Técnica de desarrollo de sistemas de objetos. (TDSO)

Isabel Besembel C.
Grupo de Ingeniería de Datos y Conocimiento(GIDyC).
Departamento de Computación.
Escuela de Ingeniería de Sistemas.
Universidad de Los Andes. Mérida-Venezuela.

Resumen:

Entre las múltiples metodologías de desarrollo de sistemas programados existentes hoy día, se encuentran el método deductivo MEDEE propuesto por el investigador francés Jean F. Dufourd como un método que engloba todas las tareas básicas del diseño de software, y la metodología OMT de Rumbaugh et al. especialmente diseñada para modelar sistemas orientados por objetos. El primero está basado en el refinamiento paso a paso, en la especificación formal de tipos abstractos de datos y en el diseño estructurado de sistemas programados. La segunda está basada en la combinación de tres modelos denominados el modelo de objetos, el dinámico y el funcional, todos ellos presentes en cualquier programa orientado por objetos. La nueva técnica presentada aquí, denominada Técnica de Desarrollo de Sistemas de Objetos (TDSO), es el producto de la integración y extensión de los métodos arriba mencionados, como una adaptación hecha por la autora para contener el análisis, la especificación, el diseño y la documentación, todo ello utilizado en la parte de diseño de programas y a la inclusión, de una manera sencilla, de la especificación de los casos de prueba de tales programas. Asimismo, se incluyen en TDSO la especificación de las funciones y de los eventos de los sistemas programados orientados por objetos.

I. Introducción:

Existen diversas formas tradicionales de expresar la resolución programada de problemas a través del computador digital anteriores al uso de la orientación por objetos, entre ellas se tienen: la técnica HIPO [Stay-76], el refinamiento paso a paso [Wirt-71], el diseño modular y estructurado [StMC-74, Mont-87], los algoritmos estructurados [Mont-82], el método deductivo MEDEE [Dufo-88], etc, que pueden ser

utilizados independientemente unos de otros, aunque ellos, normalmente se usan en conjunto, para analizar, diseñar y documentar sistemas programados. Con el advenimiento de la orientación por objetos aparece la metodología OMT (Object-oriented modeling technique) [RBPE-91], que describe todo el proceso de modelado de clases de objetos en el modelo de objetos, y que además incluye el soporte de las relaciones dinámicas y funcionales entre las clases a través de los modelos dinámico y funcional. La Técnica de Desarrollo de Sistemas de Objetos (TDSO) está basada en el método deductivo y en la técnica OMT, ambos mencionados anteriormente. Del primero contiene todas las fases, incluyendo además las de especificación formal [Gutt-77, Gutt-78, Hoar-69, Crai-86, Hoar-87]. De la segunda se toman algunos de los diagramas que fueron transformados y adaptados para TDSO, que soportan todos los constructos de la orientación por objetos. La extensión de tales métodos se hace con la inclusión de una guía para el desarrollo de las pruebas de tales sistemas basada en [JoEr-94, McKo-94, Post-94], utilizando el paradigma de la programación orientada por objetos.

Se inicia la presentación con un breve resumen de la orientación por objetos, que incluye las técnicas de especificación de los tipos abstractos de datos en base a axiomas algebraicos [Gutt-77, Gutt-78]. Luego se muestran los conceptos más importantes de la notación gráfica de OMT y una variante adaptada de la misma, que será utilizada en TDSO, y por último, la técnica de desarrollo de sistemas de objetos producto de la integración del método deductivo MEDEE y de OMT, además de la extensión realizada para contener la especificación de los casos de prueba de los sistemas de objetos.

II. Orientación por objetos:

La orientación por objetos es un nuevo enfoque para el desarrollo de sistemas programados, que permite la representación del dominio de una aplicación en forma natural y directa, en términos de los objetos que intervienen en dicha aplicación. La representación de tales objetos se hace en forma abstracta definiendo su estructura y su comportamiento.

Los conceptos de la orientación por objetos se remontan a los años 60 con la aparición del lenguaje de simulación llamado Simula'67, donde los objetos existen por sí mismos conteniendo sus datos y sus operaciones, y además se comunican entre sí por medio de mensajes. También fue en este lenguaje donde se introdujo la noción de clase de objetos, que describen la estructura y el comportamiento de todos

los objetos que pertenecen a la clase, la noción de herencia entre las clases siguiendo una jerarquía de las mismas, y por último, la noción de dos tipos de igualdad: la igualdad en valores y la identidad.

Los lenguajes de programación que aparecieron después y que siguieron estas premisas, fueron Smalltalk, Eiffel, Trellis/Owl, etc. La extensión de lenguajes existentes arrojaron nuevas versiones de: C en C++, Objective C y Borland C, LISP en Flavors, Object LISP y CLOS, y Pascal en TurboPascal6 y CLASCAL, entre otros.

Los conceptos de la orientación por objetos se basan en la manipulación de representaciones abstractas de los objetos plasmadas en su definición y su comportamiento. Un **objeto** se define entonces como la representación de algo que se describe mediante una *estructura* y un *comportamiento*. La **estructura** de un objeto describe aquellas características de interés presentes en el objeto y que sirven para plasmar el *estado* de ese objeto, siendo el **estado** de un objeto el conjunto de valores actuales almacenados en su estructura. El **comportamiento** del objeto está representado por una serie de operaciones, funciones o métodos que modifican o no el estado del objeto, haciendo que ocurra un cambio de estado en el mismo, el cual representa el comportamiento del objeto en la realidad. Así, el comportamiento del objeto está dado por sus cambios de estado.

II.1. Conceptos básicos:

Un objeto puede ser conocido y descrito por medio de sus propiedades o atributos que son *ilimitados*. Un objeto puede componerse de dos o más objetos, conformando así un **objeto compuesto**. Cada objeto tiene un único *identificador* que es *asignado* por el sistema, para aquellos sistemas que soportan la identidad del objeto.

Los **atributos** son las propiedades relevantes de un objeto que representa su estructura. Ellos pueden ser simples o monovaluados, en el caso que ellos contengan un único valor a la vez, y multivaluados cuando pueden contener varios valores a la vez.

La existencia de un objeto es independiente de los valores de sus atributos, así dos objetos son **idénticos** si ellos son **el mismo objeto** —sus identificadores son iguales—, o ellos son **iguales** si tienen **los mismos valores** en sus atributos. En un sistema o lenguaje que no soporta identidad del objeto la representación gráfica de los objetos es un *árbol*. En los sistemas que si la soportan es un *grafo*, ya que los objetos compuestos pueden compartir componentes.

Una operación es una función asociada al objeto que efectúa una acción atómica sobre el mismo. Tal acción puede o no modificar el estado del objeto.

II.2. Tipos abstractos de datos y su especificación:

Un tipo abstracto de dato (TAD) se basa en la separación clara entre su implantación y el uso del TAD a través de su interfaz, que es la cara visible del TAD. Ello implica que *un TAD tiene una interfaz* y puede tener varias implantaciones. La definición de un TAD se hace en dos partes: la especificación y la implementación.

Una especificación formal es la acción de determinar o explicar en términos formales o matemáticos la cosa deseada, en este caso un tipo de dato que es necesario para la resolución del problema. Este tipo de dato **T** se define "como una clase de valores y una colección de operaciones sobre esos valores. Si las propiedades de esas operaciones son especificadas solamente con axiomas, entonces **T** es un tipo abstracto de dato o una abstracción de dato" [Gutt-78,p.1049]. Una implantación **correcta** del TAD cumple con todos los axiomas especificados para él.

La especificación por axiomas algebraicos para el tipo **T** se compone de: una especificación sintáctica donde se definen los nombres, dominios y rangos de las operaciones sobre **T**; y una especificación semántica que está compuesta del conjunto de axiomas en forma de ecuaciones, que dicen como opera cada una de las operaciones especificadas sobre las otras.

La implementación se compone de: una representación que especifica como los valores del TAD serán almacenados en la memoria, es decir su estructura de datos, y los algoritmos que especifican como será usada y manipulada la estructura de datos, es decir las operaciones del TAD.

El acceso al TAD es hecho a través de su interfaz que es visible para los usuarios de ella. La implementación del TAD es invisible para el usuario y es visible para el que desarrolla el TAD.

Como un ejemplo de esto se presenta aquí la especificación del tipo abstracto de dato: *Pila*. Tal especificación será hecha para una pila no limitada.

Tipo de dato: Pila[elemento:tipoEle]

Especificación sintáctica:

creaPila()→ Pila,	Crea la pila
meterElePila(Pila,tipoEle)→ Pila,	Inserta un nuevo elemento en el tope
sacarElePila(Pila)→ Pila,	Elimina el elemento que está en el tope
conTopePila(Pila)→ tipoEle∪{TipoEleNoDef},	Consulta el elemento que está en el tope
vacíaPila(Pila)→ Lógico,	Verifica si la pila está vacía
destruyePila(Pila)→ .	Destruye la pila

Especificación semántica:

Declaración: *P:* Pila*, el:* tipoEle*;*
sacarElePila(creaPila()) = creaPila()
conTopePila(creaPila()) = {TipoEleNoDef}
conTopePila(meterElePila(*P,el*)) = *el*
vacíaPila(creaPila()) = Verdadero
vacíaPila(meterElePila(*P,el*)) = Falso

Con la especificación sintáctica se pueden ver claramente cuáles son las operaciones primitivas válidas sobre la estructura y cuáles son los tipos de datos que cada una regresa, luego que se ha efectuado la operación. Para mayor claridad, véase la operación sacarElePila() que necesita para operar la *Pila*, devolviendo la misma *Pila* pero disminuida en tamaño por un elemento, ya que ella suprime el elemento que está colocado en el tope de la *Pila*. En esta especificación se incluyen únicamente aquellas operaciones que son básicas para el TAD, es decir las operaciones que sirven de base para construir cualquier otra operación sobre el tipo. Por ejemplo, en la mayoría de las estructuras de datos es útil tener una operación que vacíe o limpie la estructura, en el caso de la *Pila*, sería la operación vaciarPila(Pila)→Pila. Dicha operación no aparece en la especificación del TAD *Pila*, pues ella puede ser construida invocando varias veces la operación sacarElePila(Pila) hasta que vacíaPila(Pila) sea verdadero. Por lo tanto, las operaciones que aparecen en una implementación cualquiera de un TAD no tienen por que ser iguales, en número, a las de la especificación del mismo.

Con la especificación semántica se observa el efecto que tiene cada una de las operaciones especificadas sobre las otras. Por ejemplo: ¿Qué sucede si se desea saber si la *Pila* está vacía, habiéndose recién creado?, esto es vacíaPila(creaPila()), el resultado es *Verdadero*, ya que aunque la pila existe, ella ha sido recientemente creada y por ello, está vacía. En el caso de conTopePila(creaPila()), el resultado es

un valor especial denominado *TipoEleNoDef*, esto es tipoElemento no definido, para expresar que no puede regresarse elemento alguno, pues la pila está vacía. Este valor especial para ***el*** debe estar definido en la implementación del tipo abstracto de dato.

Para llevar a la práctica esta especificación debe escogerse una representación física para el tipo *Pila*, ella puede ser una de las siguientes: la secuencial, donde los elementos de la *Pila* están contiguos en memoria; o el enlazado, donde esos elementos están esparcidos en la memoria y encadenados mediante punteros al siguiente. Ambas implantaciones, las hechas con las representaciones mencionadas, deben tomar en cuenta que la memoria es finita, y por ello tal especificación es lo que es, una especificación para un tipo abstracto de dato. Ella diferirá en algunos detalles de la implementación realizada basándose en ella.

II.3. Mecanismos de la abstracción de datos:

La abstracción consiste en enfocar los aspectos esenciales inherentes a una entidad e ignorar sus propiedades accidentales. El uso de la abstracción preserva la libertad de movimiento para tomar decisiones, ya que evita la acometida prematura de los detalles propios de las implementaciones en los lenguajes de programación. Esto es, hacer un estudio de los objetos antes de decidir como implantarlos. Los mecanismos de la abstracción de datos son: la clasificación, la composición, la generalización y la particularización.

La **clasificación** es la agrupación de objetos con propiedades y comportamiento similares dentro de una clase. Una **clase** es un objeto que define la estructura y el comportamiento de un conjunto de objetos que tienen el mismo patrón estructural y de comportamiento. Cada objeto que pertenece a una clase es una **instancia** de ella. El conjunto de todas las instancias de una clase es la **extensión** de la clase. La **instanciación** es el proceso de generación o creación de las instancias de una clase. La definición de una clase normalmente contiene: su nombre, el nombre de sus superclases –clases de las que se hereda–, su interfaz, su estructura y su implementación. Las clases formarán una jerarquía denominada **jerarquía de clases**, donde las clases superiores a una clase particular se denominan **superclases** y a la clase particular se le llama **subclase**. La relación entre una superclase y una subclase se denomina ES-UN(A). La **metaclase** es la clase que define todas las clases del sistema. La figura 1 presenta un ejemplo de una jerarquía de clases en notación gráfica de TDSO. Cada rectángulo representa una clase de

objetos. Sus divisiones indican: primero, el nombre de la clase; a continuación, la estructura de las instancias de la clase; y por último, las operaciones o métodos que trabajan sobre las instancias de la misma. Esta representación es muy simplificada y está basada en OMT.

La **composición** o agregación permite definir clases de objetos compuestos, ya que permite colocar atributos en la clase que pertenecen a otra clase. Ella representa la relación ES-PARTE-DE entre una entidad compuesta y sus entidades componentes. La figura 2 muestra el diagrama TDSO de una clase de objetos compuestos.

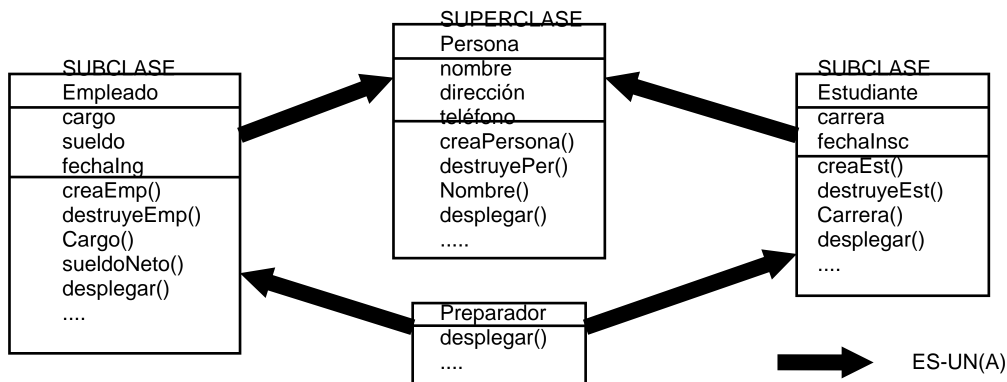


Figura 1. Jerarquía de clases.

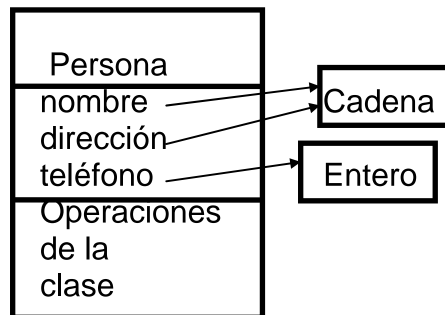


Figura 2. Composición.

La **generalización** es un proceso de abstracción en el cual un conjunto de clases existentes, que tienen atributos y operaciones comunes es referido por una clase genérica a un nivel mayor de abstracción. La generalización corresponde a la relación ES-UN(A) entre las clases. En la figura 1 pudo haber habido una generalización, si las clases Empleado y Estudiante fueron realizadas antes y una

vez que se tuvieron, se decidió crear una nueva clase genérica denominada Persona, para contener los atributos y operaciones comunes a las clases ya implementadas, ya que tanto Estudiante como Empleado son Personas.

La **especialización** o particularización es el proceso inverso de la generalización, ya que una subclase se crea a partir de una clase ya existente. Este es el proceso que soportan la mayoría de los sistemas orientados por objetos. La figura 1 muestra un ejemplo de ello, si se crea la clase Preparador como subclase de Estudiante y de Empleado.

El **protocolo** o interfaz es el conjunto de operaciones o métodos declarados como posible de *invocar o activar* por los otros objetos. Los **mensajes** son la especificación de un objeto junto con la invocación de uno de sus métodos. Por lo general, el pase de mensajes tiene la forma

objetoDestino.nombreDelMétodo(listaDeArgumentos)

La **encapsulación** es la propiedad que permite que los objetos sean definidos en su estructura y su comportamiento, obligando al uso del pase de mensajes o de la invocación de sus métodos, si se quiere acceder al objeto.

La **herencia** es la propiedad que tienen las clases de heredar de sus superclases estructura y/o comportamiento. La herencia estructural es aquella donde se hereda la estructura y la herencia de comportamiento cuando se heredan los métodos. La herencia se puede tipificar como: simple o múltiple. **La herencia simple** se da cuando la subclase hereda solamente de una superclase, y es **múltiple** cuando hereda de varias superclases. La herencia múltiple presenta problemas de ambigüedad cuando hay atributos o métodos con el mismo nombre en varias de las superclases. La solución a dicho problema es la declaración explícita de cada uno de ellos junto con la superclase de la que se hereda.

En la figura 1 se puede ver un ejemplo de herencia simple, la subclase Estudiante, y un ejemplo de herencia múltiple, la subclase Preparador. Esta última debe explicitar la herencia de los atributos y métodos de Persona a través de una de sus dos superclases, para evitar ambigüedades en el pase de mensajes.

Polimorfismo significa que se puede usar el mismo nombre para la definición de un método en varias clases sin importar la relación entre las mismas. Ejemplo: el operador + para tipos numéricos y para tipos caracter. Para que el polimorfismo sea posible se utilizan los conceptos de: reescritura y encadenamiento tardío o dinámico. La **reescritura o sobrecarga** es la que permite nombrar código diferente con el mismo nombre para más de una clase de objetos. El **encadenamiento tardío**

permite seleccionar el código adecuado al objeto definido en la invocación del método. Un ejemplo de ello se puede observar en la figura 1 donde el método *desplegar()* se define para cada una de las clases de dicha figura.

II.4. Ventajas y desventajas de la orientación por objetos:

Las ventajas de la programación orientada por los objetos son las siguientes:

- La abstracción de datos y el ocultamiento de la información aumentan la confiabilidad y ayudan a separar la especificación de la implantación.
- El encadenamiento dinámico incrementa la flexibilidad.
- La herencia junto con el encadenamiento tardío permite la reusabilidad aumentando así la productividad.

Entre sus desventajas se encuentran:

- El costo de tiempo de ejecución del encadenamiento tardío puede llegar a ser importante dependiendo de la aplicación.
- La implantación con lenguajes orientados por objetos es más compleja que con los lenguajes convencionales.
- El programador debe leer con frecuencia extensas librerías de clases.

III. Notación gráfica de TDSO:

Las clases de objetos muestran los aspectos estáticos y dinámicos del sistema y ellas se expresan en los diagramas de la notación gráfica mostrada en la figura 3.

<nombreDeLaClase>
<nombreDelAtributoDeLaInstancia>[: <tipo> [= valorPorOmisión]] $<nombreDelAtributoDeLaClase>[: <tipo> [= valorPorOmisión]]
<nombreDeLaFunciónDeLaInstancia>[(<listaDeArgumentos>) [: <tipoDeRegreso>]] $<nombreDeLaFunciónDeLaClase>[(<listaDeArgumentos>) [: <tipoDeRegreso>]]
<nombreDelEventoDeLaInstancia>[(<listaDeFunciones>)] $<nombreDelEventoDeLaClase>[(<listaDeFunciones>)]

Figura 3. Notación gráfica de las clases de objetos en TDSO.

Los nombres de los atributos de la instancia son las variables instancia para cada ocurrencia de la clase que tienen definido un tipo o clase de valores que pueden contener y opcionalmente, un valor de inicio en el caso que dicho valor no se especifique. Los nombres de los atributos de la clase son aquellas propiedades de la clase que son compartidas por todas sus instancias, ellas también tienen un tipo de valor y un valor inicial por omisión. Los nombres de las funciones de las instancias son aquellas funciones –operaciones o procesos– que trabajan sobre las instancias de la clase. Ellas son acompañadas opcionalmente por su lista de argumentos, compuesta por el tipo del argumento y el nombre de la variable, todos ellos separados por coma. De igual manera se especifican las funciones de la clase, que son aquellas funciones que trabajan sobre los atributos de la clase.

Los nombres de los eventos de la instancia son las funciones que implementan los eventos que trabajan en base al estado de las instancias de la clase. Un **evento** es un estímulo de un objeto sobre otro, que sucede en un momento determinado en el tiempo y que no tiene duración, por lo cual es instantáneo en relación a la duración de las demás operaciones definidas para la clase. Los nombres de los eventos de la clase son aquellos eventos que funcionan con el estado de los atributos de la clase.

El diagrama de las clases se complementa con la notación gráfica de las jerarquías de las mismas. Las clases de objetos pueden formar una jerarquía de herencia que expresa las relaciones ES-UN(A) representadas con una flecha gruesa en el diagrama de la figura 4. Asimismo, la jerarquía de composición correspondiente a las relaciones ES-PARTE-DE o agregación, se representan mediante arcos y también se muestran en la figura 4.

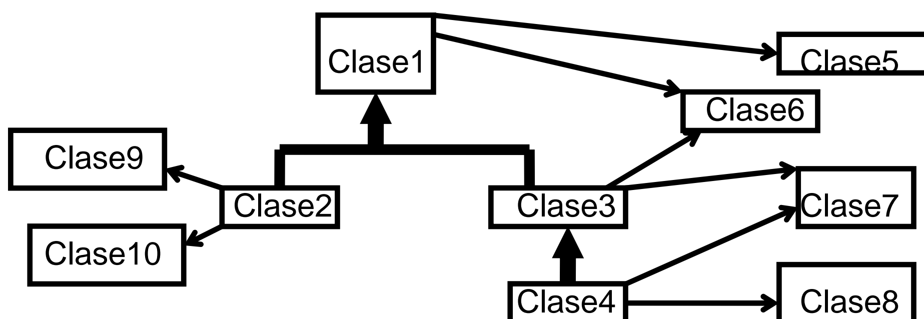


Figura 4. Notación gráfica de las jerarquías de herencia y de composición en TDSO.

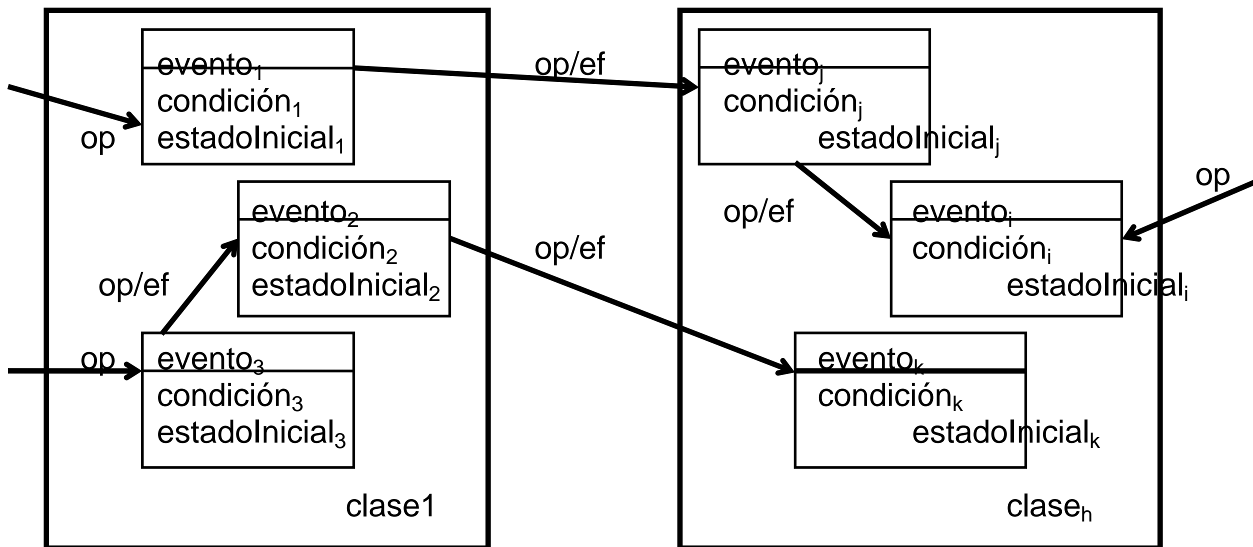
Este diagrama da una visión global de todas las clases del sistema. Los eventos asociados a cada clase forman parte del comportamiento de dicha clase y por lo tanto se heredan y encapsulan con la clase. La especificación detallada de las clases del sistema se presenta en la sección siguiente.

Para expresar la dinámica de los sistemas programados en TDSO, se utilizará la notación expresada en la figura 5, donde las clases se muestran como rectángulos con su nombre, que contienen rectángulos más pequeños con el nombre de sus eventos y sus condiciones de disparo. Las flechas indican las operaciones que disparan los eventos y los estados finales obtenidos al final de la operación.

Los eventos tienen asociada información sobre sus condiciones de disparo y las operaciones que ellos invocan, las cuales producen los cambios de estado en instancias de una o más clases. La descripción de los estados del sistema podrá ser expresada gráficamente mediante la especificación de las condiciones de disparo asociadas al evento particular, así como sus precondiciones, que muestran el estado inicial del o los objetos antes de presentarse el evento y que al cumplirse hacen que se accione o dispare el mismo, produciendo la invocación de las operaciones asociadas, y llevando los objetos a un estado final, expresado en las poscondiciones de las operaciones efectuadas.

Dos eventos pueden relacionarse o no, en el caso que uno sea continuación de otro o no. Dos eventos no relacionados se dicen que son **concurrentes**, ya que uno no tiene efecto sobre el otro. El tiempo en que ocurre un evento es un atributo implícito del mismo, por lo tanto la estructura de la clase debe soportar el último tiempo de ocurrencia de sus eventos.

Como ya se dijo anteriormente, una operación es una función de la clase que efectúa una acción atómica sobre uno o varios atributos de la clase. Cuando una función esté compuesta por otras funciones entonces su nombre cambia por el **proceso**, así un proceso es una función de alto nivel que está compuesta por subprocesos u operaciones. La descripción de los procesos del sistema programado se expresa con el diagrama ilustrado en la figura 6, que muestra como un proceso invoca otros procesos u operaciones indicando los caminos método-mensaje (M-M) [JoEr-94] o caminos de invocación de las funciones. Un proceso será activado por medio de una invocación al mismo o por medio de uno o más eventos que realicen la invocación ya mencionada.



op: operación ef: estado final

Figura 5. Notación gráfica de la dinámica de las clases en TDSO.

La notación gráfica para los procesos en TDSO mostrada en la figura 6 contiene cada clase como un rectángulo con el nombre de la clase y subrectángulos con los nombres de sus procesos y de las operaciones involucradas. Las flechas con líneas entrecortadas indican los mensajes y las líneas cuyo comienzo es un círculo pequeño indican los caminos M-M.

A continuación se presenta la técnica de desarrollo de sistemas de objetos aplicada a un caso particular.

IV. Técnica de desarrollo de sistemas de objetos:

Tomando del método deductivo sus principales conceptos, se tiene que las etapas de la resolución de problemas se expresan en dos partes: un **universo** de objetos dentro del cual se evoluciona y un **enunciado** del problema a resolver en ese universo.

Se comienza con un enunciado **Eo** en lenguaje natural, con el que se describe el problema en su primera impresión. A partir de él se profundiza en la comprensión del mismo buscando alternativas de solución, es decir, se sigue un proceso de refinamiento paso a paso expresado por los diferentes enunciados del problema,

hasta llegar a una colección de ellos, con la que se completa el enunciado del mismo y así, éste puede ser resuelto.

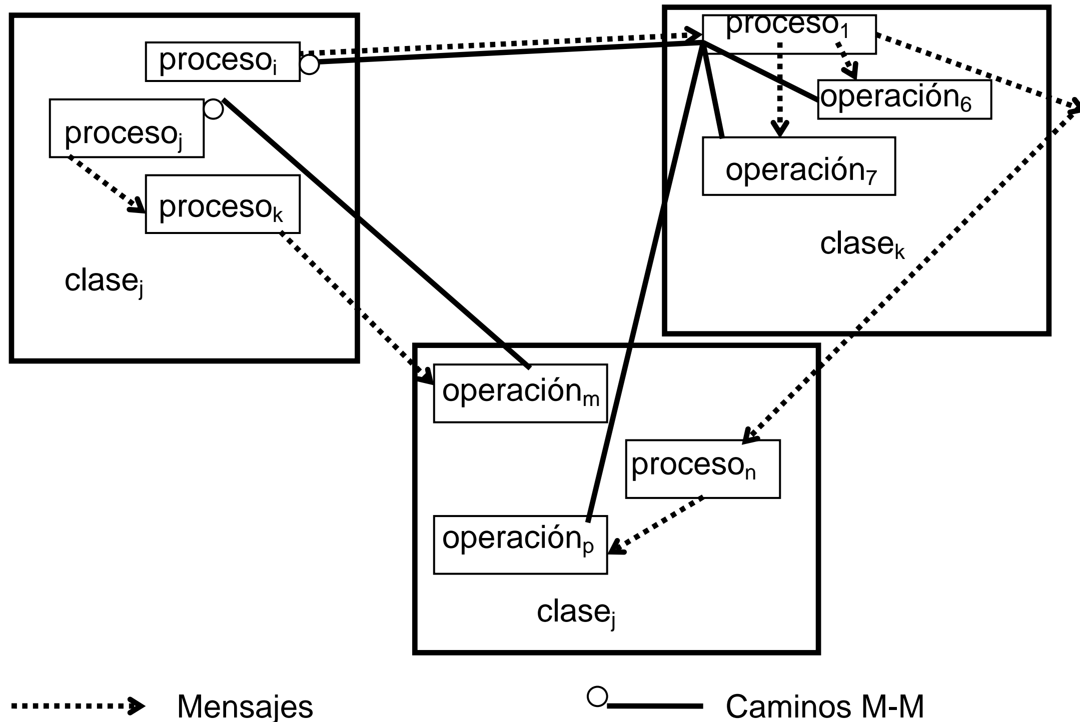


Figura 6. Notación gráfica para los procesos en TDSO.

En este proceso de refinamiento paso a paso, se identifican clases de objetos junto con sus atributos y operaciones, las comunicaciones entre las clases, sus casos de prueba y se aplica la herencia en aquellas clases donde sea conveniente su uso.

Para ejemplificar lo antes dicho y a la vez desarrollar TDSO, se muestra el enunciado cero del problema que será resuelto aquí.

Eo:

> Detectar si una cadena de caracteres dada pertenece o no al conjunto de las cadenas de la forma wCw^i, donde w son cadenas formadas en el alfabeto α={'a','b'} y w^i es la cadena inversa.

Se plantea el problema como un sistema de ecuaciones explicitas o definiciones formales, donde **n** de ellas pueden escribirse como: **X1,...,Xn = e1,...,en**, donde el identificador **Xi** está asociado a la expresión **ei**. Cuando los enunciados **Ei** son muy

complejos, ellos deben ser divididos y se obtiene un conjunto de ecuaciones por subproblema denominados **módulos** o rutinas. Una rutina **f** puede ser una función en el caso que se defina un único valor de **X1,...,Xp**. El módulo que no sirve para definir otros se llama **principal** o programa. Cada módulo se escribirá con un encabezado, un léxico o definiciones informales de los **Xi** y un sistema de ecuaciones o definiciones formales como se observa en la figura 7.

ENCABEZADO		
#	**ECUACIONES** (Definiciones formales)	**LEXICO** (Definiciones informales)

Figura 7. Especificaciones en TDSO.

En el encabezado se colocan las precondiciones y poscondiciones del módulo. Las precondiciones son condiciones que deben ser cubiertas en algunas variables para considerar válida la entrada al módulo (restricciones de entrada), ellas se expresan como aserciones. Similarmente, las poscondiciones son las aserciones de salida del módulo o efecto del mismo. La primera columna enumera las ecuaciones para la obtención del algoritmo del mismo, donde las definiciones se expresan de manera semiformal siguiendo la tabla 1. Las definiciones informales serán hechas colocando su descripción, por cada **Xi** y por cada operación especificada en las ecuaciones de la columna anterior.

Para la definición del universo del programa se utilizan los tipos abstractos de datos, en tanto que ellos son la base de las clases de objetos. Por ello, se deben definir allí los objetos utilizados en el programa. La selección de las clases de objetos es un paso que influye en la estructura y en el desempeño del programa. Para iniciar este paso es aconsejable partir de los documentos de requerimientos y el proceso se hace descendentemente o de arriba-hacia-abajo.

Para la definición del universo del programa se utilizará en TDSO la forma ilustrada en la figura 8, donde se observa el encabezado con la fecha de realización y el nombre del programa. A continuación la enumeración de las definiciones formales de los TAD o de las clases de objetos utilizadas y sus descripciones en la columna siguiente.

ASIGNACIONES	
	X1,...,Xn = E1,...,En
DEFINICIÓN CONDICIONAL	
SIMPLE	X1,...,Xn=Si (C) entonces E1,...,En sino E’1,...,E’n fsi
MÚLTIPLE	En caso que Xi=E1: X’1,...,X’n=E’1,...,E’n ... Ej: Y’1,...,Y’m=F’1,...,F’m en otro caso En: Z’1,...,Z’p=G’1,...,G’p
LAZOS	
REPITA PARA j DESDE Vi HASTA Vf CON INCREMENTO inc	(X1,...,Xn = E1,...,En) j = Vi, Vf, inc
REPITA MIENTRAS C SE CUMPLA	(C) [X1,...,Xn = E1,...,En]
REPITA HASTA QUE C SE CUMPLA	[X1,...,Xn = E1,...,En] (C)
ENTRADAS	
LECTURA DESDE EL TECLADO	X1,...,Xn = valor suministrado
LECTURA DESDE UN ARCHIVO Aj DE ACCESO SECUENCIAL	X1,...,Xn = registro siguiente en Aj
LECTURA DESDE UN ARCHIVO Aj DE ACCESO DIRECTO	X1,...,Xn = registro k de Aj
LECTURA DESDE UN ARCHIVO Aj DE ACCESO ALEATORIO O INDIZADO	X1,...,Xn = registro de Aj según CLAVE=valor
SALIDAS	
ESCRITURA EN PANTALLA	Despliegue X1,...,Xn
ESCRITURA EN PAPEL	Imprima X1,...,Xn
ESCRITURA EN ARCHIVO Aj DE ACCESO SECUENCIAL	Escriba X1,...,Xn en el registro siguiente de Aj
ESCRITURA EN ARCHIVO Aj DE ACCESO DIRECTO	Escriba X1,...,Xn en el registro k de Aj
ESCRITURA EN ARCHIVO Aj DE ACCESO ALEATORIO O INDIZADO	Escriba X1,...,Xn en Aj según CLAVE = valor
INVOCACIÓN DE OPERACIONES	
FUNCIONES	Xj = objeto.operación(parámetros)
PROCEDIMIENTOS	objeto.operación(parámetros)

Tabla 1. Definiciones semiformales para los algoritmos.

Para las clases o tipos generados y usados en el problema se realizan sus respectivos enunciados Ei, colocando las funciones y los axiomas y precondiciones, y a ello se le denomina especificaciones algebraicas, las cuales se presentan según la forma mostrada en la figura 9.

<table>
<tr><td colspan="3"><fecha>
Universo <nombreDelPrograma></td></tr>
<tr><td>#</td><td>Definiciones formales de las clases o tipos</td><td>Definiciones informales de las clases o tipos</td></tr>
</table>

Figura 8. Definición de todos los tipos o clases del programa.

En dicha figura 9 se puede observar en la parte del encabezado, la fecha de realización de la especificación, el número del TAD o de la clase, el cual se corresponde con el número utilizado en el universo, y el nombre del TAD o de la clase. A continuación se presenta como número 1, la especificación sintáctica; como 2, las declaraciones de las variables utilizadas en la especificación semántica; y por último, el número 3 es la especificación semántica en sí compuesta por los axiomas correspondientes. En la última columna, como ya se ha dicho, la descripción de cada una de las operaciones del TAD o clase.

La especificación algebraica debe ser consistente y completa. Ella es consistente si no altera las especificaciones de los tipos que utiliza. Es suficientemente completa, si para todo término sin variables de la forma **s(a1,...,ap)**, donde **s** es el selector, puede ser reescrito, gracias a los axiomas, en un término que no contiene ninguna de las operaciones del tipo especificado, es decir, hay suficientes axiomas para salir de ese tipo.

<table>
<tr><td colspan="3"><fecha>
Especificación (#) <nombreDelTipo o Clase></td></tr>
<tr><td>1
2
3</td><td>Especificación sintáctica:
Declaraciones:
Especificación semántica:</td><td>Definiciones informales de las operaciones especificadas</td></tr>
</table>

Figura 9. Especificaciones algebraicas.

Las operaciones definidas sobre la clase se catalogan como: constructores, observadores, transformadores, convertidores y destructores. Un **constructor** es una operación que crea un objeto de ese tipo o clase y eventualmente, puede iniciarlo

con algún valor por omisión, lo cual es altamente recomendable en el momento de la implementación. Un **observador** es una operación que no modifica el estado del objeto y eventualmente puede mostrar el contenido de alguno de sus atributos. Un **transformador** es aquella operación que modifica el estado del objeto y que eventualmente, puede realizar algún cálculo y regresar algún valor que indique si el cambio se efectuó exitosamente. Un **convertidor** es una función que crea un objeto nuevo a partir de otro objeto que pertenece a otra clase diferente de la clase del objeto nuevo. Y finalmente, un **destructor** es una operación que destruye el objeto. Asimismo, se especifican las funciones de la clase como: operación, proceso o evento.

Teniendo ya definido el enunciado inicial Eo, se pasa ahora a expresar el enunciado siguiente E1 con el que se refina el anterior:

E1:

> Construir un programa que detecte si una cadena de caracteres dada por el usuario, con longitud máxima de 255 caracteres, es una cadena que pertenece a la familia de las cadenas que tienen como forma general: $\mathbf{wCw^{i}}$. Donde **w** es una cadena formada con la concatenación de los símbolos del alfabeto α=**{'a','b'}**, $\mathbf{w^{i}}$ es la cadena inversa de **w** y **C** es un caracter que no pertenece a α.

Se puede ver que E1 está más detallado que Eo, por lo que se pasa ahora a la segunda fase de refinamiento: la construcción de E2. Para E2 se utilizará la forma definida en las figura 10, la cual ilustra como realizar los algoritmos generales y su documentación. En el encabezado se coloca la fecha, el número de la clase o TAD colocado en la especificación, el número de la función en esa especificación, el tipo de función, su tipo de acceso, el nombre de la función o módulo seguida de sus parámetros encerrados en paréntesis y separados por comas, sus precondiciones y sus poscondiciones, si ellas existen. El tipo de función se define como: ‘P’ para los procesos, ‘E’ para los eventos y sin tipo para las operaciones. El tipo de acceso se especifica como: ‘R’ para las funciones privadas, ‘O’ para las protegidas y sin tipo para las públicas. Luego se coloca el algoritmo con su documentación de variables a la derecha, y al final, los casos de prueba del algoritmo con su respectiva documentación a la derecha también. A continuación se muestra el algoritmo general utilizando las definiciones de la tabla 1.

La figura 11 muestra el programa denominado **familia** que realiza la función pedida en E1. La presentación de E2 se realiza mediante el refinamiento siguiente

donde se muestran los algoritmos generales que ilustran una forma de resolver el problema planteado, que ha sufrido un proceso de refinamiento sucesivo hasta la etapa actual.

<table>
<tr><td colspan="3"><fecha>
(#Clase.#Función)(tipoDeFunción.tipoDeAcceso)
nombreDeLaFunción(Tipo: parámetro,...)[:tipoDeResultado]
{pre: <precondiciones>} {pos: <poscondiciones> }</td></tr>
<tr><td>#</td><td>Algoritmo</td><td>Documentación</td></tr>
<tr><td>#</td><td>Casos de prueba</td><td>Documentación de las pruebas</td></tr>
</table>

Figura 10. Especificación de los módulos.

E2:

<table>
<tr><td colspan="3">15/3/95
(0)Programa familia
{pos: Detecta si una cadena dada pertenece o no a la familia wCw^i }</td></tr>
<tr><td>1
3

2
4</td><td>cad=valor suministrado
per=Si (Lon(cad) es impar y cad contiene
una ‘C’ y cad ≠ Λ) entonces
TratarCadena(cad)
fsi
per=Falso
Si (per o cad = Λ) entonces
Despliegue “Cadena válida”
sino
Despliegue “Cadena inválida”
fsi</td><td>-cad:Cadena: Cadena leida.
-per:Lógico: Es verdadero si cad pertenece a la familia deseada.
-Lon(): Operación del tipo Cadena.
-TratarCadena(): Función del programa familia que efectua el tratamiento de cad, devolviendo verdadero si cad pertenece a la familia.</td></tr>
<tr><td>1
2
3
4
5
6</td><td>cad=‘aaCaa’ ⇒ Cadena válida
cad=‘abbCba’ ⇒ Cadena inválida
cad=‘a’ ⇒ Cadena inválida
cad=‘’ ⇒ Cadena válida
cad=‘asCsa’ ⇒ Cadena inválida
cad=‘aaCaabb’ ⇒ Cadena inválida</td><td>Cadena válida.
Cadena inválida en longitud.
Cadena inválida pues no tiene C.
Cadena válida, por ser la nula.
Cadena inválida por el alfabeto.
Cadena inválida en secuencia.</td></tr>
</table>

Figura 11. Enunciado dos para el programa familia.

<table>
<tr><td colspan="3">27/3/95
(0.1)(P)TratarCadena(Cadena cad):Lógico
{pre: cad≠Λ} {pos: Regresa Verdadero si cad ∈ familia wCwⁱ }</td></tr>
<tr><td>1
2
4

3</td><td>w=Sub(cad,1,Ind(cad,’C’))
wi=Sub(cad,Ind(cad,’C’)+1)
TratarCadena=Si (Lon(w)=Lon(wi)) enton.
Verificar cada caracter
de w y wi en (‘a’,’b’)
Compare w_j con wi_k , si son
iguales incremente j y de-
cremente k, sino regrese
fsi
TratarCadena=Falso</td><td>-w:Cadena: Cadena antes del caracter ‘C’.
-wi:Cadena: Cadena después del caracter ‘C’.
-j:Entero: Posición de un caracter en w.
-k:Entero: Posición de un caracter en wi.
-Sub(), Ind(), Lon(): Operaciones del tipo Cadena.</td></tr>
<tr><td>1
2
3</td><td>cad=‘aaCaa’ ⇒ TratarCadena=Verdadero
cad=‘asCsa’ ⇒ TratarCadena=Falso
cad=‘aaCaabb’ ⇒ TratarCadena=Falso</td><td>Cadena correcta.
2do. caracter fuera del alfabeto.
Se vacia la pila antes de terminar.</td></tr>
</table>

Continuación de la figura 11.

Note que el enunciado anterior aún debe ser refinado, pues dentro de programa familia quedan todavía muchas cosas por definirse completamente, por ejemplo: cómo se evalúa que cad contenga un caracter 'C' o que la longitud de cad sea impar. Asimismo, la función TratarCadena contiene una asignación que es un lazo, para detectar si los caracteres de w y wi son los mismos. Esta misma función puede resolverse con el manejo de una pila para detectar el orden de los símbolos de la subcadena w que deberá ser comparada con w^i, que es la misma subcadena w pero en orden inverso. Todo ello será resuelto completamente en el próximo enunciado (E3) donde, si se quiere, se presenta un algoritmo estructurado tan detallado que es equivalente a un código de programa. Llegar a tal extremo depende del diseñador, seguramente una persona con experiencia en la resolución de problemas, le basta con el enunciado dos para expresarlo en código fuente en algún lenguaje de programación. Para los que no sea tan obvio, seguramente preferirán el próximo enunciado de la figura 12.

E3:

15/3/95 (0)Programa familia {**pos:** Detecta si una cadena dada pertenece o no a la familia wCw^i }		
1 3 2 4	cad=valor suministrado per=Si (Mod(Lon(cad),2) $\neq$ 0 $\wedge$ cad $\neq \Lambda \wedge$ Ind(cad,‘C’) $\neq$ 0) entonces TratarCadena(cad) fsi per=Falso Si (per $\vee$ cad = Λ) entonces Despliegue “La cadena $\in$ a la familia” sino Despliegue “La cadena $\notin$ a la familia” fsi	-**cad**:Cadena: Cadena leida. -**Lon(), Ind()**: Operaciones del tipo Cadena. -**Mod()**: Función que calcula el resto de una división. -**per**:Lógico: Es verdadero si cad pertenece a la familia deseada. -**TratarCadena()**: Función del programa familia que efectua el tratamiento de cad, devolviendo verdadero si cad pertenece a la familia.
1	cad=‘aaCaa’ $\Rightarrow$ Cadena válida	Cadena válida.
2	cad=‘abbCba’ $\Rightarrow$ Cadena inválida	Cadena inválida en longitud.
3	cad=‘a’ $\Rightarrow$ Cadena inválida	Cadena inválida pues no tiene C.
4	cad=‘’ $\Rightarrow$ Cadena válida	Cadena válida, por ser la nula.
5	cad=‘asCsa’ $\Rightarrow$ Cadena inválida	Cadena inválida por el alfabeto.
6	cad=‘aaCaabb’ $\Rightarrow$ Cadena inválida	Cadena inválida en secuencia.

Figura 12. Enunciado tres.

<table>
<tr><td colspan="3">27/3/95
(0.1)(P)TratarCadena(Cadena cad):Lógico
{pre: cad≠Λ} {pos: Regresa Verdadero si cad ∈ familia wCwⁱ }</td></tr>
<tr><td>4

3

5

2
1
6</td><td>[Sub(cad,j,1)≠‘C’ ∧ Sub(cad,j,1)∈{‘a’,’b’}]
(p.meterElePila(Sub(cad,j,1)
j=j+1)
j=1

[Sub(cad,j,1) = car ∧ j ≤ Lon(cad) ∧
¬p.vacíaPila()]
(j=j+1
car=p.conTopePila()
p.sacarElePila())
TratarCadena=Falso
creaPila p
TratarCadena=Si (j > Lon(cad) ∧
p.vacíaPila()) entonces
Verdadero
fsi</td><td>-car:Caracter: Caracter que actualmente está en tratamiento.

-j:Cardinal: Índice para el barrido de la cadena cad.
-creaPila(), meterElePila(), conTopePila(), sacarElePila(), vacíaPila(): Operaciones del tipo Pila.

-Sub(), Lon(): Operaciones del tipo Cadena.
-p: Pila: Pila donde se almacenarán los caracteres de la cadena antes de encontar la ‘C’.</td></tr>
<tr><td>1
2
3</td><td>cad=‘aaCaa’ ⇒ TratarCadena=Verdadero
cad=‘asCsa’ ⇒ TratarCadena=Falso
cad=‘aaCaabb’ ⇒ TratarCadena=Falso</td><td>Cadena correcta.
2do. caracter fuera del alfabeto.
Se vacia la pila antes de terminar.</td></tr>
</table>

Figura 12 (continuación). Enunciado tres.

Este enunciado final debe ser completado con la especificación del universo de los tipos de datos necesarios para el programa familia, el mismo se presenta a continuación en la figura 13.

<table>
<tr><td colspan="3">27/3/95
Universo familia</td></tr>
<tr><td>2

1
3
5
6
4</td><td>NodoPila: Estructura
el: Caracter
sig: ApNodoPila
fin
ApNodoPila: Apuntador de nodoPila
Cardinal: Entero+
Cadena
Entero
Pila[el: tipoEle]</td><td>-Cardinal: tipo: Subconjunto de los números enteros positivos, incluyendo el cero. (tipo básico)
-Pila[el: tipoEle]: tipo: Pila
-Cadena: tipo: Cadena de caracteres. (tipo básico)
-Entero: Tipo básico.
-NodoPila: tipo: Tipo compuesto soportado por la mayoría de los lenguajes. (tipo básico)
-ApNodoPila: tipo: Dirección de un dato tipo NodoPila.</td></tr>
</table>

Figura 13. Universo del programa familia.

Para aquellos tipos o clases que aparecen en el universo y que no son tipos básicos, se necesita la especificación del mismo, junto con su implementación. Asumiendo que los tipos: Cardinal, Entero, Estructura, Apuntador y Cadena son tipos básicos, se presenta la especificación y la implementación del TAD pila, siguiendo la forma ya expresada para la especificación de los TAD y utilizando una nueva forma para la implementación de los mismos que se ilustra en la figura 14. Esta forma contiene el número (#) del tipo especificado en el universo para el tipo en cuestión, el tipo de clase (tipoDeClase), que puede ser ‘G’ para las clases genéricas, ‘A’ para las clases abstractas y ninguno para las clases que no son genéricas ni abstractas. Luego el nombre de la clase y debajo de todo lo anterior, el listado de las clases o tipos necesarios para la implementación de la clase especificada, separadas por coma.

27/3/95 Implementación TAD(#) (<tipoDeClase>)Clase:<nombreDeLaClase> Clases: <listaDeClasesYaDefinidas>		
1 2 3	Superclases: Estructura: Atributos de las instancias: Atributos de la clase: Funciones: sobre las instancias: sobre la clase:	Descripción de cada superclase, atributo y función, catalogadas en constructores, convertidores, observadores, transformadores, eventos, procesos y destructor. (Los tipos de los atributos y las superclases deben ser clases ya definidas e incluidas en la lista)

Figura 14. Implementación de una clase en TDSO.

El universo del programa debe, por lo tanto, ir acompañado de la figura 15 que muestra como el TAD pila se especifica y como el mismo será implementado.

27/3/95 Especificación (4) Pila[el: tipoEle]		
1	Especificación sintáctica: creaPila()→ Pila, meterElePila(Pila,tipoEle)→ Pila, sacarElePila(Pila)→ Pila, conTopePila(Pila)→ tipoEle, vacíaPila(Pila)→ Lógico, destruyePila(Pila)→ .	-**creaPila()**: Crea una pila vacía. -**meterElePila()**: Ingresa un nuevo elemento a la pila por el tope de la misma. -**sacarElePila()**: Elimina el elemento que está actualmente en el tope de la pila. Si la pila está vacía no hace ninguna eliminación.
2	Declaraciones: p: Pila e: tipoEle	-**vacíaPila()**: Regresa Verdadero si la pila está vacía. -**destruyePila()**: Destruye la pila

3	Especificación semántica: vacíaPila(creaPila())=Verdadero vacíaPila(meterElePila(creaPila(),e))=Falso conTopePila(creaPila())={TipoEleNoDef} sacarElePila(creaPila())=creaPila()	-**conTopePila()**: Devuelve el elemento que se encuentra actualmente en el tope de la pila. Si la pila está vacía devuelve un valor especial que lo indica.

Especificación de la clase Pila en TDSO.

27/3/95 Implementación TAD(4) Clase: Pila Clases: Caracter, Lógico, NodoPila		
1 2 3	Superclases: Ninguna Estructura: tope: ApNodoPila Operaciones: creaPila() meterElePila(e: Caracter) sacarElePila() conTopePila():Caracter vacíaPila():Lógico destruyePila()	-**tope**: Apuntador al nodo que está en el tope de la pila. -**creaPila()**: Constructor e iniciador de la pila. -**meterElePila()**: Transformador que inserta un nuevo elemento en la pila. -**sacarElePila()**: Transformador que elimina el elemento en el tope de la pila, si él existe. -**conTopePila()**: Observador que regresa una copia del elemento en el tope de la pila. -**vacíaPila()**: Observador que regresa Verdadero si la pila está vacía y Falso en caso contrario. -**destruyePila()**: Destructor.

Figura 15. Implementación de la clase Pila en TDSO.

Nótese que en la especificación algebraica del constructor de tipos Pila[el], la referencia a la pila se hace con **p** dentro de los argumentos de la función, mientras que en la figura 12, la invocación de las operaciones de una clase se hace colocando la variable que referencia al objeto antes del nombre de la operación y utilizando el operador punto.

A continuación se presenta, a manera de ejemplo, la implementación de cada una de las operaciones del constructor de tipos o clase Pila[el] bajo tal método, véase la

figura 16. Cada una de estas operaciones contiene sus casos de prueba, en el caso de que ello sea necesario, ya que algunas de ellas son tan simples que es innecesario colocarlo.

27/3/95 (4.1)creaPila() {**pos:** Crea la pila vacía. tope = Nulo}		
	tope = Nulo	-**tope**: Definido en TratarCadena.
27/3/95 (4.2)meterElePila(e: Caracter) {**pre:** e $\neq$ '255'} {**pos:** Inserta e en el tope de la pila.}		
1	Si (e $\neq$ '255') entonces p.el = e p.sig = tope p = dirección de un NodoPila disponible tope = p sino Despliegue "Elemento no válido" fsi	-**e**: Caracter: Elemento a ser insertado. Debe ser diferente de '255' = tipoEleNoDef. -**p**: ApNodoPila: Variable auxiliar para referenciar el nuevo nodo antes de insertarlo.
1	e='255' $\Rightarrow$ Elemento no válido	Despliega "Elemento no válido"
2	e='a' $\Rightarrow$ Regresa la pila con un nuevo e	Inserta e en el tope de la pila

27/3/95 (4.3)sacarElePila() {**pre:** $\neg$vacíaPila()} {**pos:** Elimina el elemento en el tope de la pila }		
1	Si (tope=Nulo) entonces Despliegue "Pila vacía" sino r = tope tope = tope.sig regrese r a los NodoPila disponibles fsi	-**r**: ApNodoPila: Variable auxiliar que contiene la dirección del nodo que será eliminado de la pila, para luego regresarlo a los NodoPila disponibles. -**vacíaPila()**: Definida en el tipo Pila.
1	p.vacíaPila()=Verdadero $\wedge$ p.sacarElePila() $\Rightarrow$ Pila vacía	Despliega "Pila vacía"
2	p.vacíaPila() = Falso $\wedge$ p.sacarElePila()$\Rightarrow$ Regresa la pila sin el elemento del tope.	Elimina el elemento en el tope.

Figura 16. Implementación de la clase Pila.

27/3/95 (4.4)conTopePila():Caracter {**pos:** Extrae el elemento en el tope de la pila, si existe }		
1	conTopePila = Si (tope = Nulo) entonces ‘255’ sino tope.el fsi	-**vacíaPila()**: Definida en el tipo Pila. -**’255’**: Caracter: Es el caracter correspondiente al tipoEleNoDef.
1	p.vacíapila() = Verdadero $\wedge$ p.conTopePila()$\Rightarrow$ Regresa ‘255’	Regresa ‘255’
2	p.vacíapila() = Falso $\wedge$ p.conTopePila()$\Rightarrow$ Regresa el elemento solicitado.	Regresa el elemento en el tope.

27/3/95 (4.5)vacíaPila():Lógico {**pos:** Regresa Verdadero si la pila está vacía}		
1	vacíaPila = (tope = Nulo)	

27/3/95 (4.6)destruyePila() {**pos:** Regresa todos los nodos de la pila, dejándola vacía }		
1	[$\neg$ vacíaPila()] (sacarElePila())	

Figura 16 (continuación). Implementación del tipo Pila utilizando el método enlazado simple.

Es de suma importancia a la hora de implementar un tipo abstracto de dato, hacerlo utilizando las facilidades y principios de la programación orientada por objetos. Ciertamente, a la implementación presentada en la figura 16 deberían hacérsele ciertos cambios para que dicha implementación pueda ser usada con cualquier tipo de elemento, es decir una pila genérica, ¿Qué cambios puede usted sugerir?,¿Puede usted probarlos?.

En relación a la prueba de programas orientados por objetos, se describen tres niveles de prueba, de los presentados de P. Jorgensen y C. Erickson en [JoEr-94], ellos son: Pruebas de cada operación, pruebas del estado final de cada mensaje

y pruebas de las funciones atómicas del sistema, es decir las pruebas de los procesos y de los eventos.

1. Pruebas de cada operación, las cuales deben ser básicamente aquellas especificadas en G. J. Myers [Myer-83], pero cuyos casos de prueba, específicos para la operación que se está elaborando, serán colocados en la misma forma de la definición de la operación, especificando en el centro las descripciones formales de cada caso de prueba, bajo la forma:

   entradas(X1=c1,...,Xn=cn) ⇒ salida(Xi=ci,...,Xp=cp)

   en el lado izquierdo la enumeración deseada para la realización de tales pruebas y en el lado derecho la descripción informal de cada caso de prueba. La ilustración de este punto se pudo observar en las figuras anteriores.
2. Pruebas del estado final de cada mensaje, para lo cual se debe utiliza el gráfico de los procesos. En dicho gráfico, véase la figura 17, se muestran los caminos M-M, que son una secuencia de ejecuciones de operaciones encadenadas por los mensajes. Un camino método-mensaje se inicia en un proceso y termina en una operación, denominándose a este punto, el estado final del mensaje. Este tipo de prueba se especifica en la misma forma y se mostraron en un las figuras anteriores.
3. Pruebas de las funciones atómicas del sistema o programa. Una función atómica del sistema se inicia en un puerto de entrada seguido por un conjunto de caminos método-mensaje y terminando en un puerto de salida de la función mencionada. En nuestro caso son las pruebas de los procesos mencionadas anteriormente, incluyendo además los eventos. En este caso particular la figura 12, ilustra las pruebas para el programa familia.

Cuando el programa involucra varias clases o tipos y existe herencia entre algunas de ellas, según J. McGregor y T. Korson en [McKo-94], se debe tener en cuenta adicionalmente que:

- las precondiciones de cada método de una subclase **no** deben ser más fuertes que las de sus superclases.
- las poscondiciones de cada método de una subclase **no** deben ser más débiles que las de sus superclases.
- la invariante de una subclase es un superconjunto de las invariantes de sus superclases.

Por último es interesante mencionar la existencia de herramientas automatizadas para probar un modelo de objetos, en específico para OMT reseñada por R. Poston

en [Post-94]. Dicha herramienta permite la generación automática de los programas de prueba para las clases definidas en dicha técnica, al momento de la realización del diseño del sistema con la misma herramienta.

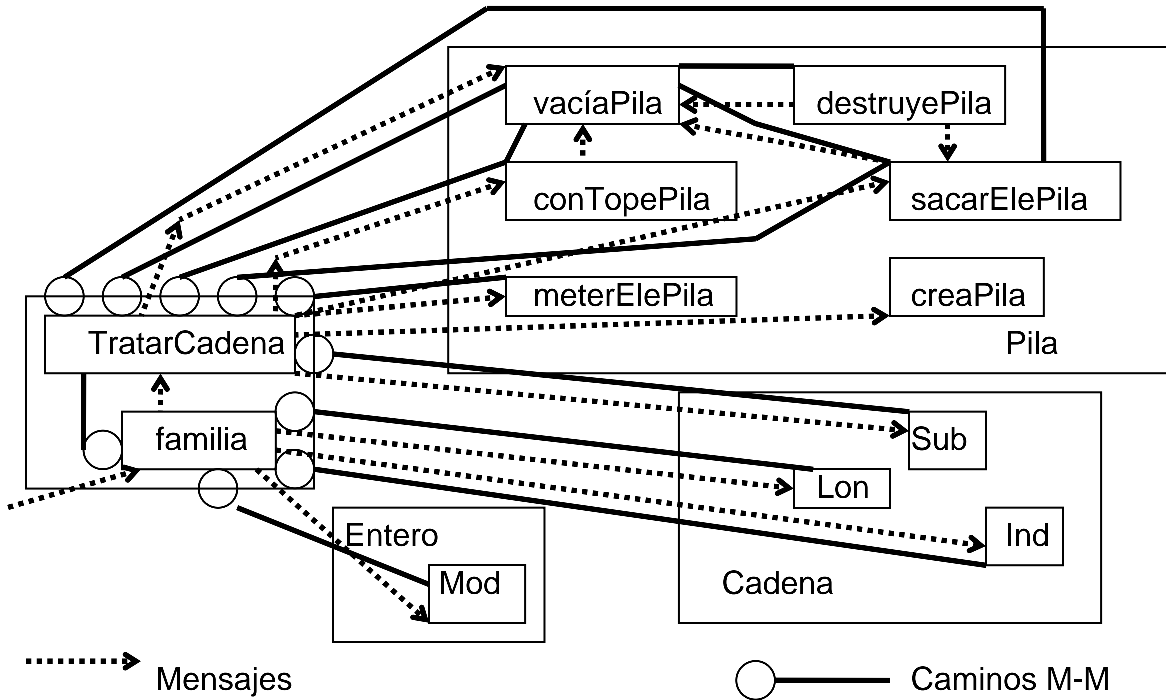


Figura 17.
Gráfico de los procesos del programa familia.

Conclusiones:

Con este escrito se ha pretendido demostrar la versatilidad y facilidad de la técnica de desarrollo de sistemas de objetos basada en el método deductivo MEDEE modificado y adaptado, y en la técnica de modelado de objetos OMT con ciertas características adicionales.
Las ventajas más resaltantes que se pueden mencionar aquí son:

- Permite modelar las propiedades estáticas y dinámicas de los objetos involucrados en el sistema.
- Se sigue manteniendo la descomposición modular-jerárquica de los problemas y el refinamiento paso a paso.
- Se construye, a medida que se analiza el problema, la documentación y las pruebas del mismo, en forma clara, modular y jerárquica.

- Se incluyen los casos de prueba, no presentes en el MEDEE original.
- La especificación se hace con miras a la utilización de herramientas de programación orientadas por objetos, aunque no imposibilita el uso de las herramientas de programación que no lo sean.
- La secuencia de análisis, especificación, diseño, documentación y pruebas en varias etapas hace que dichos procesos se sustenten en bases más seguras, donde sea poco factible el olvido de algún detalle importante.
- Permite abordar soluciones tanto iterativas (la presentada aquí), como recursivas.
- Permite cierta libertad a la hora de incluir detalles olvidados en alguna función, así como la libertad de comenzar el desarrollo de los módulos por aquel aspecto que primero viene a la mente, para una vez finalizado, hacer la secuenciación de las ecuaciones o estructuras ya definidas.

Solo resta decir que falta en este trabajo, una definición exhaustiva de todas aquellas maneras de escribir las acciones a realizar para problemas de otros tipos, sin embargo no hay ninguna limitante en definirse uno mismo las acciones necesarias para un problema particular.

Bibliografía:

[Crai-86] J. Craig. *An introduction to data types*. Addison-Wesley. 1986.

[Dufo-88] J. -F. Dufourd. *Vers un cadre unique pour spécifier et construire des programmes. Une expérience avec la méthode déductive et les types abstraits algébriques*. T.S.I.-AFCET. Vol.7 Nº3. 1988.

[Gutt-77] J. V. Guttag. *Abstract data types and the development of data structures*. Comm. A.C.M. Vol.20. Nº6.1977.

[GuHM-78] J. V. Guttag, E. Horowitz y D. R. Musser. *Abstract data types and software validation*. Comm. A.C.M. Vol.21. Nº12.1978.

[Hoar-69] C. A. R. Hoare. *An axiomatic basis for computer programming*. Comm. A.C.M. Vol.12. Nº10.1969.

[Hoar-87] C. A. R. Hoare et al. *Laws of programming*. Comm. A.C.M. Vol.30. Nº8.1987.

[JoEr-94] P. C. Jorgensen y C. Erickson. *Object-oriented integration testing*. Comm. A.C.M. Vol.37. Nº9.1994.

[McKo-94] J. D. McGregor y T. D. Korson. *Integrated object-oriented testing and development processes*. Comm. A.C.M. Vol.37. Nº9.1994.

[Mont-82] J. A. Montilva. *Programación estructurada y estilo de programación*. Mimeografiado. Facultad de Ingeniería. Universidad de Los Andes. 1982.

[Mont-87] J. A. Montilva. *Diseño modular de programas*. Mimeografiado. Facultad de Ingeniería. Universidad de Los Andes. Nº10-87-139. 1987.

[Myer-83] G. J. Myers. *El arte de probar programas*. El Ateneo. 1983.

[Post-94] R. Poston. *Automated testing from object models*. Comm. A.C.M. Vol.37. Nº9.1994.

[RBPE-91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy y W. Lorensen. *Object-oriented modeling and design*. Prentice-Hall. 1991.

[Stay-76] J. Stay. *HIPO and integrated program design*. IBM Systems Journal. Vol.15. Nº2. 1976.

[StMC-74] W. Stevens, G. Myers y L. Constantine. *Structured design*. IBM Systems Journal. Vol.13. Nº2. 1974.

[Wirt-71] N. Wirth. *Program development by stepwise refinement*. Comm. A.C.M. Vol.14. Nº4.1971.