





0111877
1011
MFA 000
16

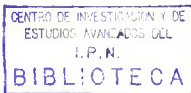
✓
CU

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

CENTRO DE INVESTIGACION Y ESTUDIOS AVANZADOS
DEL
INSTITUTO POLITECNICO NACIONAL

DEPARTAMENTO DE INGENIERIA ELECTRICA
SECCION DE COMPUTACION

UN MODELO DE HERENCIA MULTIPLE PARA TM



Tesis que presenta el Sr. Héctor Jiménez Salazar para obtener el grado de MAESTRO EN CIENCIAS en la especialidad de INGENIERIA ELECTRICA. Trabajo dirigido por el Dr. Miguel Gerzso Cady.

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

México D. F. Mayo de 1990.

X M

CLASIF.	90.16
ADQUIS.	81-11877
FECHA	24-X-90
PROCED.	RON
S	

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

agradecimientos

Al Consejo Nacional de Ciencia y Tecnología por brindarme el apoyo para realizar los estudios de posgrado.

A la Universidad Autónoma de Puebla por otorgarme todas las facilidades necesarias.

Al Centro de Investigación y de Estudios Avanzados por todas las atenciones que me fueron ofrecidas durante mi estancia.

Al asesor de esta tesis Dr. J. Miguel Gerzso Cady.

Al coasesor de esta tesis Dr. Renato Barrera Rivera.

A los profesores Adolfo Guzmán A., Manuel Guzmán R., Joseph Kolar, Sergio Chapa V. y Oscar Olmedo A., por sus valiosos consejos.

A todos mis compañeros y amigos, especialmente a: Guillermo B. Morales Luna, Isidro Romero Medina y Miguel A. Soriano J. por sus sensibles comentarios; Mady Fuerbringer Bermeo por transmitirme su gran motivación; Angel Martínez P. y Margarita Rivera (qepd) por su hospitalidad; Yolanda Martínez Rivera por su constante ayuda.

A MIS PADRES

Max y Concep

A MIS HERMANOS

Alma, Arianna, Ricardo, Azul y David

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

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

contenido

	PROLOGO	
1.	PRINCIPIOS DE DISEÑO Y TM.	11
1.1	Principios de diseño en lenguajes de programación.	11
1.2	TM.	13
1.2.1	Elementos del lenguaje.	14
1.2.2	Estructura del sistema.	20
2.	LENGUAJES ORIENTADOS A OBJETOS.	25
2.1	SMALLTALK.	25
2.2	FLAVORS.	29
2.3	TRELLIS/OWL.	32
2.4	LOOPS.	35
3.	HERENCIA.	38

3.1	Análisis de la herencia.	38
3.1.1	Herencia múltiple.	40
3.2	Un modelo de herencia múltiple.	42
3.2.1	Combinación.	44
4.	PRUEBAS.	49
4.1	Descripción general.	50
4.1.1	Mensajes.	51
4.2	Restricciones al modelo.	52
4.3	Consecuencias de la combinación.	53
4.3.1	Exploración de la red.	53
4.3.2	Verificación de tipos.	54
4.3.3	Generación de código.	54
4.4	Algoritmo de exploración	55
	EPILOGO	59
A.1	Ejemplos	62
A.2	Instrucciones de la máquina virtual de TM	70
	Bibliografía	73

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

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

PROLOGO

La "crisis del software" estimuló la formulación de nuevos principios que sin duda han influido en la definición de los lenguajes actuales, y la construcción de "programas grandes" es una de las problemáticas que enfrentan muchos de los lenguajes.

Ya que la implantación de un programa estará fuertemente marcada por los principios del lenguaje, es natural atacar el problema de construir programas grandes con lenguajes que sigan una metodología cimentada en la actividad de administrar grandes volúmenes de código, su clasificación y adecuada explotación.

Los lenguajes orientados a objetos presentan una metodología para construir "software" de una manera uniforme y sencilla. En ellos la herencia se considera como una de las características más importantes. Esta propiedad requiere concebir los elementos que maneja un sistema agrupados en clases, de modo tal que la herencia apoya a la metodología cuando se define una nueva clase con base en las ya existentes. Al organizar un sistema con este enfoque se tienen diversas consecuencias. Por ejemplo la reusabilidad, que ofrece mayor explotación de un código previamente definido. Desafortunadamente la herencia continúa sin una definición general y estándar -casi cada lenguaje orientado a objetos define de manera diferente la herencia llegando inclusive a debilitar algunos de sus principios de diseño.

La programación orientada a objetos promete ahorrar camino en el desarrollo de proyectos ambiciosos. Es por lo tanto tarea ineludible presentar una alternativa encaminada hacia el desarrollo de este tipo de proyectos. Se justifica

asi el diseño de un lenguaje que atienda a estas demandas y se sienten bases para su ulterior desarrollo. Empero, para conseguirlo no basta con el diseño de un lenguaje sino que es necesario emprender el camino poco explorado del diseño e implantación de lenguajes, semántica de los lenguajes de programación e ingeniería de software, que a su vez requiere de ingentes esfuerzos para formar grupos de investigación en las Areas mencionadas.

TM es un lenguaje que ha sido desarrollado en el Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas de la UNAM y pretende dar una alternativa al esfuerzo en la construcción de grandes proyectos en programación. Hasta ahora se han implantado dos versiones del compilador de TM. Sin embargo, ninguno de éstos incluye un manejo de herencia acorde con las necesidades establecidas en el proyecto.

El presente trabajo aborda el problema de definir un modelo de herencia para TM, problema que se encuentra ubicado dentro del diseño de una de las características del lenguaje. Para resolverlo se tomaron como base los principios de diseño establecidos en el proyecto TM y la experiencia que ofrecen otros lenguajes de programación.

En el primer capítulo de este trabajo se presenta el lenguaje TM, sus principios, en el marco general del diseño de los lenguajes de programación.

En el capítulo dos se hace una revisión panorámica de los lenguajes orientados a objetos considerados como representativos, destacando los rasgos esenciales acerca de su herencia.

En el capítulo tres se analizan los problemas que acarrea trabajar con herencia múltiple. Al esclarecer la esencia de este tipo de herencia y la manera de atenderla con los principios de diseño, se formula un modelo de herencia para TM.

Por último, en el cuarto capítulo se presentan pruebas realizadas con un prototipo, tomando como base el modelo formulado en el capítulo tres. Se describen dos módulos, editor de vistas públicas y compilador de mensajes, que sirven para definir nodos en la red de herencia y verificar la correctitud de los mensajes, respectivamente.

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

1. PRINCIPIOS DE DISEÑO Y TM

1.1 Principios de diseño en lenguajes de programación

El diseño de los lenguajes de programación se encuentra en permanente retroalimentación. Los nuevos lenguajes vienen a suplir las deficiencias de los anteriores, de acuerdo con la experiencia que se haya obtenido.

Resulta inevitable la especialización de los lenguajes. Habrá los que enfatizan más en los aspectos de la sencillez de los elementos que manejan, su significación teórica u otros. Todo ello se fundamenta en las necesidades generadas por el uso del lenguaje frente a una problemática. Tales necesidades se sintetizan en un criterio, útil en la decisión a lo largo de la elaboración del futuro lenguaje. A los criterios que determinan la esencia del significado o utilidad se le llama "principio". Debe señalarse que los principios persiguen una norma de programación como refinamiento de pasc, modularidad, eficiencia, legibilidad, separación de niveles de abstracción, etc. [10].

Generalmente el diseño de los lenguajes se basa en principios, aunque no todos los principios tendrán la misma importancia para cada lenguaje. Los principios básicos podrían expresarse como:

- 1) claridad,
- 2) economía y
- 3) seguridad.

10 De acuerdo al enfoque que da Tennent [Ten-61].

La especialización y la combinación de estos principios básicos conduce a enunciar otros principios más precisos. Actualmente se cuenta con principios que dan las pautas a seguir en el diseño e implantación para que en un determinado campo de aplicación, facilite al programador la expresión de la solución que le da a su problema. Algunos principios son excluyentes, otros antagónicos y algunos se reducen a ser corolarios. A continuación se listan aquellos principios que estaremos refiriendo a lo largo del presente trabajo [11]:

1. **Abstracción.** Significa que es posible separar y definir ciertos elementos relacionados a los cuales se les hace referencia de acuerdo a su especificación. Tenemos así expresiones con un nivel de síntesis mayor. Un claro ejemplo de ello son los procedimientos.
2. **Ocultamiento.** Establece que "el implantador debe saber sólo aquello que va a implantar y el usuario sólo aquello que va a utilizar". Este principio se evidencia plenamente en lenguajes tales como Modula-2, donde se tiene separada la definición de la implantación.
3. **Sencillez.** Se resume en reglas de construcción y significado sencillas. PASCAL cumple este principio.
4. **Regularidad.** Tiene que ver con la sencillez. Trata de reunir sus reglas de construcción y significado del lenguaje en forma breve y sin excepciones. LISP es un buen ejemplo de este principio.
5. **Ortogonalidad.** Principio que se cumple cuando las reglas que definen al lenguaje mantienen independencia. Así, con pocas reglas de construcción pueden generarse todas las posibles combinaciones del lenguaje. La regularidad deriva de este principio.
6. **Interfase manifiesta.** Las especificaciones a través de las cuales el usuario se refiera a entidades de su programa deben aparecer a la vista. En otras palabras evitar relaciones implícitas. El tipo implícito de FORTRAN no cumple este principio.
7. **Costo localizado.** Este principio se basa en el principio, muy conocido, de costo/beneficio utilizado en la toma de decisiones y se enuncia como: el costo (consumo de recursos en el sistema) no se debe propagar mientras el beneficio sea local. Un intérprete sin recolector de basura viciaría el principio.
8. **Inducción.** Es un corolario de la regularidad. Pretende que las reglas de un lenguaje sean generales respecto a la

11 [Mac-83].

aridad. Por ejemplo, ningún manual dice como declarar un procedimiento con "trece" parámetros, pero el usuario podrá hacerlo (inducción) correctamente, si es válido el principio de inducción.

9. Consistencia Sintáctica. Por ella se entiende que "casos similares tiene significados similares y casos diferentes significados diferentes".

10. Seguridad. La seguridad nos dice que: los programas que infrinjan las definiciones del lenguaje deberán detectarse. Principio que se ejerce en las primeras etapas de la compilación de un programa.

La claridad se refleja en los principios 1, 2, 3, 4, 5, 6, 8 y 9. La economía aparece como base en los principios 1 y 7, asimismo, de manera implícita en 2 y 6. Los principios 2, 6 y 10 se fundamentan en el principio básico de seguridad.

Para definir un lenguaje adecuado a una metodología de programación deben seguirse ciertos principios. Por ejemplo incluir la abstracción obedece al refinamiento de programas.

En la siguiente sección se plantean las necesidades del lenguaje TM que permiten destacar sus principios de diseño.

1.2 TM

TM se ha ceñido al uso de cierta metodología que es requisito para satisfacer necesidades del diseño arquitectónico ayudado con computadora.

Las características deseables según Gerzso [12] son:

- a) Manejo de objetos a través de mensajes.
- b) Representación de literales.
- c) Invocación mediante patrones.
- d) Manejo de vistas pública y privada y
- e) Uso opcional de excepciones.

Características que podrían expresarse en lineamientos metodológicos de programación como "abstracción de datos, particionamiento (en forma de vistas pública y privada), extensibilidad y reusabilidad" [13].

12 [Ger-84].

13 *ibidem*.

Con esta descripción de las especificidades del lenguaje, se pueden enunciar tres principios generales de diseño:

- 1) Ocultamiento (vistas y objetos).
- 2) Abstracción (reusabilidad).
- 3) Simplicidad (mensajes, patrones y literales).
- 4) Seguridad (excepciones).

El manejo de objetos y paso de mensajes tienen implícitamente como base los principios de regularidad y portabilidad.

1.2.1 Elementos del lenguaje

El lenguaje TM está influenciado principalmente por PLASMA y SMALLTALK. Para abocarnos a los aspectos de interés citamos a continuación los conceptos sintácticos y semánticos más distintivos.

TM es un sistema que maneja una colección de objetos. Los objetos pueden ser de dos tipos: *pasivos* o *activos*. Todo objeto tiene un *estado* que se representa a su vez por uno o varios objetos.

Un objeto se opera mediante el envío de *mensajes* (principio de regularidad). La forma de una expresión de mensaje es:

```
objeto_receptor <= patrón_del_mensaje
```

La forma del *patrón del mensaje* se especifica por:

```
selector { parametros ; palabras_clave }*
```

El *objeto receptor* puede ser una variable que representa un objeto, una literal, un mensaje o bien la palabra reservada *instance* que significa referencia al objeto mismo. Las *literales* permiten definir objetos constantes siempre que se conozcan la representación del estado. Por ejemplo

```
[.coord 72000 413]
```

es una literal que representa una coordenada (administrador .coord).

Los *administradores* son los encargados de atender los mensajes. Se dice entonces que todo objeto tiene un administrador. La composición de mensajes se llama *mensaje anidado*. Por ejemplo en:

```
una_persona <- dame el_nombre <- concat "del D.F."
```

"dame" es el selector, "el_nombre" una palabra clave. A la respuesta del primer mensaje se le envía otro, donde "concat" es el selector que, dependiendo del administrador del objeto obtenido en el primer mensaje, invocará la operación correspondiente. Asimismo el receptor puede mantenerse "fijo" y enviarle diferentes mensajes, *mensajes en cascada*:

```
tuberia <- lista_caracteristicas . <- cotización forma 3
```

A "tuberia" se le envía un primer mensaje sin parámetros y selector "lista_caracteristicas", luego nuevamente a "tuberia" se le envía el mensaje "cotización" con los parámetros "forma" y "3".

El lenguaje TM se constituye por expresiones de mensaje, declaraciones y asignamientos. El sistema además proporciona administradores básicos como por ejemplo .fix, .string, e .if que administran números enteros, cadenas y bifurcación de código, respectivamente. El usuario puede definir nuevas administradores con base en los ya existentes.

Un administrador es un objeto activo que tiene dos vistas: pública y privada. La estructura general de un administrador es:

```
{administrator .nombre
public
superclass lista_de_administradores
respuestas
end_public
private
vista_privada
end_private
}
```

La vista pública está orientada a informar a los usuarios cuáles son las relaciones con otros administradores y las *respuestas* del administrador. Las respuestas son patrones asociados a un administrador e indican cuáles mensajes se atienden. Los objetos que tienen administrador común se conjuntan en una *clase*. Una respuesta se compone del *selector* (un nombre), la secuencia de las clases de los parámetros y palabras clave. Asimismo, en el patrón de la respuesta aparece la *clase de la respuesta* del objeto que se produce después de enviar el mensaje correspondiente. La vista pública de un administrador para la estructura de datos de cola doble, ".deque" sería:

```
{administrator .deque
public
to_instance
```



```

fpush .fix <elem> <=> <inserta al inicio elem>
  <- instance;

fpop => <elimina al inicio>
  <- instance;

ftop => <consulta el primero>
  <- .fix;

notfull? => <no llena>
  <- .boolean;

bpush .fix <elem> => <inserta al final elem>
  <- instance;

bpop => <elimina al final>
  <- instance;

btop => <consulta el último>
  <- .fix;

notempty? => <no vacia>
  <- .boolean;

end_inst

to_itself

  create .fix <m> => <crea un objeto .deque de m lugares>
  <-instance;

end_it

end_public

```

En la vista privada se tienen las *variables de instancia* (nombres de los objetos que forman el estado de cualquier objeto de esa clase), *variables de clase* (nombres de los objetos que forman el estado del administrador) y dos secciones de métodos. Los métodos dicen cómo responder un mensaje. El administrador como objeto puede recibir mensajes que él mismo atiende "to_itself" y los métodos de los objetos de la clase "to_instance". A la vista privada sólo tiene acceso el implantador del administrador. Concretamente para el administrador .deque se tendría:

```

private

instance; arreglo : .array
      ; inout : .fix
      ; outin : .fix
      ; long : .fix   fields

itself ; n : .fix   fields

to_instance

fpush x =>
  .if <= (instance <= notfull?) then
    { instance;arreglo <= set instance;inout to x;
      .if <= (instance;inout <= eq 1) then
        {instance;inout := itself;n}
      else
        {instance;inout := instance;inout <= - 1};
        instance;long := instance;long <= + 1};
    <- instance
  end=>

fpop =>
  .if <= (instance <= notempty?) then
    { .if <= (instance;inout <= eq itself;n) then
      { instance;inout := 1 }
      else
        { instance;inout := instance;inout <= + 1};
        instance;long := instance;long <= - 1};
    <- instance
  end=>

ftop =>
  .if <= (instance <= notempty?) then
    {<- instance;arreglo <= get (instance;inout <= + 1)}
  end=>

notfull? =>
  <- instance;long <= ne itself;n
end=>

bpush x =>
  .if <= (instance <= notfull?) then
    {instance;arreglo <= set instance;outin to x;
      .if <= (instance;outin <= eq itself;n) then
        {instance;outin := 1}
      else
        {instance;outin := instance;outin <= + 1};
        instance;long := instance;long <= + 1};
    <- instance
  end=>

```

```

bpop =>
  .if <= (instance <= notempty?) then
    { .if <= ( instance:outin <= eq 1) then
      { instance:outin := itself;n }
      else
        { instance:inout := instance:inout <= - 1};
          instance:long := instance:long <= - 1};
      <- instance
    end=>

btop =>
  .if <= (instance <= notempty?) then
    {<- instance:arreglo <= get (instance:outin <= - 1)}
  end=>

notempty? =>
  <- instance:long <= ne 0
end=>

end_inst

to_itself

create x =>
  .if <= (x <= ge 3) then
    { itself;n := x:
      instance:inout :=1;
      instance:outin :=2;
      instance:long :=0;
      instance:arreglo := .tm <= create 1 .. x '.fix';
      <- instance)
    end=>

end_it

end_private)

```

El comportamiento es la asociación que existe entre las respuestas de un administrador y los métodos correspondientes. Cuando en la vista publica de un administrador se establecen relaciones con otros administradores, "superclass", se entiende que el comportamiento de esos administradores (administradores superiores o superclases) se hereda al administrador. Las consecuencias típicas de esta relación son: a) un administrador puede recurrir a administradores superiores para atender algún mensaje, b) los objetos que tienen un administrador definido con "superclass" tienen un estado extendido reuniendo todas las variables de instancia de los administradores superiores y c) cualquier respuesta definida en un administrador tiene precedencia sobre las respuestas heredadas. Otras implicaciones se derivan del tratamiento más específico que se dé a las superclases. El siguiente es un administrador para una estructura de datos tipo pila:

```

(administrator .stack

public
superclass      .deque
to_instance

  fpush .fix =>
    <- instance;

  fpop =>
    <- instance;

  ftop =>
    <- .fix;

  notfull? =>
    <- .boolean;

  notempty? =>
    <- .boolean;

end_inst

to_itself

  create .fix =>
    <-instance;

end_it

end_public

private
end_private)

```

Aquí podemos observar que se hereda todo el comportamiento del administrador .deque. Sin embargo la vista pública permite al usuario enviar sólo los mensajes adecuados para una estructura de pila. En este caso no fue necesario definir método alguno para el administrador .stack.

Para procurar la flexibilidad en la programación [14] se ofrece la agregación de respuestas, siendo esta una manera uniforme de trabajar con las vistas de un administrador. Al administrador anterior puede agregarse una respuesta con:

```

add_response      .stack
to_instance

size =>
  <- instance!long
end=>

end_inst
end_response

```

1.2.2 Estructura del sistema.

Se ha observado que la metodología de programación basada en objetos y mensajes proporciona uniformidad en la interacción con los sistemas (principio de regularidad). Apoyados en lo anterior, muchos lenguajes han pasado a ser sistemas con un medio ambiente completo [15]. Considerando los motivos de su creación, TM no es la excepción. En un sistema TM se distinguen dos grandes bloques:

1. Medio ambiente de objetos. 2. Máquina virtual.

El medio ambiente de objetos consiste de la red de herencia, donde se encuentran los objetos definidos y las relaciones establecidas entre sus administradores. Además la interfaz con el usuario formada por el compilador de expresiones y editor de vistas.

Este primer bloque se interconecta al segundo mediante un cargador ligador. La máquina virtual se constituye por el hardware, un intérprete de código TM y el manejador de memoria de objetos.

objeto se representan en localidades de memoria contiguas, definiendo en la primera localidad su longitud, seguida de su clase y los valores de la instancia (según las variables de instancia). De acuerdo a esta representación, el contexto que maneja la máquina virtual de TM es como sigue:

OBJETO CONTEXTO

longitud	tamaño del objeto
cadena dinámica	instancia de .contexto: apuntador al objeto previo
instrucción	instancia de .fix: desplazamiento del código a ejecutar
stacke	instancia de .stacke: apuntador al stack de evaluación
respuesta	instancia de .pc-code: apuntador al objeto código que se ejecuta
cadena estática	apuntador al objeto receptor
temporales	zona de memoria donde se alojan variables temporales del método

El stack que refiere el contexto sirve para evaluar las instrucciones. Conviene notar que la cadena dinámica tiene también una estructura de pila pues no se maneja concurrencia. La pareja (instrucción, respuesta) son las "coordenadas" de una instrucción a ejecutar en el siguiente ciclo. Las instrucciones que modifican el contexto de la máquina virtual son MANDA (invocar un método: genera nuevo contexto) y RETURN (regreso de un método: desaloja contexto). Al ejecutar MANDA, se especifica el método, éste puede estar definido en la clase del objeto receptor o bien ser un método heredado. En este último caso MANDA requiere una búsqueda en las superclases que define el administrador del objeto receptor. Para esto se utiliza el campo cadena estática en donde habrá un apuntador al administrador que a su vez, como objeto, tendrá valores de instancia que apuntan a los administradores superiores. Para aclarar lo anterior se presenta los dos primeros eslabones de la cadena estática, empezando con un objeto receptor de clase .stack:

OBJETO RECEPTOR

longitud	
clase	apuntador al administrador .stack
arreglo	apuntador a una instancia de .array
inout	instancia de .fix
outin	instancia de .fix
long	instancia de .fix

y el administrador .stack se podría representar como:

OBJETO ADMINISTRADOR

longitud	
clase	apuntador a la clase .tm
nombre	'stack'
superclases	apuntador a una lista de objetos .tm
respuestas	apuntador a una lista de objetos .pc-code
var. de clase	zona de memoria donde se alojan los valores del estado del administrador
var. de inst.	lista de nombres de las variables de instancia

Con lo anteriormente expuesto se tiene una idea del estado del proyecto y de los módulos que componen al sistema. Ahora es necesario por un lado, actualizarlos para corregir aspectos sintácticos y proporcionar mayor correspondencia con los

principios de diseño. y por otro, definir la forma de trabajar con la herencia. Esto último es el motivo de la presente tesis.

2. LENGUAJES ORIENTADOS A OBJETOS

Para proceder a definir un tipo de herencia en TM es necesario presentar primero, así sea panorámicamente, una revisión de algunos lenguajes orientados a objetos. Se han elegido SMALLTALK, FLAVORS, TRELIS/OWL y LOOPS, debido a que presentan variantes importantes del manejo de herencia.

2.1 SMALLTALK

Se considera a SMALLTALK como el lenguaje más representativo de los lenguajes orientados a objetos. SMALLTALK [19] es un lenguaje que maneja dos tipos de elementos: objetos y clases. Las clases agrupan a los objetos caracterizados por un comportamiento común (principio de abstracción). Mediante el envío de mensajes es posible realizar alteraciones en los objetos, que responden al mensaje de acuerdo al comportamiento definido en la clase a la cual pertenecen. La composición de mensajes que se envían a los objetos, sean nombres asociados a objetos creados o bien a objetos modificados por la acción de otro mensaje, se llaman expresiones. Por tanto, las expresiones determinan el comportamiento de la interacción de varios objetos. Siguiendo el principio de regularidad la composición típica de mensajes son abstraídas por el lenguaje. Así las estructuras de control pasan también a ser mensajes, por ejemplo:

```
pluma avanza: 100.  
pluma gira: 90.
```

```

pluma avanza: 100.
pluma gira: 90.
pluma avanza: 100.           puede escribirse como:
pluma gira: 90.             4 timesRepeat
pluma avanza: 100.         [pluma avanza: 100; gira: 90]
pluma gira: 90.

```

Los elementos que definen una clase son: nombre de la clase, superclase, variables de instancia, metodos y mensajes de las instancias y metodos y mensajes de la clase.

Las clases pueden verse también como objetos, lo que conduce al concepto de metaclass. Es posible enviar mensajes a las metaclasses, siempre y cuando sean congruentes con la categoría de agrupamiento. Por ejemplo [20], si definimos la clase:

```

Class Name                Objetografico
Instance Variables        posición tamaño
Instance Message and Methods

    definido: nuevolumar contamaño: indicado
    posición <- nuevolumar
    tamaño <- indicado

    muévete: pos
    self bórrate
    posición <- pos
    self pintate
    ****

Class Message and Methods

    genera: enunlugar !nuevoobjeto!
    nuevoobjeto <- self new
    nuevoobjeto definido: enunlugar contamaño: 10
    nuevoobjeto pintate
    ~ nuevoobjeto

```

es posible enviar el mensaje:

```
otromas <- Objetografico genera: enLaEsquinaIzquierda.
```

con lo anterior se dice que se "instancia" un objeto.

Al favorecer el principio de simplicidad (lectura en inglés), en los mensajes con varios parámetros, se viola el principio de inducción.

La herencia se consigue aplicando el principio de abstracción a nivel de clases. Una clase puede ser subclase de otra si es más particular en su definición. Considérese:

```

Class Name                Rectángulo
Superclass                Objetografico
Instance Variables        pluma
Instance Message and Methods

    constrúyase ;;
    pluma levanta; muévete; posición: baja.
    4 timesRepeat [pluma avanza: tamaño: gira: 90]
    ...

```

Los rectángulos heredan de los objetos gráficos el comportamiento, variables de instancia y mensajes. Es posible, entonces, referirse al "tamaño" de un rectángulo o bien enviarle un mensaje como:

```
rectangulo29 muévete.
```

sin que hayamos definido el método "muévete" en la clase de "rectangulo". Pero además se tiene la posibilidad de especializar un método heredado, definiéndolo nuevamente en la subclase ("overloading").

Para precisar los anteriores conceptos se presenta un ejemplo más completo y conocido:

```

"Clase que define una pila: Stack"
Class Name                Stack
Instance Variables        mem inout long n
Instance Message and Methods

    def: tam "define una pila de tamaño tam"
    inout <- 1.
    long <- 0.
    n <- tam.
    mem <- Array new: tam.
    ^self

    pop "desaloja el último"
    self notempty
    ifTrue: [ inout = n
    ifTrue: [ inout <- 1 ]
    ifFalse: [ inout <- inout + 1 ]
    long <- long - 1 ].
    ^self

    push: elem "inserta elem"
    self notfull
    ifTrue: [mem at: inout put: elem.

```

```

inout = 1
ifTrue: [inout := n]
ifFalse: [inout := inout - 1].
long := long + 1].
^self

top "consulta el extremo"
self notempty
ifTrue: [inout = n
  ifTrue: [ ^mem at: 1 ]
  ifFalse: [ ^mem at: (inout + 1) ]]

notempty "prueba si no esta vacía"
^(long = 0) not

notfull "prueba si no esta llena"
^(long = n) not

```

```

Class Message and Methods
cre: tam "crea un objeto tipo Stack"
tam > 3
ifTrue: [ ^self new def: tam ]

```

A partir de la clase Stack podríamos definir la clase Dequeue:

```

"Clase que define una cola doble: Dequeue"
Class Name          Dequeue
Superclass          Stack
Instance Variables  outin
Instance Message and Methods
  fpush: e "inserta en el extrmo anterior"
  ^self push: e

  fpop      "elimina tope, extremo anterior"
  ^self pop

  ftop      "consulta tope anterior"
  ^self top

  bpush: e "inserta en el extremo posterior"
  self notfull
  ifTrue: [mem at: outin put: e.
    outin = n
    ifTrue: [outin := 1]
    ifFalse: [outin := outin + 1].
    long := long + 1].
  ^self

```

```

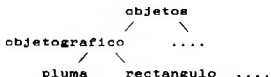
*
*
*

```

Class Message and Methods

```
cre: tam "crea una cola doble de tamaño tam"  
outin <- 2.  
^super cre: tam
```

Generalmente "objetos" es la clase inicial en un sistema desarrollado con SMALLTALK. Haciendo uso del principio de inducción y el principio de ortogonalidad, ésta pasaría a ser la raíz de un árbol de herencia. De aquí el nombre de herencia jerárquica.



Si quisieramos seguir desarrollando este árbol para especializar más los objetos gráficos, digamos dibujar elementos arquitectónicos "arquitect", en un momento un objeto que representa un elemento real requeriría otras características que no proporcionan las clases predecesoras; por ejemplo, el objeto "lavabo" requiere atención de la clase "inventario". Así se haría necesario efectuar un reacomodo de los nodos de la jerarquía. Otros lenguajes ofrecen esta posibilidad (promoción de características [21]).

2.2 FLAVORS

FLAVORS [22] es un sistema desarrollado con base en LISP. Cada objeto tiene un estado y un conjunto de operaciones que pueden ser aplicadas a él. A partir de "sabores", instancias, variables de instancia, funciones genéricas y métodos se definen los programas en FLAVORS.

Cada clase de objetos es implantada como un sabor. Un sabor resulta ser la abstracción de las características que las instancias de ese sabor tienen en común. Por ejemplo la definición del sabor "stack" se expresaría como:

```
(defflavor stack  
  (mem long n) ()  
  :readable-instance-variables  
  :writable-instance-variables)
```

y "pil20" una instancia de este sabor:

21 LOOPS lo maneja mediante el concepto "mixin" [Ste-86].

22 Descripción basada en [Moo-86].

```

(setq pil20 (make-instance 'stack
    :mem ()
    :long 0
    :n 20 ))

```

El conjunto de valores de variables de instancia representan el estado de cada objeto. El control sobre el estado de un objeto se hace a través de las opciones. "readable-instance-variables", por ejemplo, genera funciones de acceso para leer los valores de las variables de instancia.

Las operaciones que se realizan sobre los objetos son las funciones genérica. El código de la función genérica en LISP se conoce como "método". Comúnmente una función genérica tiene asociados varios métodos, los cuales corresponden a diferentes sabores. Algunos métodos del sabor "stack" son:

```

;inserta elem
(defmethod (stack :push) (elem)
  ((send self :notfull)
   (setq mem (CONS elem mem))
   (setq long (+ long 1)
   self))

;remueve el último insertado
(defmethod (stack :pop) ()
  ((send self :notempty)
   (setq mem (CDR mem))
   (setq long (- long 1)
   self))

;consulta el último
(defmethod (stack :top) ()
  ((send self :notempty)
   (CAR mem)))

(defmethod (stack :notfull) ()
  (NOT (EQ long n)))

(defmethod (stack :notempty) ()
  (NOT (EQ long 0)))

```

Comúnmente un sabor se obtiene combinando otros sabores, que se les conoce como componentes. El nuevo sabor hereda las características de sus componentes; esto constituye la herencia múltiple. El sabor "queue" pueda obtenerse combinando con "stack":

```

(defflavor queue
;sin variables de instancia:
  ()
;una superclase:

```

```

(stack)
:readable-instance-variables
:writable-instance-variables)

;forma elem en la cola
(defmethod (queue :forma) (elem)
  ((send self :notfull)
   (setq long (+ long 1))
   (setq mem (APPEND mem (LIST elem)))
   self))

```

El sistema maneja la interacción entre dos sabores, que se combinan, de la siguiente manera:

Si un sabor se define con varios componentes, éstos se ordenan y a la vez, por herencia, sucesivamente, se ordenan los componentes de los componentes. Los primeros son los más específicos y los que controlarán los métodos heredados al sabor compuesto. Para definir el orden parcial se observan tres reglas:

- 1) Un sabor precede a sus componentes.
- 2) El orden de los componentes de un sabor se debe preservar.
- 3) Si un sabor aparece varias veces, se mantiene el que se acerque más hacia el inicio mientras no viole las otras reglas.

Lo anterior significa que si tenemos:

```

(defflavor arroz-con-leche () (arroz leche))
(defflavor arroz () (cereal))
(defflavor leche () (producto-animal))
(defflavor cereal () (comestible))
(defflavor producto-animal () (comestible))
(defflavor comestible () ())

```

el sabor "arroz-con-leche" tiene los componentes:

```

(arroz-con-leche arroz cereal leche producto-animal
 comestible vanilla)

```

donde "vanilla" es un sabor que proporciona el comportamiento por omisión.

La definición de sabores no debe violar las reglas de ordenamiento. Tomemos el caso de:

```

(defflavor mazamorra-de-arroz () (arroz arroz-con-leche))

```

que tendría como componentes:

(mazamorra-de-arroz arroz ... arroz-con-leche arroz ...)

y el primer "arroz" debería eliminarse, pero esto viola la segunda regla del orden.

Las variables de instancia de un sabor vienen dadas por la unión de las variables de instancia de sus componentes.

En el caso de los métodos, cuando una función genérica se aplica a un objeto de un sabor particular, se toma el método asociado a ese sabor o a los de sus componentes. Al tener varios métodos se seleccionan e invocan en un orden particular y las respuestas de ellos se combinan. La forma más simple de heredar un método es seleccionar el método más específico. Por tanto, es necesario definir una forma de combinar las valores que regresan los métodos. De acuerdo al orden de los componentes de un sabor se elige un subconjunto de los métodos mediante un "método de combinación". Queda así establecido el orden de invocación y que hacer con los valores que regresan los métodos. Algunos tipos de "métodos de combinación" que proporciona el sistema aparecen enseguida:

- 1) Invocar sólo el más específico.
- 2) Invocar a todos en el orden dado o en el orden inverso.
- 3) Empezar con el más específico hasta invocar alguno que dé respuesta diferente de NIL.
- 4) Usar el segundo argumento de la función genérica para elegir uno de los métodos.

cuando se invocan varios métodos las respuestas pueden combinarse promediando los valores, o simplemente formando los valores en una lista, u otros.

2.3 TRELIS/OWL

Aunque TRELIS/OWL [23] tiene una sintaxis al estilo ALGOL utiliza la metodología de operaciones mediante el envío de mensajes. La abstracción de tipos y su jerarquía proporcionan una forma de herencia múltiple, que está matizada por la verificación de tipos en la compilación ("strongly typed"). Otras características son los iteradores [24] y las "excepciones".

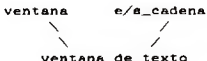
Los tipos describen comportamiento y es conveniente utilizar subtipos para organizar su abundancia. Un subtipo describe la especialización de un tipo determinado. El tipo

23 [Sch-86].

24 CLU presenta esta característica [Lis-77].

"ventana", digamos, es una región de la pantalla con propiedades de tamaño, posición y operaciones de movimiento, escalamiento o eliminación. Por otro lado, una "ventana_de_texto" es una ventana especializada y por tanto podemos verla como un subtipo de la primera. Asimismo, ya que los programas que utilizan objetos de tipo "ventana_de_texto" requieren leer y escribir desde ella y hacia ella, las operaciones de "ventana_de_texto" pueden tomarse de "e/s_cadena" (entrada y salida de cadenas).

Así pues, "ventana_de_texto" tiene el comportamiento de "ventana" y "e/s_cadena", que constituye una especialización de ambas y por lo tanto un subtipo de dos tipos.



La relación de subtipos tiene base en el comportamiento y no en la implantación. Así, la operación de lectura para "ventana_de_texto" puede tener diferente implantación a la de "e/s_cadena". Esto evidentemente ayuda a la verificación de tipos. Los tipos se definen enunciando sus operaciones. Por ejemplo el tipo de una cola doble de elementos genéricos:

```

type_module deque [elemento: type]
  operation fpush(me, elem: elemento);
  operation fpop(me);
  operation bpush(me, elem: elemento);
  operation bpop(me);
end type_module;
  
```

si se instancia:

```
var ce: deque[integer];
```

implica que las operaciones se instancian con ese tipo (integer). Virtualmente para "ce" se tiene:

```

operation fpush(me, elem: integer)
is begin
  if me.notfull then
    me.mem[me.inout] := elem;
    if me.inout = 1 then me.inout := me.n
    else me.inout := me.inout - 1
    end if;
    me.long := me.long + 1
  end if
end;
  
```

Por su parte en la definición de componentes se habilita explícita o implícitamente la consulta. Los componentes del tipo "deque" que generan implícitamente su consulta son:

```

components deque
component me.mem:      Array[1 .. me.n] of type
  is field;
component me.long:    Integer
  is field;
component me.n:      Integer
  is field;
component me.inout:   Integer
  is field;
component me.outin:   Integer
  is field;

```

Adicionalmente a las operaciones, los componentes permiten efectuar algunas operaciones sobre los valores que determinan el estado de alguna instancia del tipo. De esta manera las operaciones son privadas, sólo los subtipos tienen acceso. Para "deque" podríamos tener:

```

component me.notfull: Boolean
  get is begin
    return me.long<me.n;
  end;
component me.notempty: Boolean
  get is begin
    return me.long>0;
  end;
component me.ftop:    type
  get is begin
    return me.mem[me.inout-1];
  end;
component me.btop:    type
  get is begin
    return me.mem[me.outin-1];
  end;
end components;

```

La verificación de tipos orienta particularmente la herencia del lenguaje. Para explicar esto consideremos dos tipos S y T. Si S es subtipo de T ($S \ll T$), o bien que T es un supertipo de S, se tiene que cualquier objeto de tipo S se comporta como uno de tipo T; lo cual significa que puede ser utilizado donde se espere un objeto de tipo T [25]. Si tenemos que $S \ll T$ y

```

operation F(uno, otro: PtipoT)
  return(RtipoT);

```

se define para el tipo T, F pueda aplicarse con un parámetro de tipo PtipoT y regresa un objeto de tipo RtipoT. La operación F de S puede aceptar un parámetro de tipo "mayor" y regresar un objeto de "menor" tipo que la operación F de T. Entonces como

25 Por ejemplo con la declaración var cw: ventana_de_texto; es: e/s_cadena;
es válido: es:=cw; lee_linea(es); ... No así: cw:=es; mueve(cw,lugar);

todo tipo W: W << W. si

```
operation F(uno, otro:PtipoS)
return(RtipoS);
```

se define para el subtipo S, debe cumplir:

PtipoT << PtipoS y RtipoS << RtipoT.

Quando un tipo tiene varios supertipos, lo anterior se hace extensivo a todos ellos. Por otro lado, si S << T, una operación F de S debería tener efecto idéntico a la función F de T, pues se parte de que el comportamiento de un objeto de S debe contemplarse en el comportamiento de los objetos de T. Justamente la herencia nos permite ejercer tal característica en los subtipos de otros tipos. Así, por omisión, en un tipo se tienen todas las operaciones que proporcionan sus supertipos. También es posible "aumentar" el comportamiento del subtipo o "especializarlo", definiendo nuevas operaciones para él.

En el caso de un sólo supertipo, la operación heredada es directa. Si tiene varios supertipos que definen una operación con el mismo nombre, debe resolverse la ambigüedad. En TRELIS/OWL esta ambigüedad se elimina considerando que el usuario conoce las funciones proporcionadas por cada uno de los supertipos y, por tanto, elegir una de ellas. De esta forma el envío del mensaje se califica ("qualified"). Esto permite, colateralmente, resolver el conflicto que se origina entre una operación recursiva y otra del mismo nombre definida en uno de los supertipos. En la calificación de la operación se utiliza el nombre del tipo al cual pertenece la operación y un apostrofe como prefijos del mensaje:

```
operation despliega(me)
is
begin
    ventana'despliega(me);
    despliega_marco(me);
end;
```

es la definición de una operación del tipo "ventana_de_texto" que utiliza la misma operación de su supertipo "ventana".

2.4 LOOPS

LOOPS [26] constituye otro ejemplo importante de la familia de lenguajes orientados a objetos basados en LISP (COMMONOBJECTS, OBJECTLISP, COMMONLOOPS, etc).

Cada objeto pertenece a una única clase. Así pues, las clases son descripciones de objetos similares. Las metaclasses, a su vez, describen clases vistas como objetos.

La encapsulación se delega al usuario como mera convención de ocultamiento. No se distingue entre variable y método; podría hablarse de "valores activos" que cuando se consultan ejecutan un procedimiento (método). Una encapsulación típica podría ser:

```
MaterialConstruccion
Metaclass Class
  EditedBy (*mgc "25-jun-87 11:33")
  doc      (* Esta es la clase más general de
            materiales de construcción)
Supers (Materiales)
ClassVariables
  impuesto
InstanceVariables
  preciounitario NIL
  calidad          deprimera
Methods
  exhibe           MaterialConstruccion.exhibe
  dibujaInst      MaterialConstruccion.dibujaInst
```

Se sigue la regla de herencia por superposición: todas las descripciones de una clase (variables, propiedades y métodos) se heredan a una subclase, siempre que, en la subclase, no se tenga una descripción con el mismo nombre que la heredada.

La herencia múltiple en una red está caracterizada por el uso de una lista de precedencia de clases, ya que una clase hereda la unión de las descripciones de sus superclases, y ésta puede tener nombres de descripciones repetidas. A diferencia de otros lenguajes LOOPS permite en la encapsulación, controlar parte del paso de mensajes a las superclases (principio de seguridad). Por ejemplo dada la clase "Deque" podría definirse la clase "Stack":

```
Stack
Metaclass Class
  EditedBy (*ooy "15-feb-82 14:11")
  doc      (* ejemplo de definición de una
            clase con herencia)
Supers (Deque)
ClassVariables
InstanceVariables
Methods
  push           Deque.fpush
  pop            Deque.fpop
  top            Deque.ftop
```

Algunos nodos especiales (artificialmente contruidos en la red) juegan un papel importante en la formación de la lista de precedencia de clases. Dichos nodos se conocen como "mixin". Los mixin constan de grupos de descripciones que "factorizan" elementos comunes de otras clases. Al realizar el recorrido por profundidad en la red se da preferencia a las descripciones "más cercanas" a la clase en cuestión. Esencialmente lo anterior equivale a la obtención de componentes en FLAVORS, excepto que en LOOPS se conserva la última aparición de las descripciones repetidas. En la siguiente red de herencia "cotizaM2" representa un mixin:



En esta red "cotizaM2" contiene métodos que permiten efectuar el cálculo por metro cuadrado. Dichos métodos no podrían estar en "MaterialConstruccion" así, por ser comunes, se concentran en "cotizaM2".

No es posible, al igual que los demás lenguajes orientados a objetos, evitar el uso de "supermensajes" (<-Super). Aunque se tiene una unión de descripciones, no se opta por la combinación de respuestas de varios métodos que podrían ser invocados por un supermensaje. A excepción de una combinación particular "Superfringe", que activa los métodos de sus superclases que no han sido especializados. Asimismo, se tiene un tipo de mensaje calificado: "DoMethod". Se ofrecen, entonces, diferentes maneras de enviar mensajes. En la clase "muro" son válidos

```

(<- Super self dibujaInst)
(<- DoMethod self MaterialConstruccion.dibujaInst)
(<- self dibujaInst)
(<- Superfringe self dibujaInst)
  
```

Los dos primeros mensajes tienen el mismo efecto. Suponiendo que "cotizaM2" tuviera definido el método "dibujaInst", el tercer mensaje lo invocaría y en este caso, el cuarto, sería equivalente a los dos anteriores.

Podemos ahora hacer un análisis que dé pautas sobre la forma de definir convenientemente la herencia, que es precisamente el objetivo del siguiente capítulo.

3. HERENCIA

Podríamos afirmar que el resultado de aplicar el principio de regularidad, en el diseño de los lenguajes orientados a objetos, conlleva a mantener el principio de abstracción no sólo en la formación de clases sino además en las relaciones que se establecen entre ellas. Así pues, se concibe la especialización de clases como una abstracción sobre las clases mismas. La relación de subclasificación se ha utilizado desde SIMULA [27] como una forma de herencia.

La herencia es una característica de los lenguajes orientados a objetos que permite hacer referencia desde un objeto de una clase a las definiciones establecidas en otra. De esta forma es posible hacer refinamientos; especialización -al modificar las definiciones heredadas- o extensión -al agregar nuevas definiciones. También se logra con la herencia la reusabilidad de elementos de software. La herencia que se propone para TM parte de reutilizar ampliamente el comportamiento de los objetos previamente definidos. Para fundamentar esta propuesta en la siguiente sección se analizan diferentes enfoques que se le ha dado a esta característica.

3.1 Análisis de la herencia

²⁷ Hoare [Hoa-72] (pp 208) se refiere a una estructuración jerárquica de programas mediante la prefijación de clases (declaración de superclase).

SMALLTALK ofrece un mecanismo de herencia de tipo jerárquico para desarrollar sistemas con alto grado de complejidad. Empero, sin perder los principios establecidos para el diseño, este mecanismo resulta incompleto para modelar objetos más realistas, pues también es necesario definir objetos como combinación de especializaciones, o bien decir que pertenecen a varias clases [28].

TM no cumpliría sus principios de diseño, al usar herencia jerárquica pues, como veremos, este tipo de herencia viola dos de sus principios al intentar definir objetos que posean comportamientos combinados de clases previamente definidas:

Si interpretamos la pertenencia a una clase como subclasificación, en el esquema de herencia jerárquica, podría decirse que un objeto pertenece a varias clases; ya que la clase A de un objeto es subclase de otra, B, que a su vez tiene como superclase a C, y así sucesivamente. Lo anterior limita el comportamiento de las clases B, C... según las necesidades de la clase A, e impide utilizar las clases B, C... como superclases de otra clase diferente a A. Esto significa que parte del comportamiento de las clases B, C, tendría que ser repetido para clases semejantes a A, con la consecuente violación del principio de abstracción. Por otra parte, si se define un comportamiento especial a las clases B, C...., anularíamos el principio de seguridad; por ejemplo existe la posibilidad de enviar mensajes heredados por las superclases B, C...., inválidos al comportamiento deseado para un objeto de la clase A.

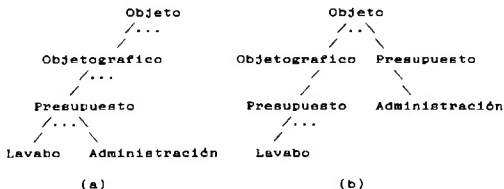


Fig 3.1 Violación del Principio de Seguridad (a) y Abstracción (b).

Se puede considerar, como lo hacen otros lenguajes, una red parcialmente ordenada en lugar de una estructura

28 Ver ejemplo al final de la sección 2.1. Cuando se desea definir la clase "inventario", esta tendría que ser subclase de "architect". Instancias que pertenecen a "inventario" y no son gráficas corren el riesgo de responder a mensajes de los objetos gráficos. De otra forma el comportamiento de ciertos objetos gráficos habría que repetirlo en la jerarquía inhibiendo el principio de abstracción.

arborescente. Con esta nueva forma de subclasificación también surgen nuevos problemas para los cuales los lenguajes con herencia múltiple presentan diversos enfoques de solución.

Otro problema implícito en el manejo de la herencia es el siguiente. En los lenguajes que definen sus variables de instancia dentro de la interfaz, el usuario puede modificar las variables de instancia de las superclases. Por tanto la herencia violaría la encapsulación, ya que tales variables no pertenecen a la interfaz de la clase utilizada. Algunas variaciones sobre esta declaración se da en lenguajes que permiten la referencia a variables de instancia de una clase sólo dentro de las clases sucesoras [29].

Por otro lado se subutilizan las operaciones de las superclases por no aparecer en la parte pública. También sería deseable "excluir" operaciones heredadas en el diseño de una nueva clase. Por ejemplo una "pila" reusa una "cola-doble", excluyendo las operaciones que refieren a la salida de la "cola-doble". Esto contrasta con la definición de herencia porque "cola-doble" se puede ver como un subtipo de "pila", ya que se refina el comportamiento de "pila". De aquí que la relación de subtipo no es la misma que la de subclase.

3.1.1 Herencia múltiple

En TRELIS/OWL se evita la problemática de elegir un elemento del comportamiento heredado calificando el mensaje. Ciertamente, de esta manera se eliminan los mayores problemas que implica la herencia múltiple. Sin embargo, con ello también se pierde la uniformidad en los niveles de abstracción que ofrece la estructuración por clases [30].

LOOPS tiene diferentes tipos de mensajes que pretenden salvar algunos problemas y además aprovechar las posibilidades de la herencia múltiple. Contrariamente a SMALLTALK, en LOOPS, se piensa que es posible proponer una relación de precedencia que no obligue al uso del mensaje calificado. La lista de precedencia de clases representa justamente el recorrido del algoritmo de búsqueda. Se ofrece pues un criterio de selección de métodos. Con todo, las ambigüedades persisten. Cuando se

29 Snyder postula que las variables de instancia no deben aparecer en la vista pública y sólo podrían tener acceso, desde las clases sucesoras, algunos métodos [Sny-86].

30 A nivel interno, el objeto con la especialización de comportamientos tiene la posibilidad de referirse a cada uno de los elementos con los que está compuesto, variables de instancia y métodos que hereda. El mensaje calificado desde el exterior haría, en cierta medida, caso omiso del principio de ocultamiento (estrictamente habría que especializar cada método heredado para utilizarlo desde el exterior; que va en contra de la abstracción) [Sny-86].

heredan varios métodos, el usuario debe tener la oportunidad de señalar cuál invoca [31]. Por otro lado, el concepto de "mixin" y la "promoción de comportamientos", es lo que nos acerca más a la esencia de la herencia múltiple [32]. Y aunque el origen de esta alternativa se encuentra en SIMULA [33] con los mixin se desfavorece el principio de regularidad en el desarrollo de grandes programas [34]; pasando a ser excepciones en la red de herencia más que nodos característicos del objeto modelado.

Al igual que LOOPS, FLAVORS calcula una lista de precedencia de clases (componentes del sabor). Ahora, son varios los protocolos que satisfacen la búsqueda, todos se invocan (como con la invocación Superfringe de LOOPS), pero además sus respuestas se combinan. Los "métodos de combinación" pasan a ser la forma de modelar la interacción de los objetos con clasificación múltiple. Este mecanismo vendría a reforzar en forma sui generis el principio de interfase manifiesta, sin detrimento de la abstracción.

La solución que se ha dado al problema de la herencia múltiple -elegir el método que será aplicado- se presenta en términos generales por:

- a) mensaje calificado.
- b) invocación múltiple y
- c) métodos de combinación de respuestas

El tratamiento que da cada lenguaje facilita el manejo de ciertas características de la herencia y en cambio desfavorece algún principio de diseño. Esperaríamos, por ejemplo, que "Superfringe", de LOOPS, además de eliminar la ambigüedad en la elección del método diera la oportunidad de uniformizar el envío de mensajes (principio de regularidad); ello implica definir un criterio general sobre la combinación de respuestas, lo que LOOPS deja exclusivo a la especialización de los objetos. La solución que va más allá de elegir un método (FLAVORS) se encuentra con el problema de efectuar una adecuada combinación de los métodos [35]. Para cada posibilidad siempre habrá ventajas y desventajas, por tanto lo que debe atenderse está en relación a cuáles son los principios de mayor importancia en el diseño del lenguaje.

En síntesis, los lenguajes con herencia múltiple requieren: determinar a que clase se "dirige" el mensaje. Esto se motiva en el aprovechamiento de todos los métodos heredados a la clase y no hacer una elección implícita que además resulta ambigua. Tenemos como muestra que "Superfringe", "Super" y el mensaje calificado son formas particulares y comunes de orientar el mensaje. Por otro lado se reducen a elegir todos o uno de los métodos heredados, sin embargo con estas formas de mensaje no se confiere calidad alguna a los métodos aplicados; la generacidad, por ejemplo, es una forma de relacionar los

métodos heredados con los métodos propios del objeto.

3.2 Un modelo de herencia múltiple

La herencia como hemos visto se ocupa de "pegar" comportamientos. No enfrenta el problema de reusabilidad en toda su extensión. Si queremos que la herencia combine otras clases debe especializarse el concepto de subclasificación. Consideremos los siguientes casos:

- 1) Generalización. Tal como se concibe en SIMULA, se concatenan las variables de instancia y los métodos de clases más generales pueden usarse para los objetos particulares.
- 2) Selección. Se elige de una o varias clases las propiedades de los objetos que pueden ser utilizadas "fuera de la clase" (se declaran en la parte pública).
- 3) Generacidad. Un clase genérica puede instanciarse con otra de tal manera que sus propiedades se comparten por ambas.

Aún esto no considera todas las posibilidades de la reusabilidad, por ejemplo efectuar reimplantaciones de un método en función de otro.

Sería deseable, entonces, formar un nuevo objeto con base en otros y expresar su interacción de acuerdo a sus características propias y las de sus componentes. Para formular un modelo que considere los anteriores planteamientos partimos de que la declaración:

```
superclass Cl, ..., Cln
```

indica que el objeto en construcción va a utilizar las clases Cl, ..., Cl_n.

31 Si la precedencia dispone el orden B, C como superclases de A ambas con métodos X e Y, los métodos de B siempre serían invocados.

32 Es natural que en el desarrollo de un sistema se encuentren casos donde, después de haber definido las clases, aparezcan comportamientos comunes y que es deseable agrupar [Ste-86].

33 [Hoa-72] (pp 203) se refiere al "bloque instanciado"

34 Puesto que los "mixin" estructuran a nivel de red, influyen en el "alcance" de los mensajes.

35 Excepto la linealización de FLAVORS este tipo de herencia vendría a ser la más acertada; es general además de ofrecer otro nivel de abstracción con la combinación de métodos.

En PRIVATE se define el nuevo comportamiento; cómo efectuar la combinación de los componentes. Por ejemplo algunos métodos heredados se superponen, redefiniéndolos. Y se dice que los mensajes al objeto que no sean atendidos por las definiciones de esta parte "pasan" (message passing) [36]. Abordar exhaustivamente los mensajes que no "pasan" es una característica de la especialización.

Sin embargo para los mensajes que si "pasan" se carece de un control más completo. Es claro que si no se utiliza el mensaje calificado se requiere de una especificación adicional para el nuevo objeto, donde se describa la manera de controlar las "múltiples" respuestas a un mensaje.

El modelo de herencia propone el uso de una definición que realice el control de la combinación. Todo objeto descrito podrá ser reutilizado al relacionar sus características con otras para describir un nuevo objeto.

La herencia puede verse como un polimorfismo por inclusión [37] que hace válidos aquellos métodos de clases superiores para atender a los objetos inferiores (subtipos). Cuando se envía un mensaje a un objeto compuesto por otros se elige el método correspondiente al recorrido en la red de herencia. Son varios los casos que se consideran:

1. La red se reduce a un árbol, el método es único.
2. En una red los métodos encontrados se pueden ordenar con algún criterio y seleccionar sólo el primero.
3. Igual que en el caso 2 pero se invocan todos los métodos encontrados.

Con el control de combinación se propone que se declare alguno de las siguientes opciones:

super) Elegir sólo algunos métodos y conformar con las respuestas un objeto de la misma clase del receptor (consistente).

selectivo) Seleccionar la parte del objeto que será afectada por el método.

genérico) Reutilizar un método genérico.

Aunque bien podrían emplearse otros nombres, por motivos de uniformidad, en lo sucesivo continuaremos empleando la declaración "superclass" para indicar el control de combinación.

36 [Hew-77].

37 En [Car-85] se definen varios tipos de polimorfismo.

A partir del análisis que se ha realizado es posible plantear una forma diferente que permita la reusabilidad, enfatizando el principio de regularidad y el principio de abstracción [38].

3.2.1 Combinación

La herencia múltiple debe ejercer un "polimorfismo extendido" invocando todos los métodos superiores asociados con un mensaje.

Por ejemplo si definimos la clase .time como:

```
(administrator .time
public
superclass .hour .min
***
)
```

el mensaje "print" a una instancia de .time podría invocar a los métodos "print" de .hour y .min. Esta combinación se describe como

```
superclass print in .hour .min
```

De esta manera se tiene implícitamente una selección de métodos: cuando se usa la combinación anterior sólo los métodos listados de las clases se consideran en la parte pública. Asimismo el principio de ocultamiento se mantiene: el usuario de la clase conoce qué métodos hereda. En tanto, el diseñador podría tener acceso a todos los métodos de la superclase: no sólo a los declarados en la vista pública.

Notemos que la anterior especificación no es exclusiva del control de combinación. Es decir, se puede implantar dentro del comportamiento del objeto, aunque esto equivale a superponer un método con otro que sólo lo invoca. Por ejemplo en el administrador .time se tendría:

```
private
to_instance
print =>
{super:hour <= print;
super:min <= print;
<-}
***
)
```

Por otra parte obsérvese que un mensaje

38 Se deriva de los planteamientos de TM [Ger-84]. En particular, la reusabilidad tiene su mayor apoyo en la abstracción y con la regularidad es más factible un ambiente amigable.

```
[.time ...] <= print;
```

tiene un efecto parecido de un mensaje en cascada porque no produce un objeto de la clase `.time`. Si en cambio declaramos:

```
superclass .hour .min
```

al enviarse un mensaje se produciría un objeto consistente. Por ejemplo si los métodos "print" de `.hour` y `.min` regresan la instancia operada y además `.time` tiene el método "set", es válido con la declaración de combinación super:

```
[.time ...] <= print <= set 10 15;
```

Es sabido que las piezas genéricas (procedimientos o tipos) son un recurso potente para el reuso de software. La forma de indicar combinaciones con piezas genéricas de software puede realizarse mediante declaraciones que instancian un objeto de una clase como parámetro de la pieza genérica [39]. Para adoptar lo anterior, en TM habría que añadir al control de combinación una declaración sobre la generacidad.

Para el caso de:

```
{administrator .ordenado
public
to_instance
minimo =>
<- anything
...

private
instance : anything
to_instance
minimo =>
(.if <= (instance <= tail <= null) then (<-instance <= head)
else <-((instance <= head) <= min (instance <= tail <= minimo)
...
)
```

el mensaje

```
[.list [.coord 2 3] [.coord 3 2] [.coord 1 4]] <= minimo;
```

sería válido siempre que `.coord` tuviera un protocolo "min" (que decide cuál es el menor entre dos instancias), `.list` tenga los métodos `head`, `tail` y `null` y se declarara en la parte pública de `.list` el control de combinación:

```
superclass ordenado(list)
```

esto es, se instancia la clase genérica `.ordenado` con `.list`.

39 El ejemplo tradicional es ADA. [Mey-86] analiza herencia vs generacidad.

Los métodos de `.ordenado` y `.list` refieren al mismo objeto ("instance" para `.ordenado` es `.list`). La instancia es común, pero se ejerce un paso de mensajes de acuerdo al orden jerárquico de las clases. La declaración de la variable de instancia `.anything` indica que `.ordenado` es una clase abstracta.

Con esta declaración podríamos corregir el "Problema del SELF" [40]. Supóngase que se tienen las clases:

```
(administrator .turtle
public
superclass .Den
to-instance
{forward <n_step>:fix =>
<-)
{backward <n_step>:fix =>
<-)
...
private
instance ; .anything
dir; .float
to-instance
{forward ns =>
declare x x0 y; float temporary
x0:= super<=xpos; y:= super<=ypos;
x := ns<=/(dir<=*dir<=+1<=sqrt)<=+x0;
y := m<=*(x<=-x0)<=+y;
super <= draw x y;
<-)
{backward ns =>
instance <= forward ns<=-;
<-)
```

```
(administrator .turtle-dash
public
superclass .turtle(turtle-dash)
to-instance
{forward <n_step>:fix =>
<-)
...
private
to-instance
{forward ns => ...
<-)
```

Aquí vemos que al enviar el mensaje "backward" a una instancia de la clase `.turtle-dash` se invoca el método que proporciona `.turtle`, debido a que éste no existe en el administrador `.turtle-dash` y además se tiene la declaración

40 Lisberman presenta a la delegación como la forma ideal para la reusabilidad y resuelve con ella el "Problema del SELF" [Lis-86].

```
superclass .turtle(turtle-dash)
```

Cuando "backward" envía el mensaje "forward" a "instance", se da prioridad al método "forward" de la clase que hereda (se superpone .turtle-dash a .turtle). Así, un mensaje dirigido por la declaración de combinación conserva los niveles jerárquicos de las clases. De otra forma el mensaje "backward" de .turtle-dash invocaría a "forward" de .turtle.

Los modos de combinación anteriores establecen relaciones poco amplias entre los componentes. Esto se debe a la falta de una descripción del objeto más completa que lo permita [41]. Algunos lenguajes han elegido el arido, aunque elegante, camino de los tipos de datos abstractos (ADT) [42] y aun cuando su desarrollo ha sido lento se prevén perspectivas atractivas. Nos interesaría poder plantear otros problemas de reuso de software, como por ejemplo: dado un objeto O1, descrito sólo por su ADT, y otro O2 descrito por su ADT y su implantación, obtener la implantación de O1 en términos de O2. A ello lo podríamos llamar "instanciar la implantación de O1 en el dominio O2".

El problema mencionado cae en las Áreas de verificación de programas y programación automática que no han tenido el éxito esperado. Una cuestión que tendría que ser explorada es abordar un problema como los anteriores relajando las condiciones, es decir, proveyendo una parte de la instancia de implantación que se desea: situación común en el desarrollo de software. Por ejemplo, sean los ADT:

```
ADT list (E) =
  new:          --> list
  cons: E x list --> list
  car:          list --> E
  cdr:          list --> list

COND
  cons(car(1),cdr(1)) == 1
  car(cons(e,1))      == e
  cdr(cons(e,1))      == 1

END list

ADT tabla (NOM VAL) =
  crea:          --> tabla
  inserta: tabla x NOM x VAL --> tabla
  elimina: tabla x NOM --> tabla
  busca: tabla x NOM --> VAL

COND
  elimina(inserta(t,n,v),m) ==
    n=m --> t, inserta(elimina(t,m),n,v)
  busca(inserta(t,n,v),m) ==
    n=m --> v, busca(t,m)
```

d1 [San-86] plantea una alternativa parametrizando las clases

d2 OBJ es un exquisito ejemplar [Gog-84]. También puede consultarse [Wul-76].

END tabla

si deseamos definir una nueva clase, .tabla_st_f que maneje una tabla de parejas de cadenas y enteros, tendríamos que declarar la combinación:

```
superclass .tabla(string fix)
superclass .list((string fix))
superclass [tabla:list inserta(t,n,v) := cons((n,v),t)]
```

que significa instanciar parámetros de los ADT respectivos e instanciar la implantación de "tabla" con ".list", conociendo la implantación de "inserta". lo cual supone declaraciones como:

```
{administrator      .tabla
****
private
instance | .anything <NOM> .anything <VAL>
****

{administrator      .list
****
private
instance | .anything <E>
****
```

Si se utiliza un algoritmo similar al de inferencia de tipos [43] obtendríamos:

```
elimina(t',m) := car(car(t'))=m --> car(t'),
              inserta(elimina(car(t'),m),car(car(t')),car(cdr(t')))
```

Claramente las ventajas de este reuso es que ni siquiera el código producido tiene que compilarse. El ADT es una descripción menos frecuente en los lenguajes de programación y su tratamiento llevaría a cambios profundos en el diseño de TM.

Es posible incorporar algunas de las ideas anteriormente vertidas, al diseño del lenguaje sin embargo habrá que considerar bastantes restricciones por la orientación inicial de este trabajo. Esta incorporación se presenta en el capítulo cuatro.

d3 Por ejemplo [Lev-84] presenta un algoritmo de inferencia de tipos que dilucida la idea para unificar ecuaciones y tipos.

4. PRUEBAS

En las anteriores secciones se ha analizado y propuesto un modelo de herencia diseñándose lo más posible a los principios de diseño del lenguaje TM. El presente capítulo describe la incorporación de este modelo al lenguaje TM. Cabe señalar que esto implica diversas modificaciones tanto sintácticas como semánticas. Es por ello que se decidió implantar un prototipo de compilador de mensajes reuniendo los elementos más viables para una etapa de experimentación.

Con las restricciones que anotaremos en la siguiente sección la tarea se concentra en definir el manejo de la red de herencia, lo cual se traduce en:

- a) Diseñar el tipo de archivos para mantener la red.
- b) Diseñar un algoritmo que efectúe la exploración de acuerdo a las necesidades de búsqueda en la red.
- c) Implantar el algoritmo de exploración de tal forma que se integre al desarrollo de la definición de las vistas públicas para formar la red de administradores.

Hasta el inciso c) es posible definir redes de herencia. El segundo paso exigiría explotar la red mediante el envío de mensajes, tenemos entonces que

- d) Analizar expresiones de mensaje que efectúen la verificación de tipos consultando la red y

e) Generar el código correspondiente a cada uno de los tipos de mensaje definidos por el modelo.

4.1 Descripción general

El requisito de compilar una vista pública para definir un nodo en la red condujo a la implantar un editor. Asimismo para verificar del funcionamiento del algoritmo de herencia se optó por la confección de un compilador de mensajes.

Tomando en cuenta que el diseño e implantación de programas conlleva al manejo de varios sistemas y subsistemas (sistema operativo, editor, compilador, etc), pretender el uso de las utilerías de un sistema de cómputo medianco de una manera uniforme resta productividad [43]. El desarrollo de grandes proyectos conduce a pensar en un ambiente donde coexistan varios programadores para los cuales el proceso de generación de código (ciclo edición-compilación-depuración) sea consistente y aproveche los recursos del sistema. A la fecha no pocos son los sistemas de desarrollo de software que ofrecen un ambiente integrado desde el cual todas las tareas cotidianas se realicen. TM tiene como objetivo proporcionar, más que un lenguaje un sistema de desarrollo con las anteriores características [44].

En los ambientes para desarrollar software es posible conjuntar diferentes herramientas de programación. Tal es el caso de los editores orientados a la estructura, donde a la vez que se codifica se analiza, realizando así dos pasos del ciclo de software (edición y compilación) [45].

Por otro lado, los módulos ahora contemplados en el proyecto de TM deben mantener una comunicación adecuada a fin de conseguir un verdadero ambiente de desarrollo. Bajo esta orientación y ya que cada vista pública representa un nodo en la red de herencia, el editor implantado constituye una primera aproximación a la integración de los módulos del sistema.

El editor dirigido por sintaxis [46] esta basado en el método de análisis sintáctico descendiente recursivo, con representación de la gramática en un árbol de atributos [47], lo cual resulta flexible, pues es posible reconfigurarlo cambiando solamente la gramática.

Dada una superclase es posible verificar para cada patrón de mensaje que define la vista pública, los mensajes consistentes. De esta manera puede modelarse el comportamiento de un administrador, habilitando o deshabilitando los patrones que se desee en la interfaz con el usuario. Esta característica semántica apoya la encapsulación.

El mecanismo de herencia adoptado puede verificarse en los mensajes. Ya que la herencia múltiple implica el manejo de un objeto compuesto de acuerdo a sus superclases, un mensaje enviado a un receptor de cierta clase obliga a verificar si existe un método en tal clase que tenga ese nombre o en su defecto llevar a cabo una búsqueda en la red de acuerdo a los componentes del receptor para obtener los mensajes que serán controlados por el tipo de mensaje. Es suficiente entonces, al momento de analizar la sintaxis, comprobar la existencia de los métodos que en la red, partiendo del nodo que denota la clase del receptor, satisfacen al mensaje enviado. Para este propósito se implantó un traductor dirigido por sintaxis que además de verificar la validez de los mensajes, es aprovechado en la generación de código para la máquina virtual de TM.

También el generador de código para mensajes se desarrolló con base en la técnica del análisis descendiente recursivo mediante código integrado a las producciones de la gramática. Este módulo contiene un procedimiento que efectúa el análisis léxico de manera directa sobre los símbolos de entrada.

Ambos módulos realizan su análisis semántico a partir de la información capturada, ya sea con acciones semánticas insertas en el análisis sintáctico; para la generación de mensajes, o bien con un recorrido del árbol de atributos generado en la edición.

Para implantar los dos módulos anteriormente mencionados se emplearon técnicas de construcción de compiladores, programación de sistemas y tecnología de software, haciendo uso de una microcomputadora compatible con IBM-PC, en el lenguaje TURBO PASCAL ver. 3.0.

4.1.1 Mensajes

La traducción que se efectúa es sobre un subconjunto representativo de las expresiones de mensaje (ver gráfica sintáctica en el apéndice 3). Dicha traducción, como ya se ha mencionado, consiste en analizar la sintaxis e invocar las acciones semánticas que correspondan. Durante este proceso se puede hacer directamente la generación de código para la

43 [Ger-84]

44 Ibidem.

45 En [Cap-85] se aborda el problema para lenguajes como ADA.

46 [Hoo-85] presenta una metodología abstracta.

47 [Zvo-86] presenta un ejemplo. La idea "árbol de atributos" la presenta Waite en "Semantic Analysis" capítulo dos de [Bau-74].

máquina virtual de TM (MVS). En el apéndice se muestra el conjunto de operaciones definidas para esta máquina, el cual fue utilizado como código intermedio en la generación de código.

Los elementos que se manejan a nivel sintáctico son las literales, los nombres de clases, el receptor de una clase, los nombres de mensajes y los parámetros de un mensaje. En el análisis semántico se verifica la correcta combinación de los elementos sintácticos y se obtiene el significado 'intermedio' para que al combinarse formen un significado de nivel mayor. El significado de un elemento sintáctico puede ser código o atributos necesarios para la definición de significados de los siguientes niveles. Los significados que se requieren en este caso son: la clase del receptor, la clase de la respuesta, la clase del administrador en curso, el código MVS y las literales definidas. La transmisión de estos significados a niveles mayores se puede comprender con el siguiente ejemplo. Tómese el mensaje:

```
instance <= oper (a <= men c) param2;
```

"instance" define la clase del receptor que es justamente la clase del administrador en curso. Por lo tanto "oper" se busca en los métodos de esta clase. Si no lo encuentra, inicia un recorrido en la red a partir de este nodo para buscar otros métodos que satisfagan el nombre y parámetros. Para este proceso primero es necesario obtener el significado de elementos de nivel menor como los parámetros. Cada parámetro tiene una clase; por ejemplo, al ser definido un parámetro como mensaje su clase será la clase de la respuesta del mensaje. Así pues, obtenidas las clases de los parámetros es posible efectuar la búsqueda en la red, de la unidad nombre-parámetros, y con ello definir el código y la clase de la respuesta.

4.2 Restricciones al modelo

De las declaraciones formuladas para controlar la combinación:

```
superclass    super-list
superclass    sel in super-list
superclass    class1(class2)
superclass    [class1:class2 implanta-list]
```

serán incorporadas sólo las primeras tres pues es factible convertirlas en calificadores de mensaje. Sintacticamente la forma de los mensajes cambia conforme las siguientes reglas:

```
super:        receptor <=                sel1 param
```

```
selectivo: receptor <= :super-list      sal2 param
genérico:  receptor <= :generic super    sal3 param
```

para conseguir las tres primeras declaraciones respectivamente. Se considera que *receptor* es un objeto de la clase donde se ha declarado "superclass", *sal1* invoca a todos los métodos de las superclases, *sal2 param* es un protocolo de las clases *super-list* y *sal3* es el nombre de un método de *super*.

4.3 Consecuencias de la combinación

Se ha dicho que una modelación más realista requiere la combinación de especializaciones previas. La organización de objetos en una red responde a tal exigencia. Pero además, de acuerdo a la orientación de TM, las características de diseño ayudado por computadora presenta necesidad de manipular objetos recursivos [48]. Este tipo de objetos se expresa con un ciclo en la red de herencia, y afecta sustancialmente el mecanismo de compilación del lenguaje.

El algoritmo de exploración de la red deberá emprender una búsqueda exhaustiva de los métodos que satisfagan un mensaje - limitada por los métodos más específicos- y además ser sensible a los ciclos. Motivado por una expresión de mensaje, llevar a cabo el recorrido en la red puede verse como un mecanismo de polimorfismo particular asociado al mensaje [49]. Cuando se han encontrado los métodos que satisfacen al mensaje, la declaración que se ha hecho sobre la combinación afectará al código generado.

4.3.1 Exploración de la red

Comunmente la organización de las clases se recorre con un criterio "depth-first" o "breadth-first", hasta encontrar el primer protocolo que satisfaga al mensaje. Esta ha sido precisamente la manera de simplificar y a la vez limitar los lenguajes con herencia múltiple que tratan de resolver la ambigüedad de los mensajes. Si se aprovechan todos los métodos, al considerar la invocación múltiple, se evita la ambigüedad, pero estamos obligados a efectuar una búsqueda exhaustiva en la red.

Se eligió el criterio "breadth-first" para realizar este

48 En [Ger-85] se dan algunas ideas.

49 Ya que se verifica que el receptor o una parte de él tiene una respuesta que se evalúa (ver [Car-85]).

recorrido lo cual quiere decir que cada componente del objeto se visita en el orden que aparece y después, sucesivamente, cada subcomponente de cada componente. La exploración se suspende en profundidad cuando un nodo ya visitado aparece nuevamente (ciclos) o bien al encontrar un protocolo que satisfaga al mensaje (métodos más específicos).

4.3.2 Verificación de tipos

A menudo la búsqueda de un método se deja al tiempo de ejecución pues se pretende efectuar enlace dinámico. Puesto que en TM se declaran tipos hay posibilidad de realizar una verificación de tipos al tiempo de compilación con lo cual se logra más seguridad.

En TM el usuario declara dentro de la vista pública la clase de la respuesta del método (principio de ocultamiento). Además, con esta encapsulación se consigue que en algunos casos el compilador verifique las expresiones de mensaje: dada la clase del receptor es posible buscar en la red el patrón del mensaje, a partir del administrador de esa clase, y generar el código correspondiente. Esto no siempre es posible pues en algunos casos tenemos declaraciones de la clase ".anything" que obliga a aplazar la búsqueda hasta el tiempo de ejecución. Lo anterior resulta plenamente necesario si queremos tener aplicaciones polimórficas. Uno de los ejemplos más claros es:

```
stack <= pop <= imprime;
```

Ya que "stack" puede estar formado por elementos heterogeneos, no es posible en la compilación saber cual es la clase del elemento que se está extrayendo de esa estructura. En otras palabras tendríamos un lenguaje de "tipo medio" [50] que el compilador deberá tomar en cuenta en el análisis semántico de las expresiones de mensaje.

4.3.3 Generación de código

Un mensaje se traduce como la invocación del método, previa inserción de los parámetros en una pila [51]. Localizados en la red los métodos que están asociados a un mensaje, hay que invocar aquellos de acuerdo al tipo de mensaje. Para el caso del mensaje selectivo se generarán

50 Apoyado en el principio básico de la compilación ("preferir el tiempo de la compilación al de la ejecución" [Aho-79]), la verificación de tipos se encontraría entre el ejercicio de tipo estático y el dinámico ("statically, dynamically typed").

51 De acuerdo con la máquina virtual [Car-86].

réplicas de código, para cada administrador, de envío de mensaje con sendas inserciones de parámetros en la pila. El mensaje genérico significa incluir en la clase corriente el método de la clase especificada, de tal manera que su código de envío de mensaje indicará que se trata de un método que lleva a cabo una preparación diferente en el proceso de cargar y ligar los métodos a los cuales hace referencia.

Se han propuesto cambios a la máquina virtual, agregando nuevas instrucciones. Si partimos de que los objetos no cambian de clase en la ejecución, el efecto de recorrer la red nos ofrece suficientes datos para identificar al método que va a ser empleado. Por tanto las instrucciones MANDA y MANDA NC [52] pueden ser precisadas con los siguientes parámetros:

MANDA x1 y1 z1 w1

donde x1 es la clase a la cual pertenece el método invocado,
y1 posición del método,
z1 el componente de la instancia al que va dirigido y
w1 número de parámetros

Para el mensaje no compilado el parámetro y1 contiene la posición de la tabla de literales donde se encuentra el nombre. En este caso la búsqueda se hará en la red en el tiempo de ejecución. El parámetro z1, cuando es un mensaje no compilado también puede indicar la compartición de una instancia mediante el envío de un mensaje que invoca algún método genérico que, como hemos visto debe definir su instancia con la del receptor del mensaje (ver sec. 4.1 y apéndice 1).

Aquellas clases que tienen en su definición referencias a sí mismas son recursivas. Los requisitos para manejar mensajes a objetos recursivos son mayores. La recursividad se puede presentar de varias maneras -directa, indirecta, sencilla, múltiple o combinando estas formas. Se analizaron diferentes alternativas para manejar los objetos recursivos y se propuso una solución para la recursividad sencilla y directa (sólo un ciclo y de longitud uno) aunque no fue plenamente desarrollada. La idea consiste en descomponer el objeto recursivo en su "cabeza" y "cola" de tal forma que pueda constituirse una nueva instancia ("instance") y con ésta trabajar los mensajes. También el algoritmo de exploración en la red considera los ciclos que implica un objeto recursivo y se definió una manera de representar el código asociado a un objeto de tal naturaleza, iterando -más que con recursión- el código que se distribuye a las partes repetitivas del objeto recursivo.

4.4 Algoritmo de exploración

Se ha propuesto que una vez obtenidos los métodos heredados éstos sean combinados de una manera particular. En la siguiente sección se expone primero la forma de recuperar los métodos sin control alguno, posteriormente se precisa el efecto de los tipos de mensajes elegidos para su incorporación al prototipo.

La información que van definiendo las vistas públicas se almacena en un archivo que constituye la red de herencia. Un registro de tal archivo contiene la siguiente información: nombre del administrador, número de superclases, lista de superclases (índices de los registros que las definen), número de protocolos y respuestas. A su vez, cada respuesta contiene: nombre del método, clase de la respuesta, número de parámetros y la lista de parámetros (fig. 4.1). El diseño del registro es flexible para posibilitar futuros cambios del sistema.

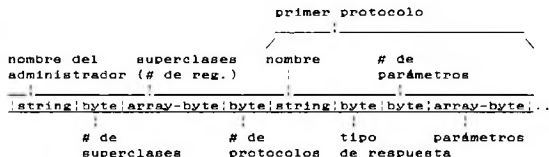


Fig. 4.1 Formato de un nodo de la red.

Los dos módulos implantados hacen acceso al archivo mediante una tabla hash que discrimina a los administradores inexistentes.

Se recorre la red con el criterio breadth-first ya que los componentes atómicos de todo objeto se ordenan según aparezcan en la definición.

Por ejemplo si se define

```

administrator      .empleado
superclass         .persona .salario

administrator      .persona
superclass         <edad>.fix <nombre>.string
  
```

los componentes atómicos que forman a empleado son:

```

persona salario edad nombre.
  
```

(en este caso edad y nombre no pueden descomponerse más, al igual que salario)

El algoritmo deberá tomar en cuenta los nodos anteriormente visitados para restringir el acceso a nodos ancestros.

Supóngase que, en una red como la que se muestra en la figura 4.2, se tiene el objeto "o" formado por los componentes c1, c2 ... cn y un mensaje

```
o <- sel param
```

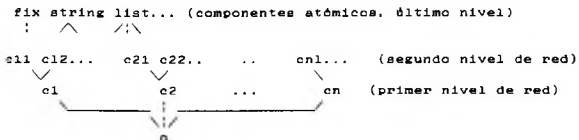


fig. 4.2 Red de herencia.

El objetivo del algoritmo es recorrer la red para el caso extremo de la invocación múltiple y verificar que el mensaje es válido. Por tanto la clase de "o" debe tener un método con nombre "sel" y parámetros "param". De no ser así, los componentes del objeto pueden responder al mensaje es decir en alguna clase c1, c2... , cn, debe existir "sel" "param" como la definición de un protocolo. Así sucesivamente para cada uno de las clases en que se descomponga el objeto en cuestión. Lo anterior constituye el recorrido simple con el criterio "breadth-first".

Al recorrer la red es necesario poder 'desactivar' la captura de métodos en algún nodo determinado, puesto que hemos dicho que deben identificar los ciclos, pero sin dejar de ascender en la red. Esto último se requiere para ir encontrando los desplazamientos a los componentes de diferentes niveles y, que van a servir al código generado.

Resumiendo las ideas anteriores tenemos el siguiente algoritmo de herencia en un lenguaje de superalto nivel:

```
herencia( (parámetros de entrada:)
selector_ parámetros, ( cadena formada por el nombre
del selector, parámetros y palabras clave)
clase_receptor, ( entero que representa la clase )
```

```

restricciones. { conjunto de clases que llevan a
                ciclos}
solo_recorre.  { bandera que inhibe la extracción de
                métodos}
                {parámetros de salida:}
desplaza.     { cadena formada por coordenadas de cada
                método a donde se distribuye el mensaje}
clase_respuesta. { cadena formada por las respuestas
                de cada método}}

lee_nodo_de_la_red      {según clase_receptor};
extrae( superclases ) {campo del registro leído};
IF NOT solo_recorre THEN
  WHILE (k IN superclases) AND (NOT encuentra)
    encuentra:= busca(k.protocolo,selector_parametros);
    IF encuentra THEN
      desplaza:= desplaza + k.coord;
      clase_respuesta:= clase_respuesta + k.respuesta
    END
  ELSE
    FOR k IN (superclase - restricciones)
      herencia(..restriccion U k..) {iguales parametros
                                     excepto las restricciones}
    END
  END herencia;

```

EPILOGO

A pesar de que han sido reducidos no dejan de ser importantes los avances que han habido en la forma de manejar la herencia múltiple. Esto se refuerza con la observación que en varias ocasiones se ha hecho: "la herencia jerárquica mas que permitir la evolución del software lo concibe estático, una la taxonomía ideal de un sistema. Lo cual contrasta con el camino que se recorre cuando se desarrollan programas.

Por lo anterior, resulta fácil aceptar que la metodología de los mixin sea una vía más natural para desarrollar software. Sin invalidar una línea muy diferente planteada por los metodos de combinación de FLAVORS. Al parecer [1] hasta el momento son las únicas alternativas para trabajar con herencia múltiple.

Uno de los primeros intentos para proveer de herencia múltiple a los lenguajes orientados a objetos fue la de una versión extendida de SMALLTALK: construir la lista de las superclases y buscar el primer método correspondiente al mensaje (linealización). Posteriormente se hicieron diferentes modificaciones superficiales para lenguajes de este tipo (LOOPS por ejemplo). En el presente trabajo se aumentó la linealización con el criterio del diseñador de una clase para que controle parte de esta búsqueda. a lo que se le ha llamado control de combinación. El control de combinación expresa una primera aproximación a la definición cabal de cómo combinar clases para formar una nueva. En él se adoptan dos mecanismos que apoyan los principios de abstracción y sencillez, conocidos

1 Haciendo algunas excepciones, que son realmente variaciones de lo anterior: Mehmet Aksit, Armand Tripathi "Data Abstraction Mechanism in SINA/ST" OOPSLA Conference Proceedings, sept 1988, pp 267.

como:

- invocación calificada y
- tipos genéricos

Para el primer caso, la necesidad de especificar la elección de un método, se traduce en todavía una falta de precisar la generación de clases a partir de otras. Sin embargo con la omisión de este modo de invocación podría caerse en especificaciones sofisticadas [2]. Por su parte la idea de los tipos genéricos ha penetrado fuertemente en los lenguajes orientados a objetos hasta llegar al concepto de FRAMEWORK [3].

Como cualquier mecanismo de reusabilidad el aquí presentado no podía desligarse de la actividad concebida alrededor del desarrollo del software. Se atiende pues a cada uno de los participantes en este proceso:

en los requisitos	-----	usuario
en la especificación	-----	diseñador
en la implantación	-----	implantador

Para TM las etapas correspondientes quedan asociadas a las vistas e interacción entre ellas, de acuerdo al siguiente esquema:

usuario	----->	requiere manejar objetos X con operaciones x1, x2, ...
		↓
diseñador	----->	especifica en la vista pública el administrador X ubicándolo en la red, con la combinación adecuada
		↓
implantador	----->	define la vista privada con operaciones x1, x2, ... usando lo necesario de acuerdo a la ubicación de X.

Como se ha dicho el modelo presentado no está acabado. algunos trabajos posteriores podrían ser:

1. Experimentar ampliamente con la herencia múltiple de TM. Por

2 Debilitando el principio de sencillez, como podría también suceder con el uso de tipos abstractos de datos.

3 Conjunto de clases abstractas, una clase abstracta contiene métodos genéricos, Johnson R. E & Foute B. "Designing Reusable Classes" JÜOP VI #2 julio 1988, pp 22.

ahora se han probado ejemplos pequeños. Escribir varios programas de tamaño mediano ayudaría a robustecer el modelo.

2. Uniformizar el lenguaje. Para que TM logre su objetivo como sistema de ayuda a la administración de programas grandes habría que efectuar algunos ajustes:

a) Definir una sintaxis más acorde con los nuevos elementos del lenguaje.

b) Establecer la comunicación entre la vista pública y la privada mediante la generación de un editor de vistas privadas.

c) Incluir el manejo de versiones, que tome en cuenta las reglas de modificación de objetos de tal forma que se optimice la ejecución (compilando todo lo que sea necesario de manera óptima).

3. Investigar otras formas de subclasificación. La relación común entre clases "es_un" puede diversificarse. Una clase abstracta como superclase es un caso muy especial de "es_un". Otras formas en torno al manejo de ADT por ejemplo la relación "hecho_de".

En la primera década de producción con los lenguajes orientados a objetos se puede estimar pocos usuarios aunque un número que va creciendo [4]. Es de esperarse que en los próximos años con las aplicaciones extensivas se robustezca el estilo de programación con esta metodología, se definan mecanismos más adecuados para el ciclo de vida del software, asimismo se adapten a las nuevas arquitecturas de computadoras.

4 Survey OOPSLA 87 Addendum to the Proceedings, oct 1987 pp 139.

A1 EJEMPLOS

En este apéndice se presentan algunas pruebas del compilador de mensajes. Los diferentes tipos de mensajes que da lugar la gramática del lenguaje son:

- a) anidado
- b) cascada
- c) parámetro como respuesta
- d) receptor como respuesta
- e) selector como respuesta
- f) heredados (invocación múltiple, seleccionado y genérico)

Para ejemplificar cada uno de estos utilizaremos la siguiente red que relaciona administradores de diferentes maneras, asociadas a las combinaciones del modelo, y que comprenderá cada uno de los mensajes.



Las operaciones que ofrece cada clase de objetos son:

```
tab
inserta : tab X nom X val --> tab
añade : tab X nom X val X indice --> tab
```

```

actualiza: tab X indice X val      --> tab

tabAtr
  actualiza: tabatr X indice X val  --> tabAtr

list
  cons      : list X E              --> list
  head      : list                  --> E
  tail      : list                  --> list

pila
  push      : pila X E              --> pila
  pop       : pila                  --> pila
  tope      : pila                  --> E

inter
  create    : fix X fix              --> inter
  eval      : inter                  --> inter

```

La red anterior representa la definición de los objetos de la clase .inter. El administrador .inter atiende los mensajes para evaluar expresiones de LISP puro. Dicha evaluación se basa en el modelo de una máquina SECD (Landin P. J. The mechanical Evaluation of Expressions, Computer J. enero 1965), cuya descripción semántica es:

El estado esta formado por tres partes:

```

Tabla asociativa  (tabAtr)
Expresión         (list)
Pila              (pila)

```

En la tabla asociativa pueden almacenarse nombres con sus características y recuperar una característica asociada a un nombre. La expresión se representa en una lista encadenada y la pila almacena elementos de cualquier tipo. La operación se realiza recursivamente cambiando el estado de acuerdo a reglas como las siguientes:

```

==constructores desmanteladores==
eval [ tabAtr, (CAR x) list, pila ] =
eval [ tabAtr, x car list, pila ]

eval [ tabAtr, (QUOTE x) list, pila ] =
eval [ tabAtr, list, x pila ]

...

==paseo de parámetros==
eval [ tabAtr, (LAMBDA (x) f) list, z pila ] =
eval [ tabAtr+(x,z), f # list, (tabAtr, list, pila) ]

==recuperación del estado anterior==

```



```
eval [ tabAtr1. # list1. v (tabAtr. list. pila) ] =
eval [ tabAtr. list. v pila ]
```

=llamado a función=

```
eval [ tabAtr. (APPLY f x) list. pila ] =
eval [ tabAtr. f list. x pila ]
```

```
eval [ tabAtr. , pila ] = (tabAtr. , pila).
```

En el siguiente fragmento del administrador .inter (vista privada) se muestran los diferentes tipos de mensajes definidos:

```
private
to_instance
```

<evalúa una expresión>

```
eval =>
  <- (instance <= tail) <= (instance <= :list head)
end=>
```

<procesa el paso de parametros>

```
LAMBDA => decl nom: .string; val: .anything;
  nom := instance <= :list head;
  val := instance <= :pila tope;
  <- instance <= tail <= :generic tab inserta nom val
  <= pop <= eval
end=>
```

<aplica una función>

```
APPLY => decl fun: .list; arg: .anything;
  arg:= instance <= :list tail <= :list tail;
  fun:= instance <= :list head;
  <- instance <= tail <= tail <= cons fun <= push arg <= eval
end=>
```

```
***
end_inst
end_private
```

Ejemplos de mensajes

a) Anidado. Estos son los más comunes:

```
_ [.boolean] <= and [.boolean] <= not;
```

TABLA DE LITERALES

```
4 0
4 0
```

```
PUSH VL 1
PUSH VL 3
MANDA
```

```
4 1 0
1
MANDA
4 2 0
0
```

b) Cascada. Mantiene el receptor para la secuencia de mensajes.

```
_ 'correcto' <= concat 's' , <= concat 'o':
```

TABLA DE LITERALES

```
2 8 99 111 114 114 101 99 116 111
2 1 115
2 1 111
```

```
PUSH VL 1
PUSH VL 11
MANDA
2 1 0
1
PUSH VL 1
PUSH VL 14
MANDA
2 1 0
1
```

obsérvese la diferencia con:

```
_ 'correcto' <= concat 's' <= concat 'o':
```

TABLA DE LITERALES

```
2 8 99 111 114 114 101 99 116 111
2 1 115
2 1 111
```

```
PUSH VL 1
PUSH VL 11
MANDA
2 1 0
1
PUSH VL 14
MANDA
2 1 0
1
```

Debido a la verificación de tipos no sucede lo mismo con:

```
_ [.pila] <= tope <= pop:
```

```
*** método indefinido ***
```

TABLA DE LITERALES

```
5 1 0
2 3 112 111 112
```

PUSH VL 1

MANDA

```
5 3 0
```

```
0
```

MANDA NC

```
0 4 0 0
```

_ [.pila] <= tope , <= pop!

TABLA DE LITERALES

```
5 1 0
```

PUSH VL 1

MANDA

```
5 2 0
```

```
0
```

PUSH VL 1

MANDA

```
5 2 0
```

```
0
```

c) Parámetro-respuesta. Este es un caso de la composición.

_ [.inter] <= create (1 <= + 2) (3 <= +1);

TABLA DE LITERALES

```
8 3 0 0 0
```

```
1 1
```

```
1 2
```

```
1 3
```

```
1 1
```

PUSH VL 1

PUSH VL 6

PUSH VL 8

MANDA

```
1 1 0
```

```
1
```

PUSH VL 10

PUSH VL 12

MANDA

```
1 1 0
```

```
1
```

MANDA

```
8 1 0
```

```
2
```

d) Receptor-respuesta. Se deriva del agrupamiento.

```
_ (435<= + 3465) <=> 64;
```

TABLA DE LITERALES

```
1 435
13465
1 64
```

```
PUSH VL 1
```

```
PUSH VL 3
```

```
MANDA
```

```
1 1 0
1
```

```
PUSH VL 5
```

```
MANDA
```

```
1 1 0
1
```

e) Selector-respuesta. Algo implícito en la sintaxis de TM, aunque poco puede hacerse en la compilación, para tener sentido, la respuesta debe ser de tipo "string".

```
_ [.tabAtr] <= ('ins' <= concat 'erta') 'str' 2;
```

TABLA DE LITERALES

```
7 0
2 3 105 110 115
2 4 101 114 116 97
2 3 115 116 114
1 2
```

```
PUSH VL 1
```

```
PUSH VL 3
```

```
PUSH VL 8
```

```
MANDA
```

```
2 1 0
1
```

```
POPVL
```

```
PUSH VL 14
```

```
PUSH VL 19
```

```
MANDA NC
```

```
7 21 0 2
```

f) Heredados. A pesar de restringirse a manejar la combinación con mensajes se pueden observar algunas características sobre la declaración que surgió en el modelo presentado. La invocación múltiple corresponde al mensaje sin restricciones y puede ser enviado por el usuario de una clase o el diseñador.

```
_ [.inter] <= inserta 'q' 2;
```

TABLA DE LITERALES

```
8 3 0 0 0
2 1 113
1 2
```

PUSH VL 1

PUSH VL 6

PUSH VL 9

MANDA

```
8 0 1
```

```
2
```

PUSH VL 1

PUSH VL 6

PUSH VL 9

MANDA

```
8 0 3
```

```
2
```

La diferencia con el selectivo es que el usuario queda restringido por la "ventana" que se define para la clase en cuestión:

```
_ [.inter] <= :tabAtr inserta 'q' 2;
```

TABLA DE LITERALES

```
8 3 0 0 0
2 1 113
1 2
```

PUSH VL 1

PUSH VL 6

PUSH VL 9

MANDA

```
8 0 1
```

```
2
```

Por último en el genérico tenemos que debe considerarse la compartición de la instancia para las dos clases

```
_ [.tabAtr] <= :generic tab inserta 'q' 2;
```

TABLA DE LITERALES

```
8 3 0 0 0
2 1 113
1 2
```

PUSH VL 1

PUSH VL 6

PUSH VL 9

MANDA

```
6 1 -7
```

```
2
```

cabe notar los parámetros con que se especifica MANDA. Los primeros dos y el último nos definen la "posición del método en la red" y el número de parámetros. El tercero se ocupa de la misma forma que antes (componente del objeto que es afectado por el mensaje), excepto que es negativo para indicar que no forma parte del objeto. Con esta descripción el ligador podrá aún verificar si procede el mensaje.

**AZ INSTRUCCIONES
MAQUINA VIRTUAL DE TM**

<u>Nombre</u>	<u>Función</u>
PushVR	Realiza el PUSH de la variable del receptor que se especifica en el siguiente byte.
PushVT	Realiza un PUSH de la variable del marco de temporales que se especifica en el siguiente byte.
PushVL	Realiza un PUSH de la variable del marco de literales que se especifica en el siguiente byte.
PushEVR	Misma función que PushVR pero extendida ya que el campo está especificado por los dos siguientes bytes.
Pushm1	Push del entero chico -1.
Push0	Push del entero chico 0.
Push1	Push del entero chico 1.
Push2	Push del entero chico 2.
PushNil	Push de Nil.
PushRec	Push del receptor.
PopVR	Se realiza el POP y STORE en una

	variable del receptor que se especifica en el siguiente byte.
PopVT	POP y STORE en una variable del marco de temporales que se especifica en el siguiente byte.
PopVL	POP y STORE en una variable del marco de literales que se especifica en el siguiente byte.
PopEVR	Misma función que PopVR pero extendida ya que el campo se especifica en los dos siguientes bytes.
Retu	Regreso de mensaje (RETURN) del tope del stack.
RetuRec	RETURN del receptor.
Retu0	RETURN del entero chico 0.
Retu1	RETURN del entero chico 1.
RetuNIL	RETURN de NIL.
Manda	Envío de un mensaje que será contestado por una RESPUESTA COMPILADA. El siguiente byte indica el desplazamiento para encontrar la respuesta en el marco de literales. El siguiente indica el número de argumentos del mensaje.
MandaNC	Se hace el envío de un mensaje cuya RESPUESTA no queda identificada en tiempo de compilación. El siguiente byte indica el desplazamiento en el marco de literales necesario para encontrar el IU (identificador único) de la RESPUESTA y el que le sigue indica el número de argumentos del mensaje.
MandaPr	Activa RESPUESTA PRIMITIVA. El índice de la primitiva está en el siguiente byte.
Salto	Salto corto incondicional. De hasta 255 bytes. La longitud del salto se especifica en el siguiente byte.
SaltoM	Salto mediano incondicional. De 256 a 511 de longitud. La longitud del salto

se encuentra sumando 255 con el valor del siguiente byte.

SaltoL Salto largo incondicional. De 512 a 767 bytes de longitud. El desplazamiento del salto se encuentra sumando el valor del siguiente byte con 512.

SaltoI Salto de un byte.

SaltoT Salto si el tope del stack es TRUE.

SaltoF Salto si el tope del stack es FALSE.

Stop Fin de interpretación.

NewObj Creación de un nuevo objeto. La longitud se obtiene del siguiente byte y el tipo del stack. Se debe realizar PUSH(tipo) antes de esta instrucción.

PushVA Push de variable de clase del administrador. La variable se especifica en el siguiente byte.

PopVA Pop y almacena en variable de clase del administrador. La variable se especifica en el siguiente byte.

PushVG Push de variable global. La variable se especifica en el siguiente byte.

PopVG Pop y almacena en variable global. La variable se especifica en el siguiente byte.

bibliografía

- [Aho-79] Aho, A., y Ullman, J., 'Principles of Compiler Design', 2a. Ed., Addison Wesley, EU, 1979.
- [Bau-74] Bauer, F. L. y Eickel, J. 'Compiler Construction An Advance Course', Springer Verlag, EU, 1974.
- [Bob-86a] Bobrow, D., Kahn, K., Kiczales, G., Manister, L. y Stefik, M., 'CommonLoops: Merging Lisp and Object Oriented Programming' en Sigplan notices of acm Vol.21 No.11, pp 17-29, EU, 1986.
- [Boo-86] Booch, G. 'Object Oriented Development' en IEEE Trans. on Soft. Eng. Vol.12 No.2, pp211-221, EU, 1986.
- [Bro-85] Bron, C., Dijkstra, E. y Rossingh, T. 'A note on the Checking Interfaces Between Separately Compiled Modules' en Sigplan notices of acm Vol.20 No.8, pp 60-63, EU, 1985.
- [Cap-85] Caplinger M. 'Structured Editor Support for Modularity and Data Abstraction' en Sigplan notices of acm Vol.20 No.7, pp 140-147, EU, 1985.
- [Car-85] Cardelli, L. y Wegner, P. 'On Understanding Types, Data Abstraction and Polymorphism' en Computing Surveys Vol.17 No.4, pp 471-522, EU, 1985.
- [Car-86] Cárdenas, G., S., 'Una Máquina virtual para TM'. Tesis de Maestría en Ciencias de la Computación UNAM, México, 1986.
- [DeR-76] DeRemer, F. y Kron, H. 'Programming -in- the -large versus Programming -in- the -small' en IEEE Trans. on Soft. Eng. Vol SE-2 No.2, EU, 1976.

- [Duf-86] Duff, C. 'Designing an Efficient Language' en Byte, pp 211-224, EU, agosto 1986.
- [Ewi-86] Ewing, Juanita J. 'An object-oriented operating system interface' Sigplan notices of acm Vol.21 No.11 Nov.86 pp 46
- [Fis-84] Fischer, C., Pal, A., Stock, D., Johnson, G. y Mauney, J. 'The POE Language Based Editor Project' en Sigplan notices of acm Vol.19 No.5, pp 21-29, EU, 1984.
- [Fos-86] Foster, D. 'Separate Compilation in Modula-2 Compiler' en Software Practice and Experience Vol.16 No. 2, pp 101-106, 1986.
- [Ger-84] Gerzso, J. M., 'Report on the TM Language Design' IIMAS-UNAM (sin publicar), México, 1984.
- [Ger-85] Gerzso, J. M. y Buchmann, A. 'TM - An Object Oriented Language for CAD and required database capabilities' en 'Languages for Automation', Editor Shi-Kuo Chang, Plenum Press, EU, 1985.
- [Gog-84] Goguen, J. 'Parametrized programming' en IEEE Trans. on Soft. Eng. Vol SE-10 No. 5 pp 528-543, EU 1984.
- [Gol-85] Goldberg, A., Robson, D. 'Smalltalk-80 The Language and its Implementation', Addison Wesley, EU, 1985.
- [Hen-86] Hendler, J. 'Enhancement for multiple inheritance' en Sigplan notices of acm Vol.21 No.10, EU, 1986.
- [Hew-77] Hewitt, C. 'Viewing Control Structures as Patterns of Passing Messages' en Artificial Intelligence (8), pp 323-364, EU, 1977.
- [Hoa-72] Hoare, C., Dijkstra, E. y Dahl, O. 'Structured Programming', Prentice Hall, G. B., 1972.
- [Hoo-85] Hood, R. 'Efficient Abstractions of the Implementation of Structured Editors' en Sigplan notices of acm Vol.20 No.7, pp 171-178, EU, 1985.
- [Jim-86] Jiménez, F. F. 'Diseño e Implementación de un Sistema Orientado a Objetos' Tesis de Maestría en Ciencias de la Computación UNAM, México, 1986.
- [Joh-86] Johnson, R., 'Type Checking in Smalltalk' en Sigplan notices of acm Vol.21 No.11, pp 315-321, EU, 1986.
- [Laf-85] Laff, M. Hailpern, B. 'SW2 An Object-based Programming Environment' en Sigplan notices of acm

- [Lan-86] Lang, K., y Pearlmutter, B., 'Oaklisp: An object-oriented Scheme with First Class types' en Sigplan notices of acm Vol.21 No.11, pp 30-37, EU, 1986.
- [Lar-84] Larson, P. y Kajla, A., 'File Organization: implementation of a method guaranteeing retrieval in one access' en Communications of acm pp 670-677, EU, julio 1984.
- [Lev-84] Levy, M. R., 'Type Checking, Separate Compilation and Reusability' en Sigplan notices of acm Vol.19 No.6, pp 285-289, EU, 1984.
- [Lie-86] Lieberman, H., 'Using prototypical objects to implement shared behavior in object oriented systems' en Sigplan notices of acm Vol.21 No.11, pp 214-223, EU, 1986.
- [Lis-77] Liskov, B., Snyder, A., Atkinson, R. y Schaffert, C., 'Abstraction Mechanism in CLU' en Communications of acm, pp 564-576, EU, agosto 1977.
- [Mac-83] MacLennan, Bruce J. Holt, 'Principles of programming Languages. Design, Evaluation and Implementation', Reinhart & Winston CBS C. Publishers, EU, 1983.
- [Mag-85] Magnenant-Thalman, N. y Magnenant D., 'Computer Animation. Theory and Practice', Springer Verlag, Tokyo, 1985.
- [McA-86] McAllester, D. y Zabin R., 'Boolean Classes' en Sigplan notices of acm Vol.21 No.11, pp 417-423, EU, 1986.
- [Mey-86] Meyer, B., 'Generacity versus Inheritance' en Sigplan notices of acm Vol.21 No.11, pp 391-405, EU, 1986.
- [Moo-86] Moon, D. A., 'Object Otiented Programming with Flavors' Sigplan notices of acm Vol.21 No.11, pp 1-8, EU, 1986.
- [Mor-86] Moriconi, M. y Hare, D., 'The Pegasys System: Pictures as Formal Documentation of Large Programs' en Transactions on Prog. Lang. and Sys. Vol.8 No.4, pp 524-546, EU, 1986. 524
- [O'K-85] O'Keefe R.A., 'Finding Smalltalk Methods' en Sigplan notices of acm Vol.20 No.6, pp 33-38, EU, 1985.
- [Pas-86] Pascoe, G., 'Elements of Object Oriented Programming' en Byte, pp 139-144, EU, agosto 1986.
- [Ram-86] Rammamcoorthy, C. V., Gaig, V. y Prakash, A., 'Programming in the Large' en IEEE Trans. on Soft.


Eng. Vol SE-12 No. 7. EU, 1976.

- [Ren-81] Rentsch, T. 'Object Oriented Programming' en Sigplan notices of acm Vol.17 No. 9, pp 74-86, EU, 1981.
- [San-86] Sandberg D. 'An alternative to subclassing' en Sigplan notices of acm Vol.21 No.11, pp 424-428, EU, 1986.
- [Sch-86] Schaffert, C., Cooper, T., Bullis, B. y Killian, M., 'An Introduction to Trellis/Owl' en Sigplan notices of acm Vol.21 No.11, pp 9-18, EU, 1986.
- [Sny-86] Snyder, Alan 'CommonObjects: an overview' en Sigplan notices of acm Vol.21 No.10, pp 19-28, EU, 1986.
- [Sny-86a] Snyder, A. 'Encapsulation and Inheritance in Object Oriented Programming Languages' en Sigplan notices of acm Vol.21 No.11, pp 38-45, EU, 1986.
- [Ste-86] Stefik, M. y Bobrow, D. "Object Oriented Programming: Themes and Variations" en The AI Magazine. pp 40-62, EU, 1986.
- [Str-86] Stroustrup, B. 'An Overview of C++' en SigPlan Notices of ACM Vol.21 No.10, pp 7-18, EU, 1986.
- [Swi-86] Swinehart, D., Zellweger, P., Beach, R. y Hagmann, R. 'A Structural View of the Cedar Programming Environment' en Transactions on Prog. Lang. and Sys. Vol.8 No.4, pp 419, EU, 1986.
- [Ten-81] Tennent, R. D., 'Principles of Programming Languages. Prentice Hall International, EU, 1981.
- [Weg-86] Wagner, Peter, 'Classification in Object-Oriented Systems' en Sigplan notices of acm Vol.21 No.10, pp 173-182, EU, 1986.
- [Wir-76] Wirth, N., 'Algorithms + Data Structures = Programs'. Prentice Hall, EU, 1976.
- [Wul-76] Wulf, W., London, R. y Shaw, M. 'An Introduction to the Construction and Verification of Alaphard Programs' en IEEE Trans. on Soft. Eng. Vol.2 No.4 pp 253-265, EU, 1976.
- [Zvo-86] Zvodnik, R. J. 'YALE: The design of Yet Another Language-based Editor' en Sigplan notices of acm Vol.21 No.6, pp 70-78, EU, 1986.

El jurado designado por la Sección de Computación del Departamento de Ingeniería Eléctrica del Centro de Estudios Avanzados del Instituto Politécnico Nacional, Aprobo esta tesis el 25 de mayo de 1990.



M. en C. Oscar Olmedo Aguirre



M. en C. Sergio Chapa Vergara



Dr. J. Miguel Gerzso Cady

CENTRO DE INVESTIGACION Y DE ESTUDIOS AVANZADOS DEL
INSTITUTO POLITECNICO NACIONAL

BIBLIOTECA DE INGENIERIA ELECTRICA
FECHA DE DEVOLUCION

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

13 DIC. 1995

DEVOLUCION

