UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

PROGRAMACION DE LOS CONCEPTOS DEL PARADIGMA ORIENTADO A OBJETOS EN SMALLTALK

T E S I S

QUE PARA OBTENER EL TITULO DE:

M A T E M A T I C O

P R E S E N T A :

MOISES BAUTISTA OSORNO

DIRECTOR DE TESIS: DRA. AMPARO LOPEZ GAONA

1999

VNIVERSIDAD NACIONAL
AVFNºMA DE
MEXICO

MAT. MARGARITA ELVIRA CHÁVEZ CANO
Jefa de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo de Tesis:

"Programación de los conceptos del paradigma orientado a objetos en Smalltalk"

realizado por Moisés Bautista Osorno

con número de cuenta 8528463-6 , pasante de la carrera de Matemáticas

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	Dra. Amparo López Gaona	
Propietario	M. en C. Gustavo Márquez Flores	
Propietario	M. en C. Juan Jesús Gutiérrez García	
Suplente	Mat. Ana Luisa Solís González Cosío	
Suplente	M. en C. Guadalupe Ibargüengoitia González	

Consejo Departamental de Matemáticas

A Olivia, para quien no existe un lenguaje que la describa.

A mis padres, quienes nos esperan en alguna parte de Veracruz.

A mi tía Mago, por sus comidas y su amor incondicional.

A Maru y Alejo, quienes siempre me apoyaron y aguantaron.

Agradecimientos

A mis hermanos: Beatriz, Marcela, Pedro y Pablo.

A mi otra familia: Jesús, Tere, Diana, Tania y Don Chucho.

A mis amigos: Gilberto, Leobardo, Alex, Gabriel, Olivia y Oscar, Víctor y Maru, Rebeca, Toño, ...

Quiero externar un sincero agradecimiento a mis sinodales, quienes participaron en la revisión de este trabajo:

M. en C. Guadalupe Ibargüengoitia González

Mat. Ana Luisa Solís González Cosío

M. en C. Gustavo Márquez Flores

M. en C. J. Jesús Gutiérrez García.

Un agradecimiento especial para mi directora de tesis, Dra. Amparo López Gaona y para el M. en C. J. Jesús Gutiérrez García por su ayuda y paciencia, sin las cuales no hubiera sido posible la conclusión de este trabajo.

I N D I C E

Introducción:

En la presente década el paradigma de programación que se ha impuesto es el conocido como Orientado a Objetos (OO), de ahí que resulte importante ahondar en los orígenes, fundamentos, lenguajes y conceptos alrededor del mismo.

Entre el paradigma OO y el lenguaje de programación llamado Smalltalk existe un enlace muy fuerte, el paralelismo en el desarrollo de ambos es notable, por lo que se pueden ejemplificar conceptos de uno por medio de las características del otro.

El presente trabajo servirá a todos los interesados en el desarrollo de los diferentes paradigmas de programación, pero en particular, resultará de mayor relevancia para aquellos que deseen familiarizarse con el paradigma OO.

El trabajo resulta ser una referencia introductoria en dos formas: la primera hacia el paradigma OO, y la segunda sobre Smalltalk, ya que en nuestro país poca gente conoce y todavía menos programa en él. También es importante realzar la belleza y elegancia de este lenguaje de programación, que por razones de carácter comercial no ha tenido la difusión que debiera, únicamente es conocido en el ambiente académico.

El objetivo fundamental y más ambicioso que se desea alcanzar es despertar la motivación y el interés del lector sobre esta área de los lenguajes de programación, para ello se le familiariza con los conceptos del paradigma OO. La manera de presentarlos es muy natural, ya que la filosofía OO resulta ser transparente con el entorno que nos rodea. También se pretende mostrar cómo dichos conceptos son representados de manera directa mediante Smalltalk.

El trabajo consta de 5 capítulos, los cuales son:

Capítulo 1	Paradigmas de programación
Capítulo 2	Conceptos del paradigma OO
Capítulo 3	Interfaz y lenguaje de Smalltalk
Capítulo 4	Implementación en Smalltalk
Capítulo 5	Uniendo todo.

El capítulo 1 proporciona un marco histórico en el desarrollo y evolución de los principales paradigmas de programación, los cuales permiten enmarcar los lenguajes más representativo. El capítulo aborda diferentes paradigmas pero pone especial énfasis en el paradigma Orientado a Objetos. En el capítulo 2 se estudia el marco conceptual del paradigma OO mientras que el capítulo 3 introduce el

medio ambiente del lenguaje de programación Smalltalk. Con la finalidad de ilustrar los elementos y conceptos del paradigma OO así como la implementación de ellos en dicho lenguaje, el capítulo 4 muestra la manera de codificar en Smalltalk dos problemas: el problema de hallar la ruta más corta en una red de nodos y la codificación del tipo de dato abstracto árbol. El 5o. y último capítulo conjunta lo expuesto en los capítulos previos, mediante la puesta en práctica de los dos problemas mencionados. Asimismo, provee de más detalles relativos de la codificación en Smalltalk. Finalmente se concluye con una explicación sobre aspectos técnicos en la creación de aplicaciones independientes del medio ambiente de Smalltalk.

Capítulo 1

Paradigmas de Programación

1.1 Antecedentes

La historia que aquí se desarrolla es la historia de los paradigmas de programación en general y en particular del Orientado a Objetos (OO). La palabra paradigma (Latín *paradígma*, Griego *paradeigma*) originalmente significa un ejemplo ilustrativo (Budd, 1991), pero desde la perspectiva que aquí se usa se dirá que *un paradigma es un conjunto de teorías y métodos que representan en conjunto una forma de conocimiento organizado* (Kuhn, 1970). Un paradigma de programación es un estilo o forma de programar, dentro de los paradigmas más importantes están el lógico, el funcional, el distribuido, el imperativo, y desde luego el OO (orientado a objetos). Todos estos descansan en una *filosofía particular*; esto quiere decir que cada paradigma ejecuta el cómputo de manera particular. Los paradigmas establecen condiciones para distinguir entre los lenguajes que pertenecen al mismo de los que no, esto es, existen criterios que definen condiciones paradigmáticas tales como estructura del programa, estructura del estado y metodología.

Algunos paradigmas retoman conceptos de otros y agregan conceptos nuevos, es así como surgen nuevos paradigmas y también nuevos lenguajes. A continuación se exponen los paradigmas de programación más importantes (fig. 1.1) y se habla, en cada caso, de algunos de sus lenguajes de programación más representativos.

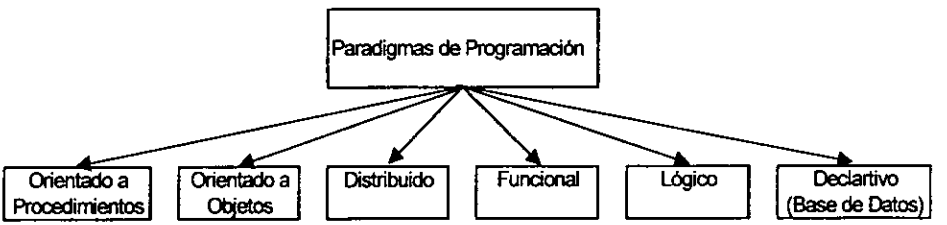


Fig. 1.1. Principales paradigmas de programación.(OOPSLA/ECOOP 90)

1.2 Paradigmas de programación

- **Paradigma orientado a procedimientos**

Lenguajes representativos: **Algol, Pascal, PL/I, Ada, Modula, C, FORTRAN, COBOL**

El paradigma orientado a procedimientos fue el paradigma primario en los 50´s, 60's y 70's. Bajo este éste, el programa es un conjunto de bloques y procedimientos anidados por lo que los programas se descomponen en subtareas, cada una de las cuales realiza operaciones específicas. Las rutinas o procedimientos se utilizan como unidades modulares para definir las subtareas. La parte principal se encuentra en el diseño del procedimiento, el algoritmo necesario para realizar la tarea deseada. La Programación con el uso de procedimientos crea orden y coherencia en el conjunto de algoritmos.

La literatura acerca de este paradigma está cubierta con discusiones sobre cómo pasar argumentos y de cómo distinguir diferentes tipos de funciones y argumentos. FORTRAN es el lenguaje que introdujo el concepto de procedimientos; ALGOL-60, C y PASCAL son invenciones sucesoras de la misma tradición.

Los primeros lenguajes de este paradigma emplearon un vocabulario netamente matemático por lo que dichos lenguajes se emplearon en su totalidad para aplicaciones científicas o ingenieriles. Esto marcó un paso que acercó al usuario hacia el dominio del problema y lo alejó más del lenguaje de máquina.

Posteriormente, la atención recayó en las abstracciones algorítmicas, aquí se enfocó el problema de cómo decirle a la máquina lo que tenía que hacer. Muchas tareas se automatizaron, principalmente en los negocios, y se insistió en acercarse más al dominio del problema que al lenguaje de máquina. Se hizo énfasis en el concepto de "subprogramas", los programas escritos en estos primeros lenguajes, tenían el gran inconveniente de que un error en alguna parte del programa podía afectar devastadoramente todo el programa; esto debido a la exposición de los datos a la vista de todos los subprogramas. Se desarrollaron muchos conceptos básicos como declaraciones, listas, pilas, arreglos, expresiones aritméticas y subrutinas.

Las nuevas tecnologías a principios de los 60's incrementaron la capacidad del hardware hasta entonces disponible, lo que conllevó a una mayor flexibilidad en el manejo de diferentes tipos de datos. Lenguajes como ALGOL 60 y Pascal aportaron la abstracción de datos.
En los 70's hubo un auge muy importante en cuanto a la investigación de lenguajes de programación, a tal grado que en su momento llegaron a existir un par de miles de lenguajes y sus respectivos dialectos. El desarrollo del concepto de subprograma tuvo tres importantes consecuencias:

1. Los lenguajes que se crearon proporcionaron una variedad de mecanismos de paso de parámetros.
2. Se facilitó aún más el anidamiento de subprogramas, así como el desarrollo de teorías con relación a las estructuras de control y al alcance y visibilidad de las declaraciones hechas.
3. Surgieron métodos de diseño estructurado.

Finalmente, otro mecanismo importante hizo su aparición: el concepto de modularidad. Los proyectos de desarrollo largo implicaban la formación de un equipo de desarrollo, y con esto algunas partes del proyecto tenían que desarrollarse independientemente, esto es, se requería una estructura modular. Cada módulo puede ser compilado por separado, algunos lenguajes facilitaban algún tipo de estructura modular, pero tenían poca consistencia semántica para la interacción entre módulos.

El concepto de módulo permitió "encapsular" y "ocultar" código y datos, lo que solucionó el problema de exposición de datos y código que se tenía con los primeros lenguajes.
Con respecto a los lenguajes, FORTRAN continúa siendo el de mayor uso para cálculos numéricos, COBOL cuenta con un gran número de aplicaciones, Pascal ha hecho una enorme contribución en el ámbito académico y C ha influido directamente sobre UNIX y C++.

- **Paradigma distribuido**

Lenguajes representativos: **CSP, Argus, Actor, Linda, Monitors**

Bajo este paradigma, los programas se ocupan de llevar a cabo tareas múltiples, sincronización y comunicación.
El Concepto fundamental de la programación distribuida es el de "proceso". La interacción entre procesos tiene dos formatos: la comunicación y la sincronización.

1. La "comunicación", que provoca el intercambio de datos entre procesos se da por medio de un mensaje explícito o a través de valores de variables compartidas. Una variable es "compartida" si su código es visible para los procesos que la comparten.
2. La "sincronización" relaciona los enlaces de un proceso con los de otro.

La necesidad de comunicación y sincronización puede verse en términos de competencia y cooperación entre procesos. La competencia ocurre cuando los procesos requieren el uso exclusivo de algún recurso, por ejemplo cuando los procesos compiten por usar la misma impresora, la sincronización es necesaria para poder otorgar a un proceso la exclusividad en el uso de un recurso.

La concurrencia es un concepto que aparece en diferentes ámbitos: en arquitectura de máquinas, sistemas operativos, bases de datos y en particular en

sistemas distribuidos. Los programas consisten de un cierto número de unidades, llamadas *procesos* las cuales se ejecutan en paralelo (lógica o físicamente). Los lenguajes de programación deberán proveer de declaraciones de sincronización, lo anterior con el propósito de asegurar la correcta interacción entre procesos.

- **Paradigma de programación funcional**

Lenguajes representativos: **LISP, FP, MIRANDA, ML, Haskell**

Resulta ser un paradigma sin efectos laterales que incorpora los conceptos de funciones de primera clase y evaluación floja.

El estilo de programación funcional tiene sus raíces en la teoría de funciones matemáticas. Hace énfasis en el cómputo de *valores* mediante el uso de *expresiones* y *funciones*. Las funciones son los bloques primarios de construcción para un programa; estas pueden pasarse libremente como parámetros y pueden construirse y regresarse como parámetros para otras funciones. Un impacto fuerte de la programación funcional reside en la eliminación de efectos laterales.
Otra característica de los lenguajes de programación funcional, es que los usuarios no tienen porque preocuparse de la manipulación de la memoria para el almacenamiento de los datos, en este paradigma se trata a las funciones como *elementos de primer orden*. Las funciones tienen el mismo status que cualquier otro valor.

La ejecución de programas bajo este paradigma se basa en dos mecanismos fundamentales: aplicación y ligado. La *aplicación* de una función se utiliza para computar nuevos valores. El *ligado* es utilizado para asociar valores con nombres. Tanto datos como funciones pueden ser usados como valores. Una contribución importante de estos lenguajes es el hecho de mostrar que los lenguajes de programación pueden ser poderosos en la computabilidad además de proveer la capacidad de evitar ciertos errores antes de llegar a la ejecución del programa.

Una herramienta que es útil para el modelado preciso del comportamiento de funciones por medio de aplicación y ligado, es el cálculo lambda. LISP fue el primer representante de este paradigma, la familia de lenguajes LISP es grande y popular La llamada "recolección de basura", fue creada para manejar demandas pesadas de LISP en materia de asignación dinámica de memoria. CLOS es una extensión OO de LISP, el cual provee de clases, funciones genéricas y herencia múltiple. ML es un miembro más reciente de la familia de lenguajes de programación funcional, el cual aporta control fuerte de tipos al paradigma.

- **Paradigma de programación lógico o declarativo**

Lenguajes representativos: **Prolog, Concurrent Prolog, GHC, Vulcan, Polka**

Los programas están conformados de relaciones, variables lógicas y unificación.

El concepto de programación lógica está ligado históricamente al lenguaje de programación Prolog, que fue desarrollado en 1972, fue aplicado en primera instancia para el procesamiento de lenguaje natural. También ha sido usado para especificar algoritmos, bases de datos de búsqueda, escribir compiladores, construir sistemas expertos, es decir todas las aplicaciones en las cuales LISP ha sido y puede ser usado. Prolog está ligado con aplicaciones que involucran coincidencia de patrones, búsqueda con mecanismos de backtrack o con información incompleta. La programación manipula relaciones antes que funciones, y se basa en la premisa de que la programación con relaciones es más flexible que la programación con funciones (Sethi, 1990 y Ghezzi, 1998).

En programación lógica, la descripción de los problemas está dada dentro de un formalismo lógico, el cual se basa en el cálculo de predicados de primer orden. Es por esto último que a la programación lógica a menudo se le dice paradigma de programación *declarativo*.

El bloque básico para construir programas en Prolog es el término. Un término es una constante, una variable, o un término compuesto. Un término compuesto es escrito como un símbolo de función ("functor") seguido por uno o más argumentos entre paréntesis, los cuales son a la vez términos. Hechos y reglas son utilizados para proveer cierta especificación declarativa de un dominio particular de conocimiento. Dado un conjunto de hechos y reglas, podemos resolver un problema haciendo consultas de dicho conjunto *(query)*, la consulta puede verse como una *meta* (*goal*), la cual debe ser alcanzada.

- **Paradigma declarativo (base de datos)**

Lenguajes: **SQL, Ingres, Encore, Gemstone, O2**

Bajo este paradigma los programas manejan persistencia de datos, manipulación de datos, control de concurrencia (Grady, 1991). El programador declara lo *que* desea, y no se preocupa de *cómo* realizar el cómputo.

La base de este paradigma recae en los llamados Sistemas Manejadores de Bases de Datos (con siglas en inglés DBMS). Estos son sistemas de software centralizados o distribuidos, los cuales ofrecen capacidades para definir bases de datos, búsqueda y almacenamiento de información entre otras cosas. Estas capacidades pueden obtenerse de manera interactiva o vía algún lenguaje de programación.

Los primeros manejadores se crearon bajo el modelo jerárquico. La siguiente generación de estos, trajo consigo el modelo relacional a principios de los 70´s. Cabe mencionar que las bases de datos relacionales han sido instaladas en prácticamente cualquier equipo de cómputo. El diseño simple en los mecanismos de abstracción del modelo relacional ha dado lugar al desarrollo de los llamados

lenguajes de consulta (Simple Query Languaje). Ejemplos de tales lenguajes son: SQL, QUEL (Ingres) y QBE (desarrollado por IBM).

Los manejadores relacionales han contribuido considerablemente al desarrollo de la tecnología de base de datos.
Los manejadores se caracterizan por el "modelo de datos" que utilizan. Por ejemplo, el modelo relacional está basado en una simple estructura de datos: la relación. Una relación puede verse como una tabla con renglones (eneadas) y columnas (atributos) que contienen cierto tipo de datos, que pueden ser: enteros, caracteres, números, imágenes, gráficos, etc.
Las tendencias que los manejadores de bases de datos han seguido, tienen diversas vertientes, por lo cual existen diferentes tipos de sistemas.

- **Sistemas relacionales extendidos**.- La tendencia consiste en extender los manejadores relacionales para que manipulen objetos más complejos.
- **Sistemas manejadores OO**.- Estos sistemas combinan la tecnología OO con la de bases de datos.
- **Sistemas manejadores deductivos**.- Aquí se combinan la tecnología de bases de datos con la programación lógica.
- **Sistemas manejadores "inteligentes"**.- Extienden la tecnología de base de datos, incorporando paradigmas y técnicas desarrolladas en el campo de la inteligencia artificial.

- **Paradigmas basado en objetos y orientado a objetos**

Lenguajes representativos: **Ada, Modula, Simula, Smalltalk, C++, Eiffel, Flavors, CLOS**

El programa es una colección de objetos que interactúan. Simula 67 proporcionó los fundamentos para la creación de Smalltalk, y este último a su vez, dio origen a muchos lenguajes en los 80's.
El programa, el estado y la computación son abstracciones complementarias que capturan la visión del diseñador del lenguaje, el implementador y el agente ejecutor respectivamente. Son tres formas de ver la solución del problema para definir distintas clases de lenguajes. Los paradigmas robustos, como el OO, se pueden definir intercambiablemente por estructura de programa, estado o computación.

En contraste al modelo de memoria compartida (los datos están expuestos de manera global) de la programación orientada a procedimientos, la POO se particiona en pedazos encapsulados, cada uno asociado a una máquina virtual, autónoma y potencialmente concurrente.

En una arquitectura de memoria compartida las acciones individuales, incluyendo los procedimientos, comparten un estado desprotegido y global; es responsabilidad de los procedimientos asegurarse de que los datos sean

accesados en una forma autorizada, los procesos deben asumir la responsabilidad para sincronizarse y accesar los datos compartidos.

La partición que hace el paradigma OO, se asocia con objetos, donde cada pedazo es responsable de su propia protección en vez de accesar operaciones no autorizadas. En un medio ambiente concurrente, los objetos se protegen ellos mismos en vez de accesar sincronizadamente.

Particionando en pedazos encapsulados se está definiendo la característica del paradigma distribuido; aquellos paradigmas basados en objetos son distribuidos lógicamente.

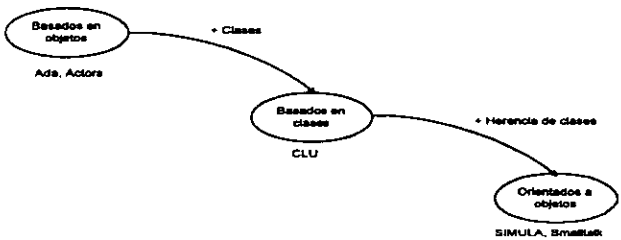


Fig. 1.2. Diferencia de atributos entre los lenguajes de programación basados y orientados a objetos.(OOPSLA/ECOOP 90)

Los programas OO están esencialmente conformados de módulos, los cuales agrupan clases y objetos en lugar de subprogramas, como se acostumbraba en los lenguajes pioneros. Esto podría resumirse como sigue: *si a los procedimientos se les llaman verbos y a los datos pronombres; un programa orientado a procedimientos se desarrolla alrededor de los verbos, mientras que los programas OO se desarrollan alrededor de los pronombres* (Grady, 1991).

A pesar de que los lenguajes basados en objetos, basados en clases y orientados a objetos proveen el marco conceptual OO tienen grandes diferencias ya que por ejemplo, los basados en objetos no proveen el concepto de clase y por tanto el de herencia y los lenguajes basados en clases no proveen de herencia, ver (figura 1.2)

Los paradigmas mencionados no son mutuamente excluyentes; por ejemplo Ada es tanto estructurado por bloques como basado en objetos, Concurrent Prolog es un lenguaje de programación tanto lógico como concurrente. Otro ejemplo de lenguaje híbrido es C++, ya que se forma de los paradigmas orientado al procedimiento y OO. A los sistemas que facilitan estilos de programación en más de un paradigma se les llama sistemas multiparadigma.

Existe una distinción que cabe señalar: un lenguaje proporciona un estilo de programación si provee facilidades de seguridad y eficiencia que lo hacen conveniente para su uso (razonablemente fácil). Se dice que un lenguaje no facilita o no soporta una técnica si le toma un esfuerzo excepcional escribir tales

programas. Cuando se habla de soporte para un paradigma no solo se habla de la forma obvia de facilidades que le permiten usar directamente el paradigma; sino también de sutilezas como el control de los tiempos de compilación y de ejecución, control de tipos, detección de ambigüedades (soporte lingüístico) y de facilidades extralingüísticas tales como bibliotecas y medio ambiente.

Un lenguaje no necesariamente es mejor que otro por que tenga una característica que el otro no tiene, la cuestión importante no es cuántas características tiene un lenguaje, sino que las características con las que cuenta sean suficientes para facilitar el estilo de programación deseado en el área de aplicación. Con respecto a las características, es importante que:

A. Todas las características sean claras y elegantemente integradas en el lenguaje.
B. Combinar características para resolver situaciones que en otro caso podrían requerir características extras por separado.
C. Exista el menor número de características especiales.
D. Elaborar una característica no impone significado por encima de programas que no lo requieran.
E. El programador no necesita conocer todo el lenguaje para escribir un programa.

Historia de los lenguajes OO

El primer ancestro de la POO es el lenguaje de programación SIMULA (véase fig. 1.3 la cual muestra una cronología de la aparición de los lenguajes más populares). SIMULA nació en Noruega, el proyecto estuvo comandado por Dahl y Ngaard, se buscaba modificar los lenguajes entonces disponibles, los cuales estaban diseñados para aplicaciones numéricas.

El objetivo era transformarlos para poder programar simulaciones discretas de problemas del mundo real, tales problemas se conformaban por objetos que cooperaban o interactuaban entre sí en una forma variada. SIMULA nació antes de su tiempo, y en un país chico, sin un gran apoyo de la industria de las computadoras y alejado de los grandes centros de investigación mundial.

La primera aparición de la noción de "objeto" vino con SIMULA, esto como consecuencia natural de modelar directamente los objetos de una simulación como *objetos* de software. Resultó sorprendente el hecho de que objetos de software fueran útiles no sólo para programación de simulaciones, sino también para prototipos y desarrollo de aplicaciones.

SMALLTALK tuvo una suerte muy distinta a la de SIMULA, éste llegó a la POO como un modelo pulido, y en contraste con SIMULA, evolucionó a través de varias versiones en los 70's, hasta llegar a SMALLTALK 80, el cual se hizo popular en

las estaciones de trabajo a fines de los 80's, esto como resultado de un modelo de interfaces bien diseñadas, y una promoción agresiva entre otras cosas. La idea implícita, incluyendo el término *Orientado a Objetos* (OO) provino de SMALLTALK. Dicho lenguaje es uno de los más fuertes representantes de la POO en el sentido de que se apega más al paradigma OO. La POO se originó cuando SMALLTALK se desarrolló y la historia de SMALLTALK sirve para mostrar la historia de la POO (Grady, 1991). La popularidad de la POO se debió al éxito alcanzado por Smalltalk a finales de los 70's y, en particular a la implementación de la versión OO de C, C++.. Lo anterior permitió que un gran número de programadores migraran de un paradigma orientado a procedimientos a uno orientado a objetos.

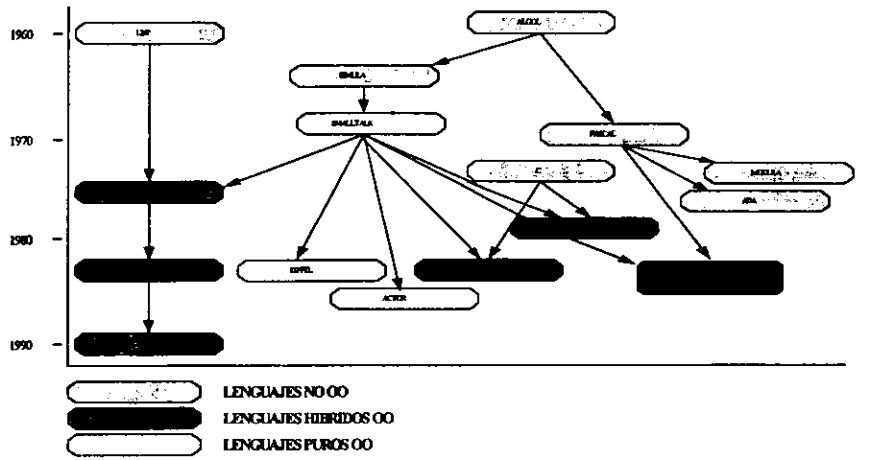


Fig. 1.3. Cronología y consecución de la aparición de algunos de los lenguajes más populares y representativos.

Programación OO

La POO puede ser definida como: "Un método de implementación en el cual los programas están organizados como un conjunto de objetos que cooperan entre sí, cada uno de los cuales representa una instancia de alguna clase, y todas las clases son miembros de alguna jerarquía de clases, esta se desprende de las asociaciones de herencia" (Grady, 1991).

Se pueden señalar tres aspectos de esta definición (1) se usan "objetos" y no algoritmos como la esencia del paradigma, esta es la llamada jerarquía *"parte de"*; (2) cada objeto es una instancia de alguna clase; (3) las clases se relacionan entre sí por medio del mecanismo de herencia o la llamada jerarquía *"tipo de"*.

Se dice que un lenguaje es OO si satisface las siguientes condiciones:

- Facilita que los objetos sean abstracciones de datos, con un protocolo de operaciones declaradas y un estado de ocultamiento local.
- Los objetos tienen un tipo asociado (clase).
- Los tipos (clases) pueden heredar atributos de super tipos (super clases).

Partiendo de estas características, un lenguaje que no provea asociaciones "*tipo de*" o lo que es lo mismo, herencia, se le llamará solamente "basado en objetos", de aquí que Smalltalk, Objective Pascal, C++ y CLOS sean OO (ver figuras 1.2. y 1.3), mientras que Ada es solamente basado en objetos; ambos tipos de lenguajes proveen de clases y objetos.

Smalltalk adopta de entrada la definición de un lenguaje POO, como uno que provee los conceptos de clase y herencia. Desde un punto de vista parece restrictiva la definición, ya que puede haber muchos argumentos que se aproximen al paradigma OO y no dependan de la herencia. Si se eliminara la restricción de proveer o facilitar el concepto de herencia cualquier lenguaje que explote la encapsulación de datos es OO. Usando tal definición no es necesario contestar a preguntas como: ¿Son Ada y Modula OO?. Tal vez sería mejor preguntar *¿en qué forma son Ada y Modula OO?* Y también preguntarnos por qué tal o cuál lenguaje es *más* OO que otros, la respuesta reside en qué tan homogéneo es el sistema.

Historia de Smalltalk

SMALLTALK es el software de un proyecto ambicioso conocido como *The Dynabook*, el cual pretendía ser un cierto tipo de computadora, cuyo objetivo primordial era el de ser una verdadera computadora personal. Alan Kay fue el responsable del proyecto Dynabook, hacia fines de los 60's trabajó con una versión preliminar conocida como Flex, a comienzos de los 70's trabajó en el centro de investigación de Xerox en Palo Alto y ahí formó un grupo. La meta siguió siendo la creación de una computadora personal real. Las raíces de SMALLTALK provienen de una versión preliminar llamada Flex pero además se evidencia influencia de LISP en la estructura más profunda de Smalltalk (esto se verá en los capítulos 3 y 4). La noción de "clase" predomina en todo el diseño.

El lenguaje resulta estar completamente basado en el concepto de clase como la única unidad estructural, con instancias de clase u objetos, siendo éstas las unidades concretas que habitan el mundo de Smalltalk. En su momento el proyecto Dynabook llegó a parecer una idea inconcebible, en tal proyecto los usuarios serían niños y Smalltalk sería el principal vehículo de programación. El desarrollo tomó diez años a Alan Kay y su colega Adele Goldberg, las principales versiones de Smalltalk fueron: Smalltalk - 72, Smalltalk - 76, Smalltalk - 78 y Smalltalk - 80. Las primeras versiones eran más bien lenguajes imperativos, los cuales se acompañaban de objetos y mensajes. Las versiones más recientes dejaron atrás esto último pero conservaron la esencia OO: incluso enteros y caracteres son objetos. Además, se tuvo una aportación adicional, se introdujeron

conceptos totalmente nuevos: menúes, el ratón, gráficos, ventanas, íconos. Hoy en día todos estos conceptos son familiares a los usuarios de las computadoras. La creación de Smalltalk y su medio ambiente se convirtieron en una de las piedras milenarias en la historia de la computación (Wegner, 1976). Smalltalk preparó el camino para introducir en el mercado económico la industria de la computación en forma masiva (Savic, 1990).

Además, planteó la pregunta crucial a la industria de la computación, ¿cómo facilitar aún más el uso de las computadoras? Sin embargo, Smalltalk y sus ideas no estuvieron accesibles a las masas por un largo período. Una de las razones principales era que Smalltalk requería de fuertes plataformas de Hardware. A medida que la industria del hardware avanzó e IBM introdujo las PC´s, Smalltalk fue implantándose en diversos equipos hasta que se llegaron a instalar algunas versiones en las VAX y las 2020, entre otras; pero Smalltalk y su medio ambiente tuvieron que esperar la aparición de Steve Jobs, el cofundador de Apple. En una visita de Jobs al centro de investigación de la Xerox en Palo Alto California (centro de operaciones de A. Kay y su equipo), pudo observar las pantallas, ratones, menúes, íconos, y Smalltalk mismo. Todo lo anterior lo impactó de tal manera que decidió importar todo esto en su nueva computadora, "Lisa". Dicha computadora fue muy costosa por lo que decidieron sacar al mercado la Macintosh. Aunque esta primera aparición de Mac no fue muy poderosa, tenía el medio ambiente WIMP (Windows-Incons-Menus-Pointers), a la gente le agradó. La visión de Alan Kay de una computadora de fácil uso se volvía una realidad, aunque no el sentido original de la concepción del Dynabook. Mac estaba lejos de ser una computadora amigable para un niño, pero no por ello dejaba de introducir características del Dynabook. Smalltalk lanzó sendas versiones para Lisa y Mac.

Los usuarios de otras computadoras notaron la propuesta que Mac ofrecía y comenzaron a buscar características similares en máquinas que ya existían en el mercado, por ejemplo la PC de IBM XT/AT y compatibles.

Muchos programas de graficación fueron influenciados fuertemente por los gráficos de Mac, y a través de ellos, las ideas una vez desarrolladas por Smalltalk comenzaron a ser útiles a millones de usuarios. Durante los 70´s el lenguaje en sí no estaba disponible para una PC ordinaria, la razón era simple: se necesitaba de una computadora poderosa (para aquella época) para ejecutar Smalltalk, y la PC disponible en esa época, con 64 K en RAM, no era suficiente. Los objetos y la metáfora utilizada por los mensajes de Smalltalk están lejos del lenguaje de máquina; Smalltalk requiere de un procesador rápido y con ranuras de expansión de memoria para una velocidad aceptable en la ejecución. A comienzos de los 80´s la PC común evolucionaba e incorporaba los *Megabytes* en memoria y procesadores veloces como los Intel 80286, 80386 y 80486, y el más pequeño de la familia el 8088 con 640 K, este último era suficiente para los requerimientos de la versión de Smalltalk - 80.

Con todos estos elementos presentes hizo su aparición el producto "Smalltalk / v", desarrollado por una compañía de Los Ángeles, Digitalk. Fue barato ($ 100.00

U.S. D.), poderoso, y bonito. En dos años se vendieron 25000 copias en todo el mundo, congregando la mitad de las instalaciones orientadas a objetos. Para los procesadores 80286, 80386, y 80486 existió una versión llamada Smalltalk /V286. Existe también una versión de Digitalk para la Mac. Smalltalk siempre ha inspirado la creación de nuevas plataformas especiales de hardware. Existen otras implementaciones de Smalltalk además de la ya mencionada Smalltalk /V de Digitalk, por ejemplo en Atari ST y en Macintosh (Savic, 1990). Cabe señalar que la versión utilizada en el presente trabajo se realizó con la versión 2.0 para Windows de Smalltalk /V, por Digitalk, la cual era la versión comercial que se adecuaba a los propósitos del mismo.

Rasgos característicos de los lenguajes de programación, en la actualidad.

Se ha buscado por décadas el lenguaje de programación ideal, pero ahora se sabe que la elección correcta depende de la aplicación en cuestión.

En el mundo de los sistemas de información, el cual fue dominado por COBOL durante muchos años, existe un número cada vez más creciente de generadores de aplicaciones. Estos generadores pueden producir código desde pantallas o funciones por medio de las cuales se realizan búsquedas, filtros y reportes con la información de un conjunto de bases de datos. En ciertos dominios de aplicación, los programadores sin experiencia y usuarios pueden desarrollar aplicaciones no triviales, sin necesidad de los servicios de un programador profesional. A las herramientas de programación de este tipo se les ha llamado *lenguajes de cuarta generación*.

Se pueden desarrollar aplicaciones con un alto grado de interacción con *lenguajes visuales*, tales como Visual Basic ó Visual C++.
También existen herramientas específicas, lenguajes, y medios ambientes para el desarrollo de sistemas expertos, tales como: CLOS, Prolog, etcétera.

Finalmente, C++ parece haber ganado la aceptación general como una herramienta de programación de propósito general, principalmente por dos razones, es OO y no se aparta de formas de programación más convencionales. Para los nuevos desarrollos, dentro del área de cómputo en redes ha iniciado una nueva e importante dirección, su nombre es Java.
Java es un lenguaje derivado de C++ que provee de facilidades para crear código transportable sobre Internet, este lenguaje puede ser visto como el punto de partida para una nueva generación de lenguajes de programación.

Capítulo 2
Conceptos del Paradigma OO

¿Qué se entiende por Orientado a Objetos (OO)?

Los lenguajes de programación son el corazón de la ciencia de la computación, ellos constituyen las herramientas que utilizamos para comunicarnos con las computadoras.
Existe una fuerte relación entre lo que se conoce como diseño de software y los lenguajes de programación. Los métodos de diseño descomponen un problema o sistema en componentes lógicos, los cuales eventualmente deberán codificarse en un lenguaje de programación.
Para comprender mejor esta relación, es importante darse cuenta de que los lenguajes de programación se clasifican de acuerdo al estilo de programación que ellos ofrecen, a esta clasificación se le denomina paradigmas de programación(capítulo 1). El término Orientado a Objetos (OO) se utiliza para señalar a aquellos lenguajes de programación que basan el desarrollo de sus programas en clases de objetos. Esto último significa que los programas en estos lenguajes están formados por objetos, los cuales interactúan unos con otros para realizar una tarea. Asimismo, por ejemplo, lenguajes como C, FORTRAN y Pascal fueron originalmente definidos como lenguajes orientados a procedimientos o imperativos, ya que la unidad básica de los programas desarrollados en estos lenguajes es la rutina o procedimiento.

Se dice que el método de diseño y el lenguaje de programación que será utilizado para la implementación, deberán ser el mismo, es decir, si se tiene un diseño OO, el lenguaje idealmente debería ser también OO. Es posible no tener concordancia entre diseño y lenguaje, por ejemplo el diseño puede ser OO y el lenguaje no serlo, el precio que deberá pagarse será en el doble esfuerzo que implicará el implementar no sólo la solución del problema sino también los conceptos del diseño.

2.1 Clases y Objetos

Se ha dicho que la unidad fundamental del paradigma OO son las clases de objetos, para a bordar y comprender mejor los conceptos alrededor del paradigma OO, iniciamos describiendo precisamente lo que son objetos y clases.

2.1.1 Objeto

Informalmente podríamos decir que un objeto es una colección de operaciones o métodos que comparten datos (identidad). La manera de comunicarnos con los objetos es por medio de mensajes "comprensibles" para tales objetos. Cuando a

un objeto se le "ordena" algo, lo que se está haciendo es pedirle que nos responda a un mensaje. El objeto que recibe el mensaje puede responder de dos maneras, dependiendo de si entiende o no el mensaje. Si el objeto reconoce la petición, este ejecutará alguna acción para satisfacer la demanda; si por el contrario el objeto no reconoce el mensaje, este responderá precisamente con un mensaje de "no entiendo el mensaje".
Para conocer a un objeto debemos primero entender su comportamiento, ya que si se desea establecer una comunicación con él es necesario saber qué es lo comprensible por dicho objeto. Al conjunto de mensajes que un objeto entiende se le conoce como interfaz, esta define el comportamiento de un objeto.

Vale la pena señalar, que la relación que nosotros tenemos con nuestro entorno funciona precisamente con la interacción que tenemos con todos los objetos que nos rodean: personas, animales, plantas, carros, sillas, elevadores, etc.; cada uno de estos objetos, desde el punto de vista totalmente abstracto, responde a mensajes muy particulares. Por ejemplo los seres humanos reconocemos lenguajes específicos (Idiomas, dialectos, mímica, etc.).
Otra forma de definir a un objeto es como algo tangible y que muestra un comportamiento bien definido. Desde esta perspectiva un objeto puede ser:

- Una cosa visible y tangible
- Algo que puede ser comprendido intelectualmente
- Algo hacia lo cual puede ser dirigido el pensamiento o una acción

Con todo lo mencionado, podemos dar la definición formal:

Un objeto o instancia tiene un estado, un comportamiento, y una identidad; la estructura y comportamiento de objetos similares está determinado por su clase que es común. Los términos instancia y objeto son intercambiables (Grady, 1991).

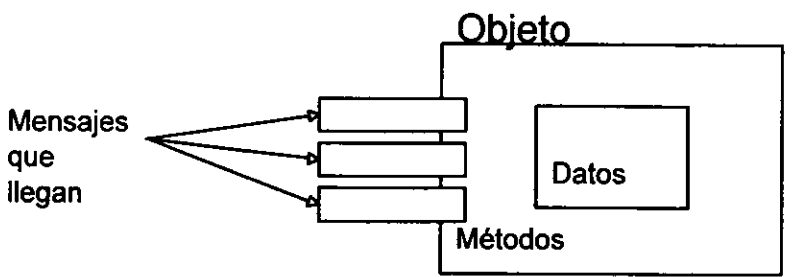


Fig. 2.1. Anatomía de un objeto.

A continuación se describirán con mayor detalle los componentes de un objeto.

Estado

El estado de un objeto se compone de todas las propiedades (usualmente estáticas) del objeto y de los valores actualizados (usualmente dinámicos) de cada una de esas propiedades (Grady, 1991).

Una propiedad es una característica inherente o distintiva que hace a un objeto diferente, las propiedades son usualmente estáticas, ya que por lo general no se pueden cambiar. Todas las propiedades pueden tener algún valor; este valor puede ser una simple cantidad, o puede denotar a otro objeto; de aquí que se pueda hacer la distinción entre los objetos y los valores. Por ejemplo, un carro puede tener el estado apagado denotando esto que el carro no está en marcha.

Comportamiento

El cómo actúa y reacciona un objeto es lo que define su comportamiento, en función de sus cambios de estado y paso de mensajes (Grady, 1991).

Se puede decir que el comportamiento de un objeto está completamente definido por sus acciones. Una operación es algún tipo de acción que un objeto realiza sobre otro para provocar una reacción. En lenguajes de programación basados en objetos y OO, las operaciones que los clientes pueden realizar sobre un objeto son lo que se conoce como *métodos*, los cuales son parte de la declaración de la clase a la que pertenece el objeto; en particular, en Smalltalk es conocido como método ya sea de instancia o de clase.
Son cinco los tipos de operaciones que por lo general se realizan sobre los objetos, los tres más comunes son:

- Modificadoras. Son operaciones que alteran el estado de un objeto; como pueden ser una operación de escritura o acceso.
- Selectoras. Son operaciones que accesan el estado de un objeto, pero no lo alteran, como puede ser una operación de lectura.
- Iteradoras. Son operaciones que permiten accesar todas las partes de un objeto de una manera bien ordenada.

En lenguajes como Smalltalk, C++, y CLOS, se pueden declarar otros tipos de operaciones:

- *Constructoras*. Son operaciones que crean a un objeto y/o inicializan su estado.
- *Destructoras*. Son operaciones que liberan el estado de un objeto y lo destruyen al mismo tiempo.

En Smalltalk tales operaciones son parte de la interfaz de la clase misma, las operaciones pueden ser declaradas como métodos, pero no permite declarar métodos por separado. En cambio en otros lenguajes como Objective Pascal,

C++, CLOS, y Ada se permite escribir operaciones como subprogramas libres o *utilidades de clase.*

Las utilidades de clase son funciones que actúan sobre uno o varios objetos de la misma o de diferentes clases (Grady, 1991).

Todos los métodos junto con las utilidades de clase asociadas a un objeto forman lo que se conoce como interfaz del objeto. La interfaz define lo que el comportamiento del objeto admite, y abarca toda la parte externa del mismo tanto la parte dinámica como la estática.

Identidad

La identidad es la propiedad que tiene un objeto que lo distingue del resto de los objetos (Grady, 1991).

Muchos lenguajes de programación utilizan nombres de variable para distinguir a los objetos que son temporales mezclando la direccionabilidad e identidad, así como muchos sistemas manejadores de bases de datos utilizan nombres clave para distinguir a los objetos que son persistentes, mezclando valores de datos e identidad. La falla en reconocer la diferencia entre el nombre de un objeto y el objeto mismo, es fuente de muchos errores en programación OO. Debido a que un objeto es distinguido de todos los demás, su identidad se preserva aun cuando el objeto cambie completamente su estado.

Los objetos pueden clasificarse en funcionales, imperativos y activos; según correspondan a valores, variables, y procesos respectivamente (ver fig. fig. 2.2).

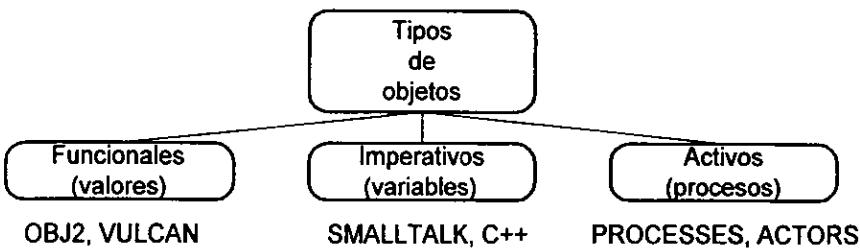


Fig. 2.2. Tipos de objetos existentes.

Tipos de Objetos

Objetos funcionales

Los objetos funcionales se originan como es de suponerse en los lenguajes lógicos o funcionales, además tienen una interfaz que no es actualizable, no tienen identidad pero logran persistir entre cambios de estado. Sus operaciones se

invocan por llamadas a funciones con expresiones cuya evaluación provoca un efecto lateral libre, como en los lenguajes funcionales. Los objetos funcionales son considerados como degenerados, ya que se forman a partir de un conjunto de funciones pero sin un estado.

Objetos imperativos

Estos objetos tienen un estado actualizable que es compartido por operaciones en la interfaz. Los objetos imperativos son los tradicionales de Simula, Smalltalk, y C++. Estos tienen un nombre (identidad), una colección de métodos que se activan al recibir mensajes y variables de instancia compartidas por los métodos del objeto, las cuales son inaccesibles a otros. Los objetos imperativos son pasivos a menos que un mensaje los active. Las operaciones sobre variables compartidas sacrifican su reusabilidad en otros objetos en aras de facilitar la eficiencia en la coordinación con las operaciones de un objeto dado. Los lenguajes tradicionales OO tienen objetos imperativos, por lo que un lenguaje como Smalltalk, puede referenciarse como un *lenguaje imperativo OO(orientado a objetos)*.

Objetos activos

Son activos en el momento de recibir un mensaje, los mensajes deben sincronizarse para regular las actividades del objeto, estos provienen de lenguajes de programación concurrentes, los cuales están basados en objetos. Los objetos activos son ejecutantes en el momento en que un mensaje está llegando. Estos tienen los modos: latente (dormant), activo(ejecutante), o expectantes (por recursos o término de subtareas).

2.1.2 Clase

¿Qué es una Clase?

De manera general podemos decir que una clase agrupa a un conjunto de objetos de acuerdo a ciertas características comunes. Una de las ventajas de usar objetos como base de la descomposición de un programa es que los sistemas son modelados con más cercanía al mundo real.
En muchos lenguajes OO el concepto de clase abarca tres conceptos diferentes, pero muy relacionados.

- Una clase es un *molde*, por medio del cual se generan numerosas instancias (representantes), objetos que son idénticos tanto en el comportamiento como en la estructura. El conjunto de operaciones declaradas en el molde representa la interfaz que define la parte externa de las instancias.

- Una clase también define un tipo de dato abstracto, el cual puede usarse para determinar la compatibilidad de los objetos en un sistema.
- Por último, un molde puede instanciarse muchas veces para dar como resultado un conjunto de objetos estructuralmente idénticos por lo que una clase abarca el concepto de "conjunto" de objetos.

Los conceptos de clase y objeto están fuertemente ligados, no se puede hablar de un objeto sin mencionar su clase, pero claro, existen diferencias importantes entre ellos, mientras que un objeto es una entidad tangible, una clase representa una abstracción, esta última es la esencia de un objeto. En lenguaje coloquial se dice que una clase es un conjunto marcado por uno o varios atributos comunes, pero en términos del paradigma OO se puede definir como:

Una clase es un conjunto de objetos que comparten una estructura y comportamiento común (Grady, 1991).

Un objeto es solamente una instancia o ejemplar de una clase, mientras que un objeto lleva a cabo una tarea concreta dentro de una aplicación la clase retoma la estructura y comportamiento comunes de todos los objetos relacionados. De aquí se puede pensar en una clase como un contrato entre una abstracción y todos sus clientes; el verla así nos ayuda a distinguir las partes interna y externa de una clase. La interfaz de una clase le provee su parte externa y remarca la abstracción mientras que oculta su estructura y los secretos de su comportamiento. Por otra parte, la codificación conforma la parte interna de la clase.
Se puede definir la interfaz de una clase en tres partes:

- Pública. Es la parte de la interfaz visible para todos los clientes que la accesan.
- Protegida. Es la parte de la interfaz que solo es visible a sus subclases.
- Privada. Parte de la interfaz que no es visible a ninguna otra clase.

Las constantes y variables que forman la representación de una clase son conocidas por varios nombres, dependiendo del lenguaje de programación; Smalltalk usa el término *variable de instancia*, Objective Pascal *campo*, C++ *objeto miembro*, y CLOS el término *slot*.

Las clases que no tienen instancias se les llama *abstractas*, la generalización en una estructura de jerarquía es la llamada *clase base*, en algunos lenguajes esta se ubica en la parte más alta de la jerarquía, es decir sirve como la superclase de todas las clases. En Smalltalk a esta se le conoce como *object*.

Con respecto a la asociación entre clases se tienen tres tipos de relación.

- La ***generalización***: Denota una asociación "tipo de". Por ejemplo, un "barco" es un tipo de transporte, con esto lo que se dice es que *barco* es una subclase especializada de una clase más general, la clase *transporte*.
- La relación de ***agregación***: Denota una asociación "parte de". Por ejemplo, un "mástil" no es un barco, es parte de un barco.

- La ***asociación***. Denota una conexión semántica entre clases que no tienen una relación directa. Por ejemplo, las clases automóvil y carretera, son independientes, pero ambas se pueden relacionar para podernos trasladar de un lugar a otro.

Varios conceptos de lenguajes de programación expresan matices de estos tres tipos de asociación, algunos lenguajes OO y basados-en-objetos facilitan alguna combinación de las siguientes asociaciones entre clases:

I. Asociaciones de herencia
II. Asociaciones de uso
III. Asociaciones de instanciación
IV. Asociaciones de metaclases

Quizás la más poderosa de todas estas asociaciones es la de herencia y puede ser usada para expresar tanto generalización como asociación. La herencia es una asociación entre clases, donde una clase comparte la estructura y comportamiento definido en una o más clases, herencia simple y múltiple respectivamente. La herencia es necesaria pero todavía insuficiente para expresar toda la riqueza de asociaciones que pueden existir entre las abstracciones clave en el dominio de algún problema específico. Las asociaciones usuarias son necesarias para facilitar la agregación. Las asociaciones de instanciación, como las de herencia, facilitan tanto la generalización como la asociación, pero lo hacen de una manera completamente diferente.

Las asociaciones de metaclase no son facilitadas por todos los lenguajes de programación OO y basados en objetos. El concepto de metaclase surge al pensar a la clase misma como un objeto, de ahí que la metaclase se defina brevemente como la clase de una clase.

I. Asociaciones de Herencia

El concepto de herencia es un mecanismo importante en el paradigma OO, sin él sería imposible hablar de la asociación entre clases, la herencia es un mecanismo para compartir código y comportamiento.

Codificación de la herencia

Considérense una clase A con instancia *a* y subclase B con instancia *b*. Tanto A como B definen un comportamiento común por operaciones compartidas por sus instancias, y tienen variables de instancia que provocan que se cree una copia privada de las mismas para cada instancia de la clase o subclase. La instancia *a* tiene una copia de las variables de instancia de A y un apuntador a su clase base. La instancia *b* tiene una copia de las variables de instancia tanto de B como de su superclase A y un apuntador a su clase base B. La clase B apunta a su clase base A, mientras A podría apuntar hacia la clase base.

Cuando *b* recibe un mensaje para ejecutar un método, busca primero en los métodos de B. Si encuentra el método, usa las variables de instancia de B como datos de *b*. En otro caso, el apuntador va hacia su superclase, si encuentra el método lo ejecuta con los datos de b, y en otro caso busca en la superclase de A, si tampoco encuentra el método en la superclase sigue una búsqueda ascendente hasta llegar a la clase genérica, si en toda la búsqueda no encuentra el método asociado, se produce una falla.

El método hallado como resultado de esta búsqueda deberá ser ejecutado en el medio ambiente de la clase base. La referencia a sí mismo (por ejemplo, *sela* en Smalltalk) deberá interpretarse como referencia a la clase base antes que a la clase en la cual se ha declarado el método.
La restricción de que las instancias tienen precisamente una clase base deberá entenderse en términos de codificación, las instancias deben saber donde empezar a buscar los métodos cuando reciben un mensaje para ejecutarlos.

La herencia como mecanismo de mayor modificación

Una modificación a los sistemas de software por medio de la herencia, distingue dos nociones: refinamiento y similitud. Decir que "B refina A" significa que B no solo preserva A, sino que le añade propiedades.
La herencia combina un padre P y un modificador M sobre un resultado R, donde R = composición (P,M). Cuando los atributos de M son disjuntos a los de P, R tiene p + m atributos consistiendo de la unión de P y M. Para atributos que coinciden, los de M redefinen a los de P. A continuación se citan algunos de los mecanismos más relevantes para modificar atributos de P que coinciden con atributos de M:

- Comportamiento compatible: El comportamiento de subclase es "compatible" con la superclase.
- Compatibilidad de asignación: la asignación de subclases es sintácticamente compatible con subclases.
- Compatibilidad de nombre: los nombres de operaciones de una superclase se preservan en las subclases.
- Cancelación: modificación sin restricciones de atributos de superclases por subclases.

Agrupamiento de la herencia

La herencia es un mecanismo de comportamiento que se complementa con valores compartidos invocando métodos en tiempo de ejecución.

El comportamiento compartido a nivel de objetos es referido como un *delego*. Los objetos pueden delegar responsabilidad para ejecutar operaciones que no son locales, hacia instancias padre llamadas *prototipos*, los cuales sirven para compartir comportamiento.

Los lenguajes de programación que toman su fundamento de las matemáticas clásicas, distinguen entre clases e instancias y usan a la herencia primordialmente para compartir comportamiento.

Las clases por otra parte separan asociaciones de estructura y comportamiento de aquellas de cálculo en tiempo de ejecución; esto ayuda en el diseño, ya que durante éste, es de interés primordial especificar el comportamiento del dominio de una aplicación sin ninguna especificación computacional. Los prototipos son útiles para representar tipos teniendo solamente una instancia, ya que no hay necesidad de representar al tipo como distinto de su instancia. Sin embargo, cuando existen muchas instancias potenciales de un tipo, es útil distinguir de un tipo y sus instancias, tanto conceptualmente como a un nivel de implementanción.

Herencia Simple

Una subclase comúnmente aumenta o redefine la estructura y comportamiento existentes de sus superclases. La disponibilidad en los lenguajes para facilitar este tipo de herencia distingue a los lenguajes OO y basados en objetos.
Un buen diseño en la jerarquía de clases será recompensado con una buena comprensión y mejor codificación del problema en cuestión. Se espera que algunas clases tengan instancias y otras no, a estas últimas se las llama *clases abstractas*. Una clase abstracta es creada con la intención de que sus subclases agreguen más estructura y comportamiento, esto se logra completando e implementando sus métodos que usualmente están incompletos.
La clase más generalizada dentro de una estructura de clase es la llamada *clase base* , ésta representa la generalización más abstracta dentro del dominio del problema. Por ejemplo algunos lenguajes incluyen una clase base que está en la parte más alta de la jerarquía, y sirve como la superclase de todas las clases.

II. Asociaciones de Uso

Hay dos alternativas para dar cuenta de las asociaciones de uso: la interfaz y la codificación de la clase. En la primera de estas, la clase usada debe ser visible a todos los clientes, en la segunda en cambio, la clase usada está oculta.

III. Asociaciones de Instanciación

Una clase contenedora (container class) es aquella cuyas instancias se forman de colecciones de objetos. Las colecciones pueden ser homogéneas, lo que significa que todos los objetos pertenecen a la misma clase, opuestamente están las colecciones heterogéneas que son objetos que provienen de diferentes clases pero que tienen en común alguna superclase. Las clases heterogéneas más comunes incluyen, entre otras, a las pilas, listas, cadenas, colas, bicolas, árboles, y gráficas.

Existen básicamente cuatro maneras para crear clases contenedoras. En la primera de ellas, se emplean macros, esta forma se utiliza, hasta ahora, en C++, pero el mismo Stroustrup apunta que esta "no funcionará bien, salvo en pequeña escala" (Budd, 1991), ya que el tratar de mantener tales macros resultará algo torpe de manipular, además, de que cada instanciación provoca que se cree una nueva copia del código. La segunda es la técnica usada en Smalltalk a través de herencia y ligamiento tardío. Con esta técnica sólo se pueden construir clases contenedoras homogéneas, ya que no hay forma de asegurar quién es la clase específica que contiene a los elementos; todo elemento es tratado como si fuera una instancia de la clase base *object*. La tercera, se puede tomar como otra técnica implementada que generaliza clases contenedoras como en Smalltalk pero en este caso se utiliza un control fuerte de tipos para garantizar que todos los objetos sean de la misma clase, lo cual se asegura cuando el objeto contenido es creado. La cuarta técnica provee de un mecanismo de parametrización de clases. Una clase parametrizada, también conocida como una *clase genérica*, es aquella que puede servir como molde para otras clases, dicho molde puede parametrizarse por otras clases, objetos, y/o operaciones. Una clase parametrizada debe ser instanciada, es decir, sus parámetros deben ser proporcionados antes de que los objetos sean creados.

Las clases parametrizadas pueden usarse mucho más que solo para construir clases contenedoras. Desde una perspectiva de diseño, las clases parametrizadas son útiles también para resolver ciertos problemas de decisiones en el diseño de la interfaz de una clase.

IV. Asociaciones de Metaclase

Las tres asociaciones mencionadas cubren casi todo lo que alguien pueda necesitar ya que cubren además las asociaciones más importantes entre las clases, sin embargo, existe un tipo más de relación: las asociaciones de Metaclase.

Se ha dicho que todo objeto es instancia de alguna clase ¿y qué pasa si pensamos a una clase como a un objeto? O ¿qué es la clase de una clase? La respuesta es que la clase de una clase es una metaclase. Dicho de otra forma, una metaclase es una clase cuyas instancias son clases.

Se justifica la necesidad de las metaclases diciendo "en un sistema en desarrollo, una clase provee una interfaz para que el programador interactúe con la definición de los objetos. Para este tipo de uso de las clases, es muy útil el hecho de que ellas mismas sean objetos, de tal forma que puedan manipularse en la misma forma que los objetos" (Robson, 1981).

Con todo la anteriormente expuesto es posible definir el resto de los conceptos que rodean al paradigma OO.

Mensajes y métodos

La acción en un programa OO es iniciada cuando a un objeto, que será considerado con el responsable de la acción, se le transmite un *mensaje.* El mensaje es acompañado por cualquier información adicional (argumentos) necesaria para despachar la petición. El *receptor* es el agente al que se le manda el mensaje Si el mensaje es aceptado, el receptor ejecutará algún método para satisfacer la petición.

Se señalan distinciones importantes entre mensajes y llamadas a procedimientos, en POO se designa un receptor del mensaje y la *interpretación* (esto es, la selección del método para satisfacer la petición) puede variar dependiendo del tipo de receptor de que se trate. En una llamada a procedimiento no existe receptor designado; usualmente el receptor específico para cualquier mensaje dado no será conocido sino hasta el tiempo de ejecución, y también la determinación del método. Aquí se dice que existe ligamiento tardío entre el mensaje (nombre de la función o procedimiento) y el fragmento de código (método) usado para responder al mensaje. Esta situación contrasta con el ligamiento temprano (tiempo de compilación o ligado) que nombra el código en llamadas convencionales de procedimientos.
Es importante notar que el comportamiento es estructurado en términos de *responsabilidades.*

Todos los objetos son *instancias* de una *clase*, el método utilizado por el receptor en respuesta a un mensaje es determinado por la clase a la que pertenece, todos los objetos de una clase dada usan el mismo método para responder a mensajes similares.

Para encontrar un método invocado en respuesta a un mensaje dado, se comienza a explorar en la clase del receptor, si no se halla algún método apropiado, la búsqueda se continúa en la superclase de la clase, esta búsqueda continúa hacia arriba en una cadena, hasta que el método es hallado, o la cadena de superclases es agotada. En el caso de encontrar el método adecuado, este será ejecutado. Por otro lado, si no existe tal método se despliega un mensaje de error, esto se conoce como método de ligamiento. Todo lo anterior se permite porque las clases pueden organizarse jerárquicamente y la subclase heredará atributos de una superclase más alta en el árbol jerárquico.

2.2 El Marco conceptual del Paradigma OO

El paradigma OO centra su marco de trabajo conceptual en el *modelo de objeto*, el cual cuenta con las siguientes características esenciales (Grady, 1991):

- Abstracción
- Encapsulación
- Modularidad
- Jerarquía

Un modelo sin alguna de estas cuatro características no es OO. Existen otras características del modelo objeto que son deseables pero no esenciales:

- Tipos*
- Concurrencia
- Persistencia

Es posible programar en cualquier lenguaje OO sin el marco conceptual del modelo objeto, pero el diseño se parecerá a un lenguaje orientado a procedimientos, como podría ser el caso de Modula-2. La abstracción de un problema la podemos tener en todos los diferentes paradigmas de programación, en un lenguaje orientado a procedimientos como Modula-2 es posible encapsular información declarando como locales ciertas variables en el módulo de implementación, en este lenguaje se definen perfectamente los módulos (de definición e implementación), además de existir jerarquía entre módulos. Sin embargo, debido a que la jerarquía no es "ordenada" no se tiene el concepto de herencia.

Abstracción

La abstracción denota las características esenciales de un objeto, que lo distinguen de los otros tipos de objetos y además provee limitantes conceptuales, relativas al observador.(Grady, 1991).

Una abstracción se centra en la visión exterior de un objeto, de esta forma se puede separar el comportamiento de su codificación. Un *cliente* es cualquier objeto que usa los recursos de otro, por lo que un objeto se caracteriza por las operaciones que pueden realizar sus clientes sobre él así como de las que a su vez él puede realizar sobre otros objetos. Al conjunto entero de operaciones que un cliente puede realizar sobre un objeto se le llama interfaz. Una interfaz denota el modo en que un objeto puede actuar y reaccionar, conforma toda la visión externa estática y dinámica de la abstracción. Las operaciones que podemos realizar sobre un objeto y la forma en la que reacciona son las que constituyen el comportamiento de un objeto.

La abstracción de un objeto deberá preceder a las decisiones acerca de su codificación. Una vez seleccionada una codificación deberá de ocultarse de casi todos los clientes, pero también se dice que "ninguna parte de un sistema complejo deberá depender de ningún detalle interno de otra parte" (Budd, 1991). La abstracción y la encapsulación son conceptos complementarios: la abstracción se enfoca sobre la parte externa de un objeto y la encapsulación sobre el ocultamiento de los detalles de la parte interna de un objeto.

* Veremos que Smalltalk no obstante ser un lenguaje llamado "puro" Orientado a Objetos es no tipificado, es decir, no tiene tipos.

En la práctica cada clase debe tener su parte de interfaz y su parte de codificación. La *codificación* es la representación de la abstracción así como los mecanismos que hacen posible el comportamiento deseado.

Encapsulación

La encapsulación es el ocultamiento de todos los detalles de un objeto, aquellos que forman parte de las características esenciales (Grady, 1991).
Por tanto, la encapsulación es el ocultamiento del código del objeto.

Modularidad

Dividir el programa en subpartes reduce la complejidad del sistema en cuestión. En cierto sentido los módulos pueden ser vistos como una técnica mejorada para crear y manipular nombres de "espacios". El *módulo* mejora la capacidad para dividir un nombre de espacio en dos partes. La parte *pública* que es accesible fuera del módulo, y la parte *privada* que sólo es accesible dentro del módulo.
David L. Parnas sugiere que:

- Se debe proveer toda la información necesaria al usuario, para el correcto uso del módulo, y nada más (Parnas,1972).

- Se debe proveer al implementador de toda la información necesaria para completar el módulo, y nada más (Parnas,1972).

La filosofía sostiene que si no se necesita saber cierta información no se tendrá acceso a ella. Algunos investigadores sostienen que un objetivo esencial de la descomposición en módulos, es la reducción en el costo del software, esto se logra permitiendo que los módulos sean diseñados y revisados independientemente. Cada módulo debe ser lo suficientemente simple para entenderse cabalmente, deberá ser posible cambiar la codificación de algunos módulos sin afectar el comportamiento de otros. En la práctica, el costo de recompilar el cuerpo de un módulo es relativamente bajo, la unidad en cuestión es recompilada y el resto es ligado nuevamente; sin embargo, el costo de recompilar la *interfaz* de un módulo es relativamente alto, en especial con lenguajes fuertemente tipificados ya que se debe recompilar la interfaz del módulo, su cuerpo, y todos los módulos que dependan de esta interfaz, además de los módulos que dependen de estos módulos, y así sucesivamente. Para un programa largo, algún cambio en la interfaz de un módulo puede provocar horas de recompilación. De aquí que la interfaz de un módulo deba ser tan pequeña como sea posible, pero sin dejar de satisfacer las demandas de los módulos que sean usuarios. Con este panorama se puede definir el concepto de modularidad:

La *Modularidad es la propiedad de un sistema que ha sido descompuesto en un conjunto de módulos acoplados amplia y cohesivamente* (Grady, 1991).

Podemos decir que dentro de las claves para un buen diseño de software está la modularización. Una buena modularización representa una abstracción útil; la cual interactúa con otros módulos de manera bien definida. Asimismo, esta abstracción puede ser entendida, implementada, compilada y manejada para la especificación de otros módulos, únicamente mediante la especificación (sin proveer detalles de la implementación).

Como se puede ver, los principios de abstracción, encapsulación, y modularidad trabajan conjuntamente. Un objeto da una clara visión de una simple abstracción, mientras que la encapsulación y la modularidad son una especie de barreras que se le ponen a dicha visión, esto en el sentido de que el objeto tiene cierta privacía y pertenece a un cierto tipo específico de objetos (clase).

Jerarquía

En programas grandes, se hallan diferentes tipos de abstracciones, y cada una de estas se puede comprender por separado. La encapsulación ayuda a ocultar la parte interna de las abstracciones, y la modularidad nos posibilita la agrupación de relaciones entre las abstracciones. Un conjunto de abstracciones así, siempre forman una jerarquía y hallar ésta, simplificará mucho la tarea de trabajar con la complejidad de todo el programa.

La jerarquía consiste de un ordenamiento de las abstracciones (Grady, 1991).

Los tipos de jerarquía más importantes dentro de un sistema son: su estructura de clase (la jerarquía "tipo de") y su estructura de objeto (la jerarquía "parte de").

Tipos

El concepto de tipo es útil para clasificar, organizar, abstraer, conceptualizar y transformar grupos de valores. Los tipos fueron introducidos en FORTRAN y Algol a principios de los 50's y fueron extendidos a tipos numéricos, tipos de registros, procedimientos, clases y tipos de datos abstractos.

El concepto de *tipo* se deduce de la teoría de datos abstractos. Los conceptos de clase y tipo resultan parecidos en el contexto OO, pero el tipo se considera como un elemento separado del modelo de objeto ya que hace énfasis en una forma muy diferente sobre lo que significa abstracción, he aquí la definición:

El concepto de tipo es la puesta en vigor de la clase de un objeto, de manera que los objetos de diferente tipo, salvo en formas muy restringidas, no pueden intercambiarse (Grady, 1991).

Al tipo se le considera un elemento no esencial del modelo de objeto, ya que existen lenguajes que tienen un fuerte control de tipos, un débil control de tipos, y hasta otros que no tienen tipos, y sin embargo son OO o basados en objetos; en particular Smalltalk es no tipificado.

Tipos de datos abstractos

Un *tipo de dato abstracto* es un tipo de dato definido por el programador que puede manipularse de manera semejante a los tipos de datos definidos en el sistema. Como con estos últimos, un tipo de dato abstracto tiene asociados un conjunto de valores legales y operaciones que puedan ejecutarse con dichos valores. Los módulos son usados para implantar tipos de datos abstractos, cabe señalar que el concepto de módulo es una técnica de codificación. En la construcción de un tipo de dato abstracto, se debe ser capaz de :

1. Exportar una definición de tipo.
2. Crear un conjunto de operaciones disponibles que puedan usar y manipular instancias de ese tipo.
3. Proteger los datos asociados con el tipo, de tal forma que ellos puedan ser solo operados por rutinas provistas.
4. Crear instancias múltiples del tipo.

En cierta forma, un objeto es simplemente un tipo de dato abstracto, esto es cierto en el caso de Smalltalk, ya que en particular una clase es un objeto. Entonces la definición de tipo (nombre de la clase) es exportable, las operaciones que las instancias de ese tipo pueden usar y manipular son los métodos de instancia de dicha clase, de tal forma que los datos (variables de instancia) solamente pueden accesarse por medio de dichos métodos y de ésta forma se da la protección de datos. La creación de instancias se realiza invocando a los métodos de clase.

Tipos y Clases

Los principales propósitos al introducir tipos son los siguientes (OOPSLA/ECOOP 90):

- *especificar la estructura de expresiones.*
- *especificar el comportamiento de clases para el desarrollo de programas, su incremento y ejecución.*

Los tipos son usados sobre expresiones para tener control de las mismas, mientras las clases son usadas para manejar y generar objetos con propiedades y comportamiento uniformes. Toda clase es un tipo, definido por un predicado que especifica su "constitución". Sin embargo, no todo tipo es una clase, ya que los predicados no necesariamente determinan la constitución de un objeto.

Estas diferencias en cuanto a propósito y especificación causan la derivación en diferentes formas de subclases y subtipos con respecto a las de sus padres.

Concurrencia

La POO se centra en la abstracción de datos, encapsulación y herencia, mientras que la concurrencia se basa en los procesos de abstracción y sincronización. El concepto de objeto unifica estas dos visiones, cada objeto puede representar la abstracción de un proceso por separado; un objeto de este tipo se dice que es *activo*. Un sistema basado en diseño OO (DOO) se puede ver como un conjunto de objetos cooperativos, algunos de los cuales son activos y sirven para señalar las tareas que son independientes; con esto en mente se puede dar la siguiente definición:

La concurrencia es la propiedad que distingue a un objeto activo de uno que no lo es (Grady, 1991).

Algo que debe tomarse en cuenta al introducir concurrencia en un sistema, es sincronizar a los objetos activos y sus actividades con aquellos objetos que no son activos; aquí es donde interactúan las ideas de encapsulación, abstracción y concurrencia. Cuando hay concurrencia, no basta con definir los métodos de un objeto, se debe asegurar que la semántica de los métodos se preservará en presencia de multiproceso. Existen algunos lenguajes OO concurrentes, que proveen mecanismos para objetos activos y sincronización. Entre los lenguajes conocidos, Smalltalk y Ada son los que proveen directamente multitarea (Smalltalk tiene la clase Process y Ada incorpora el tipo task), en C++ es posible la concurrencia a través de UNIX llamando a la función *fork* (Grady, 1991).

Persistencia

Introduciendo el concepto de persistencia al modelo de objeto se obtienen bases de datos OO. Tales bases de datos se construyen sobre tecnología ya implantada, secuencial, indexada, jerárquica, redes, o modelos de bases de datos relacionales, que dan al programador una interfaz OO. Las operaciones en estas bases se realizan a través de objetos, cuyo tiempo de duración traspasa el tiempo de duración de un programa particular. La unificación permite utilizar, para una cierta aplicación, los mismos métodos de diseño a la base de datos así como a las partes que no están en la base. Los lenguajes que se han estado mencionando sólo permiten dar la ilusión de persistencia, pero ninguno de ellos la provee directamente. La persistencia es más que preservar la duración de datos, en bases de datos OO, no sólo el *estado* de un objeto persiste, sino su *clase* también traspasa el tiempo de ejecución

2.3 Polimorfismo

Hasta aquí se ha hablado de la herencia y sus asociaciones pero también es necesario hablar del *polimorfismo* entre clases. El polimorfismo es un concepto de la teoría de tipos que permite que un nombre (tal como un parámetro B) pueda denotar objetos de las más variadas clases pero que se relacionan por alguna superclase que tienen en común. De esto se sigue que cualquier objeto denotado por este nombre (B) está dispuesto a responder a un conjunto común de operaciones en diferentes formas. El concepto fue descrito por primera vez por Strachey, quien en su momento habló de polimorfismo *ad hoc*, por el cual símbolos como "+" podían ser definidos con un significado determinado para diferentes contextos. Hoy en día, a este concepto se le llama *sobrecarga*, Strachey también habló de *polimorfismo paramétrico*, el cual se conoce hoy simplemente como polimorfismo.

Sin polimorfismo se acabaría escribiendo código conteniendo una gran cantidad de enunciados de "opción"(switch). Algunos dicen "el polimorfismo es lo más útil cuando pueden existir clases con los mismos protocolos (Kaplan, 1986)" (ver capítulo 4, la codificación del método `ins:` para la clase MiArbol y sus subclases). La herencia sin polimorfismo es posible, pero la verdad es que no resulta muy útil. El polimorfismo y el ligamiento tardío van de la mano; con el polimorfismo, el ligamiento de un método a un nombre no es determinado sino hasta el tiempo de ejecución.

Por otro lado hablando del polimorfismo y su influencia en los tipos de datos abstractos se dirá que las técnicas OO agregan varias ideas importantes al concepto de tipo de dato abstracto. Entre estas, la del *paso de mensaje* del cual ya se ha hablado. La herencia le permite a diferentes tipos de datos compartir código, reduciendo el tamaño de la codificación e incrementando la funcionalidad; el polimorfismo permite que el código compartido sea adaptado individual y convenientemente a las circunstancias específicas de cada tipo de dato (ver codificación del métodos `ins:` en el capítulo 4).

2.4 Más acerca de la POO

"¿Que es Programación OO?" Muchas veces la respuesta tiende a enfatizar características introducidas en lenguajes como Object Pascal o C++, en oposición con sus ancestros, las versiones no OO: Pascal o C respectivamente. Las conversaciones sobre POO recaen en los conceptos de clase y herencia, paso de mensajes, métodos virtuales y estáticos, y así sucesivamente pero olvidan una parte fundamental de la POO, la cual no tiene que ver con sintaxis. Este aspecto de la POO es la técnica de diseño que está dirigido por el delego de responsabilidades. Esta técnica es llamada *diseño de responsabilidad dirigida*.

El secreto para un buen diseño OO es establecer primero quien será el responsable de cada acción que será realizada.

Una de las primeras decisiones que debe hacerse al crear una aplicación OO, es la selección de clases, las clases en POO tienen diferentes tipos de responsabilidades, entonces no sorprende que halla diferentes tipos de clases. Las siguientes, sin embargo, cubren la mayoría de los casos :

- **Clases Estado.** Estas son las clases cuyo principio es mantener los datos o el estado de la información de un tipo u otro.
- **Clases Fuente.** Son las clases que generan datos, tales como generadores de números aleatorios, o aceptan datos y además los procesan, tales clases tienen salida a disco o archivo.
- **Clases de Visión.** Una parte esencial es el despliegue de la información en un dispositivo de salida, pero el código que realiza tal tarea es, a menudo, complejo y frecuentemente modificado. Por ello, en una buena técnica de programación, es conveniente aislar las clases que no mantienen los datos desplegados de aquéllas que sí lo hacen.
- **Clases Ayudante.** Estas clases mantienen poco o ningún estado de información, pero asisten en la ejecución de tareas complejas.

Lo anterior, aunque no es una lista completa, contiene las clases de uso más común, y son una buena ayuda para la fase de diseño de POO. El siguiente paso en el proceso de especificación enfatiza la manipulación de valores de datos.

El siguiente principio puede ser descrito como el básico para el manejo de datos: "Los valores que son accesados o modificados constantemente o que existan por un periodo significativo deberán ser "administrados". Es decir, una y solo una clase tendrá la responsabilidad sobre las acciones que consistan en accesar o alterar tales valores. El resto de las clases que necesite obtener dichos valores deben realizar peticiones a la clase que los administra antes que accesar a los datos mismos" (Budd, 1991).

Dos asociaciones son importantes cuando se llega a esta etapa en el diseño OO, estas son conocidas coloquialmente como la asociación *es un* y la asociación *parte de*, de las cuales ya se ha hablado.
La asociación *es un* define jerarquía de clase-subclase, mientras la asociación *parte de* describe datos que se mantienen dentro de una clase. En la búsqueda de código que pueda ser reusable o pueda ser mucho más general, se enfatiza en el reconocimiento de estas dos asociaciones.

2.5 Clases y Métodos

Encapsulación

Aquí los objetos se verán como ejemplos de tipos de datos abstractos. Como ya se ha visto un tipo de dato abstracto tiene dos caras: la primera es su comportamiento que se refiere al conjunto de operaciones que lo definen (esto es

lo que cliente ve) y la segunda es la abstracción que ven los datos o variables y que mantiene el estado interno del objeto.
Por *variable de instancia* se entiende una variable interna que pertenece a una instancia; cada instancia tendrá su propio conjunto de variables de instancia, estas variables no serán cambiadas directamente por los clientes, solamente por métodos asociados con la clase (métodos de instancia).
Entonces, una simple visión de un objeto es una combinación de estado y comportamiento; el estado es descrito por las variables de instancia, mientras que el comportamiento lo describen los métodos. Desde afuera, los clientes pueden ver solo el comportamiento de los objetos; desde adentro, los métodos proveen el comportamiento apropiado a través de modificaciones del estado.

Interfaz y Codificación

Existen ciertos principios de división que son aplicables a las técnicas OO en términos de objetos, estos principios son:

1. Una definición de clase debe proveer al usuario de toda la información necesaria para manipular correctamente la instancia de la clase, y nada más.
2. Un método debe proveerse con toda la información necesaria para llevar a cabo sus responsabilidades asignadas, y nada más.

Con esto termina la descripción de los conceptos del paradigma orientado a objetos, ahora toca llevarlos a la práctica, lo cual se hará en el resto del presente trabajo, explicando cómo quedan enmarcadas cada una de estas características.

Capítulo 3
Interfaz y Lenguaje de Smalltalk

Elementos del lenguaje y medio ambiente

En este capítulo se describen y exponen las características fundamentales de Smalltalk/V (Digitalk Inc., 1992), tanto de su interfaz del usuario como del lenguaje de programación. Las características del lenguaje permiten implementar de manera natural los conceptos del paradigma OO, a medida que se avance en la exposición de los conceptos se describirán detalles de codificación propios de Smalltalk.

Se explica en primer lugar la manera de interactuar con la interfaz del usuario, ésta tiene multipropósito, ha sido diseñada para crear y manipular gráficos, desarrollo de programación, almacenamiento y recuperación de información, se hace también la presentación del árbol jerárquico de las clases (ver fin del capítulo) que provee Smalltalk. Smalltalk provee como otros lenguajes utilerías comunes del sistema operativo de una computadora personal: compilador, rastreador (*debugger*), editor de texto, etc. pero la peculiaridad es que lo hace mediante ventanas. En una ventana es posible editar texto o código de programación, detectar y corregir errores, así como enmarcar pedazos de código y evaluarlo para su ejecución sin necesidad de cambiar hacia modos de edición, compilación, o ejecución respectivamente, bastará con accesar los menúes que pone a la disposición del usuario dentro de la misma ventana.

El medio ambiente de programación de Smalltalk resulta ser de una peculiaridad sobresaliente junto a otros medios ambientes de programación, el ambiente está hecho de ventanas, así que se comenzará por describirlas.

Las ventanas tienen características comunes, tales como: etiqueta (nombre), tamaño, menúes y diferentes tipos de divisiones. Solamente una ventana puede estar activa a la vez, y se le reconoce por que su etiqueta aparece resaltada. La ventana se activa llevando el cursor hacia ella y presionando el botón izquierdo del ratón. La comunicación en Smalltalk se lleva a cabo por medio de una ventana activa, la manera más fácil de comunicarse con Smalltalk es mediante menúes.

Al ingresar en el medio ambiente de Smalltalk existe una ventana que resulta ser generalmente la salida de los programas que se ejecutan. Esta venta está etiquetada con la leyenda `System Transcript` (Fig. 3.1) y resulta ser la ventana principal en Smalltalk.

3.1 Menúes

Al menos un menú siempre está al alcance del usuario en todo momento, a éste se le conoce como el menú principal (**System Menu**) y aparece siempre en la parte superior de cada ventana. Para abandonar cualquier menú bastará con apartar el cursor de este y presionar el botón izquierdo del ratón.

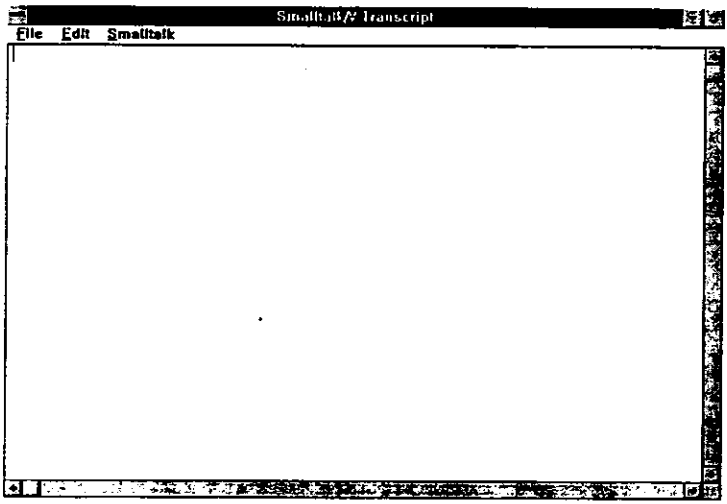


Fig. 3.1. Ventana principal de Smalltalk

3.1.1 Menú principal (System Menu)

Cada ventana contiene este menú, el cual cuenta con tres opciones: **File**, **Edit** y **Smalltalk**. Eligiendo Edit se obtienen las opciones habituales para el texto, a saber: **cut**, **paste**, **copy**, **find**, **find/replace**, etc. La opción llamada Smalltalk se relaciona con la ejecución del código, la primera de tales opciones es **show it**, la cual muestra el resultado en la misma ventana en que se ejecuta el código. La segunda es la opción **do it**, esta función también ejecuta el código pero no despliega en pantalla el resultado. Con la opción **Inspect it** lo que resulta (si el código no tiene errores) de evaluar el código, es mostrado en una ventana, en la cual se pueden examinar con detalle variables, expresiones, etc.

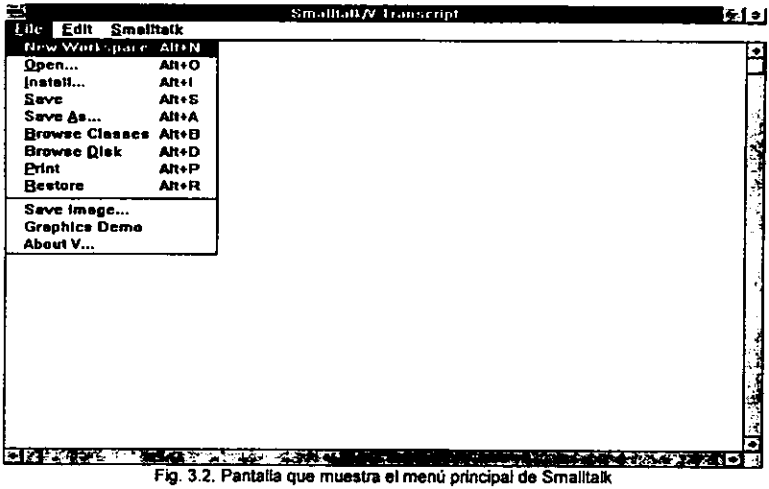


Fig. 3.2. Pantalla que muestra el menú principal de Smalltalk

Obsérvese la figura 3.2, en la cual se muestran las opciones del menu File: **New WorkSpace** crea una ventana nueva donde podemos editar texto y código, probarlo, etc., **Open** abre un archivo para su edición, **Save** salva el archivo que se edita, mientras **Save as** permite guardar el archivo que se edita con otro nombre. **Browse Classes** permite visualizar el árbol de clases de Smalltalk, similarmente existe **Browse Disk** con la cual visualizamos la estructura del árbol de archivos del disco duro o unidad lógica en que estemos posicionados. La opción **restore** limpia la ventana, se asemeja al comando cls de DOS, **Save Image** salvará la imagen actual que se tiene, de tal forma que al tener posteriormente otra sesión Smalltalk traerá la misma imagen que se guardó, se recomienda el uso de esta opción hasta salir completamente de Smalltalk.
Con respecto a esta última opción, al querer salir de la sesión actual de Smalltalk, se le presenta al usuario una ventana de diálogo con tres opciones:

1. **Cancel**, provoca que continuemos en Smalltalk como si nada hubiera ocurrido.
2. **Yes,** salva la imagen o sesión actual para que sea utilizada la próxima vez que ingresemos a Smalltalk. En el paradigma OO a esto se le conoce como persistencia.
3. **No,** sale de Smalltalk sin salvar nada de lo que se ha hecho durante la última sesión.

3.2 Ventanas

En esta sección se ahondará más sobre las ventanas y ciertos tipos de ellas. En la siguiente gráfica (Fig. 3.3) se observan tres ventanas: una de **Workspace**, otra es un **Browse disk**, y la ventana de la parte inferior corresponde a un **Browse classes**.

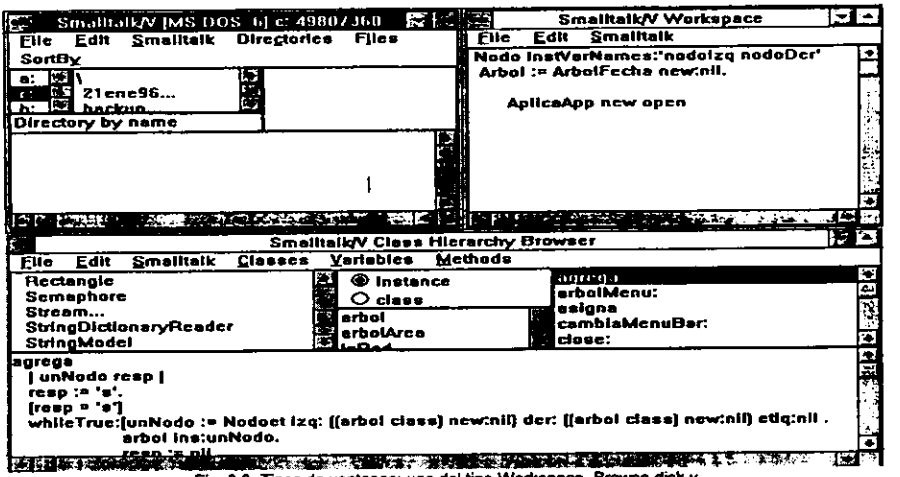


Fig. 3.3. Tipos de ventanas: una del tipo Workspace, Browse disk y otra del tipo Class Hierarchy Browser

La ventana del **Browse disk** tiene tres divisiones, dos superiores y una inferior. En la primera de ellas aparece el árbol de directorios donde se encuentra instalado Smalltalk; la segunda división muestra los archivos del directorio que esté enmarcado en la primera división; y la división de abajo muestra el contenido de algún archivo enmarcado. La división del árbol de subdirectorios tiene las opciones de **create**, que permite crear un nuevo subdirectorio, **update** que cambia la unidad de disco o repite la lectura de la misma unidad, **remove** que borra un subdirectorio y todos los archivos que este contenga y **hide/show** que oculta o muestra un subdirectorio. En esta ventana se tiene la opción **Files**, la cual ofrece las siguientes opciones: **remove** (borra un archivo), **print** (imprime una archivo), **mode** (cambia los atributos de un archivo), **rename** (renombra un archivo), **copy** (copia un archivo), **create** (crea un archivo). La división inferior ofrece las opciones comunes de un editor de texto.

Cuando se elige **Browse classes** dentro del menu **File** aparece la ventana llamada **Class Hierarchy Browser**, la cual contiene todo el conjunto de clases que proporciona Smalltalk, además contiene el código fuente de las mismas y a la vez nos permite crear el código propio.

El programador tiene también a su disposición otras ventanas que son herramientas como: *Debuggers, Prompters, Inspectors, y Walkback* (ventanas con mensajes de error).

Cuando se abre una ventana del tipo **WorkSpace**, esta nos servirá para escribir código, probarlo, depurarlo, etc., ofrece también las opciones comunes en la edición de texto.

3.3 Ventanas y menúes

Las operaciones sobre las ventanas son análogas a las que proporciona Windows, como son: minimizar, maximizar, mover, cerrar, etc.

La mayoría de las ventanas tienen una división para edición de texto, la edición es posible solamente si el cursor se encuentra dentro de esta división. Para ejecutar un pedazo de código se tiene que seleccionar dicho texto o pedazo de código, lo cual se logra colocando el cursor al principio del texto, presionando el botón izquierdo del ratón y arrastrándolo hasta el final del texto. Para ejecutar el código enmarcado, el usuario presionará el botón derecho del ratón, accesando al menú correspondiente. De las opciones presentadas **show it** y **do it** son por mucho las más importantes. Activando alguna de éstas, se ejecutarán dentro del bloque seleccionado, las instrucciones que ahí se encuentren, esto es un "programa" en Smalltalk.

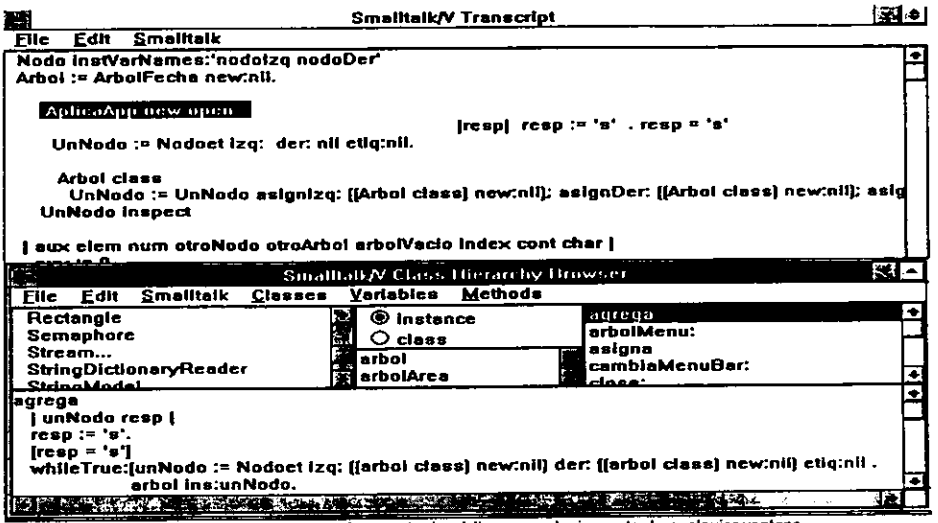


Fig 3.4. La ejecución de un fragmento de código en cualquier parte de cualquier ventana

Cabe resaltar que basta únicamente con teclear fragmentos de código, enmarcarlos y se estará en condiciones de evaluarlo (figura 3.4), esto es de mucha ayuda para el programador, ya que no se requiere todo un bloque , o todo un programa para probar el código.

Todo programa en Smalltalk regresa un valor, la única diferencia entre **show it** y **do it** es el hecho de que el primer comando si despliega en la pantalla el resultado del cómputo. El resultado de evaluar la expresión enmarcada dentro de la figura previa se muestra a continuación (Fig. 3.5).

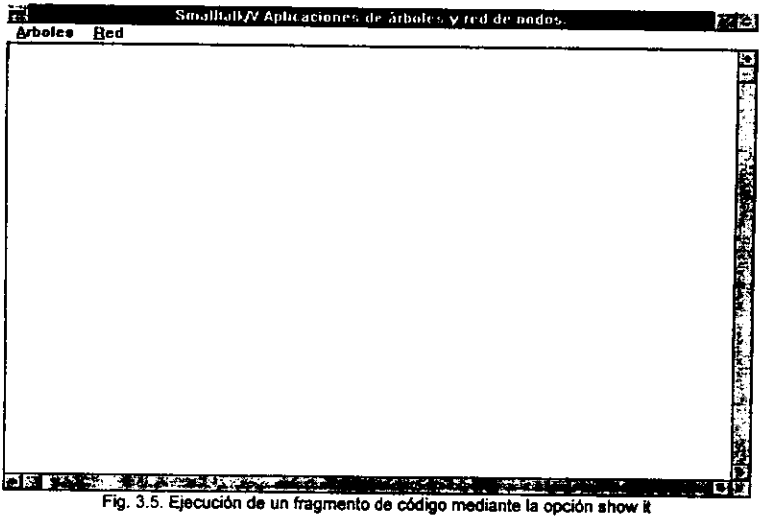


Fig. 3.5. Ejecución de un fragmento de código mediante la opción show it

Antes de hablar de los métodos ó mensajes, describiremos primero los objetos, ya que hacia ellos se dirigen los mensajes. Para hablar de ellos, necesariamente se deben crear instancias, y para crear instancias, se deben crear clases, así que empezaremos por describir como crear clases en Smalltalk.

3.4 Clases e Instancias

Como se expuso en el capítulo previo una clase describe la implementación de un conjunto de objetos; los objetos que como entes individuales son descritos por una clase son llamados instancias. Una clase describe cómo serán las memorias privadas (a esto se le llama *identidad*, 2.1) de sus instancias y cómo ejecutarán sus operaciones. La programación en Smalltalk consiste en crear clases e instancias de éstas y de especificar un conjunto de mensajes para que los objetos (instancias) puedan interactuar entre sí. Las propiedades públicas de un objeto son los mensajes que recibe su interfaz. Todas las instancias de una clase tienen la misma interfaz de mensaje, ya que ellos representan a la misma cosa. Las propiedades privadas de una instancia son sus variables de instancia y un conjunto de métodos que describen como ejecutar sus operaciones; cada instancia tiene su propio conjunto de variables de instancia. El concepto de *modularidad* queda definido en Smalltalk de manera natural, ya que las clases mismas representan módulos por medio de los cuales subdividimos nuestro problema (secc. 2.1).

Los valores (variables de instancia) por ejemplo de un Rectángulo son dos, ambos son instancias de la clase `Point` y conforman las esquinas opuestas que delimitan el área del rectángulo. Una clase incluye un método para cada tipo de operación que sus instancias puedan realizar. Un método puede provocar algunos

cambios en la memoria privada de un objeto y/o a su vez enviar otros mensajes; también especifica el valor que será devuelto al mensaje invocador. Los métodos de un objeto pueden accesar as variables de instancia del objeto en el que residen pero no a las variables de instancia de otros objetos, esto es lo que llamamos *encapsulación*.

3.5 Clases y métodos (mensajes)

Previo a la exposición de cómo implementar un método dentro de un clase, cabría mencionar que *todo en Smalltalk es un objeto*; este axioma es aplicado consistentemente a todos los aspectos del lenguaje. Las estructuras de control, rastreador, compilador, supervisor de código (code inspector), enteros, *booleanos*, reales, cadenas, rectángulos, puntos, líneas, formas (gráficos), pantallas y ventanas son objetos, los cuales reaccionan a mensajes. Un mensaje tiene tres componentes: el *receptor*, el *selector*, y (opcionalmente) *parámetros*. En Smalltalk, mensaje, método y selector son sinónimos.

Esta sección está dedicada a la descripción y creación de una clase y sus correspondientes métodos en Smalltalk.

Clases

Algo de suma importancia es el hecho de que Smalltalk proporciona el código fuente de la mayoría de sus clases, por lo que se puede accesar cualquier método (código fuente) de alguna clase por medio de la ventana **Class Hierarchy Browser** (de aquí en adelante se hará referencia a ésta como CHB) y el código queda a disposición del programador. A cada clase se le pueden agregar métodos de nuestra propia creación, y se puede decir que la ventana CHB es el editor, compilador e interprete del código a la vez.

Antes de explicar cómo crear una clase, es importante mencionar que gran parte del éxito de un buen desarrollo de cualquier aplicación OO recae en el hecho de que se debe estar completamente consciente en elegir con cuidado las clases que servirán como soporte para el propósito en cuestión. Es decir, de todo el conjunto de clases con que cuenta Smalltalk, incluyendo las creadas por el programador, se deben elegir aquellas cuyas instancias se adecuen lo mejor posible al modelo que se pretende simular.

La manera de crear una clase es la siguiente, en la parte que corresponde a las clases del CHB, elija la clase que funcionará como la clase padre de la que se está creando (nos encontramos con el concepto de *jerarquía*, secc. 2.1.1). A continuación presione el botón derecho del ratón; aparecerá un menú con algunas opciones en la parte inferior, elíjase la opción **add subclasses**, posteriormente

Smalltalk pregunta por el nombre de la clase mediante un **Prompter**, se proporciona el nombre y con esto se ha creado la nueva clase.
Si elegimos la nueva clase, en la división inferior aparecerá algo similar a lo siguiente:

```
<Nom. Clase Padre> variableSubclass: #<Nom. Nuestra Clase>
   instanceVariableNames:
        ''classVariableNames: ''
        ''poolDictionaries: ''
```

En el extremo superior izquierdo aparece el nombre de la clase padre, y en el extremo derecho el nombre de la clase que nos ocupa. A continuación, aparece la cadena *instanceVariableNames* , inmediatamente de la cual se muestran entre apóstrofos los nombres de las variables de instancia de la clase. Podemos ampliar nuestro conjunto de variables de instancia por medio del mensaje (a la clase) llamado `instVarNames:`. Dicho mensaje o método es entendido por todas las clases y se encuentra en la clase `Behavior`. A continuación de la cadena *classVariableNames* se listan las variables de clase. En la parte más baja aparece *poolDictionaries*, esta parte hace referencia a constantes del lenguaje como son `eof` (fin de archivo), `cr` (retorno de carro), etc.
Una vez que se han declarado cuáles son las variables de instancia, variables de clase y constantes; el siguiente paso es la creación de la interfaz de la clase.

Métodos

Smalltalk tiene dos tipos de métodos:

1. métodos de instancia (*instance methods*) y
2. métodos de clase (*class methods*)

Los métodos de instancia de una clase, forman la interfaz de las instancias de dicha clase con el medio ambiente.

Los métodos de clase son utilizados para crear instancias inicializadas, en C++ a este tipo de métodos se les llama **constructoras**. Cabe señalar que Smalltalk no maneja la contraparte de las constructoras de C++, las llamadas **destructoras**; para esto, Smalltalk hace uso de un "recolector de basura", de tal suerte que el programador no necesita preocuparse por la liberación de memoria.

Ciertos métodos conocidos como **primitivos** no pueden ser modificados, es decir, el código no está a la vista del programador. Tales métodos están escritos en una máquina virtual (lenguaje de máquina) y no pueden ser alterados por el programador. Ellos se invocan de la misma forma que otros métodos, y permiten tener acceso a hardware básico. Por ejemplo, las instancias de un entero utilizan un método primitivo para responder al mensaje +, otros métodos primitivos realizan interacciones entre disco y terminal.

Los *métodos* en Smalltalk resultan ser código, son los algoritmos que determinan la ejecución y comportamiento de un objeto, un ejemplo

```
| y |
y := Array new:10
```

Se tiene una variable temporal a la cual se le asigna un arreglo de 10 elementos. Existen mensajes para la Clase y mensajes para sus instancias, de aquí que los mensajes de clase sean métodos de clase.

En el ejemplo, quien recibe el mensaje es la clase `Array`, el mensaje (un método de clase) es `new:`, quien a su vez recibe un parámetro, en este caso 10. En el ejemplo anterior, el método `new:` crea una instancia inicializada de la clase `Array`, reservando dicho arreglo para 10 elementos. Con lo anterior se ha creado una instancia de la clase `Array`, una vez creada la instancia de la clase, podemos proceder a enviarle mensajes que le resulten comprensibles (esto lo podemos ver como la *abstracción* del objeto del mundo real), dichos mensajes son los métodos de instancia.

A la instancia del ejemplo anterior podemos enviarle mensajes como `at:put:`, el cual accesa y modifica el contenido de un arreglo.

```
1 to:10 do:[ :i| y at:i put:1]
```

En la expresión anterior se creó la instancia "y" para contener 10 objetos, ésta recibe el mensaje `at:put:` el cual modifica todo el contenido del arreglo, por medio del ciclo `to:do:`.

Tipos de mensajes

De acuerdo al número de parámetros, existen tres tipos de mensaje: *unario*, *binario* y *keyword*. Los mensajes unarios no tienen parámetros, solamente un receptor y un selector, por ejemplo

```
beta sin
```

El mensaje `sin` es enviado al número identificado por `beta` (receptor). Otro tipo de mensajes son los binarios, los cuales tienen un parámetro, algunos de éstos son:

```
1+3
```

1 es el receptor, + es el selector y 3 es el argumento. Los mensajes aritméticos son los principales representantes de los mensajes binarios, los cuales son siempre computados de izquierda a derecha. Por ejemplo, la operación

```
1+2*3
```

será evaluada incorrectamente, el resultado es 9 en lugar de 7. El uso de paréntesis es necesario para una correcta evaluación.

Los mensajes con dos parámetros `(at:put:)`:

```
#(1 2 3 4 5 6) at:2 put:3
```

El mensaje `at:` determina la posición dentro del receptor, en este caso el objeto es `#(1 2 3 4 5 6)`, mientras que `put:` sustituye el valor de su argumento en el objeto que es receptor. Para resaltar un mensaje *keyword*, quítense los parámetros y queda `at:put:`, lo cual provocará que el receptor cambie a

```
#(1 3 3 4 5 6).
```

En la siguiente sentencia

```
(beta)sin+2
```

los mensajes unarios son evaluados en primera instancia. Primero se computa `(beta)sin`, el mensaje unario `sin` es aplicado a `beta` y el resultado es sumado a 2. Un concepto elegante en Smalltalk es la *cascada de mensajes,* que resulta ser una serie de mensajes que tienen como destinatario al mismo receptor. Cada mensaje después del primero es precedido por punto y coma. Considérese la siguiente secuencia de declaraciones

```
#(1 2 3 4 5 6) at:2 put:3.
#(1 2 3 4 5 6) copyFrom:2 to:4
```

La primera declaración termina con un punto, en Smalltalk las expresiones o sentencias se separan unas de otras por un punto. En ambos mensajes el receptor es el mismo, `#(1 2 3 4 5 6)`, así que una *cascada de mensajes* podría ser más conveniente, hela aquí

```
#(1 2 3 4 5 6) at:2 put:3;copyFrom:2 to:4
```

donde el punto y coma nos sirve para separar cada mensaje. El primer mensaje at:put: coloca un 3 en la posición 2 del receptor, y el mensaje copyFrom:to: copia los elementos situados entre las posiciones 2 a 4, es decir los números 3,3 y 4.

Agregar y/o modificar un método

Si se desea modificar un método ya existente en alguna clase, colóquese en la segunda división del CHB (Fig. 3.6), en esta división aparecen en la parte inferior las leyendas **instance methods** y **class methods**.

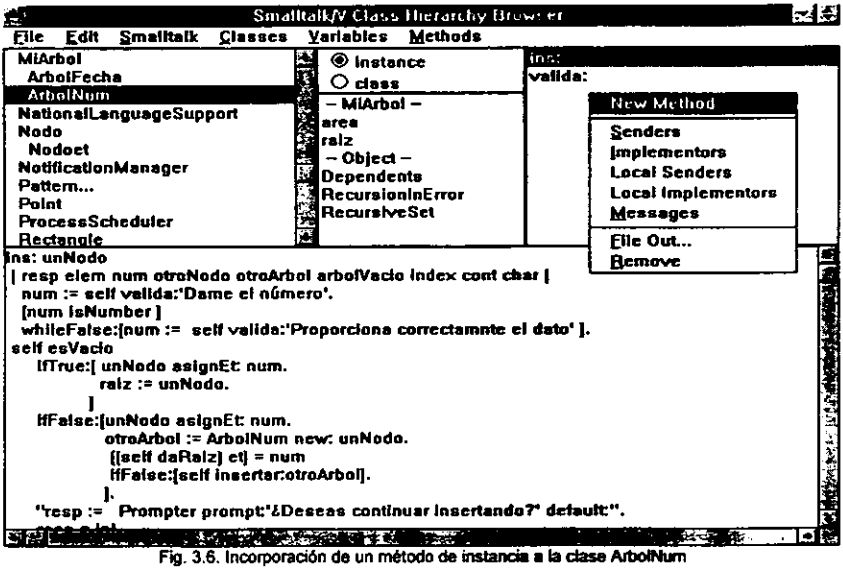


Fig. 3.6. Incorporación de un método de instancia a la clase ArbolNum

Habiendo seleccionado la clase deseada, coloque el cursor en alguno de los letreros mencionados y aparecerán los nombres (según la elección) de los métodos de la clase o los de instancia. Seleccionar alguno de estos métodos provocará que aparezca el código correspondiente en la división inferior del CHB. Posteriormente, coloque el cursor en donde desee realizar los cambios en el código, el ejemplo de la parte inferior muestra los métodos de instancia de la clase ArbolNum, la cual solamente tiene dos métodos de instancia. Al seleccionar el método, la parte inferior de la CHB cambia, mostrando el código fuente del método.

Para agregar un método haga lo anterior hasta la selección del tipo de método deseado (ver Fig. 3.7); a continuación sin salirse de la segunda división presione el botón derecho del ratón, aparecerá un menú, elija la opción `new method`, y automáticamente se nos envía a editar el código correspondiente en la división inferior del CHB, en la cual aparece

```
messagePatern
     "comments"
     | temporaries |
     statements
```

Lo anterior es el formato de un método; sus componentes son un patrón de mensaje, es decir, cómo será invocado el mensaje, si es unario, binario o *keyword*. El resto de los componentes de un método son: los comentarios (en Smalltalk van entre comillas), las variables locales si son necesarias y el código (*statements*). Una vez que ha terminado la edición del código, el siguiente paso es

salvarlo: presiónese una vez más el botón derecho del ratón, del menú que aparezca elija **save**, en ese momento Smalltalk se comporta como un compilador, revisando la sintaxis y la coherencia en el uso de las palabras reservadas. Si no existen problemas el nombre del nuevo método (*messagePatern*) se incrusta en la división correspondiente. Algo que llama la atención es el hecho de que un vez terminado de codificar un método, en la misma división, esto es, en la parte inferior de CHB, podemos probar el código que se acaba de escribir, es decir, prácticamente en cualquier lugar de una ventana que permita edición, se puede probar código.

Hasta aquí se ha mencionado todo lo que a interfaz y conceptos básicos de Smalltalk se refiere, se ha dicho como interactuar con Smalltalk a través de menúes. En POO como en otros estilos de programación, un programa se compone de tres cosas: secuencia, iteración y decisión. Los programas escritos en Smalltalk son una secuencia de instrucciones, las cuales pueden ser iterativas en algún punto y además es necesario tomar decisiones en ciertos momentos. La relevancia de la POO y en general del paradigma OO estriba en el hecho de que los programas se componen de "objetos" y peticiones entre ellos. Antes que nada se deben crear objetos(instancias), posteriormente se busca que dichos objetos interactúen a través de mensajes, para que al final se llegue a un resultado; pero, ¿esta forma de comportamiento no es acaso la misma que conforma nuestro entorno?. Smalltalk implementa lo anterior de una manera muy parecida a la realidad, nuestro entorno se compone de objetos lo mismo que el de Smalltalk. Para que los objetos cumplan la petición internamente utilizarán sus interfaces para establecer comunicación con otros objetos y así poder satisfacer la petición.

3.6 Variables

En Smalltalk existen dos tipos de variables, las *compartidas* y las *privadas*, cabe señalar que en este lenguaje las variables no tienen tipo asociado. Las primeras comienzan su nombre con mayúscula y las segundas con minúscula; las variables compartidas contienen solamente un objeto. Si se invoca una variable compartida que no se ha creado, Smalltalk pregunta si se desea crearla, si se confirma, el lenguaje usará dicha variable para comunicación con uno o varios programas; cuando se salva la imagen dichas variables quedan permanentemente como parte de Smalltalk (aparición del concepto de *persistencia*, ver secc. 2.1). De las llamadas variables compartidas; existen tres tipos de ellas (ver Fig. 3.7).

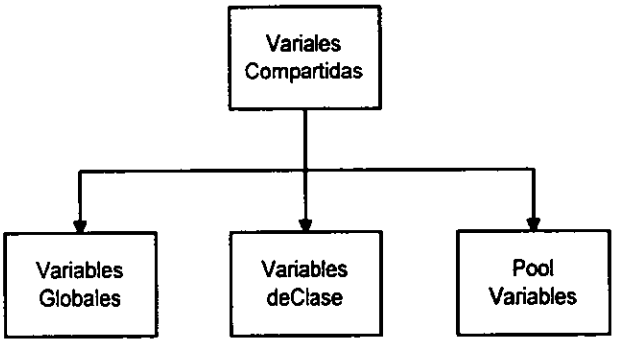


Fig. 3.7. Tipos de variables compartidas en Smalltalk

La diferencia entre estos tipos de variables reside en el grado de disponibilidad para los diferentes objetos que deseen usarlas. A continuación una breve descripción de ellas.

- ***Variables globales.*** Compartidas por todos los objetos.
- ***Variables de clase.*** Compartidas por todas las instancias de una clase. Se pueden accesar desde cualquier método de instancia y como es de esperarse las subclases tienen acceso a tales variables.
- ***Variables tipo "Pool".*** Son las compartidas por un subconjunto bien definido de subclases.

Las variables globales son almacenadas en una instancia especial de la clase `Dictionary` llamada `Smalltalk`. Los diccionarios y colecciones (arreglos, conjuntos, etc.) son asociaciones entre nombres (o llaves) y valores. Algunas variables globales ya están predefinidas en Smalltalk. Por ejemplo `Display` es una instancia especial de la clase `Screen` (clase con propósito gráfico) la cual hace referencia a la pantalla activa, y `Transcript` es una instancia de la clase `TextWindow` que permite que el texto sea desplegado en una ventana dentro de la pantalla, la cual se conoce como `System Transcript`. Para agregar una nueva variable global al diccionario llamado `Smalltalk`, se utiliza el mensaje `at:put:`, el cual sirve también para darle un nombre y un valor a la nueva variable.

```
Smalltalk at:#UnaVariable put:nil.
```

"Agrega el nombre UnaVariable y su valor inicial nil, al diccionario de variables globales llamado Smalltlak"

Ahora hablaremos de otro tipo de variable muy importante; la variable de instancia. Al hablar de una instancia (objeto) necesariamente se tiene que hacer referencia a su estado interno, que es lo que la diferencia de las demás instancias de la clase a la cual pertenece. El estado interno de una instancia de cualquier clase es definido por sus **variables de instancia** (conceptos de *identidad* y *encapsulación*, ver 2.1).

El término variable de instancia se refiere a un cierto tipo de variable global para las instancias de una clase. Ellas son accesadas por una instancia de la clase, es decir, por un objeto en particular. Las variables son globales en el sentido de que todas las instancias de dicha clases las pueden accesar y no globales porque son únicamente las instancias mismas quienes pueden accesar a ellas, de esta manera conservan su privacía. Los nombres de las variables de instancia son conocidos dentro de la clase pero son a la vez desconocidos fuera de ella; esto se conoce como ocultamiento de información.

Como hemos visto, además de las variables locales, globales y de instancia, también existen las llamadas variables de clase, estas últimas únicamente están disponibles para todas las instancias de dicha clase, se utilizan por lo regular para transferir datos de una ventana a otra. Por ejemplo, al cortar (`cut`) texto en una ventana y pegarlo (`paste`) en otra, dicho texto se almacena en variables de clase para editores de texto.

Las variables locales son descartadas tan pronto el programa ha terminado su ejecución, las variables locales son declaradas entre un par de barras verticales. Por ejemplo, cont y aux son un par de variables locales:

```
|cont aux|
cont := 0.
aux := 1.
cont := cont + aux.
```

El símbolo := es el operador de *asignación*, igual que en Pascal. Pero una asignación en Smalltalk es muy diferente, en Pascal una variable es identificada con una localidad de memoria y un valor contenido en dicha localidad, mientras que en Smalltalk la variable *apunta* a un objeto. Esto tiene consecuencias de largo alcance, Pascal (y en general en la programación bajo el paradigma orientado al procedimiento) requiere que se decida previamente el tipo de variable y éste será permanente mientras que en Smalltalk puede cambiar de tipo tantas veces como el programa lo requiera, esto se conoce como ligamiento tardío o *late binding*. En el siguiente ejemplo se ilustra como es que funciona esto y el manejo que hace Smalltalk de la memoria.

Al realizar una asignación en Smalltalk, se realiza polimorfismo, ya que una variable se transforma en una instancia de la clase a la cual pertenece el objeto involucrado en la asignación. Pasemos al ejemplo:

```
(1) |tot a1 a2|
(2) a1 := 5.
(3) a2 := 3.
(4) tot := a1 + a2.
(5) a1 := (a1 = a2).
(6) a2 := tot.
(7) tot := a1.
(8) ^a2.
```

Se tienen tres variables locales, al ejecutarse las cuatro primeras líneas del código las variables `a1`, `a2` y `tot` se convierten en instancias de la clase `SmallInteger`, al llegar a la quinta línea `a1` recibe el objeto `false` (que resulta ser una constante de Smalltalk) el cual es el resultado de la comparación entre `a1` y `a2`, el polimorfismo aparece aquí, en la manera como Smalltalk realiza una asignación. Ahora bien, se dijo que una asignación en Smalltalk es equivalente a decir que se crea un apuntador hacia ese objeto, lo relevante de este asunto es el hecho de que no se fija el contenido de la dirección, veamos lo que esto significa.

En la sexta línea del código `a2` recibe el entero 8, posteriormente a `tot` se le asigna otro valor, podría pensarse que `a2` tomará el último valor que le ha sido asignado a `tot`, pero esto no es cierto, es decir, la asignación en Smalltalk resulta ser un apuntador pero no en el sentido de C o Pascal. El símbolo ^ provoca que se regrese el resultado de la expresión, el equivalente del `return` en Modula 2, de hecho es una *expresión de retorno*, en el ejemplo regresa el valor de `a2` el cual es 8. Puede haber más expresiones después de una de retorno, pero no serán ejecutadas; en el ejemplo, no existen más instrucciones a ejecutarse.

De esto último podemos contestar una pregunta, ¿cómo crear una instancia idéntica a partir de otra, y que éstas sean independientes?. Para dar respuesta recordemos lo siguiente, lo que distingue a dos instancias de una clase, es su estado interno, pero solamente los métodos de instancia de la clase pueden accesar dicho estado, tales métodos serán útiles para poder duplicar una instancia. Para crear una copia de un objeto, se puede utilizar el método `copy` de la clase Object, el cual regresa como resultado un objeto que es copia del que recibe el mensaje. Por ejemplo, supongamos que deseamos crear una copia en `unaCopia` de `unObjeto`.

```
| unObjeto unaCopia |

unaCopia := unObjeto copy.
```

3.7 Operadores lógicos y bloques

Los operadores lógicos: <, >, <=, >=, =; son implementados por mensajes binarios, las comparaciones regresan **true** o **false**, por ejemplo usando **show it** para las siguientes expresiones se tiene

```
'Juan' = 'Juan'
```

regresa **true**, y

```
4+2=5
```

regresa **false**.

Todos los objetos entienden el mensaje de igualdad (=), y la mayoría de los objetos entiende el resto de las comparaciones.

Gran parte del poder de Smalltalk proviene de los llamados bloques (*blocks*), los cuales están asociados con expresiones condicionales, un bloque es algo análogo a un bloque **BEGIN...END** en Pascal. Un bloque en Smalltalk es una secuencia de instrucciones que serán ejecutadas en algún momento, la sintaxis de estos objetos es simple, una secuencia de expresiones encerradas entre corchetes.

Un bloque es evaluado después de haber recibido el mensaje **value**, el cual puede ir acompañado de uno o más argumentos. A continuación un bloque con un argumento

```
[:d| a:=d.
       a at:1 put:0]
```

La variable `d` es un argumento o parámetro del bloque, ya que está precedida por los dos puntos. La barra vertical denota el fin de la declaración de parámetros, y el comienzo del código ejecutable dentro del bloque; tal bloque no puede existir por sí solo, algo debe relacionarlo con el valor del parámetro; enseguida un ejemplo de un bloque con dos argumentos

```
|aux|
...
1 2 [:aux2:y| aux2:=aux+y]
```

El valor de `aux2` al igual que el de `y` será local al bloque. La manera de pasar argumentos a un bloque es anteponiéndole dos puntos, si se desea pasar mas de un argumento se debe dejar un espacio en blanco y anteponer nuevamente los dos puntos para el segundo argumento y así sucesivamente para n argumentos. Las siguientes dos expresiones tienen efectos idénticos:

```
 indice := indice + 1.
[indice := indice + 1] value.
```

Un bloque puede ser también asignado a una variable, de tal forma que si la expresión:

```
incrementaBloque := [indice := indice + 1]
```

es ejecutada, entonces la expresión

```
incrementaBloque value
```

incrementa el índice. Después de que el mensaje **value** se envía al bloque el objeto regresado es el valor de la última expresión. Así, si la expresión

```
sumaBloque := [indice + 1]
```

se ejecuta, otra forma de incrementar el índice es evaluar

```
indice := sumaBloque value
```

Un bloque vacío, al recibir el mensaje `value`, contendrá **nil**. La expresión

```
[] value
```

regresará `nil`.

Examinar la evaluación de la siguiente secuencia de expresiones, proporciona un ejemplo más extenso de la manera en que trabajan los bloques.

```
incrementaBloque := [indice := indice + 1].
sumaBloque := [suma + (indice * indice)].
suma := 0.
indice := 1.
suma := sumaBloque value.
incrementaBloque value.
suma := sumaBloque value
```

Lo anterior puede verse como sigue:

Asigna un bloque a `incrementaBloque`
Asigna un bloque a `sumaBloque`
Asigna el número 0 a `suma`
Asigna el número 1 a `indice`
Envía el mensaje `value` al bloque `sumaBloque`
Envía el mensaje *1 al número 1
Envía el mensaje +1 al número 0
Asigna el número 1 a `suma`
Envía el mensaje `value` al bloque `incrementarBloque`
Envía el mensaje +1 al número 1
Asigna el número 2 a `indice`
Envía el mensaje `value` al bloque `sumaBloque`
Envía el mensaje *2 al número 2
Envía el mensaje +4 al número 1
Asigna el número 5 a `suma`.

3.8 Controladores de flujo y expresiones lógicas

Los bloques son manipulados mediante la ejecución iterativa y condicional. Smalltalk tiene cuatro mensajes para la ejecución condicional: **ifTrue:**, **ifFalse:**, **ifTrue:ifFalse:** y **ifFalse:ifTrue:**. En el siguiente fragmento de código, si se evalúa cierta la desigualdad se regresa el objeto a, en otro caso se regresa b.

```
a < b
   ifTrue:[^a]
  ifFalse:[^b].
```

Smalltalk cuenta con dos pares de operadores booleanos predefinidos. El primer par es & y | ; y el segundo es **and:unBloque** y **or:unBloque**. Los operadores & y | toman siempre tanto al receptor como al argumento y nunca terminarán prematuramente, en cambio los operadores **and:** y **or:** provocan que la evaluación termine de manera prematura. Técnicamente a esto se le conoce como "evaluación booleana en corto circuito". Si el receptor de **and:** es falso, entonces a todo el mensaje se le asigna el valor de **false**, y el bloque que corresponda a la opción verdadera no será ejecutado, análogamente con **or:** y el valor **true**.

En el siguiente ejemplo, la segunda condición de un mensaje **or:** ó **and:** siempre lleva los corchetes, especificando que es un bloque. El mensaje **class** puede ser recibido por cualquier objeto y regresa como resultado la clase a la cual pertenece el objeto. Evaluándose la expresión lógica, se regresa una cierta cadena, dependiendo del resultado. Los nombre de clases en Smalltalk igual que los de las variables globales, empiezan con mayúscula.

```
| x y |
x:='mi cadena'.
y:=5.
x class=Integer and:[y class=String]
 ifTrue:[^'Cierto']
ifFalse:[^'Falso'].
```

3.9 Iteradores

Smalltalk cuenta con varias expresiones de iteración. La más básica es **to:do:**. Dicho mensaje tiene dos argumentos, el receptor resulta ser el límite inferior de la iteración; el primero de los parámetros es el límite superior, mientras que el segundo argumento es un bloque de código que será evaluado tantas veces como la iteración lo indique. El siguiente ejemplo inicializa un arreglo con ceros.

```
|a|
a:= Array new:10.
1:to:10 do:[:i|a at:i put:0].
```

Se declara una variable local y se crea un arreglo de 10 lugares enviándole un mensaje **new** a la clase Array para que cree el arreglo. Teniendo el arreglo ya inicializado, quisiéramos asignar un 1 a las posiciones impares, esto se obtiene como sigue:

```
1 to:10 by:2 do:[:j|a at: j put:1],
```

La sintaxis anterior provoca saltos de 2 posiciones en el arreglo por lo que el arreglo queda inicializado como deseamos.

3.10 Arbol jerárquico de clases en Smalltalk

Se mostrará el árbol jerárquico de clases en Smalltalk (Fig. 3.8), detallando aquellas ramas o subramas que se relacionan con las clases de interés para el presente trabajo.

La clase genérica es `Object`, la cual es la base para las demás clases, se dice que `Object` es indirecta o directamente la clase padre de todas las clases.
Una subclase directa de `Object` es `Collection`, la cual podemos observar más abajo con detalle (Fig. 3.9). Cuando al nombre de una clase lo suceden puntos suspensivos se está indicando que tiene subclases. `Collection` tiene varias subclases que dan origen a la clase `Array`; esta última es el soporte principal para la aplicación de hallar la ruta más corta dentro de una red de nodos.

Object
AnimatedObject
Behavior...
Boolean...
CallBack
ClassComparisonTool...
ClassReader
ClipboardManager
Collection...
Compiler
Context...
CursorManager
DIB
Directory
Dos
DynamicDataExchange
Font
GraphicsMedium...
CraphicsTool...
Icon
Magnitude...
Menu
MiArbol...
Nodo...
NotificationManager
ObjectFiler
Point
ViewManager...
Window...

Figura 3.8. Arbol de clases en Smalltalk

Las clases MiArbol y Nodo son también subclases directas de Object; MiArbol tiene tres subclases: ArbolFecaha, ArbolCadena y ArbolNum las cuales hacen referencia o objetos de las clases Date, String y Number respectivamente. De las últimas tres clases, dos son subclases directas de la clase Magnitude: Date y Number, String está al mismo nivel que la clase Array (ver árboles, Fig. 3.9).

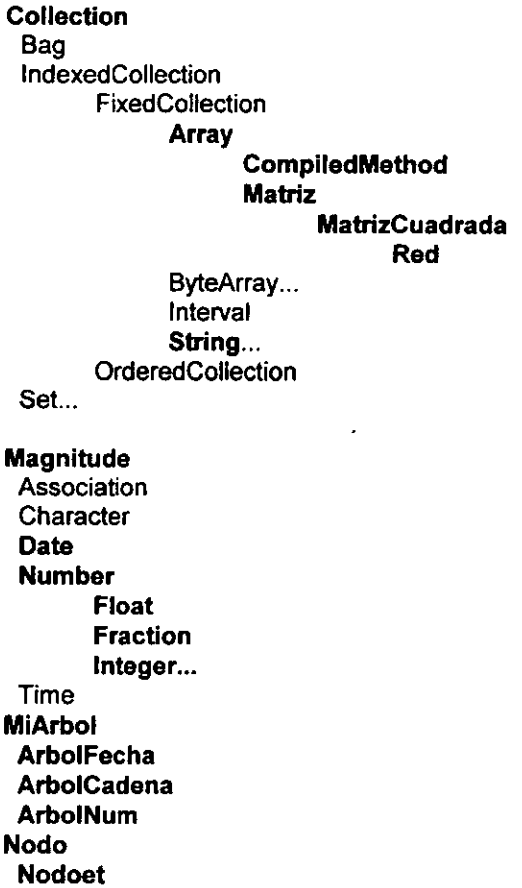


Figura 3.9. Las clases Collection y Magnitude

Otra clase muy importante es ViewManager, ésta permite crear aplicaciones independientes de Smalltalk (archivos ejecutables) que pueden interactúan en el medio ambiente Windows (Fig. 3.10). Las aplicaciones para el presente trabajo quedarán enmarcadas bajo la clase AplicaApp, esto quedará más claro al llegar al capítulo cinco ya que en él se muestran los detalles de cómo crear aplicaciones independientes de Smalltalk, basándonos en la clase ViewManager.

ViewManager
- **AplicaApp**
- ClassBrowser
- ClassHierarchyBrowser
- **DiskBrowser**
- GraphicsDemo
- Inspector
  - **Debugger**
  - DictionaryInspector
- MethodBrowser...
- **TextWindow**
- WindowDialog
  - AboutDialog
  - ObjectLoadDialog
  - PrintAbortDialog
  - **Prompter**
  - **SaveImageDialog**

Figura 3.10. La clase ViewManager

Capítulo 4
Implementación en Smalltalk

En este capítulo se exponen dos problemas que involucran los conceptos del paradigma OO mencionado en el capítulo 2 y se ve como se lleva a cabo la implementación de la solución a tales problemas en Smalltalk/V.

El primer problema se refiere a la implementación del tipo de dato abstracto árbol y el segundo al de encontrar la ruta más corta en una red de nodos. El autor del algoritmo que detecta el camino más corto es Dijkstra (Cormen, 1990), existen dos maneras de implementar este algoritmo, mediante apuntadores o a través de la matriz de incidencias. Se optó por la última, por lo cual fue necesario crear algunas clases: la clase `Matriz`, así como su subclase `MatrizCuadrada`.

En ambos problemas la intención es mostrar en primer término cómo se implementan en Smalltalk de manera natural, los conceptos del paradigma OO, y en segundo término realzar las características que hacen de este lenguaje de programación uno de los más bellos e interesantes. No es un objetivo del presente trabajo el mostrar una metodología de diseño OO.

Cabe señalar en este punto que el diseño de software es un proceso creativo y para el cual no existen algoritmos que garanticen un éxito completo, el diseño es algo que todavía depende de la habilidad, intuición, creatividad y experiencia del diseñador. Sin embargo, la definición más general de lo que es un diseño satisfactorio es aquella que cumple los requerimientos del sistema de la mejor manera posible.

Con el propósito de realizar un diseño satisfactorio de un modelo OO es necesario tener presente: (1) las clases disponibles, (2) la herencia entre las clases, (3) el comportamiento del software en función de interacciones entre objetos, (4) plantear el problema en componentes manejables. La manera en que cada uno de estos puntos sean cumplidos dependerá del conocimiento que el diseñador tenga del dominio del problema.

Ahora bien, la manera de abordar cada uno de los problemas mencionados es la siguiente:

- Se plantea el problema;
- Se realiza la abstracción de los objetos que conformarán el problema;
- Se determinan dentro de las clases existentes, las que serán necesarias para el problema;

- Se crean las clases que sean necesarias, partiendo de las existentes;
- Se describe la identidad y el comportamiento de los objetos que serán instancias de las clases que se crearon.

Para los tres últimos puntos será necesario introducir detalles técnicos propios de la implementación, en este caso de Smalltalk.

4.1 Implementación del tipo de dato abstracto árbol

Antecedentes

La idea original de este desarrollo partió de un curso de lenguajes de programación impartido en la Facultad de Ciencias de la UNAM, bajo la dirección de la Dra. Hanna Oktaba.

Planteamiento del problema

Se desea implementar el tipo de dato abstracto árbol.

Abstracción del problema

Un tipo de dato abstracto árbol tiene una raíz en ella se encuentra un nodo y este a su vez contiene dos "árboles".

Figura 4.1.1. La abstracción enmarca el comportamiento y las características que hacen diferente al objeto de los demás, y centra la visión externa del objeto relativa al observador

Clases disponibles en Smalltalk

De las clases provistas por Smalltalk resulta que solamente la clase `Object`, es decir, la clase que es padre de todas las clases directa o indirectamente, resulta útil para el propósito de crear "árboles".

Clases creadas

En la abstracción del problema se ha mencionado que el problema se compone de dos objetos: árboles y nodos. Las clases que se generaron para crear árboles y nodos son MiArbol y Nodo respectivamente, ambas tienen subclases.

Identidad y comportamiento de los objetos que conforman el problema

Se describen los objetos árbol y nodo, definiendo para cada uno de ellos su estado ó identidad y su comportamiento ó interfaz. Cabe señalar que únicamente se listan cada una de estas propiedades, el detalle vendrá en la implementación en Smalltalk.

Objeto árbol
- Estado(identidad): raíz, área.
- Comportamiento(interfaz):
- De clase: new:
- De instancia: inicializa:, área:, daRaiz, esVacio, max1:max2:, altura:, alto:, ins:, insertar:,
  Impdesp:, imprime:, do:, creaArbol, crea, arbolMenu.

Objeto nodo
- Estado(identidad): nodoIzq, nodoDer.
- Comportamiento(interfaz):
- De clase: izq:der:
- De instancia: inIzq:inDer, derecho, izquierdo, esHoja, etiquetas, do:.

A continuación se implementan ambos objetos en Smalltalk.

4.1.1 La jerarquía de la clase MiArbol.

La clase `MiArbol` es la responsable de crear "árboles" y es subclase directa de `Object`, a su vez `MiArbol` tiene subclases como veremos más adelante, por lo pronto baste con observar la gráfica

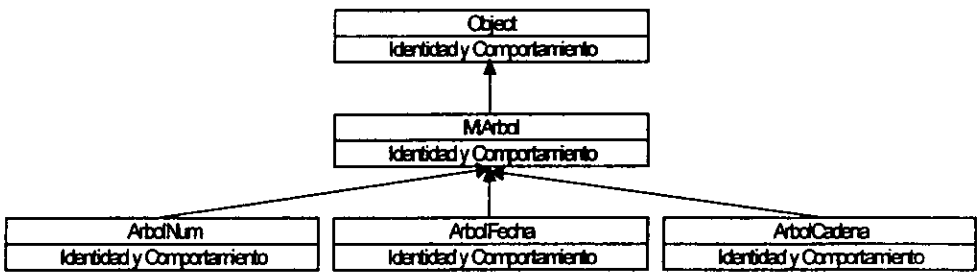


Fig. 4.1.2. Gráfica que muestra la jerarquía de la clase MiArbol en Smalltalk

En el siguiente fragmento de código se define la declaración de la clase `MiArbol`.

```
Object subclass: #MiArbol
  instanceVariableNames:
    ' raiz area'
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants'
```

La clase `MiArbol` tiene dos variables de instancia, `raiz` y `area`, de aquí que la identidad (Cap. 2 secc. 2.1) de las instancias de la clase `MiArbol` quedará determinada por el contenido de estas variables. La variable `area` es "la ventana" que nos servirá de salida para nuestras aplicaciones con árboles, por ejemplo, para desplegar el contenido del árbol o saber la altura del mismo.
La encapsulación (Cap.2 secc. 2.1) queda manifiesta con la creación de las variables de instancia, éstas definen el estado interno del objeto y ningún objeto puede modificarlas, a lo más, puede copiar el estado interno (Cap.2 secc. 2.1).
Se declara que se hará uso de constantes predefinidas como son entre otras `eol`(fin de línea), `cr` (retorno de carro), haciendo referencia a las constantes predefinidas en la parte referente a `'poolDictionaries'`(cap. 3 secc. 3.5) sobre las llamadas variables compartidas y diccionarios.

4.1.1.1 Métodos de clase (constructores) de la clase MiArbol.

Una vez creada `MiArbol` como subclase de `Object` se procede a crear instancias de dicha clase; como se ha dicho, Smalltalk tiene para este propósito los métodos de clase (Cap. 3 secc 3.6) y en el caso de la clase `MiArbol`, tenemos solamente uno de tales métodos, a saber, `new`:. Esto último queda mejor expresado si se da un ejemplo de cómo crear instancias de la clase `MiArbol` y se muestra el código de tal método.

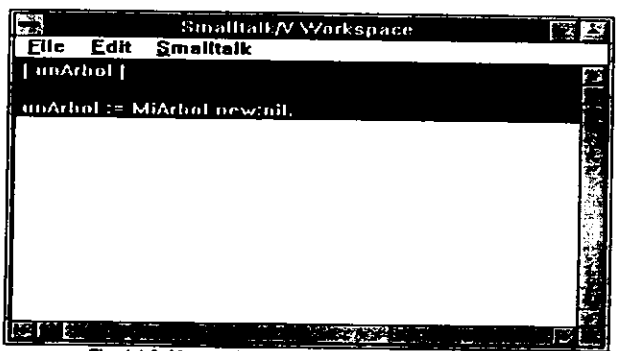


Fig. 4.1.3. Manera de crear instancias de la clase MiArbol.

Smalltalk tiene lo que se conoce como *pseudovariables*, y una de ellas es `self`, que resulta ser el receptor de algún mensaje. Obsérvese que en la figura 4.1.3 la clase `MiArbol` (vista como objeto) recibe el mensaje `new:`, la codificación de tal mensaje o método se expone a continuación.

```
new: unArbol
^self new inicializa: unArbol
```

En este caso, al invocar el mensaje, `self` es quien lo recibe, es decir, la clase misma (`MiArbol`) quien es receptor de `new`. El método `new` es genérico para cualquier clase ya que pertenece a la clase `Object` (Cap. 3 secc. 3.6). Hasta este momento se ha creado una instancia de la clase `MiArbol`, esto mediante el mensaje `new`; una vez creada la instancia ya podemos enviarle mensajes comprensibles a la misma (Cap. 3 secc. 3.6). Para tal caso se le envía el mensaje `inicializa:` con un parámetro (`unArbol`) y finalmente se regresa (Cap. 3 secc. 3.5) un objeto inicializado del tipo árbol.

4.1.1.2 Métodos de instancia (interfaz) de la clase MiArbol.

A continuación se describe la interfaz de la clase, es decir, los mensajes que las instancias de la clase `MiArbol` entenderán (Cap. 2 secc. 2.2.2); se exponen uno a uno todos los métodos que conforman la interfaz de las instancias de dicha clase. El primer método en turno es `inicializa:` del cual se ha dicho ya su propósito.

```
inicializa:arbol

"Este metodo es utilizado por el metodo de clase new: para
 crear e inicializar un arbol."

raiz := arbol
```

Este método asigna un valor a la variable de instancia `raiz`. El método `area:` recibe como parámetro (`unArea`) una instancia de la clase `TextPane` (en Smalltalk los comentarios están entre comillas); tal parámetro es asignado a la variable de instancia `area`. Esto quedará más claro cuando se genere la aplicación y se pongan todas las piezas juntas.

```
area:unArea

"Este metodo tiene como propósito inicializar la variable de
 instancia area."

area := unArea.
```

El método `daRaiz` regresa el contenido de la variable de instancia `raiz` y el método `esVacio` regresa cierto o falso dependiendo si `raiz` es o no `nil`. Smalltalk tiene un mensaje `isNil` que evalúa si el receptor es `nil`, al igual que asigna `nil` a toda variable que se crea y no ha sido inicializada[**] . Cuando se crea una instancia la variable `raiz` tiene como valor `nil`, es por tal motivo que se invoca al mensaje `inicializa:`, ya que este último le asigna un objeto a la variable de instancia o lo que es lo mismo le proporciona "identidad" a la instancia.

```
daRaiz

"Regresa la variable de instancia raiz"

^raiz


esVacio

"Regresa cierto si la raiz es nil e.d. el arbol es vacio."

^raiz isNil
```

El método `max1:m1 max2:m2` regresa el máximo de dos números, `m1 y m2`.

```
max1:m1 max2:m2

"regresa el máximo de dos números, m1 y m2"

m1 < m2
 ifTrue:[ ^m2 ]
ifFalse:[ ^m1 ]
```

Toca el turno al método para hallar la altura de un árbol, y para explicar éste método supóngase por un momento que en la variable `unArbol` está contenido el siguiente árbol

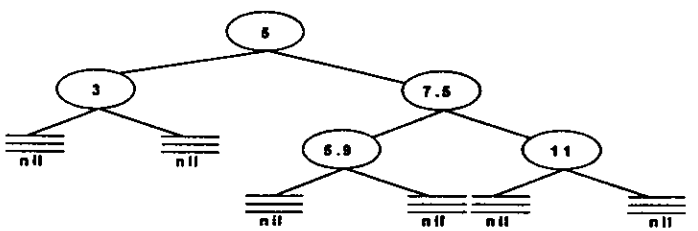


Fig. 4.1.4. Contenido de la variable unArbol.

[**] Cada vez que se crean instancias de cualquier clase sus respectivas variables de instancia contienen nil.

Ahora bien, la manera de invocar al mensaje `altura` es mostrado en la figura 4.1.5. El resultado del mensaje se muestra en la figura 4.1.6.

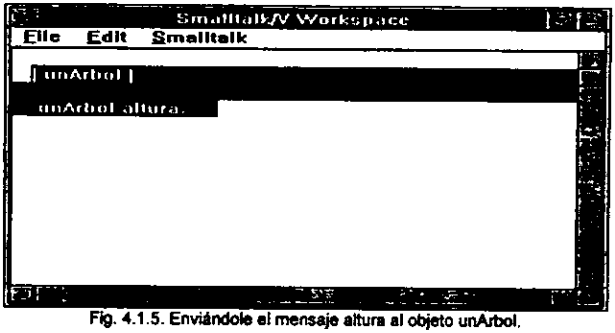


Fig. 4.1.5. Enviándole el mensaje altura al objeto unArbol.

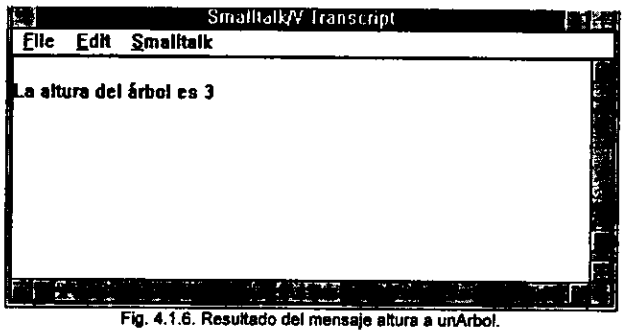


Fig. 4.1.6. Resultado del mensaje altura a unArbol.

La petición original la podemos ver en la Fig. 4.1.5; el objeto `unArbol` debe responder al mensaje `altura`. El objeto receptor del mensaje lo comprenderá ya que la interfaz de este contiene tal método.

```
altura

 | temp |

 temp := self alto.
 area contents:''.
 area appendText:'La altura del arbol es ',temp printString.
 area cr.
```

El mensaje `altura` invoca a su vez al mensaje `alto`, éste último responderá regresando un valor (la altura) el cual será asignado a la variable temporal `temp`. Sabemos que las instancias de `MiArbol` contienen dos variables de instancia: `raiz` y `area` (ver Fig. 4.1.2), la variable de instancia `area` contiene una instancia de la clase `TextPane`. Las tres últimas líneas del método contienen mensajes

para el objeto area, los cuales resultarán en el despliegue de la altura del árbol (ver Fig. 4.1.6).

Como se ha observado, un mensaje enviado a un objeto desencadenará a su vez el envío de más mensajes o peticiones hacia otros objetos con el fin de satisfacer la petición original, ésta es la esencia de la POO.

Al principio de este capítulo se mencionó (Sección 4.1.1.) que un objeto de la clase `MiArbol` se caracterizaba por contener entre otras cosas, un nodo, el cual a su vez contenía dos objetos del tipo árbol; tales objetos son los subárboles derecho e izquierdo.

Es importante tener presente lo anterior ya que el método `alto` y otros más "visitan" estos subárboles. A continuación el método `alto`.

```
Alto

self esVacio
 ifTrue:[^0]
ifFalse:[raiz esHoja
          ifTrue:[^1]
         ifFalse:[^1 + ((self max1:((raiz derecho) alto)
                              max2:((raiz izquierdo) alto)))]
        ]
```

El algoritmo es recursivo, si el árbol es vacío la altura es cero, si no fuera así entonces revisamos el contenido de la raíz, si la raíz es hoja entonces querrá decir que tenemos únicamente un nodo y regresamos uno como altura. Si no es ninguno de los casos mencionados entonces se realiza la recursión sobre los subárboles izquierdo y derecho y la altura del árbol será el máximo de las alturas de los subárboles izquierdo y derecho.

La variable de instancia `raiz`, la cual contendrá el nodo, recibe los mensajes `derecho` e `izquierdo`, estos mensajes regresan los subárboles derecho e izquierdo respectivamente, entonces el mensaje `alto` es invocado recursivamente para cada subárbol. Finalmente se comparan los resultados de los dos subárboles por medio del método `max1: max2:`.

La posibilidad de insertar nodos en el árbol se implementa por cada una de las subclases de MiArbol, de la figura 4.1.2 se ve que el tipo de elementos que se pueden insertar en el árbol; pertenecen a la clase `Magnitude` (ver fin Cap. 2, secc 2.11), algunas de sus subclases son: `Date`, `Number`, `Time`, etc. A su vez por ejemplo, `Number` tiene algunas subclases como: `Integer`, `Float`, `Fraction`[**] . Es decir, todas las instancias de estas clases atenderán a los mensajes <, > , <= , >=, =, etc; lo cual permite insertar elementos en el árbol en forma ordenada. En el código de éste método se observa que la codificación es

[**] Cabe aclarar que un árbol solamente puede contener subárboles del mismo tipo

responsabilidad de alguna de sus subclases, en otras palabras, cada una de las tres subclases de `MiArbol` (`ArbolNum`, `ArbolDate` o `ArbolCadena`, gráfica 4.1.2) implementa el mensaje `ins:`.

```
ins: unNodo

^self implementedBySubclass
```

Este método ejemplifica el concepto de polimorfismo (Cap. 2 secc. 2.3), es decir, el mensaje `ins:`, que permite la inserción de nodos en el árbol, es implementado por las tres subclases de MiArbol. En otras palabras, al momento de realizar una inserción de algún nodo, el mensaje `ins:` "sabrá" cual de los tres métodos utilizar dependiendo del tipo de objeto que contenga la etiqueta del nodo a saber: una cadena, una fecha o un número.

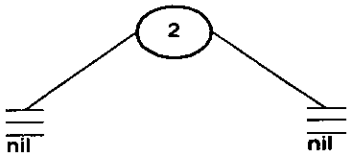
La inserción funciona como sigue; si el árbol es vacío entonces el parámetro es colocado en la raíz del árbol, si resulta que no es vacío entonces existe al menos un nodo en la raíz, a continuación comparamos las etiquetas tanto del parámetro como de la raíz del nodo y dependiendo del resultado de la comparación se inserta ya sea en el subárbol izquierdo, derecho, o se regresa sin llevar a cabo la inserción.

```
(1)insertar:unArbol

(2) | e1 e2 |
(3)
(4) e1 := (unArbol daRaiz) et.
(5) e2 := (self daRaiz) et.
(6) e1 = e2
(7)  ifTrue:[^self].
(8) e1 > e2
(9)  ifTrue:[ ((self daRaiz) derecho) esVacio
(10)            ifTrue:[ (self daRaiz) asignDer:unArbol.
(11)                     ^self]
(12)           ifFalse: [ ((self daRaiz) derecho)
                          insertar:unArbol]]
(13)ifFalse:[ ((self daRaiz) izquierdo) esVacio
(14)             ifTrue:[ (self daRaiz) asignIzq:unArbol.
(15)                      ^self]
(16)             ifFalse:[ ((self daRaiz) izquierdo)
(17)                      insertar:unArbol]]
```

El método `insertar` es utilizado por las subclases de `MiArbol`, más específicamente por el método `ins:`. A continuación se explica cada línea del método. Se recibe como parámetro `unArbol`, se declaran dos variables locales `e1` y `e2`, las cuales guardan las etiquetas de las raíces del parámetro y el receptor (la instancia misma) respectivamente.

Para facilitar la explicación, supóngase que se tiene un árbol de números y que solamente se tiene un nodo, como se muestra en la siguiente figura.



Figurra 4.1.7

Se desea insertar un nodo más, digamos el número 7.4, entonces el árbol se vería así después de la inserción.

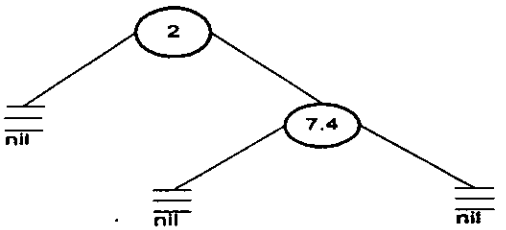


Figura 4.1.8

Como se dijo, el algoritmo compara la etiqueta del nodo que está en la raíz del árbol, en el ejemplo la etiqueta resulta ser 2, contra la etiqueta del nuevo nodo que se va a insertar (línea 8). Si la etiqueta del parámetro resulta ser menor que la etiqueta de la raíz del árbol entonces el nodo será colgado en el subárbol izquierdo (línea 13) en caso contrario el nodo será colgado en el subárbol derecho (línea 9), como sucedió en el ejemplo. Si se diera el caso de que las etiquetas fueran iguales entonces el algoritmo regresa el mismo árbol sin insertar el nodo (líneas 6 y 7).

En la octava línea se realiza la comparación de las etiquetas, analizaremos el caso en el cual la etiqueta del parámetro resulta ser mayor que la etiqueta de la raíz del árbol, el caso contrario es análogo.

La variable de instancia `raiz` de un árbol siempre contendrá un nodo etiquetado, de aquí que lo que regresa `daRaiz` es una instancia de la clase `Nodoet`. Por lo anterior se puede responder al mensaje `derecho` (ver sección 4.1.2.3), el cual regresa el subárbol derecho. A continuación se pregunta si éste último es vacío, en caso afirmativo el método `asignDer:` (secc. 4.1.2.6) asigna el nodo a la raíz del subárbol derecho, y entonces se regresa el árbol con un nodo más (líneas 9, 10 y 11). Si resulta que el subárbol derecho no es vacío, entonces se invoca recursivamente el método pero ahora con receptor el subárbol derecho del árbol original (línea 12). En el ejemplo que se expuso, tanto el subárbol derecho como el izquierdo eran vacíos.

Este método es otro buen ejemplo de lo que significa programación bajo el paradigma OO, ya que dado un mensaje dirigido a un objeto, se desencadena la creación de objetos y estos se envían mensajes entre sí, cooperando de esta forma para responder a la petición hecha. Esto también es motivo para mostrar la terminología utilizada: objetos, mensajes, respuestas, receptores de mensajes etc., para ello citemos la siguiente línea de código del método expuesto en cuestión.

```
((self daRaiz) derecho) esVacio
```

Muchas veces se desea saber si uno de los subárboles es vacío, el hecho de ser vacío implica que la variable de instancia raiz sea `nil`. Al receptor del mensaje (`self`) se le envía el mensaje `daRaiz`, este último regresa la "raíz" de un árbol o lo que es lo mismo la variable de instancia `raiz`, la cual contiene un "nodo". Los nodos "entienden" el mensaje `derecho` (ver secc. 4.1.2.3), el cual tiene como receptor un objeto del tipo nodo y lo que regresa es el subárbol derecho del nodo (ver 4.1.2.3). Una vez obtenido el subárbol derecho "se le pregunta" si es vacío.

Continuando con la descripción de la interfaz de la clase `MiArbol`, toca el turno al método `imprime:`, éste será responsable de imprimir el contenido del árbol en un ventana e invocar al método `impdesp: en:`, el cual recibe como parámetros `nivel` y `unArea` (en ese orden). El primero de los parámetros se utiliza para controlar la indentación en la impresión y el segundo es el lugar donde será impresa la salida.

```
imprime: unArea

unArea contents:''.
self impdesp:0 en: unArea.
```

El método `impdesp:en:` es invocado por `imprime:`.

```
(1)impdesp:nivel en: unArea

(2)| aux1 blancos aux2 aux3 |

(3) aux2 := ' '.
(4) blancos:= 0.
(5) self esVacio
(6) ifTrue:[UnArea appendText:'nil'.
(7)         UnArea cr.
(8)         ^self].
(9) ((self daRaiz) esHoja)
(10) ifFalse:[aux1 := raiz.
(11)          UnArea appendText:(raiz et) printString.
(12)          UnArea cr.
(13)          aux3 := nivel + 1.
```

```
(14)            1 to:aux3 do:[:i|UnArea appendText:' '.].
(15)            (raiz izquierdo) imp:aux3.
(16)            1 to:aux3 do:[:i|UnArea appendText:' '.].
(17)            (raiz derecho) imp:aux3.
(18)          ]
(19) ifTrue:[UnArea appendText:(raiz et) printString.
(20)            UnArea cr.]
```

Como consecuencia del código anterior, se imprime la constante `nil` si el árbol resulta ser vacío (líneas 5 a 8); si por otra parte el árbol no tiene subárboles, únicamente se imprime la etiqueta de la raíz por medio de la sentencia `unArea appendText:(raiz et) printString` (líneas 19 y 20). El mensaje `et` pertenece a la clase `Nodoet` y su propósito es regresar la etiqueta del receptor, además de la etiqueta se imprime también un retorno de carro (`unArea cr`) lo anterior con las dos últimas líneas del código.

Si el nodo no es hoja se imprime la etiqueta de la raíz del árbol (línea 11), la variable `aux3` incrementa el nivel en el cual nos encontramos, esto último para llevar la indentación en el despliegue del árbol (línea 13). Se imprimen tantos espacios en blanco en función del nivel de profundidad en que nos encontremos (líneas 14 y 16). La recursión se presenta al invocar nuevamente al método pero recorriendo el subárbol izquierdo, y de manera análoga para el subárbol derecho. El resultado de imprimir `unArbol` se muestra en la gráfica 4.1.9

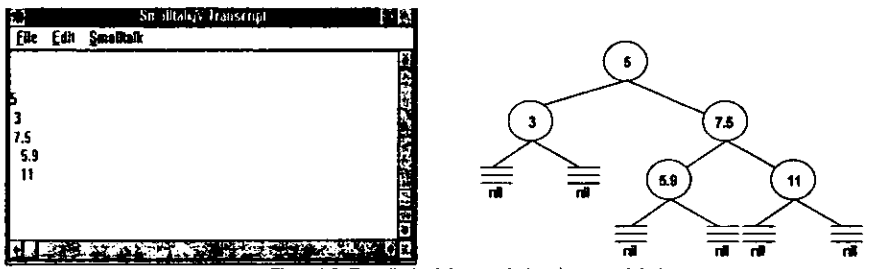


Fig. 4.1.9. Resultado del mensaje imprime a unArbol.

Nótese la indentación, la "raíz" es 5, sus subárboles izquierdo y derecho son 3 y 7.5 respectivamente, de estos solamente el subárbol derecho el cual está etiquetado con el numero 7.5 tiene subárboles, de los cuales el izquierdo es 5.9. y el derecho 11.

En el capítulo 3 se dijo que gran parte de la fuerza de Smalltalk proviene del uso de los bloques (Cap. 3 secc. 3.8). Pues bien, el método `do:` evalúa el bloque que se le pase como parámetro cada vez que se encuentre o visite un nodo dentro del

árbol. El mensaje se le envía a la raíz del árbol y a las respectivas raíces de los subárboles. El algoritmo evalúa si el árbol es vacío, si resulta que no lo es, se invoca al mensaje `do:`, el cual tiene a la variable de instancia `raiz` del árbol como receptor del mensaje.

```
do:unBloque

" Metodo que evalua unBloque en la raiz de cada instancia de
  la clase   MiArbol."

self esVacio
 ifFalse: [raiz do:unBloque]
```

Este método (`do:`) ejemplifica el concepto de polimorfismo (Cap.2 secc. 2.3), ya que el método que se presenta pertenece a la interfaz de la clase `MiArbol`, mientras que el mensaje `do:` de la última línea del código se dirige a los objetos que sean nodos; recuérdese que la variable de instancia `raiz` contiene una instancia de la clase `Nodo`. Podría parecer que es recursión pero el llamado ligamiento tardío (late binding) provoca que sea invocado el método de la clase `Nodo`, pues el receptor es un "nodo" y no un "árbol".

El mensaje `etiquetas` se vale del mensaje anterior para recorrer todo el árbol y obtener en un arreglo todas las etiquetas del árbol en forma ordenada, este método ilustra la belleza y el poder de los bloques, ya que en forma compacta y eficiente se evalúan expresiones a medida que se recorre el árbol (ver secc. 4.1.2.3).

La variable temporal `cont` recibe el número total de nodos que existen en el árbol, esto es posible invocando el mensaje `do:` (secc. 4.1.2.3), tal número servirá para declarar el tamaño del arreglo que contendrá las etiquetas. El arreglo es declarado en la línea 2 y la variable se llama `arrAux`, el arreglo se irá formando a medida que se recorra el árbol a través de la invocación al mensaje `do:`, finalmente lo que se regresa como resultado es el arreglo de etiquetas.

```
(1)etiquetas

(2)  | cont cont2 arrAux tmp |

(3)   cont := 0.
(4)   cont2 := 0.
(5)   self do:[ cont := cont + 1 ].
(6)   arrAux := Array new:cont.
(7)   self do:[cont2 := cont2 + 1.
(8)            tmp := raiz et.
(9)            arrAux at:cont2 put:tmp ].
(10)  ^arrAux
```

Al comienzo de este capítulo se mencionó que se crearían árboles de diferentes clases: fechas, números y cadenas** (ver Fig. 4.1.2). El método `creaArbol` es el responsable de crear un árbol de cualquiera de estos tipos. Se declaran tres variables locales, una de ellas (`resp`) contendrá la respuesta del usuario 's' ó 'n'. Al usuario se le cuestiona si desea crear un árbol nuevo, si la respuesta es negativa se regresa el árbol con que inicia la aplicación (Fig. 4.1.10).

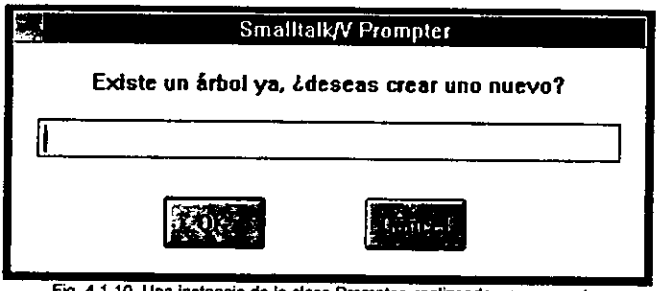


Fig. 4.1.10. Una instancia de la clase Prompter, realizando una pregunta.

Se menciona que ya existe un árbol , esto se debe a que se tiene una variable global llamada `Arbol`, la cual *persistirá una vez que ha sido declarada como global* (Cap. 2 secc. 2.1). Dicha variable contendrá una instancia de alguna de las subclases de la clase `MiArbol`. Por lo que en el *prompter*, en la parte que menciona la existencia de un árbol, se refiere es a la variable global `Arbol`.

Adelantándonos un poco observemos la ventana final de la aplicación (gráfica 4.1.11), obsérvese que existe un menú de barra con las opciones de "árboles" y "red". Accesando la opción de "árboles" se mustran las opciones disponibles, la primera de las cuales es "Crear un árbol".

El método `creaArbol` muestra el prompter de la figura 4.1.10 y si se contesta afirmativamente se invoca al método `crea`, en caso contrario se copia el contenido de la variable global `Arbol` y se regresa en uno u otro caso un árbol del tipo deseado, o el árbol que ya existía.

```
creaArbol

| resp aux1 aux2 |

resp := Prompter prompt:'Existe un árbol ya,¿deseas crear uno
nuevo?'
        default:''.
[resp = 's' or:[resp = 'n']]
```

** De igual forma se pueden crear árboles de otros objetos que puedan ser comparados.

```
  whileFalse:[resp := Prompter prompt:'Teclea "s" o "n"'
default:''].
resp = 's'
  ifTrue:[aux1 := MiArbol new: nil.
        aux2 := aux1 crea.
        Arbol := aux2 copy.
        ^Arbol]
ifFalse:[^Arbol]
```

Ahora se revisa el código del método `crea`, el cual es invocado, como se ha dicho, por `creaArbol`. Se declaran algunas variables locales, la variable `arbolAux` recibe un árbol vacío (línea 3). En la siguiente línea del código `unMenuArbol` recibe un "menú de árboles" esto es posible mediante la invocación del método `arbolMenu`, a continuación se capta la opción elegida por el usuario a través del método `popPup:`, la cual es asignada a la variable `op`. Una vez, ya con la elección del usuario debemos ejecutarla, si éste no abandonó el menú o escogió la opción "salir" (línea 7), entonces dependiendo del valor de la variable `op`, regresamos un determinado tipo de árbol (de fechas, números o cadenas). El árbol es vacío es este momento ya que lo inicializamos con `nil`.

```
(1)crea

(2)  | op aux unMenuArbol unArbol arbolAux |
(3)
(4)  arbolAux := MiArbol new:nil.
(5)  unMenuArbol := self arbolMenu.
(6)  op := unMenuArbol popUp.
(7)  (op isNil or:[op = #exit])
(8)   ifFalse:[(op = 2)
(9)              ifTrue:[^ArbolFecha new:nil]
(10)             ifFalse:[ (op = 3)
(11)                        ifTrue:[^ArbolNum new:nil)]]
(12)   ifTrue:[^nil]
```

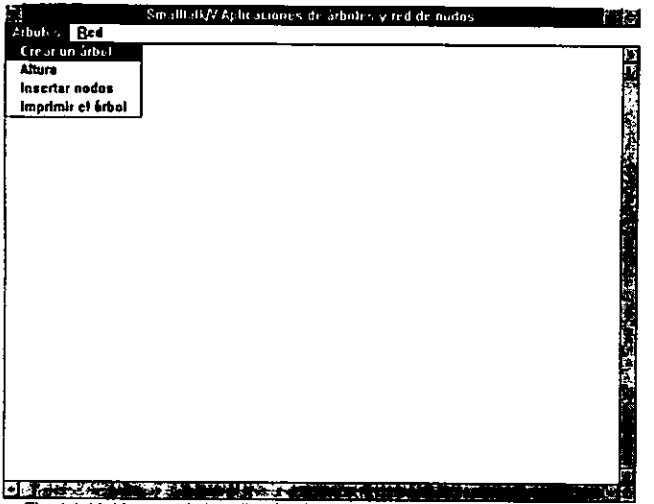


Fig. 4.1.11. Ventana de la aplicación final, mostrando las opciones disponibles.

El método `arbolMenu` presenta un menú , dicho menú tiene las opciones para crear un árbol de fechas o un árbol de números.

```
ArbolMenu

^Menu
 labels:'Salir\Arbol de fechas\Arbol de números\' withCrs
 selectors:#()
```

Con este método termina la descripción de la interfaz de la clase `MiArbol`, en la siguiente subsección se pasará a describir las subclases de `MiArbol`.

4.1.2 Implementación de las subclases de MiArbol

Las clases: `ArbolFecha`, `ArbolNum` y `ArbolCadena` son análogas en su interfaz, las tres responden al mensaje `ins:` el cual, como se ha dicho, funciona para insertar nodos al árbol de manera interactiva. Ninguna tiene métodos de clase, los heredan de la clase padre, así como las variables de instancia y las constantes (concepto de herencia, secc. 2.2.2)
La primera clase a describir es `ArbolCadena`, por ser la más sencilla, contiene solamente un método en su interfaz.

```
MiArbol subclass: #ArbolCadena
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

Obsérvese que no se declaran variables de instancia, variables de clase ni tampoco pool dictionaries, todas se heredan, (ver Fig. 4.1.2). La clase, al igual que las de `ArbolNum` y `ArbolFecha` no tiene métodos de clase.

4.1.2.1 Implementación de la clase ArbolCadena.

Para el método `ins:` se declaran tres variables locales: `resp`, `elem` y `otroArbol`; `resp` nos permitirá el control en el ciclo de la inserción (líneas 15 a 18), se pregunta si se desea continuar con la inserción, en caso afirmativo se invoca recursivamente el método (línea 18), teniendo como receptor al mismo árbol. El método recibe como parámetro `unNodo` (línea 1), dicho parámetro resulta ser una instancia de la clase `Nodoet`, tal nodo no trae consigo una "etiqueta", por lo que se le solicita al usuario que la proporcione (línea 4).

```
(1) ins:unNodo

(2)  | resp elem otroArbol |
(3)
(4)  elem := Prompter prompt:'¨Proporcione la cadena?' default:''.
(5)
(6)  self esVacio
(7)    ifTrue:[unNodo asignEt:elem.
(8)            raiz := unNodo.
(9)           ]
(10) ifFalse: [unNodo asignEt:elem.
(11)           otroArbol := ArbolCadena new:unNodo.
(12)           ((self daRaiz) et) = elem
(13)             ifFalse:[self insertar:otroArbol].
(14)          ].
(15) resp:= Prompter prompt:'¨Deseas continuar insertando (s/n)?'
            default:''.
(16) resp = 'n'
(17)  ifTrue:[^self]
(18) ifFalse:[self ins:unNodo].
```

Una vez proporcionada la "etiqueta", nos preguntamos si el árbol es vacío (línea 5), si esto es cierto la etiqueta se asigna a `unNodo` y tal nodo es "colocado" directamente en la raíz del árbol (líneas 7 y 8); en otro caso de igual forma se realiza el "pegado" de la etiqueta al nodo, se crea una instancia de la clase la cual contendrá en su raiz `unNodo` y dicha instancia es asignada a la variable `otroArbol` (línea 11). La línea 12 lleva a cabo una comparación entre la etiqueta pegada en el nodo del árbol a insertar y la etiqueta del nodo de la raíz del árbol sobre el que se realiza la inserción (self), si resultan ser diferentes se invoca el método `insertar:` , heredado de `MiArbol` (línea 13). Las clases `ArbolFecha` y `ArbolNum` incorporan al método `ins:` de igual forma: en su interfaz; pero con la diferencia con respecto a la clase `ArbolCadena` de que aquellas incorporan un método más. Enseguida se describe `ArbolFecha`.

4.1.2.2 Implementación de la clase ArbolFecha.

```
MiArbol subclass: #ArbolFecha
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

Se tienen dos métodos de instancia únicamente, `ins:` y `mes:`, el primero de los cuales pide interactivamente el día y el número del mes, el año se ha fijado al actual. Para el método `ins:` las variables `día` y `mesn` recibirán los números que corresponden al día y mes respectivamente de las fechas que se irán creando interactivamente, el año también podría ser proporcionado por el usuario. Se validan los números de mes y día (líneas 4 a 12). El número del mes proporcionado por el usuario y almacenado en la variable `mesn`, sirve de parámetro para el método `mes:`, este último regresa el nombre del mes correspondiente al número que recibe como parámetro (línea 13).

La clase `Date` tiene el método de clase `new:Day:month:year:`, el cual será utilizado para crear instancias del tipo `Date`, esto mediante el paso de los parámetros: `dia`, `mess` y `año`; la instancia creada se asignará a la varible local `fecha` (línea 14). Posteriormente se pregunta si el árbol es vacío (línea 16), si es el caso, entonces se crea un nodo etiquetado (la etiqueta será instancia de la clase `Date`), el cual será "trasplantado" en la variable de instancia `raiz` del árbol (línea 18); en otro caso y condicionado a que la etiqueta del nodo que se encuentra en la raíz del árbol no sea igual a la instancia de la clase `Date` que se creó (la variable `fecha`) se invocará al mensaje `insertar:` heredado de `MiArbol` para realizar la inserción del nodo etiquetado (líneas 20 a 24). El resto del algoritmo es idéntico al expuesto para el método `ins:` de la clase `arbolCadena`.

```
(1)ins:unNodo

(2) | resp elem dia mesn mess fecha otroNodo otroArbol arbolVacio
|
(3)
(4) dia := Prompter prompt:'¨Dame el d¡a?' default:''.
(5) dia := dia asInteger.
(6)[(dia isKindOf:Number) and:[dia >= 1 and:[dia <= 31]]]
(7) whileFalse:[dia := Prompter prompt:'Proporciona el tipo
correcto de
                          eiqueta.' default:''.].
(8)
(9) mesn := Prompter prompt:'¨Dame el mes?' default:''.
(10)mesn := mesn asInteger.
(11)[(mesn isKindOf:Number) and:[mesn >= 1 and:[mesn <= 12]]]
(12) whileFalse:[mesn := Prompter prompt:'Proporciona el tipo
                         correcto de etiqueta.' default:''.].
(13)mess := self mes:mesn.
(14)fecha := Date newDay:dia month:mess year:1996.
```

```
(15)
(16)self esVacio
(17) ifTrue:[unNodo asignEt:fecha.
(18)          raiz := unNodo.
(19)         ]
(20) ifFalse: [unNodo asignEt:fecha.
(21)           otroArbol := ArbolFecha new:unNodo.
(22)           ((self daRaiz) et) = fecha
(23)            ifFalse:[self insertar:otroArbol].
(24)          ].
(25)resp:= Prompter prompt:'¨Deseas continuar insertando (s/n)?'
          defaultExpression:''.
(26)resp = 'n'
(27) ifTrue:[^self]
(28)ifFalse:[self ins:unNodo].
```

Para el caso del método `mes:` se utiliza una cascada de mensajes, tal cascada se envía a la clase `Date` para poder regresar el nombre del mes al cual corresponde el número `m` pasado como parámetro.

```
mes:m

| mes |                                                          .
^(mes := Date initialize;nameOfMonth:m).
```

La estructura de interfaz que tiene `ArbolNum` es muy semejante a la de la clase `arbolFecha`, el método que incorpora aquella clase a diferencia de esta última es `valida:`, el cual verifica los datos proporcionados por el usuario (en este caso números). Con esto termina la descripción de objeto "árbol", ahora toca describir al objeto "nodo".

4.1.3 La jerarquía de la clase Nodo.

La identidad del objeto nodo quedará determinada por el contenido de sus nodos, izquierdo y derecho. Existe un tipo especial de nodo, tal nodo es aquél que tiene asociada una "etiqueta", con esto último se quiere decir que la etiqueta del nodo puede ser una instancia de alguna de las siguientes clases: `Number`, `Date` y `String`. Es posible extenderse hacia cualquier objeto que sea comparable en magnitud. Resulta que la clase que nos proporciona objetos con la característica de ser nodos y además ser "etiquetados" es la clase `Nodoet`, siendo esta última subclase de `Nodo` y por tanto heredera de las variables de instancia `nodoIzq` y `nodoDer`, ella misma incorpora la v.i `et`.

La clase `Nodo` es subclase directa de `Object` (véase gráfica 4.1.12), y a su vez tiene a la clase `Nodoet` como subclase, tiene dos variables de instancia : `nodoIzq` y `nodoDer`, no hace uso de constantes predefinidas ni de variables de clase. A continuación la definición de la clase

```
Object subclass: #Nodo
  instanceVariableNames:
    'nodoIzq nodoDer '
  classVariableNames: ''
  poolDictionaries: ''
```

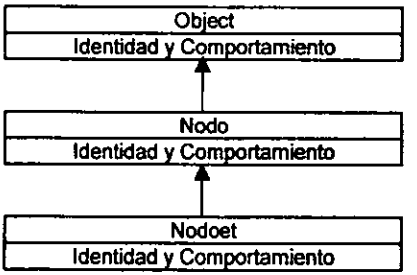


Fig. 4.1.12. Jerarquía de las clases Nodo y Nodoet.

4.1.3.1 Métodos de clase (constructores) de la clase Nodo.

El método `izq:der:` se vale del método de instancia `inIzq:inDer:` para inicializar las variables de instancia, se reciben dos parámetros:nodoI y nodoD, los cuales serán asignados a las variables de instancia nodoIzq y nodoDer respectivamente.

```
izq:nodoI der:nodoD

"Método que crea e inicializa una instancia de la clase Nodo"

^self new inIzq:nodoI inDer:nodoD
```

4.1.3.2 Métodos de instancia (interfaz) de la clase Nodo.

El método `inIzq:inDer:` mencionado anteriormente inicializa las variables de instancia.

```
inIzq: i inDer: d
" Este método es utilizado por el método de clase 'izq:
der:', para
inicializar las variables de instancia del nodo."

nodoIzq := i. nodoDer := d.
```

Los métodos `izquierdo` y `derecho` regresan los subárboles izquierdo y derecho, o lo que es lo mismo las variables de instancia `nodoIzq` y `nodoDer` respectivamente.

```
Derecho

"Regresa el subarbol derecho"

^nodoDer


izquierdo

"Regresa el subarbol izquierdo"

^nodoIzq
```

El método `esHoja` evalúa si un nodo es hoja, es decir, si tanto el subárbol derecho como el izquierdo son vacíos, el operador `&` resulta ser el "y" lógico en Smalltalk.

```
EsHoja

" Regresa cierto si el nodo es hoja e.d. si su subarbol
izquierdo y
derecho son ambos vacios"

^(self derecho esVacio & self izquierdo esVacio)
```

Para analizar el método `do:` (para nodos), necesitamos regresar a la sección 4.1.1.3, precisamente donde se describe el método `etiquetas`. Por tal motivo citamos el código del método `etiquetas`.

```
(1)etiquetas

(2) | cont cont2 arrAux tmp |

(3) cont := 0.
(4) cont2 := 0.
(5) self do:[ cont := cont + 1 ].
(6) arrAux := Array new:cont.
(7) self do:[cont2 := cont2 + 1.
(8)          tmp := raiz et.
(9)          arrAux at:cont2 put:tmp ].
(10)^arrAux
```

Supondremos que se tiene un árbol como el que se observa en la gráfica y que dicho árbol se encuentra contenido en la variable `arbolC` (sólo por asignarle un

nombre), y lo que se desea es colocar en un arreglo en forma ordenada, las etiquetas de dicho árbol; para tal propósito se hará uso del método `etiquetas`, el cual invoca al método `do:` de la clase `MiArbol` y este último a su vez al método `do:` de la clase `Nodo` (polimorfismo).

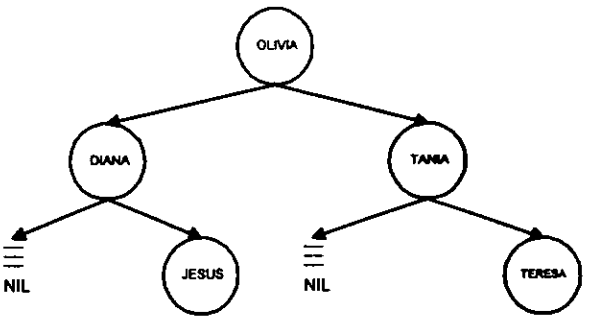


Fig. 4.1.13. Un árbol de cadenas.

En la implementación del método `etiquetas` se declaran cuatro variables locales (línea 2), de las cuales dos son incializadas con cero (líneas 3 y 4), enseguida se envía el mensaje `do:` a nuestro "árbol" llamado `arbolC` (línea 5), en nuestro caso el árbol no es vacío (ver `do:` de la interfaz de la clase MiArbol) y por tal motivo se invoca a `do:` y con receptor la variable de instancia `raiz` de `arbolC`. Recordemos que las variables de instancia de una instancia de la clase `Nodoet` son: `et`, `nodoIzq` y `nodoDer`, de donde `et` resulta contener la cadena "OLIVIA" y las otras dos contienen los subárboles izquierdo y derecho respectivamente que se observan en la gráfica.

La esencia del método `do:` (de la clase `MiArbol`) es evaluar el bloque de instrucciones que recibe como parámetro en cada nodo que se encuentra dentro de un árbol.

El método se invoca en dos ocasiones (líneas 5 y 7), en la primera, el bloque a evaluar únicamente incrementa en uno el contenido de una variable local, `cont` (línea 5), esto servirá para saber el número de nodos del árbol y así declarar el tamaño del arreglo `arrAux`, que contendrá las etiquetas (línea 6); en la segunda invocación (línea 7), se visita cada nodo y se rescata la etiqueta colocándola en el arreglo, para finalmente regresar como resultado dicho arreglo (línea 10).

Dentro de este método fijemos nuestra atención en el punto en el que invocamos por vez primera el método `do:` (línea 5), es análogo para la segunda invocación en la línea 7. El receptor del mensaje es un árbol no vacio y por lo tanto su `raiz` recibe el mensaje `do:`; la raíz contiene un objeto del tipo nodo; es en este momento en el cual empieza la ejecución
del mensaje `do:` de la clase `Nodo` (ver secc. 4.1.1.3.

El algoritmo funciona de la siguiente forma:

1. Si el nodo es hoja entonces evaluamos el bloque de instrucciones que se ha pasado como parámetro al método (línea 8)
2. De no ser hoja (línea 5), utilizando la recursión, se invoca nuevamente el método pero ahora sobre la raíz del subárbol izquierdo.
3. Una vez recorrido el subárbol izquierdo se evalúa el bloque (línea 6), podemos decir que cada vez que visitamos un nodo se evalúa el bloque.
4. Se repite el punto dos, pero para el subárbol derecho (línea 8).

```
(1)do:unBloque

(2)" Evalua un bloque en orden 'transversal' e.d. visita al nodo y a
     sus subarboles izquierdo y derecho y en cada nodo evalua unBloque."

(3) | aux |
(4) self esHoja
(5) ifFalse: [ (aux := self izquierdo) isNil ifFalse:[aux do: unBloque].
(6)            unBloque value:self.
(7)            (aux := self derecho) isNil ifFalse: [aux do: unBloque]]
(8) ifTrue: [ unBloque value:self]
```

Con este método finaliza la descripción de la interfaz de la clase Nodo.

4.1.3.3 Subclases de la clase Nodo

La única subclase de `Nodo` es `Nodoet`, a continuación la definición de esta última.

```
Nodo subclass: #Nodoet
  instanceVariableNames:
    'et '
  classVariableNames: ''
  poolDictionaries: ''
```

Se declara solamente una variable de instancia, `et`, no se hace referencia a variables de clase ni a constantes predefinidas (pool dictionaries).

4.1.3.4 Métodos de clase de Nodoet

Se tiene solamente al método `izq:der:etiq:`, el cual se encarga de crear y regresar una instancia inicializada, esto último vía el método de instancia `ii:dd:eett:` (izquierdo,derecho,etiqueta).

```
izq:i der:d etiq:e

^self new ii:i dd:d eett:e.
```

4.1.3.5 Métodos de instancia de Nodoet

El método `asignEt:` inicializa a la variable de instancia `et`.

```
asignEt: etiqueta

et := etiqueta
```

El método `ii:dd:eett:` es usado por el método de la clase para inicializar la instancia recien creada.

```
ii:i dd:d eett:e

nodoIzq := i.
nodoDer := d.
     et := e.
```

El método `et` regresa la variable de instancia `et`, la cual contiene la "etiqueta" del nodo.

```
et

^et
```

Los métodos `asignIzq` y `aisgnDer` se utilizan para asignar los valores que contendrán las variables de instancia `nodoIzq` y `nodoDer` respectivamente.

```
asignIzq:arbol

nodoIzq :=arbol


asignDer:arbol

nodoDer :=arbol
```

Hasta aquí la descripción de árboles y nodos, los objetos que conforman la primera de las dos aplicaciones, lo que resta es interactuar tales objetos; pero eso será parte del último capítulo; a continuación pasamos a describir los objetos que conforman otra de las aplicaciones, la de encotrar la ruta más corta en una red de nodos.

4.2 Implementación del problema de hallar el camino más corto en una red de nodos.

Planteamiento del problema

Se desea implementar el problema de hallar la ruta más corta en una red de nodos con peso en las aristas.

Abstracción del problema

Para este problema se requieren los objetos red y matriz; ya que el objeto que se utiliza es precisamente una red de nodos y para representarla requerimos de una matriz. Esta última representará la matriz de distancias o llamada también de adyacencias.

Clases disponibles en Smalltalk

Para la implementación del algoritmo se requirió la creación de la clase `Matriz`, que como tal no existe, pero que se puede deducir a partir de la clase `Array`, ya que para nosotros desde el punto de vista de la codificación una matriz puede verse como un arreglo. De la clase `Matriz` se desprende la clase `MatrizCuadrada` y de esta la clase Red (ver Fig. 4.2.2).

Clases creadas

Objeto Matriz
- Estado(identidad): renglones, columnas.
- Comportamiento(interfaz):
  - De clase: renglones:columnas:.
  - De instancia: columnas:, renglones:, columnas, renglones, loc:con:, en:en:pon:, imp:.

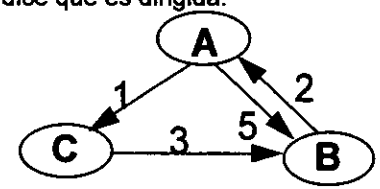
Objeto MatrizCuadrada
- Estado(identidad): (No tiene las hereda de Matriz).
- Comportamiento(interfaz):
  - De clase: new:.
  - De instancia: (todo su comportamiento lo hereda de Matriz)

Objeto Red
- Estado(identidad): d, g, e, numNodos, o, r, area.
- Comportamiento(interfaz):

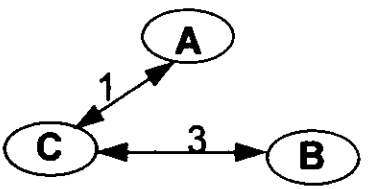
- De clase: new:.
- De instancia: red, inicializa, llena:, ruta, numeNodos, min:de:, matrizDist, area:.

Las clases que crearemos son Matriz, MatrizCuadrada y Red; las cuales crearán una matriz (de mxn), una matriz cuadrada y una red de nodos con pesos en las aristas.

Informalmente hablando podemos definir a una red como un conjunto de nodos que están relacionados por arcos. Partiendo de lo anterior un nodo es cualquier objeto y un arco es una conexión entre dos nodos, donde dicha conexión la podemos pensar como una línea (o flecha) que une dos nodos. Antes de hablar un poco más de los llamados arcos es necesario introducir un concepto relacionado con estos. Como se dijo, un arco entre dos nodos se puede pensar como un segmento de línea que une dichos nodos, ahora bien, dichos segmentos pueden ser o no "dirigidos". Denotaremos por AB, al segmento que va de A hacia B, así como por BA el segmento que va de B hacia A (ver fig 4.2.1 (A)), estos dos segmentos puden ser iguales en magnitud como lo son en la figura (B) de la gráfica, este último tipo de segmento es el llamado no dirigido (de aquí que el tipo de red B es no dirigida), no así los dos antes mencionados ya que puede darse también el caso de que existan los segmentos AB y BA pero no ser iguales (caso A) o simplemente existir en un sentido la conexión; por lo que la red del tipo A se dice que es dirigida.



(A) Ejemplo de una red dirigida



(B) Ejemplo de una red no dirigida

Figura 4.2.1

Los arcos representan magnitudes fijas y en problemas cotidianos se desea hallar la ruta máxima o mínima dentro de los caminos que generan los arcos. Las magnitudes de los arcos pueden representar tiempo, costos, distancias etc., para el caso que nos ocupa, se desea hallar la ruta más corta.

A partir de este momento se inicia la descripción de las diferentes clases (objetos) que están relacionadas con el objeto red, y la primera de ellas es la clase `Matriz`.

4.2.1 La jerarquía de la clase Matriz.

Obsérvese la gráfica 4.2.2, la cual muestra la jerarquía de las diferentes clases involucradas con la clase Red (herencia, Cap. 2 secc 2.3), se puede deducir que los objetos de la clase Red son ricos en cuanto a comportamiento heredado por parte de las clases de las cuales se deriva. El comportamiento del que más uso haremos proviene de las clases `Matriz` y `Array`, pero principalmente de la primera. A continuación el análisis de la clase `Matriz`.

```
Array variableSubclass: #Matriz
  instanceVariableNames:
    'renglones columnas '
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants '
```

La clase `Matriz` es hija directa de la clase `Array` (es muy importante recordar que todo el comportamiento (interfaz) es heredado por las clases hijas), se declaran dos variables de instancia: `renglones` y `columnas`, las cuales contendrán el número de renglones y columnas de las diferentes matrices que creemos; no se tienen variables de clase y se invoca a las constantes de tipo caracter en la parte de los `poolDictionaries`.

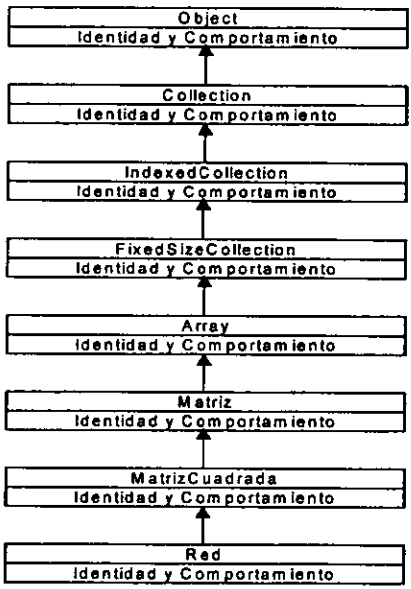


Fig. 4.2.2. La jerarquía de la clase Red.

4.2.1.1 Métodos de clase (constructores) de la clase Matriz.

Smalltalk no provee de los llamados arreglos bidimensionales (matrices) como se tiene en los lenguajes C ó Pascal, pero es posible simular tales arreglos. El método `renglones:columnas:` es el único constructor con que cuenta la clase `Matriz`, el cual regresa una "matriz" de tamaño r*c (ver código abajo), donde `r` y `c` son los parámetros que definen el número de renglones y columnas respectivamente.

```
(1) renglones:r columnas:c
(2)
(3) | resp temp |
(4) resp := super new:r*c.
(5) resp renglones:r;columnas:c.
(6) ^resp
```

Este método invoca a new: de su clase padre (`Array`), el concepto de herencia aparece muy claro aquí, ya que la clase `Matriz` está realizando una referencia hacia su clase padre a saber `Array` y lo realiza directamente a través del mensaje `super`. Lo que realmente se crea es un arreglo de tamaño r*c. Con la instancia ya creada se envían dos mensajes; `renglones:` y `columnas:`, los cuales son métodos de instancia que asignan los valores `r` y `c` a las variables de instancia `renglones` y `columnas` respectivamente. Por último el método regresa la instancia creada e inicializada (línea 6).

4.2.1.2 Métodos de instancia (interfaz) de la clase Matriz.

Ya se ha comentado el propósito de los métodos `renglones:` y `columnas:`, a continuación se expone la codificación de tales métodos.

```
columnas:unEnt

columnas := unEnt.

renglones:unEnt

renglones := unEnt
```

Dos métodos cuyo propósito es muy claro son `columas` y `renglones`, los cuales regresan el número de columnas y de renglones de una instancia; en otras palabras los valores de las variables de instancia `columas` y `renglones` respectivamente.

```
columnas

^columnas


renglones

^renglones
```

Para simular el acceso a las diferentes i,j ésimas posiciones de una matriz, se asocia a cada una de estas posiciones un número bien determinado dentro del arreglo, de la misma forma como se asigna también se debe accesar. La clave de esta simulación recae en el método `loc:con:`, el nombre proviene de "localizado con".

La manera de asociar las diferentes "columnas"** dentro del arreglo, es como sigue. El arreglo se dividirá en n partes, donde n es el número de columnas, de tal forma que los n primeros elementos conformarán el primer renglón, del (n + 1) ésimo elemento del arreglo al 2n serán los que formen el segundo renglón y así sucesivamente. A partir de pensar en esta división necesitamos una fórmula que reciba dos parámetros, que sean los números que nos coloquen dentro del arreglo en "la columna y renglón deseados". La respuesta se tiene en el método `loc:con:`, el cual se muestra a continuación.

```
loc:i con:j

^((i - 1) * (self columnas) + j)
```

Este método es muy importante, ya que a partir de él podremos accesar a las distintas posiciones dentro de la matriz.
Ahora se puede describir el resto de la interfaz a partir de este método y toca el turno al método `en:en:`, el cual regresa el i,j ésimo elemento dentro de la matriz.

```
(1)en:i en:j
(2)
(3) | valor |
(4)
(5) (i < 0 or:[i > self renglones]) ifTrue:[^nil].
(6) (j < 0 or:[i > self columnas]) ifTrue:[^nil].
(7) valor := self loc:i con:j.
(8) ^(self at:valor)
```

Las líneas 5 y 6 validan los parámetros, los cuales deben estar dentro de los valores permitidos por el número de columnas y renglones. Una vez validados los parámetros, se envía el mensaje `loc:con:`, el cual nos colocará en la posición

** Se enmarca con comillas debido a que en realidad no existen columnas, en Smalltalk sólamente se pueden crear arreglos lineales.

deseada dentro de la matriz y finalmente regresa el valor contenido en dicha posición. La fuerza de la herencia se nota una vez mas aquí, en todo momento estamos utilizando métodos que pertenecen a la clase padre (`Array`). El método `en:en:pon:` es similar al anterior, pero con la diferencia de que esta vez no regresa un valor sino que lo asignamos en la posición deseada dentro de la matriz.

```
(1)  en:i en:j pon:k

(2) | valor |
(3)
(4) valor := self loc:i con:j.
(5) self at:valor put:k.
(6) ^self.
```

El último de los métodos que conforman la interfaz de la clase Matriz es `imp:`, lo que realiza es la impresión de la matriz en la ventana de salida y el parámetro que recibe es la ventana donde ésta será impresa, una ventana semejante se utilizó en la aplicación de los árboles.

```
imp: unArea
   unArea cr.
  1 to:self renglones do:[ :i|
    1 to:self columnas do:[ :j|
                            unArea appendText:
                            (self en:i en:j) printString,''.]
    unArea cr ].
  unArea cr.
```

Realmente es pequeña la interfaz de la clase `Matriz`, pero lo es mucho más la de su subclase directa, se trata de la clase `MatrizCuadrada`. Esto último es precisamente una de las grandes ventajas de la POO, la reutilización de código.

4.2.2 Implementación de las subclases de la clase Matriz

La interfaz de la clase `MatrizCuadrada` que es descendiente directa de `Matriz` es nula ya que toda su interfaz la hereda de su superclase. La clase `Red` es descendiente directa de la primera, es más rica en su interfaz.

4.2.2.1 Implementación de la clase MatrizCuadrada

```
Matriz variableSubclass: #MatrizCuadrada
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
```

No se declaran más variables de ningún tipo, así como tampoco constantes predefinidas.

4.2.2.2 Métodos de clase (constructores) de la clase MatrizCuadrada.

```
new:tam

| matriz |

^(matriz := super renglones:tam columnas:tam)
```

Solamente se incorpora un método de este tipo y su función es regresar una "matriz cuadrada", para ello se invoca al método `renglones:columnas:` de su supercalse, de tal suerte que se crea un matriz cuadrada. El hecho de crear esta matriz, obedece a que a esta clase se le pueden incorporar más métodos los cuales son propios de las matrices cuadradas.

4.2.2.3 Implementación de la clase Red

Ahora comencemos la descripción de la clase que es responsable de crear los objetos del tipo red.

Se declaran siete variables de instancia, de las cuales `d` contendrá la llamada matriz de distancias de nuestra red, `numNodos` contendrá el número de nodos de la red y el resto serán arreglos, los cuales contendrán información referente al camino que estaremos hallando.

```
(1) MatrizCuadrada variableSubclass: #Red
(2)  instanceVariableNames:
(3)    'd g e numNodos o r area '
(4)  classVariableNames: ''
(5)  poolDictionaries:
(6)    'CharacterConstants '
```

4.2.2.4 Métodos de clase (constructores) de la clase Red.

A través del mensaje `new:`, se crea y regresa una instancia de la clase `Red`.

```
(1) new:unEnt
(2)
(3) | red |
(4)
(5) red := super new:unEnt.
(6) ^red.
```

4.2.2.5 Métodos de instancia (interfaz) de la clase Red.

El método responsable de desencadenar toda la aplicación se llama red. Con `red` se crea una cascada de mensajes para la instancia; el primer mensaje de dicha cascada es `inicializa`, posteriormente `llena:` y por último el mensaje `ruta`.

```
red

self inicializa; llena:(numNodos); ruta
```

El método `inicializa` se encarga de asignarles valores inciales a las diferentes variables de instancia del objeto. Lo primero que realiza es preguntar al usuario de cuántos nodos constará su red, el mensaje `numeNodos` es el responsable (línea 3). Dicho número será utilizado (línea 5) para crear una matriz cuadrada (variable de instancia `d`) de `numNodos * numNodos`, las siguientes cuatro líneas declaran a las variables de instancia: `g`, `o`, `e` y `r` como arreglos de dimensión `numNodos`. La matriz de distancias es inicializada con el número 10000, excepto en la diagonal principal, donde los valores son 0, esto se hace en las últimas 4 líneas del método.

```
(1)  inicializa

(2)  | resp aux |

(3)  self numeNodos.
(4)  aux := numNodos.
(5)  d := Red new:aux.
(6)  g := Array new:numNodos.
(7)  o := Array new:numNodos.
(8)  e:= Array new:numNodos.
(9)  r := Array new:numNodos.
(10) 1 to:numNodos do:[ :i|
(11)        1 to:numNodos do:[ :j| d en:i en:j pon:10000].
(12)        o at:i put:0.
(13)        d en:i en:i pon:0].
```

El método `llena:` es el que interactúa con el usuario para crear la red, se le pregunta al usuario si la red es dirigida, a continuación nos encontramos en un ciclo en el cual se pregunta por el origen, destino y distancia entre cada uno de los nodos. Para esto último se valida la entrada de tal forma que para calcular un camino se pregunta por el origen y el destino (aquí los nodos son números), de tal forma que los números estarán entre 1 y `numNodos`. Ahora bien, si resulta que la red es dirgida solamente se "tiende" el arco en una dirección, en otro caso en ambas direcciones, esto se hace en la matriz de distancias `d` con el mensaje `en:en:pon:`.

```
llena:unEnt

 | i j dist resp continua aux |

 continua := 's'.
 resp := 't'.
 i := -1.
 j := -1.
 [resp = 's' or:[resp = 'n']]
  whileFalse:[resp := Prompter
              prompt:' ¿La red es dirigida? ' default:''].

 [continua = 's']
  whileTrue:[ i := -1. j := -1.
             [i > unEnt or:[i < 0]]
      WhileTrue:[i := self valida:'Proporciona el origen del arco '].

     [j > unEnt or:[j < 0]]
     whileTrue:[j := self valida:'Proporciona el destino del arco '].

     dist := ''. aux := true.
     [dist isNumber]
      whileFalse:[dist := Prompter prompt:'Proporciona la distancia
                                          del arco ' default:0.
      1 to: dist size do:[ :index |(dist at: index) isDigit
                                  ifFalse:[aux := false:]].
      aux
       ifTrue:[dist := dist asInteger]
       ifFalse:[dist := '']].

     (i ~= 0)
     ifTrue:[d en:i en:j pon:dist.
             resp = 'n'
             ifTrue:[d en:j en:i pon:dist]].

     continua := Prompter
                 prompt:' ¿Deseas continuar (s/n)?' default:''.].
```

Ahora se ejemplifica como quedarían dos matrices, una que representa a una red no dirigida y la otra que es dirigida. Obsérvemos que la matriz b (no dirigida) es simétrica eso se debe a que por ejemplo las distancias del nodo 3 al 1 y viceversa existen y son iguales a 3. El hecho de colocar 10000 como entrada de la matriz denota que no existe conexión entre ambos nodos, en la teoría se dice que dicha longitud es infinita, pero en la práctica con un número "razonablemente" grande es suficiente; en el caso de la misma matriz b, los nodos 1 y 2 no están conectados, el resto sí. Para la matriz (a), que es dirigida, la distancia de 1 a 2 es 2 y la de 2 a 1 es 3; finalmente el hecho de tener ceros en la diagonal representa que la distancia de un nodo a él mismo es cero.

$$\begin{bmatrix} 0 & 2 & 1000 \\ 3 & 0 & 8 \\ 2 & 1000 & 0 \end{bmatrix}$$ (a)

$$\begin{bmatrix} 0 & 10000 & 3 \\ 10000 & 0 & 2 \\ 3 & 2 & 0 \end{bmatrix}$$ (b)

Ahora toca al turno al método `ruta`, el cual codifica el algoritmo, básicamente se divide en tres pasos. El primer requerimiento es segurarse de que existe una distancia osociada a cada par de nodos dentro de la red. El algoritmo visitará sistemática y gradualmente a cada nodo, asociándoles una etiqueta, esta etiqueta no es otra cosa que su distancia, es decir, el camino más corto del origen a dicho nodo. El algoritmo lo podemos exponer de la siguiente manera:

- PASO 0 .- Asignar etiquetas temporales e(i) = ∞ $\forall$ i $\neq$ s, asignar e(s**) = 0 y p = s.Hacer e(s) permanente (p es el último nodo en tener asignada una etiqueta permanente).

- PASO 1 .- Para cada nodo y con etiqueta temporal, redefinir e(i) como el número más chico entre e(i) y e(p) + d(p, i) (este último sumando es la distancia entre el último nodo etiquetado como permanente y el nodo i). Hallar entonces el nodo con la etiqueta temporal más chica; asignar a p ésta i, y entonces convertir en permanente la etiqueta e(p).

- PASO 2 .- Si el nodo t tiene etiqueta temporal entonces repetir el paso 1. En otro caso t ya tiene una etiqueta permanente, y dicha etiqueta corresponde a la longitud del camino más corto entre s y t a través de la red.

```
ruta

  | s t p t4 t8 t5 continua oli aux aa |

  t5 := numNodos.
  continua := 's'.

  [ continua = 's']
  whileTrue:[s := 0. t := 0.
 [[[[(s ~= t and:[s >= 1]) and:[t >= 1])
  and:[s <= numNodos]) and:[t <= numNodos])]
   whileFalse:[[s > numNodos or:[s < 1]]
               whileTrue:[s :=self valida:'Proporciona el origen de
                                          la ruta '].


     [t > numNodos or:[t < 1]]
      whileTrue:[t :=self valida:'Proporciona el destino de la
                                 ruta']].

" ***************PASO CERO ************************ "

 1 to:numNodos do:[ :i| e at:i put:10000.
                        g at:i put:0].

  g at:s put:1.
  e at:s put:0.
  p := s.
```

** Por s y t denotamos a los nodos inicial y final, o dicho de otro modo el nodo origen y el nodo destino a los cuales se les quiere encontrar el camino más corto entre ambos.

```
" ***************PASO UNO ************************ "

 [(g at:t) = 0]
 whileTrue:[1 to:numNodos do:[ :i|(g at:i) = 0
                                    ifTrue:[e at:i put:(self min:(e at:i)
                                                        de:((e   at:p)   +   (d   en:p
en:i))]]
                              ].
 p := 0.
 t4 := 10000.
 1 to:numNodos do:[ :i|(((g at:i) = 0) and:[(e at:i) <= t4])
                          ifTrue:[p := i. t4 := (e at:i)]
                   ].
 g at:p put:1].

" ***************PASO DOS ************************ "

 1 to:numNodos do:[ :j|(g at:j) = 1
       ifTrue:[1 to:numNodos do:[ :i|aux := (e at:i) + (d en:i en:j).
                                  (i ~= j and:[(e at:j) = aux])
                                  ifTrue:[r at:j put:i.
                                          (d en:i en:j) >=10000
                    ifTrue:[area appendText:'Se ha forzado una liga'
                                          ]
                                  ]]
                 ].
 t8 := t5.
 o at:t8 put:t.
 i := r at:t.
 [i ~= s]
 whileTrue:[t8 := t8 - 1.
                 o at:t8 put:i.
                 i := r at:i].
 area cr.
 area appendText:'El camino del nodo ', s printString, ' al nodo ', t
printString,
                 ' es ', s printString.
 t8 to:t5 do:[ :i|area appendText:'-',(o at:i) printString].
 area cr.
 area appendText:'Su longitud es ', (e at:t) printString.
 area cr.

 continua := Prompter
             prompt:'¿Otro camino?' default:'']
```

El método mumeNodos pregunta interactivamente por el número de nodos que conformarán la red, dicho valor es asignado a la variable de instancia numNodos.

```
numeNodos

 numNodos := 500.
 [numNodos > 50]
 whileTrue:[numNodos := self valida:'¿Cuantos nodos?']
```

El método min:de:, regresa el mínimo de dos números, este es invocado dentro del método ruta.

```
min:i de:j

  i < j
   ifTrue:[^i]
  ifFalse:[^j]
```

El método `matrizDist` regresa la variable de instancia `d`, en la cual se tiene la matriz de distancias asociada a la red.

```
matrizDist

^d
```

El mensaje `area:` recibe como parámetro una ventana en la cual serán desplegados los resultados del algoritmo.

```
area: unArea

area := unArea.
```

Finalmente, ha llegado el momento de conjuntar todo lo expuesto en este capítulo, lo cual se realizará en el siguiente capítulo.

Capítulo 5
Creando el Objeto "Aplicación"

Introducción

En este capítulo se expone la manera de crear una aplicación independiente (standalone application), o dicho de otra manera, la versión ejecutable de una aplicación. Lo anterior con el propósito de presentar una visión de lo que es una aplicación "pura" orientada a objetos. Asimismo, se seguirá ejemplificando la puesta en práctica de los conceptos del paradigma OO.

Debido a la consistencia de considerar como un objeto a todo lo que habita dentro de Smalltalk, no debe extrañar que la misma aplicación sea un objeto; por lo cual deberá poseer una identidad y comportamiento. Este último atributo le permitirá interactuar con el medio ambiente de Windows.

La siguiente pregunta plantea el problema que se desarrollará en el presente capítulo, ¿cómo crear una aplicación ejecutable que enmarque los problemas de implementar el tipo de dato abstracto árbol y el de hallar el camino más corto dentro de una red de nodos?

El problema será abordado con la misma secuencia de pasos que se utilizó en el capítulo anterior: planteamiento del problema, abstracción de los objetos que lo conforman, determinación de clases existentes, creación de clases a partir de las existentes y descripción de los objetos que serán instancias de las clases creadas.

5.1 Creación de una aplicación *pura* OO

Planteamiento del problema

Se desea crear una aplicación independiente de Smalltalk.

Abstracción del problema

La aplicación consiste de una ventana, en la cual por medio de un menú se podrá interactuar con los objetos del tipo árbol y red.

Clases disponibles en Smalltalk

La clase que proveerá los mecanismos para establecer una comunicación transparente con Windows se llama `ViewManager`. Esta clase contiene el soporte necesario para el manejo de eventos, que es la manera como Windows responde.

`ViewManager` es puente entre nuestra aplicación, y la unidad de interfaz gráfica (GUI) de Microsoft Windows. Un propósito importante aunque secundario de esta clase es la posibilidad del manejo de múltiples vistas o múltiples ventanas. Si se llegase a tener necesidad de crear una aplicación con manejo múltiple de ventanas, es agradable saber que `ViewManager` define todo el protocolo que sea necesario para manipular las interacciones entre esas ventanas. Por otra parte, la gran mayoría de las ventanas con las que trabaja Smalltalk, son instancias de `ViewManager` o de alguna de sus subclases. Por lo dicho, esta clase resulta la ideal para ser base de la aplicación.

Clases creadas

Se ha dicho que la aplicación es un objeto, la clase que resulta ser la responsable de crear objetos "aplicación" será `AplicaApp`.

Identidad y comportamiento de los objetos que conforman el problema

Se describe al objeto aplicación definiendo identidad y comportamiento.

Objeto aplicación
- Estado(identidad): arbolArea, arbol, laRed, alertaDatos.
- Comportamiento(interfaz):
- De clase:(no cuenta con métodos propios)
- De instancia: inicializa:, área:, daRaiz, esVacio, max1:max2:, altura:, alto:, ins:, insertar:, Impdesp:, imprime:, do:, creaArbol, crea, arbolMenu, ManejadorMensajes:mensage:.

5.1.1 La jerarquía de la clase AplicaApp

La clase `AplicaApp` es subclase de `ViewManager`, esta última es una de las clases más importantes de Smalltalk, ya que todas las aplicaciones independientes que se creen tendrán que ver con esta clase o con alguna de sus subclases.

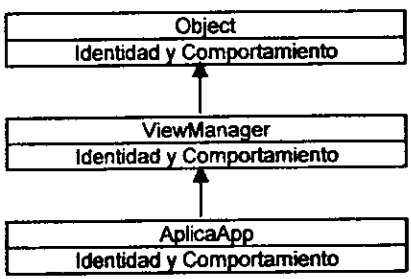


Fig. 5.1. Jerarquía de la clase AplicaApp

AplicaApp no declara variables de clase pero se declaran cuatro variables de instancia: `arbolArea`, `arbol`, `laRed` y `alertaDatos` y se hace referencia a las constantes de caracter.

```
ViewManager subclass: #AplicaApp
  instanceVariableNames:
    'arbolArea arbol laRed '
  classVariableNames: ''
  poolDictionaries:
    'CharacterConstants '
```

Los objetos "aplicación" serán un tipo especial de ventanas. La gran relevancia que tienen las ventanas para Smalltalk se ve opacada por el hecho de que la palabra Windows está bastante cargada de conceptos en este ámbito; sin embargo, desde sus inicios Smalltalk ya incorporaba y manejaba los objetos ventana, mucho tiempo antes de que existiera el concepto Windows; de hecho, Smalltalk es pionero en gran medida del manejo de ventanas tal y como lo conocemos hoy en día.

Es natural que las instancias de la aplicación resulten ser ventanas, pues por medio de estas es como nos comunicamos con Smalltalk. Utilizamos una ventana para: crear una clase, ejecutar código, inspeccionar alguna clase, utilizar el rastreador para examinar código, etcétera.

5.1.1.1 Métodos de clase (constructores) de la clase AplicaApp.

La clase como tal no incorpora sus propios método de clase, los heredará de sus clases padre, `ViewManager` u `Object`.

5.1.1.2 Métodos de instancia (interfaz) de la clase AplicaApp.

Iniciaremos con un método que tiene gran relevancia para la clase, pues por medio de este es que se inicia la aplicación, a su vez este desencadena una serie de sucesos o eventos, los cuales permitirán que finalmente se tenga un objeto aplicación y se pueda interactuar con él, este método se llama `open`. El nombre obedece a la convención de llamarlo así en todas las aplicaciones que se desarrollan en Smalltalk, aunque bien pudo llamársele `inicia` ó `abrir`.

Para construir una aplicación ejecutable en Smalltalk se deben tomar en cuenta algunas indicaciones, las cuales se encuentran contenidas en el archivo readme.txt, que viene con el conjunto de archivos de Smalltalk.
Ahora se explicarán todas y cada una de las líneas de código.

```
(1)open

(2)  self
(3)  label: 'Aplicaciones de árboles y red de nodos.';
(4)  owner: self;
(5)  when: #close perform: #close:;
(6)  when: #menuBuilt perform: #cambiaMenuBar:;
(7)  addSubpane:
(8) (arbolArea := TextPane new
(9)               owner: self;
(10)              when: #getContents perform: #muestra:;
(11)              when: #getMenu perform: #arbolMenu:);
(12) openWindow.
(13) self menuWindow addMenu: self redMenu.
(14) self asigna.
```

Es importante tener presente que AplicaApp es una ventana, por tal motivo los mensajes que se le envíen serán los relacionados con objetos de este tipo, a partir de este momento cuando se hable de ventana debe entenderse que se está hablando del objeto aplicación.
La primera línea contiene el nombre del método, a continuación a partir de las líneas 3 a 11 la ventana (`self`) recibe una cascada de mensajes a saber:

- La etiqueta de la ventana tendrá la cadena "Aplicaciones de árboles y red de nodos" (línea 3).

- Se indica que quien posee tales atributos es la ventana misma (línea 4).

Las líneas 5 y 6 le indican a la ventana que cuando suceda el evento "cierra ventana" (esta ocurre cuando se da por terminada la ejecución de la aplicación), se debe ejecutar el método `close:`. Es importante mencionar que una de las indicaciones para la creación de una versión ejecutable de una aplicación en Smalltalk, es agregar esta línea de código al método `open`. La siguiente línea de código indica que cuando se construya el menú correspondiente a la ventana, se deberá ejecutar el método `cambiaMenuBar:`.
Por medio del mensaje `addSubpane` (línea 7) se le agrega una subárea (subpane) a la ventana, por ejemplo recuérdese que la ventana `Class Hierarchy Browser` tiene tres subáreas.

- La subárea será una instancia de la clase `TextPane` (línea 8). Esta área será la que se utilizará para desplegar las salidas de la aplicación, y es una ventana de texto debido a que lo que se escribe es precisamente texto.

- Al igual que a la ventana anterior a esta se le envía el mensaje `owner` (línea 9). Ahora bien, cuando suceda el evento `getContents`, que pasará cuando la subárea desea actualizar su contenido se invocará al método `muestra:` (línea 10).

- En la línea 11 se crea el menú para esta ventana. En este caso se incorpora un menú por medio de la ejecución del método `arbolMenu:`, el cual contendrá las opciones necesarias para comunicarnos con los objetos del tipo árbol.

- En la línea 12 se le envía al objeto aplicación el mensaje de que se abra la ventana.

- El mensaje `addMenu` incorporará el menú para comunicarnos con los objetos que serán instancias de la clase `Red`.
- En la última línea se invoca al método `asigna`, el cual describiremos en su momento.

Para finalizar con la explicación de este método, se comentará un poco más acerca de cómo es que Smalltalk manipula los eventos que Windows le transfiere. Si un evento es tal que una subárea necesita responder a él, se deben tomar los siguientes pasos (ver fig. 5.2):

1. Identificar a la aplicación dueña de la ventana con el mensaje `owner:`
2. Incluir un método `when: perform:` en el método `open` para decirle a Smalltalk cual método deberá ejecutar cuando la subárea recibe el mensaje.
3. Escribir el método que será ejecutado en respuesta al mensaje.

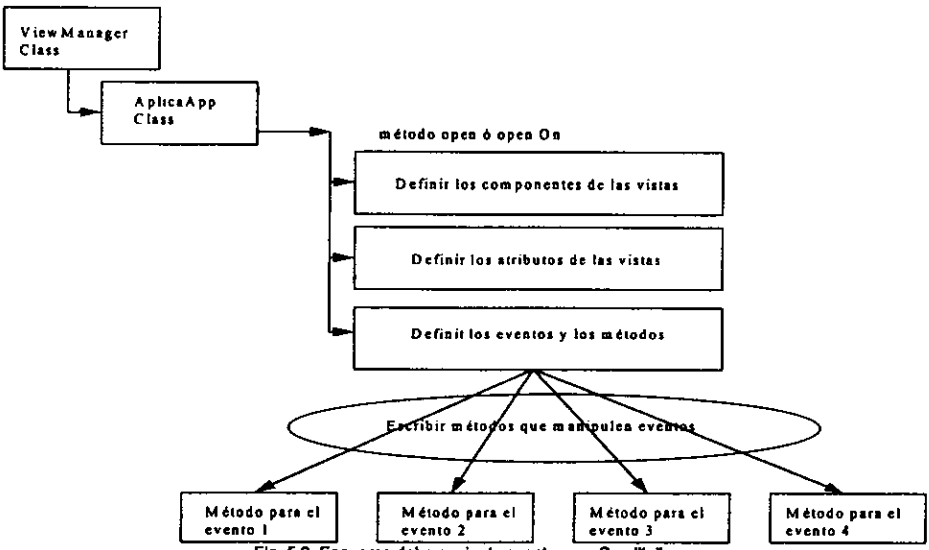


Fig. 5.2. Esquema del manejo de eventos por Smalltalk.

Ahora se explicarán los métodos que aparecen en el método `open`. Siguiendo un orden, tocaría el turno al método `close:`, pero debido a que este se relaciona al fin de la aplicación, se dejará para el final.

El método cambiaMenuBar: remueva los menúes que de entrada nos proporciona Smalltalk para una ventana.

```
cambiaMenuBar: unArea

"Quita los menúes File, Edit y Smalltalk."

unArea menuWindow
removeMenu: (unArea menuWindow menuTitled: '~File');
removeMenu: (unArea menuWindow menuTitled: '~Edit');
removeMenu: (unArea menuWindow menuTitled: '~Smalltalk');
removeMenu: (unArea menuWindow menuTitled: '~Tools').
```

A continuación se presentan los métodos asociados con la incorporación de los menúes para los objetos árbol y red.

En la segunda línea aparece el método de clase labels:lines:selectors:title de la clase Menu. El primer parámetro para este método se refiere a una cadena la cual contiene las descripciones de las opciones que aparecerán en el menú, dichas opciones están separadas por el caracter "\". El segundo de los parámetros (para la parte de labels:) se refiere a trazar una línea de división dentro de las opciones del menú cosa que en nuestro caso resulta irrelevante y por tanto ese parámetro no se pasa. Asociada a cada opción declarada en la cadena que recibe como parámetro selectors: se tiene un método de instancia. Por ejemplo, "Crear un árbol" se asocia con el método regArbol, "Altura" se asocia al método profund y de igual forma "Insertar nodos" con agrega e "Imprimir el árbol" con despliega. Se declara que el poseedor de tal menú es el receptor del mensaje (línea 8) y la asignación de tal menú es para el objeto unArea (última línea), que será la subárea incorporada.

```
(1) arbolMenu: unArea
(2) | unMenu |
(3) unMenu  := (Menu
(4) labels:'Crear un árbol\Altura\Insertar nodos\Imprimir el árbol' (5)
    withCrs
(5) lines: #()
(6) selectors:#(regArbol profund agrega despliega))
(7) title: '&Arboles';
(8) owner: self;
(9) yourself.
(10)unArea setMenu: unMenu.
```

El método redMenu regresa como valor un menú y este es agregado al menú de árboles que ya se tenía.

```
RedMenu

| unMenu |

^(Menu
 labels:'Crear el objeto red\Inicializar la red y hallar las
        rutas\Imprimir la matriz de distancias'
 withCrs
 lines: #()
 selectors:#(regRed redAp impMat))
 title: '&Red';
 owner: self;
 yourself.
```

Otro de los métodos que aparecen en `open` es `muestra:`, que como se dijo, se ejecuta como respuesta al evento `getContents`. El parámetro que recibe es la subárea (TextPane) donde se desplegarán los resultados del cómputo, es por ello que a dicha ventana se le envía el mensaje `display`, que provoca que la ventana refresque su contenido.

```
muestra:unArea

arbolArea display
```

El método `asigna` crea, invoca e inicializa los objetos árbol y red, que habitarán la aplicación. En la línea 8 se carga un objeto tipo árbol, y se le asigna a la variable de instancia `arbol`. Este fragmento ejemplifica el concepto de la persistencia[†], el cual se logra por medio del método `loadFrom:`, que pertenece a la clase `ObjectFiler` , éste proporciona la facultad de recuperar objetos dentro de un archivo físico en alguna ruta que se haya especificado.

El hecho de cargar este objeto provoca que se desencadenen una serie de mensajes, los cuales pudieran ser de error, en caso de que los hubiera, o simplemente mensajes relativos al objeto que se esté cargando. Estos aparecerán en una ventana, pero si se desea tener control sobre ellos, se utiliza el método `manejadorMensajes:mensage:`, el cual pertenece a la clase `ObjectFiler`.

Se crea un objeto del tipo `File` que contendrá la dirección donde se ubica el objeto (línea 6). Un objeto del tipo `ObjectFiler` se encargará del manejo de los mensajes provocados al cargar el objeto (línea 7), la variable de instancia. `arbol` (línea 8) contendrá al objeto Arb2 y el parámetro `árbolArea` es la subárea de la ventana (ver método `open`).

---

† Al declarar una variable global en Smalltalk, automáticamente es persistente (siempre y cuando se salve la imagen al terminar la sesión), es decir, sobrevivirá al tiempo y la ejecución.

```
(1)asigna

(2)| miManejador objetoDatos unObjFiler |

(3)  miManejador:= Message new
(4)  receiver: self;
(5)  selector: #manejadorMensajes:mensage:.
(6)  objetoDatos:= File pathName: 'c:\lengua\vWin\Arb2'.
(7)  (unObjFiler:=ObjectFiler new) clientMessageHandler:miManejador.
(8)  arbol := unObjFiler loadFrom: objetoDatos.
(9)  objetoDatos close.
(10) arbol  area: arbolArea.
(11) laRed := Red new:1.
(12) laRed area: arbolArea.
```

Las dos últimas líneas se refieren a la asignación de una instancia de la clase Red, la cual se asignará a la variable de instancia `laRed` (líneas 10 y 11).

Dentro del método asigna existen invocaciones a métodos de instancia propios de `AplicaApp`. El primer método se refiere al manejo de los mensajes provocados al cargar el objeto `Arb2`.

```
(1) manejadorMensajes: anObjectFiler mensage: assoc

(2) | msg |

(3) (assoc key = 'summary')
(4) ifFalse: [  " info or warning message "
(5)           alertaDatos isNil
(6)           ifTrue: [alertaDatos :=
(7)           File newFile: 'Foo.log'.]
(8)           alertaDatos nextPutAll: assoc value; cr]
(9) ifTrue: [  " summary count of messages "
(10)           msg := assoc value printString,
(11)           messages reported while storing ',
(12)           self printString.
(13)           alertaDatos nextPutAll: msg ; cr.
(14)           alertaDatos close].
```

El primer parámetro se refiere propiamente al manejador, mientras que el segundo es una instancia de la clase `Message`, este último se utiliza para determinar cual es el tipo de los mensajes. Si los mensajes son de advertencia o información, se ejecutarán las líneas 5 a 8, en otro caso serán las líneas 10 a 14. Los mensajes serán enviados al archivo llamado `Foo.log`, el cual puede ser guardado en disco o simplemente mantenerse en memoria hasta que sea cerrado.

El método `area:`, que aparece en `asigna` es método de instancia de las clases `MiArbol` y `Red`, los cuales se describieron en el capítulo anterior.

En este punto ya se ha creado el objeto aplicación, es decir, una ventana a la cual se le han incorporado ciertos menúes y la inicialización de sus variables de instancia, entonces lo que resta es describir los métodos asociados a las opciones de dichos menúes.
Se empieza por describir los métodos asociados con el menú de árboles. Eligiendo la opción "Crear un árbol", invocamos al método `regArbol`, el cual a su vez invoca a `creaArbol`, ambos pertenecen a la clase `MiArbol`. Al utilizar dicha opción, se preguntará si se desea reemplazar el árbol ya existente, este último es aquel que se recupera cuando se inicia la aplicación vía el método `asigna`. Si se contesta afirmativamente a la pregunta, entonces se crea un árbol vacío del tipo deseado (cadenas, fechas, números) y sus subárboles inicializados con `nil`.

```
(1)regArbol
(2)"regresa un árbol"

(3) arbol := arbol creaArbol.
(4) UnNodo:=UnNodo asignIzq: ((Arbol class) new:nil);
(5)                asignDer: ((Arbol class) new:nil); asignEt:nil.
(6) arbol  area: arbolArea.
```

El método `profund` invoca al método `altura`, este último tiene como receptor a la variable de instancia `arbol`.

```
Profund

arbol  altura.
```

El siguiente fragmento de código se refiere al proceso de inserción en un árbol, se está en un ciclo hasta que el usuario ya no desee insertar nodos. Es importante señalar que dependiendo del tipo de árbol, será el mensaje `ins:` que se invoque. Cada tipo de árbol implementa o reconoce dicho mensaje, esto es lo que se conoce como polimorfismo.

```
(1)agrega

(2) | unNodo resp |

(3) resp := 's'.
(4) [resp = 's']
(5) whileTrue:[unNodo := Nodoet izq: ((arbol class) new:nil)
(6)                          der: ((arbol class) new:nil)
(7)                          etiq:nil.
(8)            arbol ins:unNodo.
(9)            resp := nil.
(10)           [resp = 'n' or:[resp = 's']]
(11)           whileFalse:[resp := Prompter prompt:'¿Deseas
                        continuar insertando?' default:'']]
```

El método despliega muestra la estructura del árbol en la ventana valiéndose del método imprime: de la clase MiArbol.

```
despliega

arbol imprime:arbolArea.
```

Con este método se terminan los métodos asociados al menú de árboles, a continuación se describen los métodos asociados con el menú de red.
El método regRed análogamente a regArbol, regresa una instancia de la clase Red y se la signa a nuestra variable de instancia laRed.

```
regRed

"Regresa una red"

laRed := Red new:1.
laRed area: arbolArea.
```

Una vez ya creado el objeto de tipo Red, el método que inicia la interacción con él es redAp, que a su vez invoca al método red.

```
redAp

laRed red
```

El método impMat solamente imprime en la ventana la matriz de distancias asociada a la red de nodos.

```
impMat

"Imprime la matriz de distancias de la red"

((laRed matrizDist) imp: arbolArea).
```

El último de los métodos que conforman la interfaz de la clase AplicaApp es close:, el cual se ejecutará cuando se suceda el evento close, es decir, al cerrar la ventana, o dicho de otro modo, al término de la aplicación, esto es la contraparte de open.

Se utiliza al mensaje dump para escribir en disco un objeto, en este caso se trata del "árbol" con el que se trabajó durante la sesión que se está por terminar. En la última línea, explícitamente se invoca al método close de la clase padre (viewManager) para cerrar la ventana. Al igual que con open, se tiene un manejador para el control de los mensajes que resulten de la invocación de dump.

```
(1)close: unArea

(2) | miManejador objetoDatos  unObjFiler |
(3)
(4) miManejador:= Message new
(5) receiver: self;
(6) selector: #manejadorMensajes:mensage:.
(7) objetoDatos:=File newFile: 'c:\lengua\vWin\Arb2'.
(8)(unObjFiler:=ObjectFiler new)clientMessageHandler: miManejador;
(9) dump: arbol on: objetoDatos.
(10)objetoDatos close.
(11)super close.
```

Dentro de las clases que provee Smalltalk está `NotoficationManager`, la cual contiene el método de instancia `StartUpApplication:`, este tiene como tarea desatar el inicio de la aplicación. A efecto de que esta se inicie, deberá agregarse la siguiente línea de código.

```
AplicaApp new open.
```

La línea de código anterior creará una instancia de AplicaApp y posteriormente le envía el mensaje open. Esta adecuación se indica en al archivo readme.txt, dentro de la sección llamada "Starting up a standalone application".

Finalmente, al crearse una aplicación independiente del sistema de Smalltalk, se crea el archivo v.exe, el cual puede ser renombrado a la conveniencia del usuario. Cuando se está en una sesión con el sistema de Smalltalk propiamente dicho, se ejecuta el archivo vw.exe, el cual nos permite accesar el código de cualquier método en alguna clase. Esto último es de mucha importancia cuando se está tratando de crear la versión ejecutable de una aplicación, ya que en tiempo de desarrollo tenemos a nuestra disposición el código fuente de todas las clases, cosa que no es posible cuando nuestro programa ejecutable entra en acción debido a que no tenemos los métodos asociados con el compilador. Por lo anterior debemos de procurar escribir métodos que no estén invocando a su vez métodos que solamente estén disponibles en tiempo de desarrollo.

Conclusiones

El paradigma Orientado a Objetos está siendo determinante en muchas áreas, su influencia se ha dejado sentir prácticamente en todos los paradigmas de programación. Por ejemplo, la adaptación de algunos lenguajes de programación para que soporten la programación OO, ha dado origen a la aparición de C++, Object Pascal, CLOS, etcétera. Pareciera que ningún lenguaje deseara quedarse atrás y todos quisieran ser OO. Existen muchos productos y lenguajes que se presentan como OO, pero muchos de ellos solo simulan o proveen mecanismos restringidos de los conceptos del paradigma OO.

Smalltalk mapea de manera transparente los conceptos OO. Cuando se implementa en él un problema, se piensa en objetos, mensajes y la intercomunicación entre objetos.

El lenguaje posee una belleza particular: al trabajar en una sesión, todo el código, variables globales, modificaciones a métodos existentes, etc., que se realicen pasarán a formar parte del sistema.

Una característica relevante y muy cómoda para cualquier programador es el hecho de que se pueden ejecutar fragmentos de código sin necesidad de toda una estructura completa y predefinida. Esta característica es una propiedad que se presenta de manera natural en Smalltalk, debido a que con sólo crear un objeto es posible establecer desde ese momento una comunicación con él. Por esta característica a Smalltalk se le puede considerar un lenguaje interpretado, pero a la vez es compilado pues al momento de que se crean los métodos ya sean de clase o instancia, el lenguaje revisa sintaxis antes de aceptar e incorporar al método como válido.

Smalltalk ha probado ser bueno en varias áreas, algunas aplicaciones específicas son las siguientes:

- Controla y maneja redes telefónicas
- Ejecuta circuitos conmutados en tiempo real
- Ejecuta programas en lotes (batch) en mainframes
- Genera patrones de prueba para equipos semiconductores
- Genera el costo exacto de fabricación de un carro
- Sistemas de información de negocios

Actualmente circulan gratuitamente dos dialectos de Smalltalk que son: GNU Smalltalk de E. S. Byrne y Little Smalltalk de T. Budd, pero están muy limitados en lo que se refiere a la interfaz gráfica del usuario, que es done precisamente brillan las versiones comerciales. Existe una versión barata para UNIX con X-Windows, conocida como Smalltalk/X de Tomcat.

Existe una gran cantidad de productos comerciales los cuales varían de acuerdo al tipo de aplicación que se desee realizar, por ejemplo, HP ofrece HP DST, producto que permite crear aplicaciones distribuidas en un medio ambiente UNIX. Por su parte IBM ofrece ENVY/400, el cual permite desarrollos con el esquema cliente/servidor basados en Smalltalk. Es importante mencionar que tanto HP como IBM han decidido junto con sus clientes corporativos migrar todas las aplicaciones codificadas en COBOL y 4GL hacia Smalltalk. En Europa la empresa Siemens ha lanzado FINIS, que resulta ser una plataforma para una gran cantidad de sistemas bancarios. Hacia finales de 1994 se creó por parte de las compañías distribuidoras más grandes de Smalltalk, la organización STIC (The Samltalk Industry Council), la cual tiene como propósito principal promover en el mercado de software de desarrollo las ventajas de usar este lenguaje de programación.

Cabe destacar que así como se ha impuesto el paradigma OO, ha hecho su aparición JAVA, el cual promete convertirse en el nuevo estándar de lenguajes de programación. Este lenguaje desde su concepción es OO, su código es portable y lo que lo ha impulsado enormemente es que es posible escribir aplicaciones para Internet. Debido a los fundamentos OO de JAVA, se torna indispensable tanto para programadores como diseñadores, el tener conocimiento y dominio de los conceptos del paradigma OO.

Para finalizar, es importante mencionar que aunque el paradigma OO resulta ser apropiado para abordar una gran cantidad de problemas, no implica que sea la solución óptima a todos. La elección del paradigma, al igual que la del lenguaje de programación con el que se debe abordar una situación específica, dependerá en gran medida de las características mismas del problema.

Bibliografía

Aho, Hofcroft, Ullman (1975). *The Design and Analysis of Computer Algorithms.* Addison Wesley.

Budd, Timothy (1987). *A Little Smalltalk.* Addison Wesley.

Budd, Timothy (1991). *An Introduction to Object - Oriented Programming.* Addison Wesley.

Cormen, T., Leiserson, C. and Rivest, R. (1990). *Introduction to Algorithms.* The MIT Press.

Ghezzi C., Jazayeri M. (1998). *Programming Language Concepts.* 3th ed. John Wiley & Sons.

Goldberg, Adele (1984). *Smalltalk-80 The Interactive Programming Language.* Addison Wesley.

Goldberg, Adele and Robson, David (1989). *Smalltalk-80 The Language.* Addison Wesley.

Grady, Booch (1991). *Object Oriented Design with aplications.* The Benjamin/Cummings Publishing Company, Inc.

Kaplan, S., and Johnson, R. (July 21, 1986). *Designing and Implementing for Reuse.* Urbana. IL, University of Illinois, Department of Computer Science. P-8.

Kuhn, Thomas S. (1970). *The Structure of Scientific Revolutions.* 2nd ed. University of Chicago Press, Chicago, IL.

OOPSLA / ECOOP 90. *Advanced Program Overview (Oct. 21-25 1990 Ottawa, Ontario, Canada).*

Parnas, David L. (1972). *On the Criteria to be Used in Descomposing System into Modules.* Communications of the ACM, 15(12).

Robson, D. August (1981). *Object Oriented Software Systems.* Byte, Vol. 6(8).

Savic, Dusko (1990). *Object Oriented Programming With Smalltalk.* E. Horwood.

Sethi, Ravi (1990). *Programming Languages Concepts and Designs.* Addison Wesley.

Wegner, P (1976). *Programming Languages – the first 25 years*. IEEE Trans. Computers C-25:12

Wilf, R. Lalonde and Pugh, John R (1991). *Inside Smalltalk Vol. I*. Prentice Hall.

Wilf, R. Lalonde and Pugh, John R. (1991). *Inside Smalltalk Vol. II*. Prentice Hall.

Wilf, R. Lalonde (1994). *Discovering Smalltalk*. The Benjamin/Cummings Publishing Company, Inc.