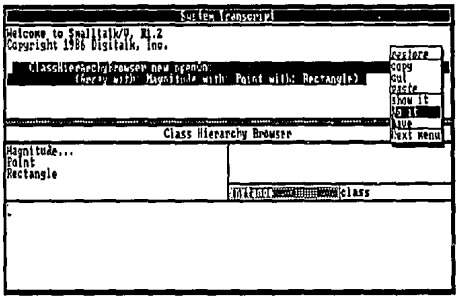


Figura 2.10 Mostrando clases específicas

Para hacer referencia a una clase, es necesario seleccionarla con el cursor, en el panel superior izquierdo. Por ejemplo, seleccionemos la clase Magnitude de la librería de clases de envase de Smalltalk/V. Ver figura 2.11.

Siempre que trabajemos con el Browser de clases, se desplegarán solamente los primeros niveles subclases de object (que es la superclase de todas las clases). Como vemos, podemos tener cierta clase que contenga a otras diferentes subclases; como en la figura anterior por ejemplo, en donde se tienen 3 puntos (Magnitude...) que nos indican que dicha clase contiene a otras subclases en ella.

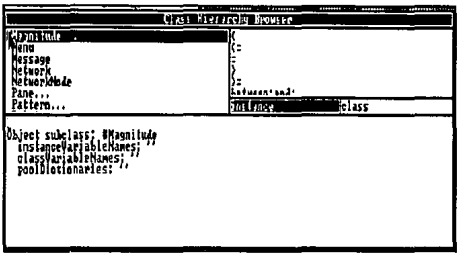


Figura 2.11 Clase seleccionada

Para poder manipular las subclases ocultas, es necesario seleccionar la clase deseada, invocar el menú de ese panel, y seleccionar la opción "hide/show" (mostrar lo oculto) que nos presentará su jerarquía de clases. Esto lo podemos observar en la figura 2.12, en donde ya fue seleccionada dicha opción.

Cuando seleccionamos una clase, aparecerá el código en el Panel de texto que indica las características de la clase. Ver Panel de texto de la figura 2.13. A éste panel también se le conoce como Mensaje de las definiciones de clase (class definition message).

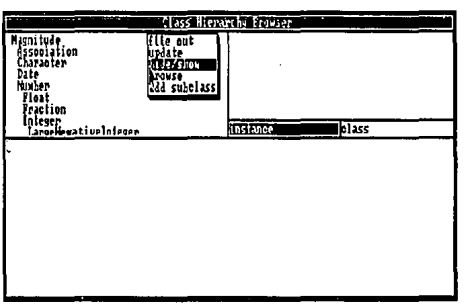


Figura 2.12 Jerarquía de la clase seleccionada

En la primera línea se indica la superclase de la clase seleccionada. En las siguientes, aparecen los mensajes de definición de variables de instancia, variables de clase y los diccionarios tipo alberca (pool).

Primeramente aparece el mensaje instanceVariableNames:, cuyo argumento es una cadena que representa a una variable miembro de la clase. Los objetos, cuentan con éstas variables internas que se conocen como "variables de instancia nombradas" (named instance variables), las cuales se accesan por medio de su nombre, y contienen objetos. El nombre de una variable de instancia siempre debe comenzar con una letra minúscula.

Por ejemplo veamos la clase Character (caracter) que se deriva de Magnitude (magnitud), en la figura 2.13. Como podemos observar, en el panel de texto del Browser de Clases, tenemos la definición de una sola variable de instancia que es asciiInteger (valor entero ASCII).

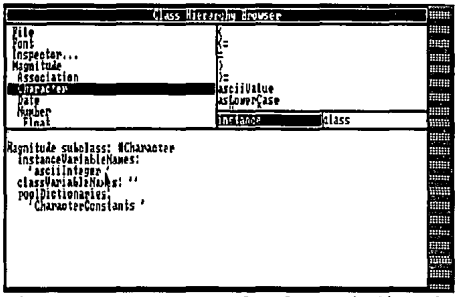


Figura 2.13 Browser de clases indicando a la clase Character

Para un objeto que representa a un caracter ^ por ejemplo, la variable de instancia asciiInteger contendrá al objeto 94, que es un entero que representa al número ASCII de ese caracter.

Por otro lado, existe otro tipo de variables de instancia, que son las indexadas (indexed instance variables). Estas variables se identifican por los números enteros comenzando desde el uno. Y se accesan únicamente vía mensajes, como por ejemplo:

```
'objeto cadena' copyFrom: 1 to: 6.
```

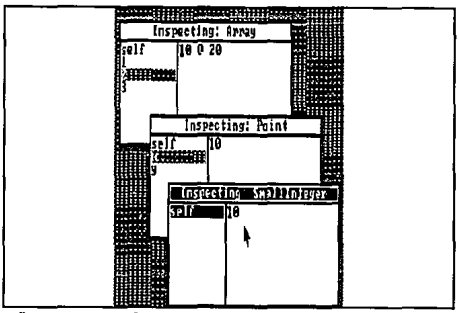
La diferencia que existe entre las nombradas y las indexadas, es que las primeras se especifican en la clase con un número fijo de ellas, y las otras no se especifican en la clase, y se definen al momento de crear el objeto. En la expresión anterior por ejemplo, el número de variables de instancia indexadas es el tamaño del arreglo (en éste caso 13).

Existe una herramienta en Smalltalk que nos permite visualizar las variables de instancia de un objeto: el uso de inspectores. Un inspector es una ventana que se define al momento de invocarlo por medio de código.

Por ejemplo, podemos inspeccionar un arreglo:

```
| arreglo |
arreglo := #(1 'dos' 3).
arreglo at: 1 put: 1/2.
arreglo at: 2 put: 10 @ 20.
arreglo at: 3 put:6.
arreglo inspect.
```

Al evaluar el código con 'do it', aparecerá una ventana de inspección; en el panel izquierdo se mostrarán los nombres de las variables de instancia nombradas, y los números de las variables de instancia indexadas.



Ilustr. 14

Como se muestra en la figura anterior, podemos aún inspeccionar sobre una ventana de inspección. La primera ventana que invocamos inspecciona el arreglo definido (inspect: Array), de donde seleccionamos inspeccionar el segundo elemento; éste elemento pertenece a un punto, por lo que aparecerá ahora, una ventana inspeccionando a un punto (inspect: Point). La clase Point tiene definidas dos variables de instancia, que son x y y. La x representa a la coordenada horizontal del punto, mientras que y representa el eje vertical. Como nosotros dimos el punto:

```
10 @ 20
```

entonces el valor de x es el entero 10 y y tiene un número 20. Por último, al inspeccionar la variable x, aparecerá una ventana que inspecciona un número entero pequeño (SmallInteger).

Un número compuesto por 4 dígitos, es considerado como un entero pequeño, mientras que un número con más dígitos es considerado largo, ya sea positivo (LargePositiveInteger) o negativo (LargeNegativeInteger).

Con respecto a los inspectores existe un tipo especial de inspector solamente para diccionarios (Dictionary Inspector). Podemos por ejemplo mirar y editar el contenido de un diccionario específico. Para ello, debemos escribir el nombre del diccionario, seguido por la palabra "inspect", y entonces se abrirá una ventana con dos paneles que nos mostrarán las claves y los valores; si una clave está marcada con el cursor, entonces en el panel derecho se imprimirá su valor.

Volviendo a las definiciones de la clase, el siguiente mensaje que aparece es classVariableNames:, que indica la definición de las variables de clase. Estas variables son variables globales que pueden ser manipuladas por todos los métodos de la clase, y deben estar definidas en el panel de definiciones de la clase. Por ejemplo, podemos utilizar una variable de clase en un método ya definido, y crear un nuevo método que retorne su valor:

```
nuevoMetodo
        ^VariableDeClase.
```

Por último (en las definiciones de clase) la última línea nos indica las variables poolDictionaries:. Una variable de éste tipo es una variable que comienza con una letra mayúscula, y se define en los diccionarios. Un diccionario de alberca (pool), es entonces una variable global que contiene a un diccionario (claves y valores), alojado en el diccionario Smalltalk. Por ejemplo CharacterConstants es una diccionario pool que asocia nombres de caracteres especiales ASCII. Veamos la siguiente expresión:

```
Cr
```

en donde la clave es Cr, y el valor es el valor ASCII 13 que indica retorno de carro. Los diccionarios pool contienen información que puede ser compartida por varias clases, como por ejemplo el diccionario pool CharacterConstants se comparte en varias de las clases de la librería en Smalltalk/V.

Pero a todo esto, ¿Cómo podemos crear nuestras propias clases?

La respuesta es muy sencilla, lo único que debemos hacer es invocar al browser de clases y elegir la opción **add subclass** del menú de opciones del panel de clases. Para ello debemos seleccionar la clase de la que deseamos heredar comportamiento y enseguida aparecerá un prompter preguntando el nombre de la nueva clase; de éste modo podemos añadir cuantas clases necesitemos, o modificar sus definiciones sobre el mismo browser de clases.

Por último, hablaremos sobre la clase Stream, que pertenece a la librería de envase de Smalltalk/V. La clase Stream es una clase que especifica el acceso a una colección, mediante una posición referente a cada elemento de la misma. Este acceso se concibe a través de los mensajes para leer o escribir dentro de la colección. Uno de los mensajes básicos de stream es next, que lee el siguiente elemento en una colección, mientras que nextPut: escribe el argumento en la colección.

La clase Stream también se utiliza para accesar archivos de disco, y de esta forma manejar entradas y salidas sobre ellos. Por ejemplo podemos manejar el nombre de la ruta de acceso de los archivos en disco:

```
File pathName: 'd:\Smaltalk\tesis2.ejm'
```

o podemos también tener un nombre parcial (incompleto) como ruta de acceso:

```
File pathName: 'tesis2.ejm'
```

en donde un objeto llamado Disk que es una variable global, completa la ruta suponiendo el directorio actual.

Por otro lado, también por medio de la especificación de archivos, podemos crear clases, aunque normalmente nosotros crearemos clases usando el Browser de Jerarquías de Clase (ClassHierarchyBrowser).

Dicho archivo, deberá tener la extensión CLS que nos indica una clase (CLasS). Lo primero que debemos hacer es sumar al ambiente el nuevo archivo, con la instrucción:

```
(File pathName:'nombrearch.CLS') fileIn
```

en donde se indica el nombre del archivo que será instalado en el ambiente, y podrá verse normalmente

a través del browser de clases.

Un ejemplo del tipo de código de un archivo que contiene las especificaciones de una clase y sus métodos se ve en el siguiente listado:

```
Object subclass: #Message
  instanceVariableNames:
        'selector arguments'
  classVariableNames: ''
  poolDictionaries: '' !

! Message class methods ! !

! Message methods !

arguments
        "Answer the arguments array for the message."
    ^arguments !

selector
        "Answer the message selector."
    ^selector ! !
```

que se refiere al archivo MESSAGE.CLS de la clase de envase Message, de donde se observan algunas notaciones ajenas a lo que hasta ahora hemos visto. Por ejemplo en el primer pedazo de código, se declaran normalmente las definiciones de la clase pero escribiendo al final un signo de admiración (!); éste signo de admiración se le conoce como "marca de pedazo" (chunk marks) y se utiliza para separar el código Smalltalk en secciones o pedazos. La primera marca de pedazo que aparece delimita las definiciones de clase, posteriormente se indica que se definirán los métodos de clase; pero si no existiera ninguno, entonces denotaríamos el término de éste bloque con otro signo de admiración. Una vez hecho esto, se procede a definir los métodos de instancia, separando cada uno nuevamente por un signo de admiración. Finalmente debe colocarse otro signo de admiración o marca de pedazo para indicar el final de las declaraciones.

Podemos observar entonces que en el listado anterior existen dos métodos, y no existen métodos de clase. Dichos tipos serán explicados a continuación en el siguiente subcapítulo que se refiere a el uso y tipos de métodos en Smalltalk.

2.5. El uso de Métodos

En los subcapítulos anteriores hemos aprendido los aspectos primordiales para el manejo del lenguaje Smalltalk, específicamente en Smalltalk/V. En éste subcapítulo estudiaremos la herramienta que hace que Smalltalk sea único en su género: el uso de métodos.

Como sabemos, un método describe la manera en que un objeto ejecutará una de sus operaciones.

Prácticamente los métodos son algoritmos que se ejecutan por un objeto que recibió un mensaje. Los detalles internos de la implementación del objeto se encuentran en los métodos, es decir, cada método en una clase nos dice como ejecutar cierta operación requerida por un mensaje.

Toda entidad en el ambiente Smalltalk, es un objeto, no existen funciones globales. Todas las funciones son procesos o métodos, que se asocian a un objeto.

Smalltalk clasifica a los objetos, de acuerdo a sus rasgos característicos. Hasta ahora hemos visto el comportamiento de los objetos, pero de forma externa; esto es, observando el paso de mensajes entre objetos y esperando cierto resultado. Para poder observar sus rasgos internos (código smalltalk) es necesario analizar algunos de los métodos disponibles para responder a los mensajes que se le envían.

Los métodos son código de Smalltalk para ejecutar cierta tarea, por lo que determinan la conducta del objeto. En los lenguajes procedurales, las funciones son similares a los métodos en donde se determina un proceso a seguir.

La relación que existe entre los mensajes y los métodos se refleja cuando un mensaje se envía a un objeto y un método es evaluado, teniendo por resultado a un objeto. Por ejemplo si evaluamos la siguiente expresión:

```
$B asciiValue  ==>>  66
```

estaremos pidiendo el valor ascci del caracter B, teniendo por resultado al objeto 66. Esto es porque cuando el mensaje asciiValue se envía al caracter B, Smalltalk ejecuta el método asciiValue definido en la clase Character.

Como hemos mencionado a lo largo de éste trabajo, la programación en Smalltalk/V se facilita con el uso de la ventana llamada Browser de Jerarquías de Clase (Class Hierarchy Browser), la cual nos permite observar y modificar código existente para las clases y las definiciones de los métodos, e incluso añadir nuevas definiciones.

Con la ayuda del Browser de clases podemos manejar el código fuente de un solo método a la vez, seleccionando el método que se desee, en el panel superior derecho. Al seleccionar uno de ellos, aparecerá en el panel de texto el código fuente correspondiente al método. Este panel se puede usar para modificar el método, pues el mismo panel es un editor de textos. Siguiendo con el ejemplo anterior, veamos la figura 2.14.

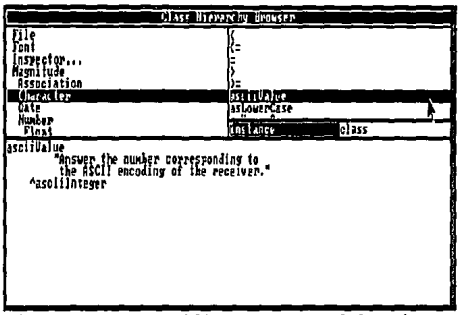


Figura 2.14 Código fuente del método asciiValue

En ella se muestra el código correspondiente al método asciiValue. La primera línea de código, define el nombre del método que fue invocado, el cual debe coincidir con el selector en el mensaje correspondiente ($B asciiValue). La siguiente línea se refiere a los comentarios que indican la acción que realiza el método. Después de ello, la expresión siguiente regresa el resultado ASCII (asciiInteger), que es la variable de instancia de la clase Character.

Podemos entonces deducir que un método consta de 3 partes: un nombre de mensaje, comentarios (opcionalmente) y expresiones Smalltalk.

Observemos ahora un método un poco más complejo, en la figura 2.15. En las especificaciones del método, se indica que se enviará el mensaje < a un objeto receptor, comparándolo con un caracter. En ésta misma figura, veamos la ventana Workspace que muestra un ejemplo de la aplicación del método. Aquí, se envía el mensaje < a el caracter 7, con el caracter A como el argumento (aCharacter) que evalúa el método. Observamos que el resultado es verdadero, pues el valor ASCII de 7 es 55, que es menor que 65 que representa al caracter A. El argumento aCharacter es usado en el proceso del método para representar un objeto argumento tipo caracter. El método retornará valor verdadero si el objeto receptor (en éste caso $7) es menor que el argumento ($A) en valor ASCII, de lo contrario regresa falso.

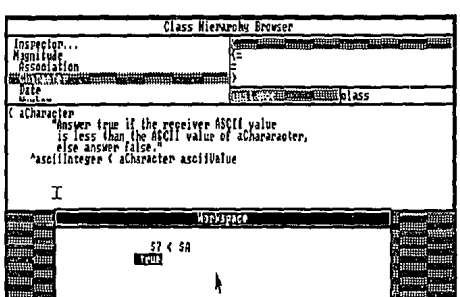


Figura 2.15 Código fuente del método <, y ejemplo de su uso

Así, podemos observar que las variables de instancia definidas en la clase (en éste caso asciiInteger, que es la variable de instancia de la clase Character), se utilizan en algunos de los métodos definidos en ella, para referenciar a los objetos receptor. Recordemos que un mensaje cuenta con un objeto que recibe

el mensaje (object receptor), un mensaje selector y su argumento correspondiente. En éste método específicamente (<), el argumento es otro objeto tipo caracter. Aún cuando éste objeto argumento pertenece a la misma clase que el objeto receptor, sus variables internas no están disponibles en éste método. Podemos hablar entonces, del principio de encapsulación, de donde Smalltalk nos brinda una completa seguridad de que no habrá una manipulación de datos no deseada. En el siguiente capítulo estudiaremos en detalle estos rasgos, por ahora solamente nos adentraremos a la manipulación de los métodos.

Ahora bien, aumentaremos un nuevo método a una clase existente; para hacerlo debemos posicionar el cursor en el panel superior derecho y activar el menú. Al hacerlo automáticamente aparecerá en el panel de texto, una plantilla de los elementos que deberá llevar un método (figura 2.16).

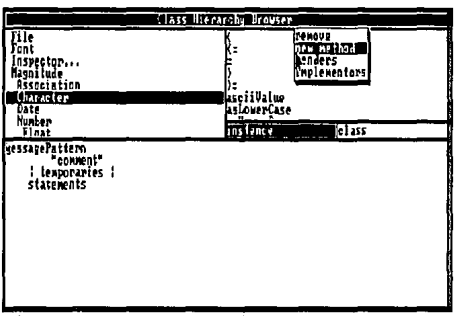


Figura 2.16 Prototipo de un método, proporcionado por Smalltalk

El modelo del nombre del mensaje (messagePattern), como hemos visto, debe ser el nombre del método que coincida con el mensaje. También contamos con un prototipo de comentarios para el método, declaración de variables temporales y el espacio para las instrucciones.

En especial la declaración de variables temporales de métodos (method temporaries), nos indica que dichas variables serán creadas durante la activación del método y existirá solamente durante el tiempo de vida de esa ejecución.

Para ilustrar todo lo anterior, se nos ocurre sumar un nuevo método que convierta un caracter numérico a su equivalente número entero. Para ello, veamos la figura 2.17.

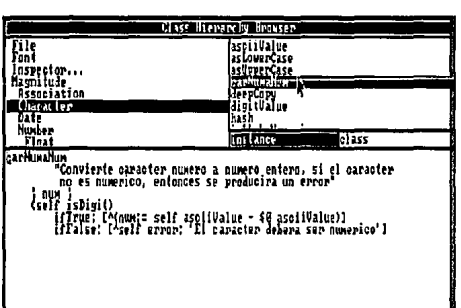


Figura 2.17 Añadiendo un método a la clase Character

El nombre del nuevo método es carNumaNum, en el que observamos la presencia de la palabra: "self". La cual es un tipo de variable especial en el lenguaje (llamada pseudo-variable), que nos permite representar al objeto receptor en el mensaje que se envíe.

Analizando el método, vemos que en la expresión:

```
(self isDigit)
```

se verifica si la variable self es dígito; isDigit es otro método ya definido en la clase Character, el cual tiene el siguiente código:

```
isDigit
        ^asciiInteger > 47 and:[asciiInteger < 58]
```

que cuestiona por medio de la variable de instancia asciiInteger, si el valor del caracter se encuentra entre 48 y 57 que son los valores ASCII para los dígitos. Como vemos, el resultado de la expresión es de tipo booleano (verdadero o falso).

Siguiendo con el análisis del nuevo método, observamos que existen dos bloques para el resultado de la expresión:

```
ifTrue: [^(num:=self asciiValue - $0 asciiValue)]
ifFalse: [^self error: 'El caracter deberá ser numérico']
```

si es verdadero se evalúa la variable self con su valor ASCII y se le resta el valor del caracter cero, que es el número 48. Esto se hace porque para obtener un número caracter a número entero, se debe restar siempre el caracter 0. Existe otra forma de referirnos al valor ASCII del objeto receptor (self asciiValue), y es a través de la variable de instancia asciiInteger que hará el mismo cálculo.

Por otra parte, si el caracter no fue dígito (ifFalse), entonces imprimiremos una ventana de error, que indique que el caracter que se teclee debe ser numérico.

```
$4 carNumaNum    ==>>  4.
```

Pero a todo esto, para poder ejecutar el nuevo método, es necesario salvarlo en el panel de texto. Al salvarlo, el método será compilado. Si no hubo errores de compilación, el método se aumentará a la lista de métodos de la clase. Una vez realizado esto, podremos ejecutarlo.

Ahora bien, si deseamos hacer parte permanente del ambiente al nuevo método, es necesario salvar en el archivo Image (como se vio en el capítulo anterior).

Es importante señalar que en Smalltalk se soporta la sobrecarga en nombres de métodos (method name overloading), esto se refiere a que podemos tener en una misma clase, el mismo nombre de un método, pero con diferente número de argumentos. Por ejemplo podemos tener el método llamado "valor" en una clase X, que será diferente al método "valor:" con un argumento, o al de "valor:

otrovalor:" con dos argumentos.

Nota: Recordemos que los dos puntos (:) al final del nombre del método, nos indican que un método espera un objeto argumento.

Y hablando en general sobre los tipos de métodos, todos los métodos a los que nos hemos referido anteriormente son los llamados "métodos de instancia" o de mensaje. Ello lo podemos notar, en cualquier Browser de clases que tendrá definidos los métodos de instancia, marcados bajo el panel derecho superior como **instance**. Existen otro tipo de métodos que son los "métodos de clase", que se invocan marcando bajo el panel derecho superior la opción **class**.

Los métodos de clase al igual que los de instancia, responden a los mensajes de los objetos de clase, excepto que generalmente se utilizan para la creación de objetos inicializados a cierto valor.

Por ejemplo en la clase Bag se tiene un método de clase que es new. Ver primer Browse de clases de la siguiente figura 2.18. Este método nos indica inicializar el objeto nuevo creado como Bag, a "initialize", quien a su vez es un método de instancia de esa clase, que inicializa los elementos de la bolsa a tipo Dictionary new (ver el otro Browser de clases de la misma figura). Esto significa que todos los elementos se inicializarán a valor 0[4].

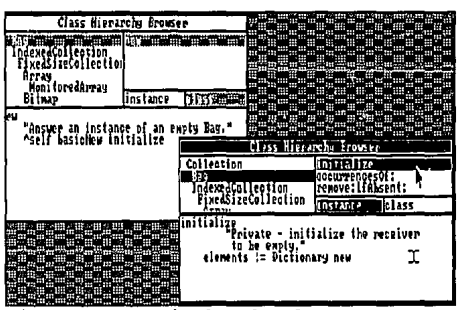


Figura 2.18 Métodos de clase

En las definiciones de clase, siempre las variables de instancia son inicializadas al objeto nulo (nil); pero como en el ejemplo anterior, podemos también crear nuestro propio método de clase para inicializar un objeto a un valor deseado.

El método de clase new puede ser implementado como caso especial para cada clase, pero si no es necesario hacerlo, utilizaremos el método new definido en la clase principal de Smalltalk: Objetc (objeto). Pues todos los objetos heredan una definición básica del método new de ésta clase base común.

Cuando un mensaje se envía a un objeto, éste checa con su clase, que exista un método definido para el mensaje. Si el objeto encuentra el método adecuado, lo recibe y lo ejecuta, de lo contrario continua buscando en las clases anteriores (en los niveles arriba de él en la Jerarquía de clases) hasta encontrarlo. De hecho, si aún no lo encuentra, puede llegar a buscar al nivel más alto -que es la clase Object.

---

[4] **Esto lo sabemos al observar el método "new" de la clase Dictionary, que en su proceso inicializa sus elementos a valor numérico cero.**

La clase Object provee un comportamiento común para todos los objetos. Entre otros, se encuentran el imprimir simbólicamente un objeto, desplegar su clase o especie, crear un nuevo objeto, etc.

En Smalltalk, todas las clases se derivan de Object. Si un método buscado no se encuentra en ninguna clase de las que se deriva el objeto, entonces se señalará un error, suspendiendo el proceso y apareciendo así una Ventana Walkback (ventana de errores), con una etiqueta de señalamiento.

Aquí podemos entonces invocar una ventana de depuración (debugger window), para explorar todos los métodos que fueron activados en el proceso de generar el error. Por ejemplo si mandamos un mensaje mal escrito, ver figura 2.19, en donde el mensaje correcto debe ser printOn:.

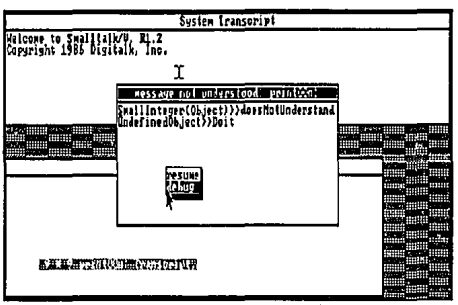


Figura 2.19 Cuando un método no se encuentra definido para el objeto, se genera un error

Una ventana Walkback describe una condición de error. La etiqueta nos muestra el tipo de error, y el panel de texto muestra los mensajes que fueron enviados a las clases, donde se generó el error. Muchas veces nosotros mismos al ver la etiqueta que señala el error, podemos determinarlo y corregirlo. Otras veces necesitaremos la ayuda del depurador para ver de donde se envió el error, marcando la línea deseada y observando todos los métodos que fueron invocados.

Un error típico en la evaluación de expresiones es que algún objeto receptor no entendió un mensaje selector:

NombreDeLaClase does not understand MensajeSelector

pues el nombre de la clase del objeto receptor no tiene definido en su protocolo a dicho mensaje selector. El protocolo de un objeto, se define como el conjunto de mensajes a los que el puede responder.

En el sistema Smalltalk, todos los procesos son inicializados con el paso de mensajes entre objetos. Aún el proceso de compilación o interpretación de código se lleva a cabo por los objetos del sistema, pues el propio intérprete por ejemplo es un objeto. Y como hemos venido mencionando, todo objeto en Smalltalk conoce su tipo, es decir, la clase a la que pertenece. Esto se logra en base a que el objeto mantiene un apuntador a la clase de la que se deriva. En cualquier momento podemos preguntar a un objeto acerca de su tipo, y el objeto contestará resultando en una cadena que indica el nombre de

la clase a la que pertenece.

CAPITULO III

"POLIMORFISMO DENTRO DE SMALLTALK"

3.1 Encapsulación

3.2 El papel del polimorfismo dentro de la P.O.O.

3.3 Herencia

CAPITULO III. POLIMORFISMO DENTRO DE SMALLTALK.

3.1. Encapsulación

La encapsulación es la forma de esconder la información contra modificaciones accidentales de datos, por lo que la encapsulación protege a los objetos de cambios no deseados en sus funciones. Los objetos solo tendrán acceso a los elementos públicos ya especificados, es decir, los objetos solamente se podrán comunicar entre si a través de la interfase pública asentada en su clase, para poder ejecutar un resultado deseado.

La memoria privada de un objeto, consiste específicamente de un componente de la clase: las variables de instancia. Como ya lo habíamos mencionado, las variables de instancia son variables privadas, que se accesan únicamente por lo métodos definidos en la clase, y su jerarquía descendente. Esto es lo que en Smalltalk se conoce como Encapsulación.

Cada miembro de una clase determinada, contará con sus propias variables de instancia. De hecho, cada clase especifica sus propias variables de instancia.

Como las variables de instancia son una parte componente del objeto (en las definiciones de clase), existirán solamente en el tiempo de vida del objeto, en su invocación.

El grado de acceso establecido en la especificación de una interfase dentro de una clase, permite que los objetos solamente se dediquen a sus funciones y no traten de inmiscuirse en el trabajo de otro objeto ajeno a él.

Los inspectores que ya hemos analizado, nos permiten mirar adentro de un objeto, pero no modificar sus variables de instancia. Por ejemplo, podemos crear una bolsa nueva y mirar adentro. Para ello veamos la figura 3.1, en donde se muestran las definiciones de clase de Bag con el browser de clases, y también se ve la creación e inspección de una nueva bolsa. En la ventana del inspector se muestran las variables de instancia de Bag y su contenido actual.

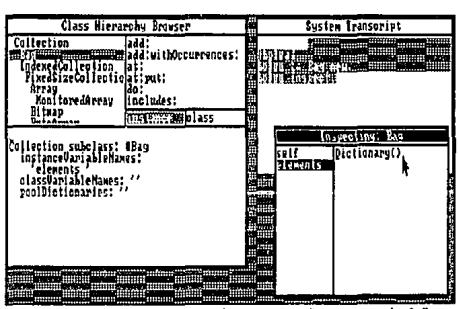


Figura 3.1 Inspeccionando las variables de instancia de Bag

Veamos como es que Smalltalk soporta una encapsulación de datos privados al 100%, esto es que no haya la posibilidad de modificar sus valores de instancia. Con el inspector podemos mirar las variables, y aunque podemos dentro de esa ventana modificar los valores y salvarlos, en realidad no se modificará su valor. La única manera de manipular los valores de las variables de instancia en el lenguaje Smalltalk/V, es a través de las definiciones de clase en la interfase del objeto.

Si el usuario lo desea, puede modificar el código de los métodos en la interfase pública, e incluso

establecer una nueva interfase y encapsulación, de acuerdo a sus propios requerimientos.

Por otro lado, existen ciertos mecanismos de orden, llamados "Verificadores de Relación" (Test Stubs), que se encargan de verificar que los datos creados sean pasados entre los objetos, a los métodos correspondientes, para tener así un resultado esperado.

Muchas veces se menciona a la programación orientada a objetos como programación con técnicas de abstracción de datos (Data Abstraction), en conjunto con la herencia de clases y el polimorfismo.

En el mundo real, todos los objetos que existen son entes individuales, con ciertas características que los hacen diferentes entre sí; pero en realidad podemos generalizar propiedades, para poder llegar a hablar de una clasificación. Esta clasificación se aplica precisamente a los principios de abstracción de datos. La abstracción o clase, conjunta las propiedades similares y omite las particularidades que distinguen a cada objeto.

Podemos ver entonces que ésta técnica de abstracción de datos, liga datos y procedimientos dentro de ciertas clases y permite a los objetos ser la herramienta de manipulación de esas clases; los datos se ocultan para llegar a ser abstracciones en el sistema.

Específicamente la abstracción de datos provee al programador la ventaja de poder definir nuevos tipos de datos sobre los ya definidos por el lenguaje. Esto es que, los tipos de datos o Clases son considerados abstractos debido a que de un tipo de dato específico se crean nuevos tipos definidos por el usuario. Esta idea se puede analizar partiendo de los conceptos mismos. Si se define a la abstracción como la separación de una cosa de otra, con la cual tiene una relación directa[1]; podemos pensar en que un objeto es por el mismo una clase, pero en acción, mientras que la clase será la definición de las características del objeto. El objeto tiene una relación directa con la clase, para llegar al sentido figurativo de separación de conceptos[2].

La abstracción de datos se alinea con el ocultamiento de datos (Data Hiding - Encapsulación) y la construcción de métodos y atributos dentro de los objetos, en base a clases ya definidas por el sistema.

El ocultamiento de datos específicamente se refiere al manejo del lenguaje con respecto a ocultar (o esconder) los detalles de las estructuras. La mayoría de los lenguajes de programación soporta el ocultamiento de datos, mediante declaraciones de variables locales (en el caso de los lenguajes que manejan procedimientos). Dichas variables locales solamente son manipuladas dentro del código en el que son utilizadas.

Al programar código en Smalltalk podemos heredar características de las clases, que son objetos que existen en el ambiente, al diseñar el programa y en el tiempo de ejecución del mismo. Pues como sabemos, las clases son objetos, es decir, las clases son tipos especiales de objetos que crean instancias de sí mismas. De este modo, podemos pensar en que no existe distinción entre el ambiente de programación y el tiempo de ejecución de código Smalltalk.

---

[1] Varios. "Enciclopedia Salvat", Salvat Editores, México, 1977, Tomo I, Pp.13.

[2] Para mayor referencia véase Clases, Jerarquías y Herencia en Capítulo II.

3.2. El papel del Polimorfismo dentro de la P.O.O.

Se le llama Polimorfismo a la capacidad de los objetos de responder al mismo mensaje, de manera singular. Los programas orientados a objetos, envían mensajes a cualquier objeto de un tipo desconocido, suponiendo que los objetos sabrán responder de cierta manera en particular, sin embargo aunque éstos objetos son diferentes, todos tienen cierta característica en particular: tener un conjunto común de nombres en su interfase pública. A ésta colección de nombres en los objetos se le conoce como "Protocolo de Comunicación". Para ejemplificar esto, veamos el diagrama 3.1, en donde se muestra una interfase pública con ciertos nombres de métodos comunes para el objeto Círculo y para Rectángulo (Dibujar, Rotar, Limpiar_pantalla).

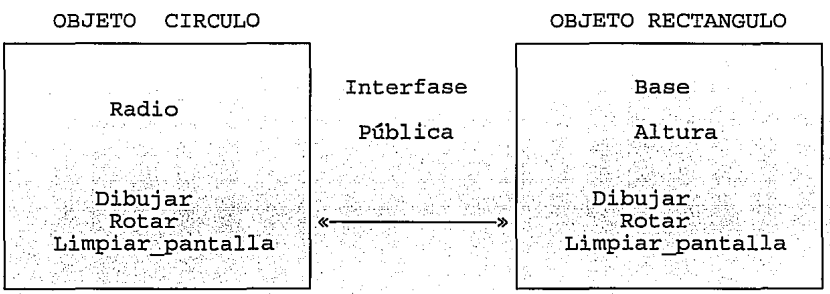


Diagrama 3.1 Protocolo de Comunicación

Aún cuando existan objetos con ciertos métodos en común, existen también los objetos que proveen métodos de los que nadie más dispone; inclusive cuando los métodos son iguales, pueden tener un significado diferente con sus propias especificaciones (en el ejemplo anterior no es lo mismo mandar el mensaje Dibujar, a un círculo que a un rectángulo), pues el polimorfismo asume que cada objeto responderá a su manera a cada mensaje.

La herencia nos auxilia en la especificación del protocolo de comunicación entre objetos. Esto es, los objetos que tienen ciertas propiedades en común son agrupados dentro de cierta clase y heredan comportamiento a otros objetos especiales. Por ejemplo podemos tener la superclase "Figuras" de la cual heredamos el método "dibujar" para poder crear un objeto "Círculo", pero añadiremos ciertas particularidades al método para poder diseñar específicamente un círculo. Esto es de gran ayuda para el programador, pues podemos heredar a varios objetos ciertos métodos con un mismo nombre, y a su vez cada subclase los modificará (si es necesario) de acuerdo a sus propios requerimientos. Así no tendremos el problema de tener que acordarnos del nombre de cada método para cada subclase diferente.

Si analizamos la palabra "Polimorfismo" podemos entenderla como una variación en la forma y

función entre los miembros de una especie en común. De esta forma, el concepto de polimorfismo se aplica a una colección de objetos que heredaron características de una misma superclase, pero que tienen ciertas propiedades que les permiten tener su propia y única respuesta a un mensaje en particular. Es por ello que debemos concebir al término "Polimorfismo" ligado de alguna manera al de "Herencia".

Un ejemplo de polimorfismo es el siguiente, donde evaluando las siguientes expresiones, tenemos dos tipos diferentes de objetos (cadenas y arreglos):

```
'cadena' at: 6   ==>> a
#(1 2 3) at: 3   ==>> 3
```

que responden al mismo mensaje (en éste caso at:) de diferente manera.

También en la mayoría de las formas de colecciones, como Set y Bag por ejemplo, no se limita el tipo de sus elementos al de Cadenas; una colección puede contener una diversidad de tipos de objetos. En la figura 3.2 se muestra como la colección Set contiene propiedades polimórficas, que le permiten soportar en un conjunto cadenas, arreglos, rectángulos y puntos al mismo tiempo.

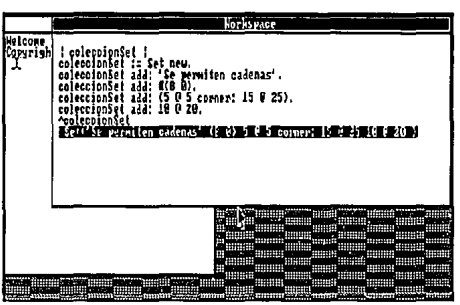


Figura 3.2 La clase Set soporta polimorfismo

Aún cuando las cadenas, arreglos, rectángulos y puntos son diferentes en su concepto, tienen algo en común: todos ellos de alguna forma, se derivan de la clase Object[3]. Esta clase establece el polimorfismo que podemos soportar en cualquier forma de colección. El protocolo de comunicación se dicta en Object, donde por ejemplo todos los objetos pueden responder al mismo mensaje printOn (enviado indirectamente por medio de la opción "show it"). Esto es, que las "clases" de una Cadena, Arreglo, Rectángulo o Punto, heredan el método printOn y entonces al trabajar con una colección no es necesario definir en su clase (Collection) código especial para imprimir su contenido.

Cuando la variable temporal (en nuestro ejemplo de la figura 3.2) coleccion_Set recibe el mensaje printOn, se itera éste mensaje pasando por cada elemento del conjunto; el iterador en éste caso va aumentando su valor de uno en uno, para referenciar a cada elemento de la colección.

El mensaje printOn también puede ser enviado directamente a un objeto, por medio de iteradores. Este mensaje debe ser enviado de una ventana a otra, como hemos visto anteriormente. En la figura 3.3 se ejemplifica la función de un iterador para utilizar directamente el mensaje printOn.

---

[3] Para mayor referencia véase Capítulo II.

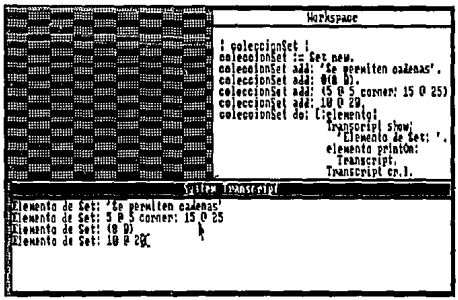


Figura 3.3 Uso de iteradores sobre colecciones para mandar imprimir a la ventana Transcript

Otra aplicación del concepto polimórfico se da internamente en la activación de un objeto. Este polimorfismo permite a un objeto en particular, funcionar de formas diferentes según el mensaje que le envío.

Por ejemplo podemos mandar el mensaje "edit" a un objeto cadena:

```
'cadena' edit
```

y la respuesta será la apertura de una ventana Workspace para poder editar la cadena. Pero también podemos mandar el mensaje "holaAmigos" (que no existe en Smalltalk/V) a ese objeto cadena, y la respuesta será una ventana de error (ver figura 3.4).

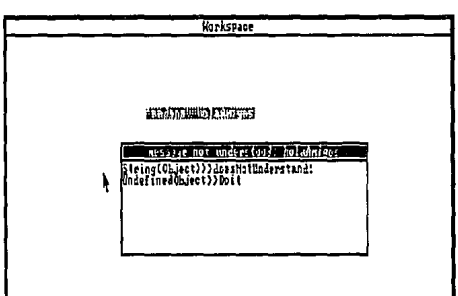


Figura 3.4 Ventana de error por un mensaje no entendido

De ésta forma vemos que el objeto puede responder a cualquier mensaje que le envíe, pues el mismo conoce los métodos que tiene definidos en su clase.

Si hablamos de que el propio lenguaje Smalltalk es un objeto, podemos pensar en el polimorfismo

que se da en la selección de opciones por medio de las ventanas de menús.

Por ejemplo, en la activación de cierta ventana por medio del mouse, es necesario saber si la posición del mouse se encuentra en una etiqueta de una ventana, en su panel o en una área fuera de ella; la técnica existente para ésta tarea consiste en la asignación de una área rectangular a cada parte de la ventana, que responderá de cierta manera a la solicitud del mouse. Esta parte se llama "Area Limitada" (bounding box). Si en el conjunto de objetos que se encuentran en la pantalla, cada uno responderá de cierta manera a la solicitud que se haga a su Area Limitada, podemos pensar claramente en un "Arreglo Polimórfico" (polimorphic array) de varios objetos con diferentes tipos y formas.

Como concepto, hemos mencionado al polimorfismo como el envío de un mismo mensaje a "distintos" objetos, para esperar una respuesta diferente en cada uno de ellos. En la práctica de la Programación Orientada a Objetos se envuelve a éstos objetos diferentes -pero de métodos similares- en una colección y activa su función enviándoles un mismo mensaje. Esta "similitud" se presenta en la herencia que tuvieron de una clase en común, mientras que la "diferencia" se debe a la definición de cada objeto con ciertas particularidades.

3.3. Herencia

La herencia es una poderosa herramienta de la programación orientada a objetos que conjuntamente con las clases nos permite reutilizar código y que con la ayuda del polimorfismo simplifica el protocolo de comunicación.

Como hemos venido mencionando, un conjunto de objetos que pertenecen a cierta clase, tienen rasgos comunes entre sí. Estos rasgos están definidos en una superclase. La herencia nos permite tomar rasgos definidos en la superclase, para crear nuestra propia Clase Derivada o Subclase. Esta técnica llamada "Programación por medio de herencia" (Programming by inheritance) nos permite a la vez poder heredar de una subclase, en donde la subclase pasa a ser la clase base (o también superclase) de la nueva clase a la que se heredó. Por ejemplo en el diagrama 3.2 se muestra la herencia entre clases de objetos animales.

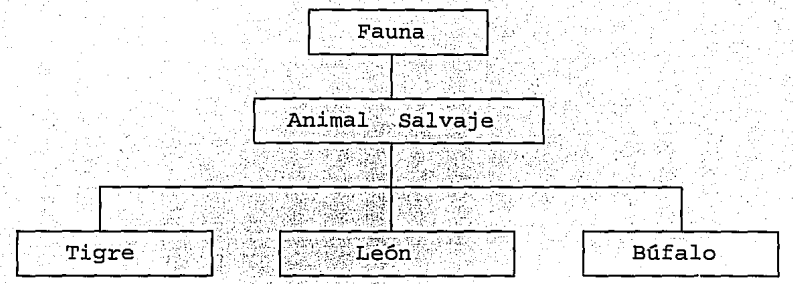


Diagrama 3.2 Herencia de Clases

Aquí podemos notar que existe cierta jerarquía descendente usualmente tomada sobre una estructura de árbol, en donde las propiedades más generales se encuentran en la raíz, y las más específicas se encuentran en las hojas, y se van dando según se va incrementando el nivel. A éstas jerarquías se les llama comúnmente "Jerarquías de Clase" (class hierarchy) ó de "Tipo" (type hierarchy).

Primeramente pensemos en que una Jerarquía de clases es una clasificación de los objetos, es decir, organización de los tipos de objetos (clases) en un orden de precedencia, donde se comparten ciertas propiedades en común.

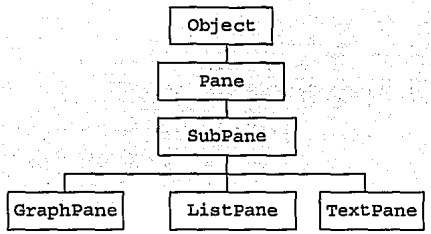
En Smalltalk cada clase tiene solamente una superclase inmediata y posiblemente más de una subclase; siempre la clase Object se encontrará en la raíz (root) de la jerarquía. Las clases más altas en la jerarquía, representan las características más generales, mientras que las más bajas (en los últimos niveles), las características particulares.

Por ejemplo mientras que la clase Fauna es muy general pues engloba a todos los animales, la clase León es más particular pues engloba a un tipo específico de animal salvaje.

Por lo tanto, podemos decir que las subclases pueden ser construidas sobre las características de sus superclases y de éste modo se simplifica la construcción de código nuevo sobre el código ya existente.

Para la reutilización de código basada en la herencia de clases, podemos auxiliarnos de las Librerías de clase, en donde se permite redefinir nuevas clases basadas sobre las clases ya existentes. La herencia provee un mecanismo para la reutilización de código que nos permite especificar solamente el código que será diferente.

Para entender mejor la herencia en Smalltalk/V, analicemos por ejemplo la clase Pane de la librería de clases de envase. Para ello, delimitaremos la jerarquía de clases a estudiar, como sigue:



La clase Pane es una subclase de la clase Object, así como SubPane es subclase de Pane, y a su vez GraphPane, ListPane y TextPane son subclases de SubPane. Las variables de instancia declaradas en el panel de definiciones de clase, para cada una de ellas son las siguientes:

para Pane: superpane, subpanes, dispatcher, model, frame, framingBlock, paneMenuSelector, paneScanner y curFont
para SubPane: name, changeSelector, margin, topCorner y scrollBar
para GraphPane: formHolder y selection
para ListPane: list, selection, currentLine y returnIndex
y para TextPane: textHolder, selection y changedArea.

Estas variables de instancia se refieren a las características necesarias para los elementos de los diferentes paneles con que cuenta Smalltalk/V (panel de tipo gráfico, de tipo lista o texto[4]).

Un objeto heredará todas las variables de instancia de las clases anteriores (en el nivel jerárquico), en adición a las definidas en su propia clase. Por ejemplo GraphPane contendrá las siguientes variables de instancia: superpane, subpanes, dispatcher, model, frame, framingBlock, paneMenuSelector, paneScanner, curFont, name, changeSelector, margin, topCorner, scrollBar, formHolder y selection.

Ahora bien, debemos pensar en que no solamente se heredarán las variables de instancia, sino también todas las definiciones de la clase. Al igual que ellas, los métodos también forman parte de la herencia. Cuando un mensaje se envía a un objeto, Smalltalk busca el método correspondiente a la clase del objeto. Si el método se encuentra, entonces se ejecuta; de lo contrario Smalltalk repite el procedimiento en cada nivel anterior a la clase, hasta llegar a la clase raíz (Object).

Por ejemplo, observemos la figura 3.5 en donde abrimos dos ventanas del Browser de clases. La primera de ellas nos mostrará los métodos definidos para la clase Pane, y la segunda los métodos de la clase GraphPane. En la clase Pane se tiene el método activatePane, mientras que la clase GraphPane reimplementa su propia versión de activatePane (ver figura 3.6).

[4] Para mayor referencia véase Capítulo I.

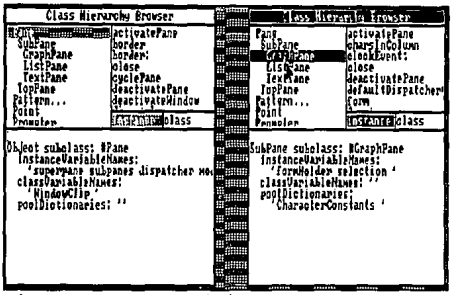


Figura 3.5 Definiciones de las clases Pane y GraphPane

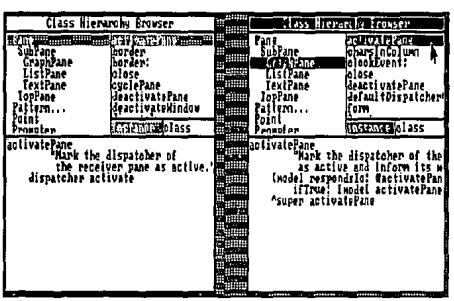


Figura 3.6 Reimplementación del método ActivatePane

Esto es porque ocasionalmente el programador querrá sobreescribir el método para un caso particular con algunas modificaciones, en lugar de usar el método de cualquier nivel anterior en su jerarquía de clases. En la figura anterior, podemos observar que en el método activatePane de la clase Pane, se realiza cierta actividad; mientras que en graphPane se reimplementa una nueva versión del método. En la línea concluyente de éste último método (ver panel de texto de la segunda ventana browser):

```
^super activatePane
```

se utiliza una pseudo-variable especial en Smalltalk: **super**. Dicha variable representa al objeto receptor en el mensaje enviado, y buscará al método (en éste caso activatePane) en la superclase correspondiente. En éste caso específico, estando en la clase graphPane, se buscará el método en la superclase SubPane, pero como ella no tiene definido ningún método activatePane, entonces Smalltalk procede a buscar en los métodos de Pane, donde precisamente se encuentra dicho método.

Como vemos, a través de la herencia los objetos pueden heredar todas o algunas de las características a otros objetos. De ésta forma, el Polimorfismo se aplica a los métodos heredados de una clase, aún cuando éstos sean modificados por la subclase.

La diferencia principal entre la herencia y el polimorfismo radica principalmente en que, mientras que en la herencia se da cierta generalidad que permite la modificación del comportamiento de un objeto, en el polimorfismo se usa a la herencia en casos particulares para expresar características en común (los nombres de los métodos). Para mayor claridad, véase el diagrama 3.3[5].

[5] Al hablar de Herencia de Métodos, nos referimos a la manipulación de los métodos para ilustrar su relación con el Polimorfismo.

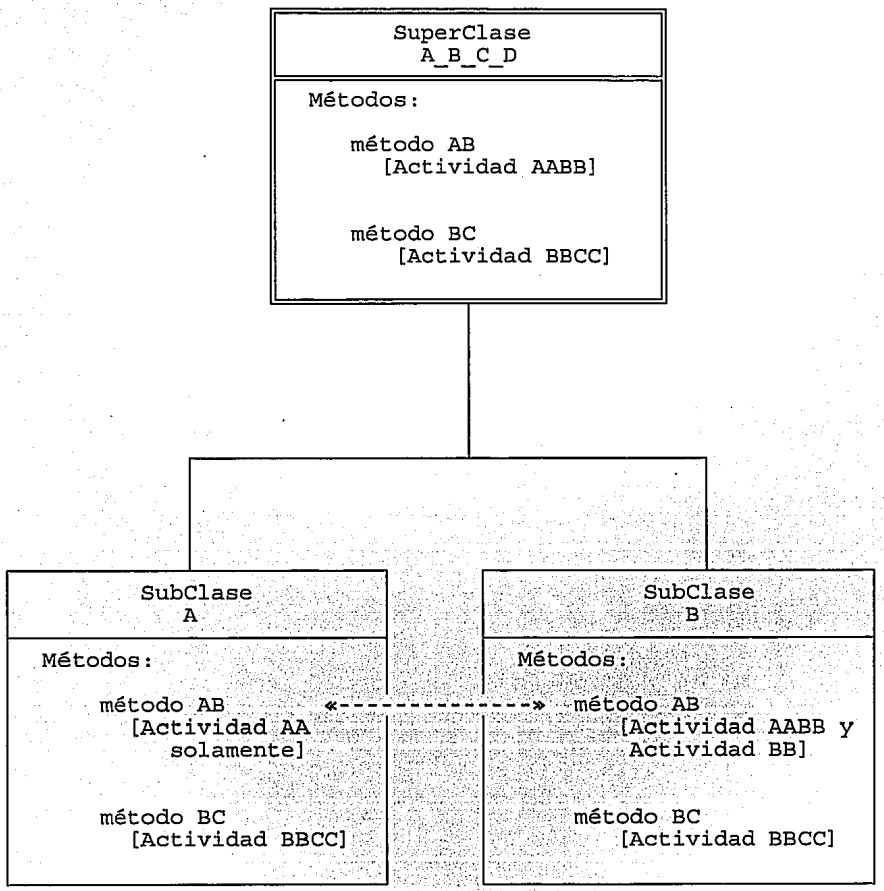


Diagrama 3.3 Herencia de Métodos y Polimorfismo, donde el método AB tiene el mismo nombre pero diferentes especificaciones para cada subclase

El manejo de Ligas Dinámicas de Smalltalk/V nos permite construir mecanismos de manejo de mensajes y funcionalidad de los objetos, para un fácil mantenimiento del control y la comunicación entre los objetos. Cuando en el tiempo de ejecución de un programa, se suman nuevos tipos de datos al sistema ó se modifica algún objeto, no es necesaria una recompilación. Las ligas dinámicas se encargan del manejo del paso de mensajes entre objetos en el tiempo de ejecución, haciendo las correspondientes extensiones a las jerarquías definidas.

Si consideramos las ventajas de las ligas dinámicas, podemos tener grandes sorpresas de tiempo y código.

Si en el tiempo de ejecución de un programa necesitáramos recompilar o modificar el código cada vez que se creara un nuevo objeto, la elaboración manual de líneas de código -para el mantenimiento del programa- y el tiempo invertido en su lógica, serían tediosos para el programador. Recordemos que los mensajes y los métodos se activan en el tiempo de ejecución de un programa.

CAPITULO IV

"ALGUNAS APLICACIONES DE SMALLTALK/V"

4.1 Conceptos básicos de gráficos en Smalltalk/V

4.2 A qué tipo de problemas podemos aplicar Smalltalk/V?

CAPITULO IV. ALGUNAS APLICACIONES DE SMALLTALK/V.

4.1. Conceptos básicos de Gráficos en Smalltalk/V

Las capacidades de gráficos en Smalltalk, se deben a las gráficas por mapeo de bits (bit-mapped graphics), que consisten en almacenar los puntos de una gráfica en mapas de bits (Bitmap).

Una línea se dibuja con un vector continuo de puntos. Por ejemplo el cursor se dibuja en base a un rectángulo de puntos blancos y negros. Los caracteres se forman por un bloque de puntos.

En Smalltalk los puntos en la pantalla se despliegan como pixeles. Los pixeles (apócope de picture alements, elementos de dibujo) son partes que delínean una figura.

Existen dos tipos básicos de modo de operación de los monitores en general: modo texto y modo gráfico. El modo texto muestra únicamente caracteres básicos, mientras que el modo gráfico (en el que trabaja Smalltalk/V) puede mostrar cualquier figura según la capacidad del monitor con el que estemos trabajando. La diferencia principal entre los monitores, es su resolución, es decir, su capacidad para mostrar pixeles por pulgada cuadrada en la pantalla.

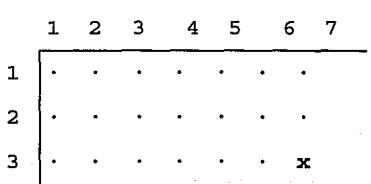
Un mapa de bits es una matriz de bits que se almacena en una forma (Form); en donde el color blanco se representa con el valor 1 (prendido), mientras que el negro se representa con el 0 (apagado).

Las áreas rectangulares envuelven a un conjunto de puntos. Estos puntos se pueden mover de un lugar a otro dentro de un mapa de bits e inclusive de un mapa a otro. Para referirnos a un punto individual nos basaremos en un punto coordenado.

Las principales estructuras de datos gráficos en Smalltalk/V son las clases Point, Rectangle y Form.

La clase Point engloba a los puntos en la pantalla. Un punto se refiere a una posición coordenada (x,y) basada en un arreglo bidimensional (dos dimensiones) de columnas y renglones. La clase Point cuenta con dos variables de instancia: x (que representa al número de columna) y y (que representa al renglón). La coordenada x se incrementa de izquierda a derecha, mientras que y de arriba hacia abajo.

Para crear un punto es necesario utilizar el mensaje binario "@" que toma como objeto receptor a la columna x y como objeto argumento al renglón y. Por ejemplo para crear un punto en la columna 7 y renglón 3:



es necesario mandar el siguiente mensaje:

```
7 @ 3.
```

Para retornar el valor de renglón o columna de un punto, se utilizan los siguientes mensajes:

```
(7 @ 3) x  ==>>  7
(7 @ 3) y  ==>>  3.
```

Podemos también así cambiar los valores de **x** y **y**. Para ejemplificar, veamos la siguientes expresiones:

```
PuntoNuevo := (3 @ 4)
PuntoNuevo x: 5.
PuntoNuevo y: 6.
^PuntoNuevo  ==>>  5 @ 6
```

en donde se retornará ahora el punto con los nuevos valores asignados.

Smalltalk nos permite comparar puntos, además de poder sumarlos, multiplicarlos, restarlos o dividirlos, por medio de los mensajes aritméticos.

Al comparar dos puntos específicos, el respectivo valor de **x** en un punto se comparará con su correspondiente valor **x** en el otro punto. Veamos los siguientes ejemplos:

```
(7 @ 3) * (4 @ 6)  ==>>  28 @ 18
(5 @ 3) > (3 @ 4)  ==>>  false
```

en la última expresión el resultado de retorno es falso debido a que, aunque en **x** si se cumple que 5 sea menor que 3, en **y** es falso; por lo tanto para que la expresión sea verdadera deben cumplirse las dos comparaciones.

Podemos también mezclar puntos con números escalares, con los mensajes aritméticos (excepto la comparación), resultando en otro punto:

```
(-3 @ 2) + 3  ==>>  0 @ 5
```

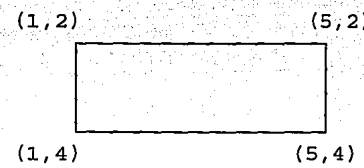
evaluando observamos que el escalar se aplica a ambos valores del punto dado.

Otra de las estructuras de datos gráficos en Smalltalk/V es Rectangle, quien maneja trazos rectangulares.

Un rectángulo se representa por dos puntos que lo delimitan. Estos puntos son el origen (origin) y la esquina (corner), y para crear un rectángulo se utiliza el mensaje corner: basado en éstos dos puntos. Al contar con ésta información, Smalltalk puede determinar su amplitud y su altura. Por ejemplo analicemos el siguiente rectángulo:

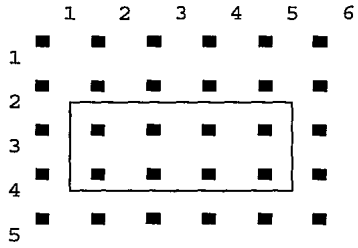
```
1 @ 2 corner: 5 @ 4
```

en donde las esquinas de él se verían como en el siguiente diagrama:



Nota: Simplemente el hecho de crear tanto rectángulos como puntos, no significa que se dibujarán en pantalla, sino que solamente se refieren a una posición.

Un rectángulo contiene un grupo de bits delimitados por el mismo rectángulo. Por ejemplo el rectángulo anterior se ilustraría con bits del siguiente modo:



en donde los cuadritos obscuros representan a los bits. El rectángulo contiene 8 bits dentro, 4 horizontales y 2 verticales.

Las operaciones que podemos realizar con los rectángulos, se basan en mensajes unarios y keyword. Ejemplo de ello son las siguientes expresiones:

```
(1 @ 1 corner: 50 @ 50) center  ==>>  25 @ 25
(1 @ 1 corner: 50 @ 50) containsPoint: 25 @ 25  ==>>  true.
```

Ahora bien, para poder visualizar una imagen gráfica de puntos o rectángulos, Smalltalk/V se auxilia de la estructura Form que es una clase dentro de las subclases de DisplayObject. Form representa a un rectángulo definiendo un área en la pantalla, pues provee una vista bidimensional para un mapa de bits delimitado por ese rectángulo.

Las variables de instancia que Form contiene son muchas, pero de acuerdo a nuestros intereses de conceptos básicos de gráficos en Smalltalk, solamente analizaremos a bits, width (anchura) y height (altura). La variable bits representa el área que delimita al rectángulo, mientras que width el ancho y height la altura del mismo. Para poder crear un objeto Form es necesario conocer solamente el objeto argumento, puesto que el origen siempre será el punto (0,0):

0 @ 0 corner:(width @ height)

con esto no se quiere decir que la posición en pantalla del rectángulo comience siempre desde la esquina superior izquierda, pues podemos (si deseamos), posicionar el rectángulo en otro lugar. Un ejemplo de la funcionalidad de Form se ve en la figura 4.1.

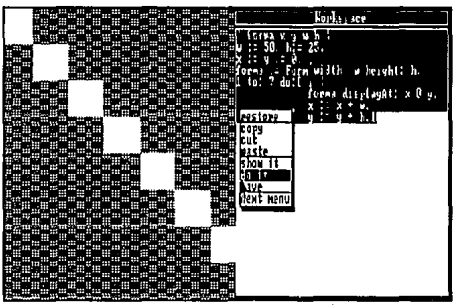


Figura 4.1 Desplegando rectángulos

Por otro lado, Smalltalk/V nos permite crear nuestras propias ventanas de edición, delimitando su área por medio de rectángulos. La forma de crear una ventana es muy sencilla (ver figura 4.2) donde declaramos una variable que representará a la ventana de edición creada por nosotros, con todas las opciones del menú de un editor de textos.

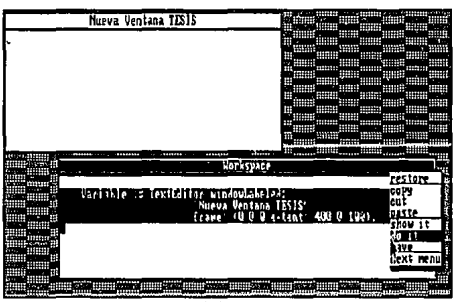


Figura 4.2 Creación de una ventana

La clase BitBlt es la clase fundamental en el manejo de todos los gráficos en Smalltalk/V. Sus siglas ("Bit Block transfer") se refieren a la generalización de transferencia de datos a locaciones de bits (o pixeles)

La función básica de BitBlt es mover un área de bits delimitada por un rectángulo en un objeto

Form, a otro lugar. El proceso que se sigue para copiar esa área, es el siguiente[1]:

- conocer el origen de Form
- conocer el origen a donde copiar
- contar con las medidas del rectángulo que se moverá

Smalltalk sabrá entonces calcular la esquina de la nueva posición a donde copiará los bits. Por ejemplo en el siguiente código:

```
(BitBlt destForm: Display sourceForm: Display)
        sourceRect:(0 @ 0 corner: Display extent //2);
        destOrigin:(Display width // 2 @ 0);
        copyBits.
```

se copiará la parte superior izquierda de la pantalla a la parte superior derecha. Aquí utilizamos la clase BitBlt como instancia, con Display como origen y a la vez destino. Recordemos que Display es todo lo que se encuentra en la pantalla. En el código anterior posteriormente se especifica el rectángulo que se moverá, delimitando así el area; y el origen del lugar a donde moveremos los bits.

Existe también la posibilidad de crear efectos de tonos grises en pantalla. Esto lo podemos hacer a través los llamados "medios tonos" o "máscaras" (mask). Estas máscaras pertenecen a la forma Form. Transformemos por ejemplo un pedazo de pantalla, en color gris:

```
(bitBlt destForm: Display sourceForm: nil)
        mask: Form gray;
        destRect: (0 @ 0 corner: 100 @ 100);
        copyBits
```

esto se lleva al cabo con la ayuda de la clase BitBlt. En éste código se hace que una parte (delimitada por un rectángulo) de la pantalla de efectos visuales grisaseos. Los mensajes que representan a los tonos que podemos manejar son los siguientes: gray (gris), black (negro), darkGray (gris obscuro), lightGray (gris claro). Para obtener una tonalidad totalmente blanca, lo único que se debe hacer es no utilizar una máscara específica:

```
(bitBlt destForm: Display sourceForm: nil)
        destRect: (0 @ 0 corner: 100 @ 100);
        copyBits.
```

BitBlt tiene dos subclases en el mismo nivel de la jerarquía de clases, que son CharacterScanner y Pen. La clase CharacterScanner convierte caracteres ASCII en su representación gráfica. Por ejemplo:

```
'cadena' displayAt: 10 @ 10
```

crea un CharacterScanner y despliega la cadena de caracteres en la posición (10,10) de la pantalla. También cuando creamos un CharacterScanner podemos dictar las características deseadas para desplegar caracteres con un tipo de letra específico. Por ejemplo:

---

[1] Manual. "Tutorial and Programming Handbook", Xerox Corp., California, 1986, Pp. 107.

```
CharacterScanner new
        initialize: Display boundingBox font:
                                        Font eightLine;
        display: 'Cadena' at: 10 @ 10.
```

en donde se inicializan los caracteres nuevos a desplegar en pantalla dentro de una caja limitada (bounding box) y con letra de 8 líneas de altura.

La caja limitada es una herramienta para delimitar un área, en éste caso, del tamaño de la cadena y tipo de letra especificados.

Muchas veces, con el uso de gráficos, es importante desplegar cierto tipo de pantalla, como una "pantalla en blanco" por ejemplo. Esto es posible con el uso del mensaje "white" (blanco) y Display:

```
Display white.
```

En esta expresión, 'Display' (desplegar) es un objeto contenido en Smalltalk, que vive como instancia de la clase DisplayScreen. El mensaje selector white se envía a Display para dibujar la pantalla en blanco total. De forma similar, contamos con un tono negro (mensaje black) y con un tono gris (mensaje gray).

Contrario a lo anterior, podemos volver a dibujar la pantalla con las ventanas que anteriormente se tenían activas, en base al objeto Scheduler (programador):

```
Scheduler systemDispatcher redraw
```

de donde Scheduler es responsable de interactuar entre el usuario y el ambiente, determinando las entradas con el mouse, las ventanas a dibujar, etcétera; en base al mensaje redraw (volver a dibujar) aunado al systemDispatcher (manejador del sistema). Otra forma de restaurar la pantalla, es por medio del menú del sistema en la opción redraw screen.

El sistema para despachar (System Dispatcher) nos sirve para interpretar las entradas del teclado o del ratón (mouse) y mandar los mensajes apropiados a las ventanas correspondientes. Entre sus funciones principales, se encuentran la activación y desactivación de paneles, retorno del cursor a la esquina superior izquierda de una ventana recién abierta, y abrir y cerrar ventanas.

Ahora bien, una forma que Smalltalk/V nos brinda de retener las figuras, es la que se basa en "menús de mensajes". Con ellos podemos mirar la pantalla hasta presionar alguna tecla[2].

```
Menu message: 'Presione cualquier tecla para continuar'.
```

Se envía el mensaje message: a la clase Menu con un argumento de tipo cadena de caracteres.

Dentro de los gráficos considerados más básicos en Smalltalk/V, encontramos a las plumas (pen), dentro de la clase Pen.

---

[2] Sería un equivalente a un REPEAT UNTIL KEYPRESSED en Pascal o a un getch() en C.

La clase llamada Pen, es una clase de la que ya hemos hablado en el segundo capítulo. Cuando le decimos a Pen que dibuje una línea de un lugar a otro, la pluma utiliza un BitBlt para trazar la línea. "Por ejemplo, para dibujar de 0 @ 0 a 9 @ 0, la pluma copia de su origen Form a su destino Form 10 veces, comenzando en 0 @ 0 y moviéndose a la derecha pixel por pixel... El resultado final se observa directamente en una línea horizontal"[3] Para cambiar el color a la pluma, se utiliza una máscara Form (mask Form) con el tono deseado.

Como ya hemos dicho, una pluma siempre recuerda su posición y su color. Dentro de su posición podemos referirnos a su sitio y a su dirección. El sitio se refiere a la posición en donde comenzará a escribir la pluma, mientras que la dirección se refiere al punto hacia donde se dirigirá.

En la figura 4.3 se muestra un ejemplo de la utilización de una pluma. Observamos que la traza resultado del código evaluado son polígonos dentro de polígonos en donde se va disminuyendo el número de lados.

La lógica del código marcado nos dice que iremos dibujando polígonos disminuyendo sus lados hasta llegar a un triángulo. La primera línea de código se refiere a las variables temporales de la creación de la pluma, numero de polígonos a dibujar y numero de lados con el que comenzaremos el polígono. En la segunda línea nos referimos a crear una pluma nueva e inicializarla:

```
defaultNib: 2.
```

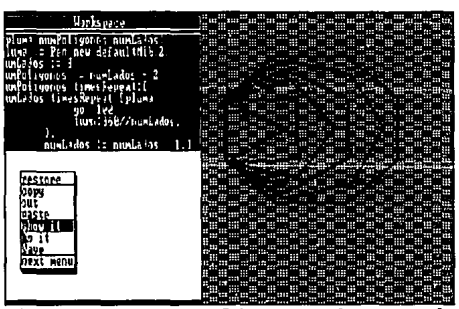


Figura 4.3 Polígonos dentro de polígonos

Un nib nos sirve para poder fijar el ancho de la pluma. Por ejemplo en el código evaluado se crea una pluma con una forma (Form) negra con dos bits de ancho. Si se desea podemos aumentar el ancho y tomar por omisión el valor del nib como 1 (sin necesidad de indicar el defaultNib:).

De forma similar a Pen, la variable global Turtle puede utilizarse para dibujar movimientos en pantalla.

Podemos aplicar el concepto de variables globales de gráficos con la ayuda de ciclos (loops) simples, para dibujar cualquier polígono. Contamos con aplicaciones de objetos gráficos que Smalltalk encierra en sus variables globales. En la figura 4.4 se muestra la variable global Turtle definida por el lenguaje, conjuntamente a los ciclos simples.

[3] Op. Cit. Manual. "Tutorial and Programming Handbook", Pp. 112.

Figura 4.4 Uso de variables globales definidas por el lenguaje y Ciclos simples

Aquí se declaran dos variables temporales que definen el número de pétalos (en nuestro caso triángulos) y la longitud de sus lados, respectivamente. Con el uso de la pluma Turtle dibujamos los triángulos, direccionándolos para así dar forma a la figura deseada. El ciclo simple timesRepeat: itera el número de pétalos que deseamos dibujar; dentro de él, Turtle dibuja hasta la longitud deseada y se utiliza otro ciclo para dibujar la forma de cada polígono e ir dando forma de flor a la unión de cada punta del triángulo.

En Smalltalk/V también contamos con la clase Commander que se encuentra entre las subclases de Pen. Commander dirige a un arreglo de plumas para dibujarlas en formas de abanico (fan out); y al igual que Pen reimplementa mensajes como go:, up, down, etcétera, para dar forma a una figura. El siguiente código nos muestra como se utiliza ésta clase, y el uso de menús de mensajes:

```
| numPen verticesMandala diametroMandala |
Display white.
numPen := 4.
verticesMandala := 10.
diametroMandala := 130.
(Commander new: numPen)
        fanOut;
        up;
        go:60; "punto donde comenzará"
        down;
        mandala:verticesMandala
                diameter: diametroMandala.
Menu message: 'Presione cualquier tecla'.
Scheduler systemDispatcher redraw.
```

Observemos su evaluación en la figura 4.5. Aquí observamos que se dibujan 4 trazas "mandala" direccionadas en forma de abanico. La variable numPen nos indica el número de plumas que serán comandadas. Entre otros, los mensajes dragon:, polygon: sides: y spiral: angle: nos permiten dibujar otro tipo de trazas.

Otra de las bondades en grafos que Smalltalk nos permite, es la percepción de redes de nodos. En los diskettes del lenguaje Smalltalk/V, se contiene un archivo para adherir a la librería de clases de envase. Este archivo se suma al ambiente con la instrucción fileIn ya anteriormente explicada, y consiste de dos clases que serán subclases directamente de Object. Estas dos clases son Network y NetworkNode que se presentan como herramientas del lenguaje para poder desplegar en pantalla una red gráfica de nodos entrelazados.

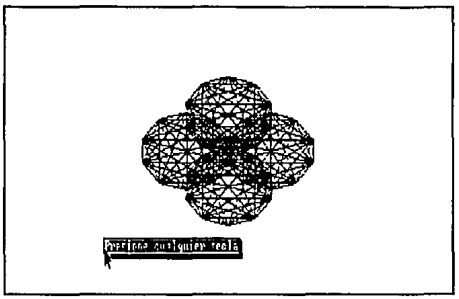


Figura 4.5 Ejemplo de Commander

Para poderlas utilizar, es necesario declarar una red nueva, dar nombre y posición a los nodos, y conectar los que se deseen:

```
NuevaRed := Network new.

NumNodo1 := Network new
                name: 'Nodo 1'
                position: 300 @ 100.
NumNodo2 := Network new
                name: 'Nodo 2'
                position: 450 @ 25.
NumNodo3 := Network new
                name: 'Nodo 3'
                position: 430 @ 110.
NumNodo4 := Network new
                name: 'Nodo 4'
                position: 550 @ 185.
NumNodo5 := Network new
                name: 'Nodo 5'
                position: 295 @ 140.
NumNodo6 := Network new
                name: 'Nodo 6'
                position: 30 @ 95.
NumNodo7 := Network new
                name: 'Nodo 7'
                position: 150 @ 15.

NuevaRed
        connect: NumNodo1 to: NumNodo2;
        connect: NumNodo1 to: NumNodo3;
        connect: NumNodo1 to: NumNodo5;
        connect: NumNodo1 to: NumNodo6;
        connect: NumNodo2 to: NumNodo3;
        connect: NumNodo4 to: NumNodo5;
        connect: NumNodo6 to: NumNodo7.
```

Ahora bien, para poder desplegar lo anterior en pantalla, es necesario mandar a dibujar la nueva red, y opcionalmente dar un color al fondo de la gráfica:

```
Display gray.
NuevaRed draw.
Menu message: 'Presione cualquier tecla'
Scheduler systemDispatcher redraw.
```

De ésta manera, podemos así dibujar el tipo de red de acuerdo a nuestros propios requerimientos, de una forma sencilla en el ambiente. La figura 4.6 nos muestra el resultado de evaluar todo el código anterior.

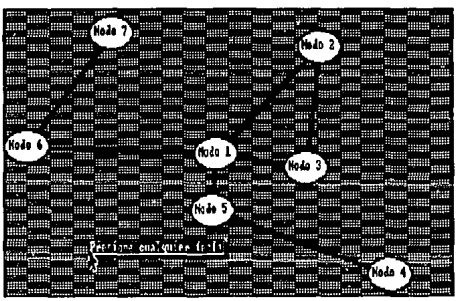


Figura 4.6 Red de 7 nodos

Como vemos, podemos sacar muchas ventajas del lenguaje a través de su manejo de gráficos, en las aplicaciones que así lo requieran, y de una forma sencilla.

4.2. A qué tipo de problemas podemos aplicar Smalltalk/V?

Posteriormente a aprender y analizar los elementos de un lenguaje, nuestros propósitos serán los de aplicar todos esos conocimientos de los conceptos fundamentales del lenguaje, sobre ejemplos prácticos.

Los ejemplos son una importante descripción del lenguaje de programación y ambiente Smalltalk/V. Muchos de los ejemplos usados a lo largo de ésta tesis se programaron para mostrar un concepto en especial; en éste subcapítulo trataremos de conjuntar algunos de esos conceptos para hacer ejemplos un poco más complejos.

Comencemos entonces con una propuesta de solución a un problema comúnmente enfrentado por el usuario de computadoras.

En Smalltalk no se tiene el problema de tener que abreviar las variables que deseamos utilizar

hasta con 8 caracteres. Muchas veces en los sistemas donde existe restricción, esto nos resulta un gran problema, al no saber que nombre poner a dicha variable, para que tenga un significado conciso. También podemos notar éste problema, al tratar de elegir el nombre adecuado a un archivo del Sistema Operativo MS/DOS que limita de igual manera a 8 caracteres el nombre del archivo.

A continuación se muestra una alternativa de solución para la elección del nombre adecuado a cierta variable o a un archivo, en su caso[4]:

```
"Programa que abrevia un nombre largo"
| nombre long |
nombre := Prompter prompt: 'Introduzca el nombre'
                                    default: ''.
nombre := nombre reversed.
long := nombre size.
^(nombre reject: [:caracter |
                    caracter isVowel and: [
                      c isLowerCase and: [
                      (long := long - 1) >= 8]]])
                            reversed,'      '
                            copyFrom 1 to: 8.
```

El programa funciona para cualquier cadena que se lea del teclado (por medio del mensaje prompt:) y que se introduzca en forma continua, separando los diferentes significados de ella, con letras mayúsculas. Como por ejemplo:

NombreDeLaVariableLarga

en donde evaluando se tendrá por resultado la siguiente abreviación:

NmbrDLVr.

Si analizamos el algoritmo, podemos ver que se basa en la eliminación de vocales de derecha a izquierda, para que el nombre abreviado pueda tener mayor significado. Para esto es necesario escribir al revés el nombre dado.

En la línea:

long := nombre size.

medimos la longitud del nombre, para ir analizando caracter a caracter. En la siguiente línea el símbolo ^ nos representa el retorno de un objeto simple; el objeto resultado proviene de evaluar cada caracter en ese orden que resulte falso a las evaluaciones -recordemos que el iterador generalizado reject: retorna los valores para los que la expresión resulte falsa.

En el bloque de instrucciones de reject: las evaluaciones consisten en verificar si el caracter es vocal y mayúscula; esto es porque para abreviar el nombre correctamente, tomaremos en cuenta los

---

[4] Idea tomada de: Manual. "Tutorial and Programming Handbook", Xerox Corp., California, 1986, Pp. 47.

caracteres que son consonantes, y los que son mayúsculas. Ahora bien, notamos que en la expresión:

```
(long := long - 1) >= 8
```

la longitud del nombre de la variable se va controlando, esto es, mientras la longitud sea mayor o igual a 8, el resultado será verdadero; y si el caracter llega a ser vocal (isVowel) y minúscula (isLowerCase) también, entonces todas las expresiones serán verdaderas:

```
true and: [ true and: true ]]]
```

y reject: no tomará en cuenta ese caracter. Si la longitud llega a ser menor que 8 (al ir disminuyendo de uno en uno la misma) entonces la expresión resultará falsa[5], y todo caracter será sumado a las evaluaciones falsas de reject:.

Finalmente, el objeto "cadena de caracteres" que resulte de lo anterior, se escribirá en reversa (para tener ya el nombre en el orden adecuado), y se sumará a la cadena de 8 caracteres en blanco, por medio del mensaje binario coma (,). Estos 8 caracteres en blanco se suman, para que en caso de existir una variable muy corta, se le sumen caracteres en blanco al final. A todo éste mensaje, se manda aún el mensaje copyFrom: to: para copiar desde el elemento 1 hasta el 8 y así tener ya como resultado un nombre abreviado con 8 caracteres.

Veamos algunos ejemplos de nombres (ya sean de archivos o de variables) que serán abreviados, resultando de evaluar el programa anterior:

```
MetodoDeGauss        ==>>  MetdDGss
ArchivoDeCaptura     ==>>  ArchvDCp
Tiempo               ==>>  Tiempo
Variable             ==>>  Variable
VariableLargaLarga   ==>>  VrblLrgL.
```

Como hemos mencionado anteriormente, en Smalltalk/V podemos tener desde las más simples, hasta las más sofisticadas aplicaciones. Podemos conjuntar algunos de los conceptos del sistema, y combinar su uso para cierto estudio. Por ejemplo, el uso de iteradores, como lo hemos visto en los capítulos anteriores, son de gran utilidad para la evaluación de los elementos de una colección.

En una simple aplicación de los iteradores sobre los diccionarios, en Smalltalk/V se utilizan pocas líneas de código para su ejecución, mientras que en cualquier otro lenguaje tendríamos una serie de declaraciones y especificaciones con mucho más código para dicha aplicación. En la figura 4.7 se muestra un "reporte" sobre una Agenda Telefónica (ya creada en el capítulo II), con el uso de iteradores para trabajar sobre los elementos de ella.

---

[5] Recordemos que en la tabla lógica de una conjunción (and), si algún valor resulta falso, entonces la evaluación será falsa.

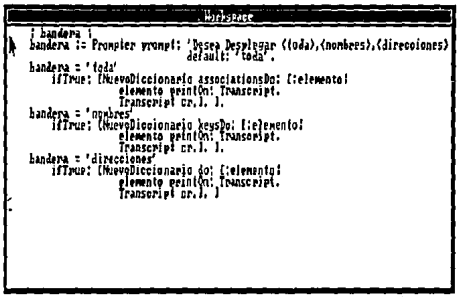


Figura 4.7 Ejemplo de una simple aplicación de los iteradores en los diccionarios

En el programa anterior, se puede desplegar desde toda la agenda telefónica por medio de asociaciones, hasta las claves o valores contenidos. El mensaje do: itera sobre los valores del diccionario NuevoDiccionario, mientras que keysDo: itera sobre las claves del mismo. Como podemos observar, para imprimir toda la asociación, es necesario el mensaje associationsDo:. Los iteradores pueden ser usados para imprimir los elementos de un diccionario, que serán direccionados a la ventana Transcript.

Y hablando también sobre colecciones, hemos mencionado también en anteriores capítulos, las ventajas sobre la construcciones de estructuras de datos un poco más complejas basadas en Colecciones dentro de Colecciones. Pues bien, podemos pensar en otra forma del manejo de la Agenda Telefónica anterior, con éstas características. Veamos la figura 4.8.

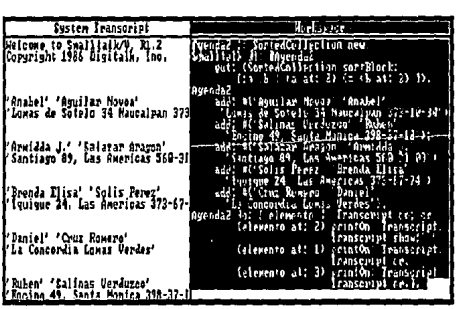


Figura 4.8 Aplicación de colecciones dentro de colecciones en una Agenda Telefónica

En éste ejemplo, podemos notar que la forma de manejo de una Agenda Telefónica, requiere del uso de la colección SortedCollection, por ser una colección que maneja el orden de los datos en forma

alfabética. Ahora bien, si intentamos introducir los datos con el mensaje add:, tendremos que insertarlos en forma de cadena.

```
ColeccionSortedNueva add: 'nombre apellido direccion y telefono'
```

Debemos pensar en que la entrada de los datos, nos resulta más fácil por medio de arreglos de cadenas (recordemos que los campos Nombre, Apellido y Dirección-Teléfono son de tipo cadena).

```
#('Apellido' 'Nombre' 'Direccion y Teléfono').
```

El código del ejemplo, nos enseña que los datos fueron precisamente insertados a una colección Sorted, en forma de arreglo de cadenas. Aquí tenemos el caso de una colección dentro de otra colección, que a su vez se encuentra dentro de una tercera. Los datos se insertaron aleatoriamente, divididos en 3 campos. La pregunta que podemos tener es que ¿Cómo funcionará la ordenación de los elementos, siendo que el Sort Block de SortedCollection no funciona sobre arreglos de cadenas?

La respuesta es la siguiente: No existen los métodos que nos permitan la comparación de arreglos de cadenas. Por consiguiente, tendremos que crear una nueva clase que soporte éstos métodos, o bien, podemos optar por una vía más fácil. Crearemos un Sort Block para la colección SortedCollection creada (Agenda2), que vaya comparando los elementos del arreglo, pues recordemos que un Sort Block tiene como entrada a 2 argumentos para su comparación:

```
Smalltalk at: #Agenda2
        put: (SortedCollection sortBlock:
           [:a :b | (a at:2) <= (b at:2)]).
```

en éste caso se eligió comparar el elemento número 2, pues por gusto particular ordenamos la agenda, por Nombres. Finalmente podemos observar la impresión de la Agenda, sobre la ventana del sistema System Transcript. La impresión se hace por Nombre, Apellido, y en otra línea, Dirección y Teléfono.

Por otra parte, es importante saber que podemos trabajar sobre archivos en el ambiente Smalltalk. Por ejemplo leamos un archivo del sistema y reportemos el número de veces que aparece en él cada letra del abecedario[6].

[6] Idea tomada de: Manual. "Tutorial and Programming Handbook", Xerox Corp., California, 1986, Pp. 84.

```
| archentr ocurrencias bolsa c d |
bolsa := Bag new.
archentr := Prompter prompt: 'Archivo a leer'
                default:''.
archentr := File pathName: archentr.
ocurrencias := WriteStream on: String new.
[archentr atEnd]
        whileFalse:[ c:= archentr next.
                (c isLetter)
                        ifTrue:[bolsa add: c asLowerCase]]
0 to 25 do:[:elem |
                d := (97+elem)
                (d = 111)
                        ifTrue:[ c := 164 asCharacteer.
                                ocurrencias
                                        cr; nextPut:c;space;
                                        nextPutAll:(
                                                bolsa occurrencesOf: c) printString.
                                        c := d asCharacter.
                                ocurrencias
                                        cr; nextPut:c;space;
                                        nextPutAll:(
                                                bolsa occurrencesOf: c) printString.
                        ifFalse:[ c := d asCharacter.
                                ocurrencias
                                        cr; nextPut:c;space;
                                        nextPutAll:(
                                                bolsa occurrencesOf: c) printString.
        ^ocurrencias contents.
```

En éste programa se utilizará un archivo cualquiera que pueda leerse para contabilizar las ocurrencias de cada letra del abecedario en él (incluyendo a la ñ), y se basa en una colección Bag para guardar el número de veces que aparece una letra. La variable temporal ocurrencias se inicializa a una cadena (String) nueva. La expresión:

```
WriteStream on:
```

utiliza el mensaje on: que se envía directamente a la clase WriteStream para actuar sobre un argumento tipo cadena y así sujetar una salida de edición (edited output), como lo serán cada letra del abecedario. Posteriormente al leer el archivo se verifica que mientras no sea fin de archivo (atEnd) se almacenen todos los caracteres que sean letras, en la bolsa (Bag).

Como sabemos, el valor ASCII de la letra **a** minúscula es el número 97, y es por ello que el siguiente pedazo de código se refiere a ir imprimiendo cada letra del abecedario con su respectiva ocurrencia en el archivo, aumentando el argumento del bloque elem de uno en uno hasta llegar a las 25 letras del abecedario. Ahora bien, para poder imprimir la letra **ñ** cuyo ASCII no es parte de ese rango de valores, se procedió a verificar que antes de la letra **o** minúscula se calculara la letra **ñ**, cuyo valor ASCII es el 164.

Al igual que en otro lenguajes de programación, Smalltalk soporta la "Recursión", como una poderosa técnica de programación. La recursión se basa en un algoritmo que se define en términos de si mismo. Veamos por ejemplo una aplicación clásica de la recursión: la sucesión de Fibonacci.

1, 1, 2, 3, 5, 8, 13, 21, ...

La lógica de la sucesión se basa en sumar dos números continuos, para obtener el término **n** siguiente:

(n-1) + (n-2)

excepto en los primeros 2 términos donde la suma siempre es 1.

Ahora bien, observamos que los valores de n para ésta sucesión, siempre deben ser enteros. Entonces tomaremos todas las características de un entero, de la clase Integer de Smalltalk/V, para crear una nueva función que calcule el correspondiente número en la serie, de un término n. Ver figura 4.9.

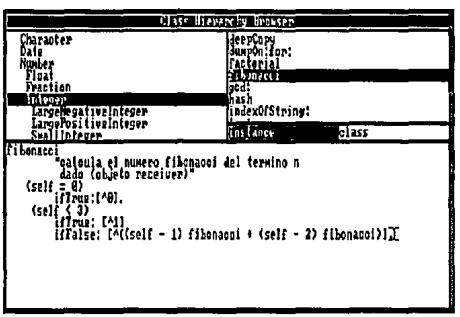


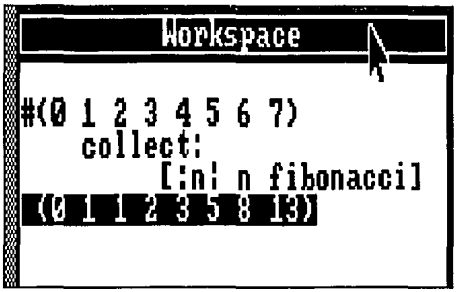
Figura 4.9 Código recursivo para el método de Fibonacci

Al referirnos al objeto receptor (self) como igual a cero, nos referimos al término 0, pues para calcular el término 1 sumaremos 0 + 1.

También podemos ver que en la expresión:

^self < 3

el resultado ejecutará cualquiera de los 2 bloques (ifTrue o ifFalse), dependiendo de esa evaluación. Si la variable self no es menor que 3, entonces aplicaremos la recursividad, retornando el valor del término indicado. Para observar como funciona el método, podemos aplicarlo a un arreglo de números enteros, con el uso del iterador collect:.



Por otro lado, en el primer capítulo de ésta tesis, se mencionó que Smalltalk tiene acceso directo sobre el lenguaje y manipulación de ciertos componentes del ambiente (como los menús que el sistema proporciona). Pues bien, para corroborar lo anterior modificaremos el menú de demostraciones gráficas que Smalltalk/V ya tiene definido. Para activar dicho menú, basta con seleccionar la opción "run demo" del Menú del Sistema (System Menu).

Podemos por ejemplo, añadir un método que dibuje el polígono con tres pétalos en forma de flor, que se programó en el subcapítulo anterior. Para ello, iremos a la clase DemoClass, que contiene todas la demostraciones de gráficos, y a la que añadiremos un nuevo método: "florTriangulos". Para mayor referencia sobre el código del método, ver figura 4.4 del subcapítulo anterior.

Posteriormente, procederemos a modificar el método demoMenu de la misma clase, añadiendo al menú de demostraciones, el método florTriangulos. Ver figura 4.10.

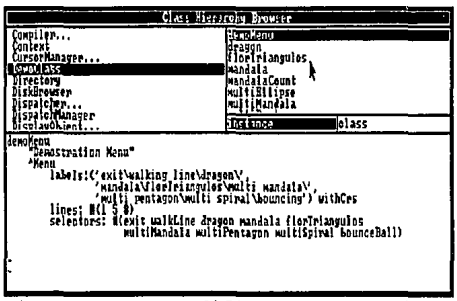


Figura 4.10 Modificando el menú de demostraciones gráficas

En éste método observamos que Menu es una variable que representa un menú con sus componentes. Para etiquetar las opciones, se utiliza el mensaje labels: en donde cada opción irá separada por un símbolo "\". Las líneas divisorias entre opciones que se observan en la figura anterior se definen

por un arreglo de números enteros, donde se indica en que renglón colocarla. Por último el mensaje selectors: (selectores) se basa también en un arreglo, pero de los métodos a ejecutar, coincidiendo su orden con los de las etiquetas definidas. Observemos en la figura 4.11, como quedará el menú modificado con el nuevo método.

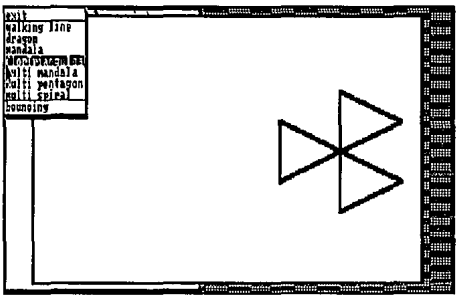
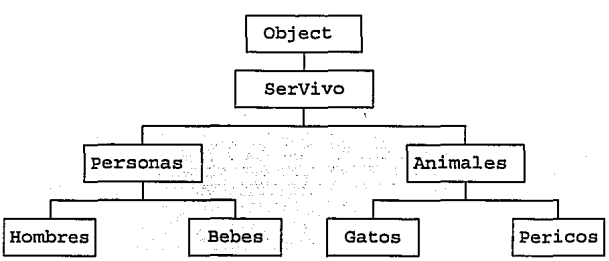


Figura 4.11 Nueva demostración gráfica

Veamos ahora otro ejemplo, basándonos en los tres capítulos anteriores, en donde aplicaremos los conceptos que delínean éste trabajo en el siguiente ejemplo de creación de nuevas clases, herencia y polimorfismo en Smalltalk/V.

Inicialmente lo que haremos es mostrar la aplicación de conceptos que hasta ahora hemos visto, en un programa que establecerá cierta comunicación entre algunos "seres vivos". Para ello definamos la siguiente jerarquía de clases:



y como siempre se define a Object como clase raíz en Smalltalk. Posteriormente se especifican las clases que formarán una jerarquía de seres vivos. En ella se definen las clases Personas y Animales, que como podemos ver, contienen la clasificación de personas (hombres y bebes) y de animales (gatos y pericos) con las que contaremos. Esta clasificación fue diseñada de ésta forma en particular, para poder mostrar los diferentes maneras de comunicación, que tienen dichos objetos.

Nota: En la clase "personas", la división entre hombres y bebés se refiere a la edad de cierta persona. Además la clasificación "hombres" incluye a hombres y mujeres.

Analicemos ahora las variables de instancia que requerirá cada clase en particular -Recordemos que dichas variables se heredarán en las diferentes subclases. Para ello, pensemos en que nuestro propósito es mostrar las diferentes maneras de comunicación, entre seres con diferentes características. Pensando en ello, enseguida se muestran las variables de instancia para cada clase:

```
SerVivo        => nombre, aptitud y edad
  Persona           => razon
    Hombre     => platica
    Bebe            => ()
  Animales          => ()
    Gato            => ()
    Perico          => palabras.
```

Las clases que tienen parénstesis () se refieren a clases que no tienen variables de instancia en sus definiciones; pero como sabemos la herencia de variables de instancia siempre se hace presente; por ejemplo la clase Gato cuenta con las siguientes variables de instancia: nombre, aptitud y edad.

Ahora, el siguiente paso es definir los métodos para sus clases por medio del browser de clases. Empezaremos por definir los métodos para la clase SeresVivos, que serán heredados a sus consiguientes subclases:

- Método nombre:

```
nombre: cadena
        "Asigna nombre al ser vivo, según su tipo.
         El objeto receptor es cualquier ser vivo"
        nombre := cadena.
```

éste método recibe como objeto argumento a una cadena de caracteres, y sirve para asignar un nombre a un objeto creado anteriormente como nuevo (new). Por ejemplo:

```
Raul := Hombre new.
```

- Método nombre

```
nombre
        "Retorna nombre. El objeto receptor
         es un ser vivo"
        ^nombre.
```

al imprimir un letrero de las palabras que indicará a cierto ser vivo, es necesario escribir su nombre, para saber quien habla. Por ejemplo:

```
Raul dice: Tal cosa.
```

Esto lo veremos más claro en los siguientes métodos que se definan.

El método para la clase Hombre es el siguiente:

- Método habla

```
habla
        "Método que tiene por objeto receptor a
         un hombre o mujer que entabla comunicación
         según el status de la variable platica"
        (platica isNil)
        ifTrue:[Transcript nextPutAll:
                self nombre,' dice: Hola Que tal?'; cr.
                platica := true]
        ifFalse:[Transcript nextPutAll:
                self nombre,' dice: Hola!!'; cr.
                platica := nil]
```

antes de hablar sobre las funciones del método, es importante saber que la clase Persona no tendrá métodos definidos en su clase, pues hereda los métodos de ser vivo, existiendo así una reutilización de código; de hecho la creación de ésta clase fue para hacer la separación con la clase Animales y hacer un englobe de características generales. Ahora bien, el método habla nos permite entablar comunicación entre hombres. Al evaluar la variable platica como nulo, se refiere a que si la variable no ha sido utilizada, tendrá valor de inicialización nulo y comenzará la platica con una frase como ésta: "Hola Que tal?" y la variable platica tendrá un valor diferente de nulo. Si la variable no es nula, quiere decir que ahora la contestación será diferente: "Hola!!". Por otro lado, el mensaje Trancript nextPutAll: escribe su argumento como cadena en la ventana Transcript.

Y el método para la clase Bebe es:

- Método habla

```
habla
        "El objeto receptor es un objeto tipo Bebe,
         para poder decir las palabras de un bebe"
        Transcript nextPutAll:
        self nombre,' dice: ggggu ggu';cr.
```

Ahora bien, para poder presentar los métodos definidos para la clase Animales, lo que haremos es mostrarlos de acuerdo a como los diseñamos, pues al hacer el análisis de que métodos debía llevar la clase, se fueron presentando detalles que debían tomarse en cuenta.

Primero se diseñaron los métodos para la clase Perico, como siguen:

- Método palabras:

```
"Definimos las palabras dentro del vocabulario
 del perico, el objeto receptor es un objeto perico"
palabras: cadena.
HablaMucho := true.
```

en éste método se definen las palabras del perico, como una cadena que se asigna a la variable de instancia palabras. En la última línea se asigna a la variable HablaMucho el valor de verdadero. Esta variable tenemos que agregarla a las definiciones de clase de la clase Perico, como variable de clase (auxiliar para el status de platica del perico). Para entender mejor esto analicemos el siguiente método

definido para la dicha clase.

- Método habla

```
habla
        "El perico (objeto receptor) habla
         palabras definidas por nosotros,
         según su status"
        (HablaMucho)
                ifTrue:[Transcript nextPutAll:
                        self nombre,' dice: ',palabras]
                ifFalse:[super habla]
```

si la variable HablaMucho se encuentra verdadera, entonces el perico hablará palabras. Esta variable de instancia, se recibe del método anterior como cadena de caracteres. Si HablaMucho es falsa, entonces se invocará a un método de la superclase (Animales) en la expresión:

```
super habla
```

recordemos que la variable **super** hace referencia a un método de la superclase, además de que actúa como objeto receptor del método. El método que definiremos entonces para la clase Animales es el siguiente.

-Método habla

```
habla
        "Método para que un animal (objeto receptor)
         perico o gato, no hable en general"
        Transcript nextPutAll:
        self nombre,' NO HABLA!!'.
```

puesto que en forma general un animal no habla, excepto el perico.

También podemos hacer que el perico no hable (cuando en el método habla de la clase Perico la variable HablaMucho es falsa) con el siguiente método.

- Método callado

```
callado
        "El perico (objeto receptor) ya no hablará
         más, a menos que se utilice el método palabras
         de la clase Perico"
        HablaMucho := false.
        Transcript nextPutAll:
        self nombre,' dice: Ya no hablaré mas!'.
```

en donde HablaMucho se asigna como falsa y el perico ya no hablará más, a menos de que se inicialice la variable de clase HablaMucho como verdadera utilizando el método palabras de la clase Perico.

Para ver cómo funcionan los métodos anteriores, probemos las siguientes instrucciones de la figura 4.12 en una ventana Workspace, para ver más claramente las salidas en la ventana Transcript.

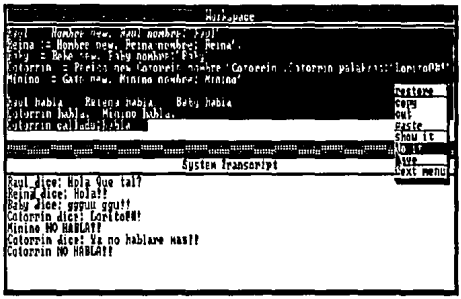


Figura 4.12 Comunicación entre seres vivos

En el primer pedazo de creación de los objetos:

Raul := Hombre new

debemos recordar que Hombre es por si mismo un objeto en el sistema Smalltalk. Cuando Smalltalk encontró las definiciones de clase para Hombre, se creó un objeto global llamado Hombre. Todas las clases saben como crear instancias de ellas mismas cuando se recibe el mensaje new, por lo tanto en la expresión anterior hemos creado un objeto (variable) global en el diccionario Smalltalk, llamado Raul (y lo mismo sucede con Bebe, Perico y Gato).

Por otra parte, si el polimorfismo se refiere a que diferentes objetos responden al mismo mensaje en forma diferente, entonces podemos observar en la figura anterior varios objetos polimórficos que responden de manera particular al mismo mensaje habla. Dichos objetos pertenecen a clases diferentes, pero tienen el mismo nombre del método "hablar" en su interfase pública.

En suma, por todo lo que hemos visto en éste capítulo, podemos mencionar a Smalltalk como un lenguaje simple, abstracto, seguro y de código pequeño (Small) en sus aplicaciones. También el número pequeño de conceptos que se manejan dentro de él, lo hacen una buena opción como lenguaje de programación.

Smalltalk demuestra su flexibilidad en la solución de problemas siendo aplicado hoy en día principalmente en problemas que incluyen la automatización de oficinas, los sistemas de información, el control de procesos, publicaciones y comerciales, la utilización de gráficos, programas académicos, juegos de computadora y los sistemas de simulaciones. Estos últimos, requieren de un mayor esfuerzo por parte del programador para, por una parte estudiar más a fondo los gráficos en Smalltalk pues en ellos podemos encontrar herramientas para poder realizar la simulación de X situaciones, y por otra dominar los conceptos de la simulación como tal.

CAPITULO V

"EL PAPEL DE SMALLTALK HOY EN DIA"

5.1 Versiones actuales en el mercado y algunos de los lenguajes más populares para la P.O.O.

CAPITULO V. EL PAPEL DE SMALLTALK HOY EN DIA.

5.1. Versiones actuales en el mercado y Algunos de los lenguajes más populares para la P.O.O.

Algunos de los programadores dentro del mundo de la computación, actualmente trabajan sobre la programación estructurada (basada en las estructuras de control) y/o la programación procedural (programación estructurada basada en módulos dentro de un solo programa llamados procedimientos), pues son las técnicas de programación que más se utilizaron en la ultima década.

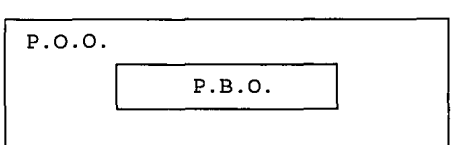
Posiblemente algunos de ellos (quizá pocos) seguirán por mucho tiempo programando con éstas técnicas, por ser ya de su dominio total, y por el miedo a un cambio de ideas.

La Programación Orientada a Objetos (P.O.O.) como tal sugiere un cambio en el estilo de programación para entender los procesos basados en la comunicación entre objetos.

Hoy en día, podemos hablar de una "moda" en ésta programación con objetos, y nos referimos así de su comercialización porque en la mayoría de las revistas de computación, se da cierta inclinación hacia este tipo de programación. Incluso ya existen varias revistas dedicadas de lleno al tema.

Por otro lado, existe una técnica llamada Programación Basada en Objetos (P.B.O.) que maneja objetos y abstracciones de datos. Sus características principales son: la construcción de datos y tipos definidos por el usuario, inicialización automática de los objetos y cierto mecanismos de acceso a los datos.

La diferencia que existe entre la P.B.O. y la P.O.O. es que ésta última se basa en la P.B.O. añadiendo a su estilo una organización de las clases por jerarquías y herencias, y una comunicación polimórfica a través de mensajes.



Debido a que en Smalltalk el sistema está diseñado como un objeto, y todos los objetos soportan herencia de métodos y polimorfismo, cualquier mínima expresión evaluada en éste lenguaje, cabe en el concepto orientado a objetos.

Recientemente (finales de 1991), Smalltalk ha sido liberado por medio de la versión Smalltalk/V Windows para operar bajo Microsoft Windows. Y un poco más recientemente para OS/2 Presentation Manager en Smalltalk/V PM, haciéndose casi indistinta la diferencia que existía entre Actor y Smalltalk. Cabe mencionar que Actor (quien opera bajo Microsoft Windows) es casi idéntico a Smalltalk, excepto en la sintaxis que se sigue en algunas instrucciones.

Si nosotros llegáramos a seleccionar Smalltalk bajo Microsoft Windows por ejemplo, sería porque entre nuestras necesidades requerimos trabajar bajo éste ambiente, pues podemos sacar las mismas ventajas tanto de Smalltalk/V como de la versión para Windows. De hecho, muchos expertos en el ramo recomiendan a los usuarios de computadoras personales trabajar con Smalltalk/V o su versión V286, puesto que nos resulta más económico al tener que adquirir un solo sistema.

Entre las principales características adicionales que tiene Smalltalk/V, tanto para Windows como para OS/2, podemos contar la capacidad para crear programas ejecutables (extensión .EXE, executable) y la portabilidad de código entre Smalltalk/V Windows y Smalltalk/V PM. Ambos productos son marca registrada de la compañía Digitalk Inc.

Actualmente varias compañías de software se esfuerzan por hacer compatibles sus productos con Smalltalk. Por ejemplo, la compañía Enfin de San Diego en cooperación con IBM han modificado las propiedades de su propio lenguaje para hacerlo compatible con Smalltalk. Este nuevo producto (ENFIN/2) provee al usuario poderosas herramientas visuales, para la creación de reportes financieros sobre grandes bases de datos, además de una librería multiusuario para la reutilización de código.

Otra de las compañías con éstas pretensiones es la llamada Servio Corp., que de igual forma ha creado compatibilidad entre Smalltalk y su sistema manejador de bases de datos. El nuevo producto renombrado como Smalltalk/GemStone, cuenta con un ambiente integrado que hace su uso de mucha facilidad al usuario.

Cabe mencionar también, que Digitalk Inc. anunció su próximo producto "Smalltalk/V for UNIX" con plataforma para el equipo RS/6000 de IBM.

Y hablando de lo último en programación con objetos, listaremos a continuación algunos de los productos en general, más populares que actualmente circulan en el mercado de la orientación a objetos:

- Borland C++ versión 3.0 (lenguaje orientado a objetos para trabajar bajo DOS con la librería Turbo Visión, o para Windows con ObjectWindows)
- OpenODB de Hewlett Packard Co. (Sistema manejador de bases de datos relacionales, orientado a objetos)
- Clipper versiones 5.0 en adelante de Nantucket Co. (Sistema manejador de bases de datos relacionales que incluye un compilador, con orientación a objetos)
- Turbo C++ for Windows de Borland (lenguaje orientado a objetos para trabajar bajo Microsoft Windows)
- Objectworks\C++ de ParcPlace Systems (programación orientada a objetos en C++ para trabajar con UNIX)
- Eiffel/S de SIG Computer de Alemania (versión de Eiffel para MS/DOS)
- Object-Oriented COBOL (Lenguaje de programación orientada a objetos, de tipo híbrido)
- Oregon C++ de Taumetric (Compilador que trabaja bajo múltiples plataformas UNIX)
- Liant C++ de Liant Software (al igual que el producto anterior, es un compilador que trabaja bajo múltiples plataformas UNIX)
- Objetive-C de The Stepstone Corp. (lenguaje de programación orientada a objetos)
- Actor 4.0 for Windows (lenguaje orientado a objetos que trabaja bajo Microsoft Windows)
- Microsoft C/C++ for Windows versión 7.0 (lenguaje orientado a objetos en su séptima versión por Microsoft Corp.)
- POET, siglas de Persistent Objects and Extended Database Technology de BKS software (manejador

de bases de datos que se basa en la estructura de definiciones de clase directamente de C++, y puede trabajar bajo ambientes que incluyen a MS/DOS, Windows y UNIX)
- Object-Oriented Turbo Pascal por Borland (lenguaje de programación orientada a objetos que puede correr en su versión 6.0 para MS/DOS y para Windows 3.0)
- DataFlex 3.0 de Data Access Corp. (Sistema manejador de bases de datos orientado a objetos)
- GeODE, siglas de GemStone Object Development Environment de Servio Corp. (lenguaje para el desarrollo de aplicaciones con bases de datos orientadas a objetos)
- etcétera.

A la información anterior, podemos agregar que C++ de Borland es el lenguaje de mayores ventas en el mercado (a principios de Noviembre de 1992 reportaba vendidas 500,000 unidades), seguido por Smalltalk/V y Objetive C, respectivamente.

Estos datos se muestran un tanto curiosos para el mundo de la computación. Según pruebas de laboratorio[1], se demostró que el lenguaje que a mayor número de aplicaciones se puede asignar es Smalltalk/V. Dichas pruebas consistieron en resolver ciertos problemas con algunos de los lenguajes orientados a objetos más utilizados en el mercado. Los resultados en esas pruebas para ciertos problemas en particular son los siguientes, de acuerdo al área de aplicación:

Lenguaje de P.O.O.	Area de Aplicación				
	Fabricac.	Ingenier.	Graficac.	Finanzas	Admón.
C++	X	X	X		
Objective-C		X			
Object Pascal				X	X
Actor					X
Smalltalk	X	X	X	X	X

Como podemos ver, el lenguaje que a mayor número de aplicaciones responde es Smalltalk; y entonces aquí surge una interrogante: ¿Porqué se percibe a C++ como el lenguaje más comercial y dominante en el mercado, y que de hecho es considerado como un estándar en el mundo de la orientación a objetos?

Para dar respuesta a ésta pregunta, existen varios factores que debemos tomar en cuenta dentro de la evolución de los lenguajes de programación, que han contribuido a ello.

Como sabemos, las primeras versiones de Smalltalk aunque toman rasgos de SIMULA, incrementan el número de elementos para proveer como objetivo principal, un ambiente de programación.

---

[1] Revista Object Magazine, "Development Environments", Número 2(1), Mayo-Junio 1992, Pp. 23, 24.

Pero la orientación del lenguaje que se presentó al mundo, fue la de un ambiente con propósitos académicos y de investigación[2]; lo cual hace que no se sitúe a Smalltalk como lenguaje para aplicaciones comerciales.

Otra de las causas, se deriva de que las primeras personas que adoptaron la tecnología orientada a objetos, era gente científica y técnica que se afirma utilizaba al lenguaje C en sus aplicaciones, y entonces al encontrar una opción de programación orientada a objetos, se inclinaron más fácilmente por C++.

Actualmente podemos ver en el mercado de la computación, la disposición que tiene C++ para funcionar bajo distintos ambientes. Esta característica, le permite al lenguaje tener una mayor publicidad, mayores vendedores, y obviamente mayor demanda del producto.

La publicidad que comenzó a hacer la firma Digitalk (productor de Smalltalk/V) sobre Smalltalk, fue mayor en comparación con el Smalltalk-80 y versiones anteriores de ParcPlace. Pero como no existe una total compatibilidad, ni escalabilidad entre el Smalltalk de las diferentes compañías, el producto sigue siendo promoción para un solo vendedor.

Con lo nuevo sobre Smalltalk (Smalltalk/V Windows y Smalltalk/V PM), se espera una mayor publicidad y demanda del producto, pues expertos en la materia consideran a Smalltalk como un poderoso enemigo de C++.

De hecho en la mayoría de los libros, conferencias y revistas que se han consultado para la realización de éste trabajo de investigación, existe una constante comparación entre los lenguajes Smalltalk y C++. Todos ellos coinciden en varios aspectos.

Uno de ellos es que la interactividad y conexión del usuario con el lenguaje, se percibe más fuertemente en Smalltalk que en C++.

En Smalltalk, como hemos estado viendo, existe todo un ambiente de interacción directa, a través de una interfase gráfica de usuario, mientras que en C++ existe una definición estricta de opciones, y archivos por separado para poder ejecutar un programa.

Por otra parte, la programación en pasos que se sigue en C++[3], no permite al usuario tener un acceso directo al ambiente de objetos. En Smalltalk es posible ejecutar Código de Smalltalk, sin necesidad de invocar a un archivo que nos permita compilar el programa. Como hemos visto en Smalltalk/V, el código que se desea compilar, debe ser salvado por medio del Browser de Clases, en donde se compilará e incrementará el ambiente[4]. Recordemos que Smalltalk cuenta con un intérprete y un compilador en un mismo ambiente.

---

[2] De hecho las primeras versiones de Smalltalk, se pretendía fueran utilizadas por niños.

[3] Editar, Compilar y Ligar.

[4] Para mayor referencia véase Capítulo I.

Abocándonos a algunos lenguajes orientados a objetos, podemos notar el gran peso que Smalltalk tiene sobre algunos de ellos.

Como ya hemos mencionado, la influencia que Smalltalk tiene sobre Actor por ejemplo, es evidente en la equivalencia de nombres y estructuras usadas para su jerarquía de clases. De hecho Actor se considera un lenguaje diseñado bajo la idea Smalltalk.

Podemos mostrar aquí la jerarquía de clases de Actor (Figura 5.1 para constatar que efectivamente Smalltalk influye directamente sobre éste lenguaje. Compárese ésta figura con la del capítulo II (2.2).

Object
- Association
- Debugger
- ErrorBox
- Menu
- MenuItem
- NilObject
- Point
- Primitive
- StopWatch
- System
- Application
  - ActorApp
- Behavior
  - Meta
- Collection
  - Bag
  - IndexedCollection
    - ByteCollection
      - String
        - Symbol
      - Struct
        - GraphicsObject
          - Rect
            - WinEllipse
          - WinPolygon
      - Interval
      - Array
        - Function
        - OrderedCollection
          - SortedCollection
          - TextCollection
  - KeyedCollection
    - Dictionary
      - MethodDictionary
      - IdentityDictionary
    - Set
      - SymbolTable
- File
  - TextFile
    - SourceFile
- Stream
- Magnitude
  - Char
  - Number
    - Int
    - Long
      - Date
        - Time
    - Real
- WindowsObject
  - Control
    - Edit
      - Button
    - ListBox
      - ScrollBar
    - Dialog
      - ClassDialog
      - DebugDialog
      - FileDialog
        - PrintDialog
      - InputDialog
        - Repleace Dialog

Figura 5.1 Librería de Clases de Actor

Actor es un lenguaje orientado a objetos que corre bajo Microsoft Windows, y de donde podemos observar una avanzada interfase gráfica de usuario (GUI).

La idea de las Interfases Gráficas de Usuario, inicialmente fue planeada por Alan Key a principios de los años 70's en su sistema Smalltalk-72. Esta interfase sirve de inspiración para el diseño de la

interfase gráfica de usuario que utiliza Macintosh de Apple, Windows de Microsoft y OS/2 Presentation Manager de IBM[5].

Incluso el propio sistema Microsoft Windows se basa directamente en la clase BitBlt (subclase de la clase Object, en Smalltalk/V) para el manejo de gráficos y transferencia de imágenes.

Lenguajes orientados a objetos que corren bajo Microsoft Windows como Turbo C++ para Windows ó C++ Views, proveen sofisticados ambientes de programación. También podemos hablar por ejemplo de conceptos tomados de Smalltalk como el de la librería de clases o la estructura de control de vistas (traslape de ventanas), la cual esta llegando a ser un poderoso modelo para la organización de programas.

Mencionaremos también que la mayoría de las librerías que C++ y Object Oriented Pascal (Pascal orientado a objetos) utilizan, se ven directamente influenciadas por la estructura de Smalltalk con respecto a las colecciones Bag, Set, OrderedCollection, Arrays, Associations y Dictionary.

Por otra parte, hoy en día existen las llamadas aplicaciones para trabajo con enmarcaciones (Application frameworks), que son tipos de librerías de clase que proveen el soporte de interfases por medio de ventanas y manejo de entradas y salidas por medio del ratón y del teclado. Estas aplicaciones automatizan el proceso de comunicación con el usuario haciendo fácil el manejo de vistas, menús y ventanas de diálogos (dialog boxes). Microsoft Windows por ejemplo, se clasifica a si mismo como una aplicación framework.

Las características de las aplicaciones framework incluyen una estructura con objetos. Aplicaciones framework son librerías de clase para facilitar el diseño de aplicaciones con interfases de usuario. Estas interfases pueden ser gráficas o de tipo texto con el manejo de ventanas.

Muy a menudo, este tipo de aplicaciones cuentan con selección en menús a base del ratón. Ejemplo de éste tipo de aplicaciones, son: Turbo Vision, C++ Views, ObjectWindows, Actor e incluso el propio Smalltalk.

Debido a que Smalltalk es un ambiente puro orientado a objetos, el mismo es considerado por los usuarios como lenguaje y aplicación framework que define su propia interfase gráfica de usuario. Turbo Vision por ejemplo es solamente una aplicación framework orientada a caracteres texto de Borland que corre bajo MS/DOS y que sirve de plataforma para trabajar en C++ o Pascal, mientras que C++ Views es una poderosa aplicación framework que trabaja bajo el ambiente Microsoft Windows para el lenguaje C++ exclusivamente. Con respecto a ObjectWindows es una aplicación framework que trabaja bajo Microsoft Windows y la utilizan lenguajes como C++ de Borland, Turbo Pascal o Actor.

Se afirma que el modelo para crear un sistema de aplicaciones framework, también estuvo inspirado en el diseño del lenguaje Smalltalk[6].

---

[5] VOSS, Greg. "Objetc-Oriented Programming, An Introduction", Editorial Osborne McGrawHill, California, 1991, Pp. 347.

[6] VOSS, Greg. "Objetc-Oriented Programming, An Introduction", Editorial Osborne McGrawHill, California, 1991, Pp. 348.

Así pues, podemos atrevernos aquí a recomendar ampliamente a Smalltalk como una opción para programar con objetos; y si bien es cierto que a los programadores que provienen de lenguajes procedurales, se les facilitará más el salto de ese tipo de programación a la programación orientada a objetos por medio de lenguajes híbridos, creemos que bien vale la pena un poco más de esfuerzo por aprender Smalltalk (y en especial Smalltalk/V para computadoras personales).

Hoy en día, podemos pensar por todo lo expuesto en éste trabajo de investigación, que muchos los programadores de hace ya tiempo, e incluso los novatos, estarán ya estudiando a la programación orientada a objetos como proyecto para sus nuevos programas. Así pues, el mayor atractivo que podemos encontrar en la programación orientada a objetos y específicamente en Smalltalk/V, es su aplicación sobre la solución de una gran variedad de problemas.

CONCLUSIONES

Muchas veces, cuando la complejidad del software que se está creando incrementa, la productividad del programador puede llegar a decrementar si el mantenimiento de éste software consume demasiado tiempo. La Tecnología Orientada a Objetos provee al respecto una solución.

La orientación a objetos busca minimizar el número de conexiones entre los componentes de un sistema, a través de la independencia de módulos (llamados objetos), que nos permiten crear software con menores posibilidades de error, y con una mayor facilidad de depuración.

También la programación orientada a objetos fomenta la organización dentro de los sistemas, identificando los rasgos comunes de un conjunto de objetos, y permitiendo la agrupación de éstos rasgos en clases dentro de una jerarquía. La programación jerárquica hereda estos rasgos comunes.

Actualmente existen en el mercado muchos productos que soportan una programación orientada a objetos que nos proporciona dichas ventajas, pero en éste mercado, Smalltalk es considerado como una de las mejores opciones de software orientado a objetos. De hecho se considera como uno de los lenguajes más potentes por su flexibilidad de programación y por ser un lenguaje puro que utiliza poco código en sus operaciones.

Una de las más importantes contribuciones de Smalltalk/V al mundo de la computación, se encuentra en su distintiva forma de programación en un ambiente integrado a través del manejo de ventanas, minimizando la complejidad para la solución de una gran variedad de problemas.

De hecho la programación orientada a objetos en Smalltalk/V busca una interacción "directa" con el usuario, para poder resolver éstos problemas. Las interfases gráficas de usuario, nos permiten un acceso sobre el lenguaje y una interacción de forma amigable; incluso, como ya hemos visto, en Smalltalk/V se permite una modificación del ambiente de programación, de acuerdo a nuestro propios requerimientos.

Smalltalk como pionero en éstas interfases a través del traslape de ventanas, sirve como modelo para mucho del software que se maneja en la actualidad. Y de hecho pretende, por su madurez como lenguaje, llegar a ser un estándar en el mercado de la programación orientada a objetos, además de poder ser un producto de múltiples vendedores; pues como vemos el lenguaje ahora se extiende para trabajar bajo sistemas altamente comerciales.

Por otro lado, hemos venido definiendo a la programación orientada a objetos como una técnica que reutiliza código; pero no podemos concebir a éste concepto sin una relación directa con la herencia. La herencia envuelve la clasificación de los objetos de acuerdo a sus propiedades. Esta herencia, se maneja de una forma más sencilla en Smalltalk/V, a través de los llamados browsers de clases.

Por ejemplo, usualmente se utiliza la ventana del browser de clases para programar dentro del sistema y heredar código de las librerías de clase que provee el sistema.

La mayoría de los ejemplos de aplicación de conceptos programados en ésta tesis, se basan directamente en la librería de clase de envase de Smalltalk/V, aunque en el cuarto capítulo, podemos ver

que se añaden algunas clases por nuestra cuenta a esa librería.

Como sabemos, la comunicación directa con los objetos, se basa en mensajes. Una crucial propiedad de los mensajes es que ellos son la única manera de invocar las operaciones de los objetos.

Orientación a Objetos envuelve programar enviando mensajes polimórficos a los diferentes tipos de objetos. Aún cuando los objetos polimórficos sean de diferentes tipos, ellos tienen ciertas propiedades comunes (nombres de métodos iguales). Si objetos de diferentes tipos no tuviesen éstos rasgos comunes, no podría existir entonces el polimorfismo, pues los objetos no responderían en forma diferente a los mismos mensajes.

Por lo tanto, para poder soportar un polimorfismo es necesario clasificar de alguna manera a los objetos de acuerdo a sus propiedades, y tener un protocolo de comunicación. Aunado a esto, la encapsulación en Smalltalk nos proporciona una alta seguridad en los datos, contra modificaciones no deseadas en el flujo de datos.

En suma, para poder hablar de una programación orientada a objetos, es necesario conjuntar la encapsulación, herencia y polimorfismo en un mismo sistema. Los lenguajes orientados a objetos de tipo híbrido permiten la utilización de estas propiedades, pero en los lenguajes puros como Smalltalk/V, estas características son nativas del lenguaje, por lo que la programación aquí lleva implícita su utilización.

BIBLIOGRAFIA

- GOLDBERG, Adele y Robson, David. "Smalltalk-80 The Language", Editorial Addison Wesley. Nueva York, Septiembre 1989. 585 ps.

- GRADY, Booch. "Transactions on Software Engineering", Volúmen SE-12. California, Febrero 1986. 320 ps.

- J. MACLENNAN, Bruce. "Principles of programming languajes: Design, Evaluation, and Implementation", Editorial Holt, Rinehart and Winston, Segunda Edición. Florida, 1987. 568 ps.

- LOPEZ de Medrano, Santiago. "Teoría de Gráficas", Editorial Diseño y Composición litográfica. Edo. México, Agosto 1993. 59 ps.

- Manual de Smalltalk/V. "Tutorial and programming hadnbook", Digitalk Inc. California, Julio 1986. 515 ps.

- Periódico EXCELSIOR. "Objetos", Sección financiera (sección computación). México D:F:, 20 septiembre 1993. Año LXXVII, Tomo V, Número 27834. 20 ps.

- Revista BYTE. "The Small System Smalltalk", Publicaciones McGrraw-Hill. Nueva York, Agosto 1981. Volúmen 6, Número 8. 396 ps.

- Revista Computer Language. "Flying high with Objects", Publicaciones Miller Freeman. San Francisco, Octubre 1990. Volúmen 7, Número 10. 144 ps.

- Revista Object Magazine. "Bringing your objects to life", Publicaciones SIGS. Nueva York, Julio-Agosto 1992. Número 2(2). 81 ps.

- Revista Object Magazine. "Connecting you now", Publicaciones SIGS. Nueva York, Enero-Febrero 1993. Número 2(5). 97 ps.

- Revista Object Magazine. "Development Environments", Publicaciones SIGS. Nueva York, Mayo-Junio 1992. Número 2(1). 89 ps.

- Revista Object Magazine. "GUI Magic", Publicaciones SIGS. Nueva York, Enero-Febrero 1992. Número 1(4). 81 ps.

- Revista Object Magazine. "Methodologies to your madness", Publicaciones SIGS. Nueva York, Noviembre-Diciembre 1992. Número 2(4). 105 ps.

- Revista Object Magazine. "Objectifying the future", Publicaciones SIGS. Nueva York, Septiembre-octubre 1992. Número 2(3). 105 ps.

- Revista Object Magazine. "Objects in Harmony", Publicaciones SIGS. Nueva York, Marzo-Abril 1992. Número 1(6). 88 ps.

- Revista RED. "Los gráficos en la computación", Mayo 1993. Año 11 Número 21. 89 ps.

- Suplemento Especial de Broland. "Happy 25th Anniversary Objects", Publicaciones SIGS. Nueva York, Febrero 1992. 16 ps.

- VOSS Greg. "Object-Oriented Programming an Introduction", Editorial Osborne McGraw-Hill. California, 1991. 584 ps.