Artículo de Investigación Revista Ciencia e Ingeniería. Vol. 30, No. 3, pp. 193-200, agosto-noviembre, 2009. ISSN 1316-7081

TDSO: Una técnica para el diseño de software orientado por objetos

TDSO: An object-oriented software design technique

Besembel, Isabel[*] y Narciso, Flor
Departamento de Computación. Escuela de Ingeniería de Sistemas. Facultad de Ingeniería. ULA
Grupo de Investigación en Ingeniería de Datos y Conocimiento (GIDyC)
Mérida, 5101-Venezuela
[*]ibc@ula.ve

Recibido: 06-03-2008 Revisado: 03-11-2009

Resumen

El objetivo de este artículo es presentar la descripción y uso de la Técnica de Desarrollo de Sistemas de Objetos (TDSO), la cual permite el análisis, especificación, diseño y documentación de clases de objetos, constituyendo una herramienta apropiada para el diseño de software orientado por objetos. Está basada en el método deductivo (MEDEE) utilizado para analizar, especificar, documentar, diseñar y probar sistemas programados, y en la metodología de modelado de objetos OMT (Object-oriented Modeling Technique), que describe todo el proceso de modelado de clases de objetos.

Palabras clave: Diseño de software e Ingeniería de Software orientado por objetos, tipos de datos abstractos.

Abstract

This article presents the description and use of the Technique for the Development of Object-Oriented Systems (TDSO) that allows the analysis, specification, design, and documentation of class objects, constituting an appropriate tool for the design of object-oriented software. This technique is based on the deductive method (MEDEE) used to analyze, specify, document, design and prove programmed systems, and in the Object-Oriented Modeling Technique (OMT) that describes the whole modeling process of class objects.

Key words: Object-oriented software design, software engineering, object orientation, abstract data types.

1 Introducción

Existen diversas técnicas para expresar la solución programada de problemas conocidas como técnicas de diseño de programas. La técnica HIPO Hierarchy plus Input-Process-Output (Stay, 1976), los diagramas de flujo (Joyanes, 1987), el refinamiento paso a paso (Wirth, 1971), la programación estructurada y modular método descendente – Top-Down y método ascendente – Bottom-Up (Dahl, 1972; Joyanes, 1987; Stevens, Myers y Constantine, 1974), los algoritmos estructurados (Montilva, 1990), el método deductivo MEDEE (Dufourd, 1988) y la programación orientada por objetos (Joyanes, 1999), entre otras, se pueden utilizar de manera independiente o en conjunto para diseñar y documentar sistemas programados.

Con la aparición de la orientación por objetos surge la metodología OMT Object-Oriented Modeling Technique (Rumbaugh, 1991), la cual utiliza el modelo de objetos para describir la estructura estática de los objetos de un sistema (identidad, atributos, operaciones y relaciones entre objetos), y 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) (Besembel, 2003) está basada en el método deductivo MEDEE y en la metodología OMT en su versión de 1998 (Rumbaugh, Jacobson y Booch, 1998) extendida con el lenguaje de modelado unificado (UML) (Booch, Jacobson y Rumbaugh, 1999; Rumbaugh, Jacobson y Booch, 2004). De OMT contiene todas las fases, incluyendo además las fases de especificación formal (Guttag y Horning, 1978; Guttag, 1977; Hoare, 1987; Hoare, 1969). De MEDEE se consideran algunos de los diagramas que fueron transformados y adaptados para TDSO, los cuales soportan

todos los constructos de la orientación por objetos. Además, TDSO incluye una guía para el desarrollo de las pruebas de los sistemas programados, utilizando el paradigma de la programación orientada por objetos basada en (Jorgensen y Erickson, 1994; McGregor y Korson, 1994; Poston, 1994).

Tomando de MEDEE 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, obteniéndose como resultado el diseño de un sistema programado o sistema de software orientado por objetos, expresado en términos de la definición de cada una de sus clases y el diagrama de clases correspondiente, la definición del universo de clases y tipos de datos abstractos (TDAs), la especificación formal de cada clase y la especificación formal de los métodos u operaciones de cada clase.

Es importante destacar que TDSO ha sido ampliamente utilizada en el ámbito académico para la enseñanza de cursos de programación básica y avanzada en la Universidad de Los Andes (ULA) en Mérida-Venezuela.

TDSO también ha sido utilizada en varios proyectos de desarrollo de software orientado por objetos (Besembel et al., 2007; Besembel et al., 2006a; Besembel et al., 2006b) y basado en agentes de software (Aguilar et al., 2005a; Aguilar et al., 2005b; Ávila, 2008, Narciso et al., 2008a; Narciso et al., 2008b).

En (Besembel et al., 2007; Besembel et al., 2006a; Besembel et al., 2006b) se presenta el diseño de una aplicación de software realizado con TDSO, en el cual se utilizan más de 100 clases. Esta aplicación de software ha sido implementada por un grupo de programadores diferente al grupo de diseñadores del mismo y se encuentra actualmente en operación (Besembel et al., 2008).

Asimismo, TDSO ha sido utilizada y probada formando parte del método Watch (Montilva, 2004; Montilva, 2000; Besembel et al., 2008) y como componente de la fase de diseño de una metodología para el modelado de sistemas de ingeniería orientado a agentes (Aguilar et al., 2008).

En la sección 2 de este artículo se describe el proceso de definición del enunciado del problema, tomando un ejemplo de bajo nivel de complejidad, como lo es el determinar el mayor o el menor de dos número, y el máximo común divisor entre ambos números, a los efectos de lograr una compresión adecuada del uso de la técnica. La definición del universo de objetos requeridos para la solución algorítmica del problema, o solución programada para el problema ejemplo, se describe en la sección 3. Finalmente, en la sección 4 se presentan las conclusiones de este trabajo.

2 Definición del enunciado del problema

La definición del enunciado del problema se realiza a través de un proceso de refinamiento paso a paso (Wirth, 1971), cuyo resultado es una colección de enunciados E_0, E_1, …, E_n.

Se comienza con el enunciado E_0, que consiste en la descripción del problema en su primera impresión, escrito en lenguaje natural. A continuación se presenta un ejemplo del enunciado inicial de un problema del menor nivel de dificultad:

> **E_0**:
> *Dados dos números enteros a y b, hallar el mayor de los dos, el menor de los dos y el máximo común divisor entre ambos números.*

Se profundiza en la comprensión de E_0, buscando alternativas para una solución programada mediante un proceso de refinamiento paso a paso, expresado por los diferentes enunciados del problema generados en cada paso del refinamiento hasta obtener una colección de ellos (E_0, E_1, E_n), con la que se completa el enunciado que permite resolver el problema. Este proceso de refinamiento del problema siempre se realiza en la etapa de análisis del problema y es por ello que TDSO la soporta. En este proceso se van descubriendo las clases necesarias para soportar la solución al problema. Para el ejemplo anterior, se podría formular el siguiente enunciado E_1:

> **E_1:**
> *Construir un sistema programado tal que, para dos números enteros a y b dados por el usuario, donde $a \geq 0$ y $b \geq 0$, determine el mayor de ambos números, el menor de ambos números y calcule el máximo común divisor entre ambos números.*

A medida que se avanza en el refinamiento, se llega a un punto en el que un enunciado E_i se puede especificar en términos de sus definiciones formales o ecuaciones explícitas de la forma $X_1,..., X_n = e_1,..., e_n$, donde cada identificador X_j está asociado a la expresión e_j, y sus correspondientes definiciones informales o léxico, según la representación tabular que se muestra en la tabla 1, en donde a cada ecuación se le asigna un número secuencial comenzando desde el 1 y su correspondiente léxico o definición informal.

Si los enunciados E_i son muy complejos entonces deben dividirse obteniéndose un conjunto de ecuaciones por subproblema denominados módulos o rutinas. Un módulo puede ser una función en el caso que se defina un único valor de $X_1,...,X_p$. Es importante destacar que toda descomposición debe incluir al programa principal.

Tabla 1. Especificación formal de un enunciado en TDSO

ENCABEZADO		
1	ECUACIÓN 1	LÉXICO 1
2	ECUACIÓN 2	LÉXICO 2
…	…	…
n	ECUACIÓN n	LÉXICO n
1	CASO DE PRUEBA 1	LÉXICO 1
2	CASO DE PRUEBA 2	LÉXICO 2
…	…	…
m	CASO DE PRUEBA m	LÉXICO m

En sus definiciones formales o ecuaciones, TDSO incluye las diferentes estructuras consideradas en la programación estructurada y modular (ver tabla 2). Las definiciones informales o léxico corresponden a la descripción de cada X_i y de cada operación especificada en las ecuaciones de la columna anterior, lo que equivale a la documentación de la ecuación.

Tabla 2. Ecuaciones aportadas por TDSO

Tipo de ecuación	Ecuación
Asignación	$X_1,\ldots,X_n = E_1, \ldots, E_n$
Definición condicional simple	$X_1,\ldots,X_n$ = **Si** (C) $E_1,\ldots,E_n$
Definición condicional doble	$X_1,\ldots,X_n$ = **Si** (C) $E_1,\ldots,E_n$ **SiNo** $E'_1,\ldots,E'_n$ **FinSi**
Definición condicional múltiple	**En caso que** X_i = E1: $X'_1, \ldots, X'_n = E'_1, \ldots, E'_n$ ….. **en otro caso** E_j: $Y'_1,\ldots,Y'_m = F'_1, \ldots, F'_m$
Repita para **j** desde $\mathbf{V_i}$ hasta $\mathbf{V_f}$ con incremento **inc**	$(X_1, \ldots, X_n = E_1, \ldots, E_n)$ **j =** V_i, V_f, inc
Repita mientras **C** se cumpla	**(C)** $[X_1, \ldots, X_n = E_1, \ldots, E_n]$
Lectura desde el teclado	$X_1, \ldots, X_n$ = valor suministrado
Lectura desde un archivo A_j de acceso secuencial	$X_1, ..., X_n$ = registro siguiente de A_j
Lectura desde un archivo A_j de acceso directo	$X_1, ..., X_n$ = registro k de A_j
Lectura desde un archivo A_j de acceso indizado o aleatorio según una **clave**	$X_1, ..., X_n$ = registro de A_j según **clave**=valor
Escritura en pantalla	Despliegue $X_1, ..., X_n$
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 indizado o aleatorio según una clave	Escriba X1, ..., Xn en el registro de Aj según clave=valor
Funciones	Xj = objeto.operación(parámetros)
Procedimientos	objeto.operación(parámetros)

En la sección 3.4 de este artículo se detalla y ejemplifica el contenido y formato de las diferentes partes que conforman la tabla 1.

3 Definición del universo de objetos

Durante el proceso de refinamiento paso a paso, se identifican clases de objetos así como 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. La interacción entre las clases siempre está presente, pues es en dicha interacción que la solución se soporta. Si hay alguna clase sin interacción, lo más probable es que dicha clase no sea necesaria en la solución diseñada.

La definición del universo de objetos consiste en definir todos los objetos (clases y/o TDAs) que se requieren en la solución programada del problema. En TDSO, este proceso se lleva a cabo mediante la definición de cada clase, la elaboración del diagrama de clases que representa las relaciones entre clases, la definición del universo de clases y TDAs, la especificación formal de cada clase y la especificación formal de cada uno de los métodos de las clases.

3.1 Definición de clases

Basándose en UML, una clase se define en términos de sus atributos (privados, públicos y/o protegidos) y operaciones o métodos (privados, públicos y/o protegidos). En la figura 1 se muestra un ejemplo de la representación gráfica de la clase denominada *DosNumeros*.

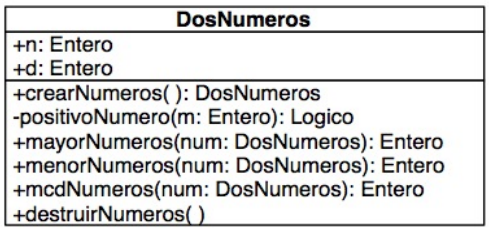

Fig. 1. Clase dos números

La selección de las clases es un paso que influye en la estructura y desempeño del programa. Por tanto, para iniciar este paso es imprescindible basarse en los documentos de requisitos obtenidos en la fase de análisis del problema.

Una vez definidas todas las clases e identificadas las relaciones entre ellas, se procede a la elaboración del diagrama de clases correspondiente y a la definición del universo de clases y TDAs, los cuales, generalmente, se corresponden con clases también.

3.2 Definición del universo de clases y tipos de datos abstractos

Para la definición del universo de objetos requeridos por la solución programada del problema se utilizan los TDAs. Un 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" (Guttag J y Horning J, 1978). Como ejemplos se pueden mencionar los TDAs Pila, Cola, Dipolo, Lista y Árbol Binario (Besembel y Rivero, 2000).

En la tabla 3 se muestra la representación tabular utili-

zada para la definición del universo de clases y TDAs de un sistema programado, ejemplificada con el sistema programado ManejoNumerosEnteros que permite comparar dos números enteros para determinar el mayor y el menor y calcular el máximo común divisor entre ambos.

Tabla 3. Definición del universo de clases y TDAs

01/31/07		Versión 1.0
Universo de clases y TDAs ManejoNumerosEnteros {Colección de clases y TDAs requerida para implantar el sistema programado ManejoNumerosEnteros}		
1	DosNumeros( )	*DosNumeros( )*: Clase que permite la creación de un objeto de la clase DosNumeros.
2	Entero	*Entero*: Tipo de datos entero.
3	Logico	*Lógico*: Tipo de datos lógico conformado por los valores cierto y falso.
4	Cadena	*Cadena*: TDA cadena de caracteres de longitud variable

Como puede observarse en esta tabla, el encabezado contiene la fecha de elaboración, la versión, el nombre del sistema programado y la descripción de la tabla. Es importante resaltar que los encabezados de las tablas en TDSO varían de acuerdo al tipo de tabla, manteniéndose siempre fijos la fecha de elaboración y la versión.

A continuación del encabezado se escribe la definición formal de cada TAD y/o clase de objetos utilizados en el sistema programado que se está diseñando y su correspondiente numeración y definición informal, correspondiendo en el ejemplo presentado en la tabla 3 a la clase DosNumeros y a los TDAs Entero, Logico y Cadena. En este ejemplo sencillo, solo habrá interacción con la clase Cadena, que es la implementación del TDA.

3.3 Especificación formal de una clase y/o TDA

Para las clases y/o TDAs generados y usados en la solución del problema se realizan sus respectivos enunciados E_i, colocando las funciones y los axiomas y precondiciones, a lo que se le denomina especificaciones algebraicas o especificación por axiomas algebraicos.

La especificación algebraica para las clases y TDAs se compone de:

- La especificación sintáctica donde se definen los nombres, dominios y rangos de las operaciones de la clase o TDA.
- La especificación semántica compuesta del conjunto de axiomas o ecuaciones, que dicen cómo opera cada una de las operaciones especificadas sobre las otras.

En TDSO, la especificación formal de una clase o TDA contempla la definición formal de sus atributos, la especificación sintáctica, las declaraciones requeridas para la especificación semántica y la especificación semántica. En la tabla 4 se muestra la representación tabular utilizada para la especificación formal de una clase o TDA, ejemplificada con la clase DosNumeros.

Como puede observarse en la especificación sintáctica de la clase DosNumeros, el enunciado E1 se ha dividido en cuatro módulos que representan las operaciones de la clase (positivoNumero, mayorNumeros, menorNumeros, mcdNumeros), añadiendo, además, las operaciones correspondientes a la creación y destrucción de objetos de la clase, tal y como lo establece la orientación por objetos.

Tabla 4. Especificación formal de la clase DosNumeros

01/31/07		**Versión 1.0**
1 DosNumeros( ) {Clase que permite la manipulación de dos números}		
	Especificación de atributos:	
1	n: Entero	*n*: Almacena el valor de un número entero.
2	d: Entero	*d*: Almacena el valor de un número entero.
	Especificación sintáctica:	
1	crearNumeros( ) → DosNumeros	*crearNumeros():* Crea un objeto de la clase DosNumeros.
2	positivoNumero(Entero) → Logico	*positivoNumero():* Devuelve cierto si el número es mayor o igual que cero y falso sino.
3	mayorNumeros(DosNumeros) → Entero	*mayorNumeros():* Devuelve el valor del número mayor.
4	menorNumeros(DosNumeros) → Entero	*menorNumeros():* Devuelve el valor del número menor.
5	mcdNumeros(DosNumeros) → Entero	*mcdNumeros():* Devuelve el mcd entre dos números.
6	destruirNumeros( )	*destuirNumeros():* Destruye un objeto de la clase DosNumeros.
	Declaraciones	
1	e: DosNumeros	*e*: Objeto de la clase DosNumeros
	Especificación semántica	
1	positivoNumero(mayorNumeros(e)) = cierto	Devuelve cierto si el valor del número mayor es mayor o igual que cero.
2	positivoNumero(menorNumeros(e)) = cierto	Devuelve cierto si el valor del número menor es mayor o igual que cero.
3	positivoNumero(mcdNumeros(e)) = cierto	Devuelve cierto si el valor del mcd entre dos números es mayor o igual que cero.
4	mayorNumeros(crearNumeros( )) = 0	Crea un objeto de la clase DosNumeros y devuelve cero como el valor del número mayor.
5	menorNumeros(crearNumeros( )) = 0	Crea un objeto de la clase DosNumeros y devuelve cero como el valor del número menor.
6	mcdNumeros(crearNumeros( )) = 0	Crea un objeto de la clase DosNumeros y devuelve cero como el valor del mcd.

Para la especificación formal de una clase o TDA, el encabezado contiene la fecha de elaboración de la tabla, la versión, el nombre de la clase o TDA y su descripción. Además, se antepone al nombre de la clase o TDA un número secuencial comenzando desde el 1, el cual permite asociar la especificación formal de cada método u operación definido en la especificación sintáctica a la clase o TDA correspondiente.

3.4 Especificación formal de los métodos u operaciones de una clase o TDA

Teniendo ya definido el enunciado E_1, tal y como se explicó en la sección 2, se pasa a la segunda fase de refinamiento paso a paso que consiste en la construcción de E_2, donde se muestran los algoritmos generales que ilustran una forma de resolver el problema planteado, el cual ha pasado por el proceso de refinamiento paso a paso hasta la etapa actual. Para representar en TDSO estos enunciados se utiliza la representación tabular que se muestra en la tabla 5.

Tabla 5. Especificación de una operación

#Clase o TDA. #Operación (tipo de operación, tipo de acceso) Nombre de la operación(Tipo parámetro, ...)[:tipo de resultado] {Descripción de la operación}	
Pre: {precondiciones}	Pos: {poscondiciones}
# ALGORITMO	LÉXICO
# CASO DE PRUEBA	LÉXICO

Como puede observarse en la tabla 5, el encabezado consta de:

- La fecha y la versión.
- El número de la clase o TDA que debe corresponder con el número asignado en la tabla de definición del universo de clases y TDAs, seguido del número de la operación que debe corresponder con el número asignado en la especificación sintáctica de tabla de especificación formal de la clase o TDA, el tipo de operación y el tipo de acceso.
- El nombre de la operación, el tipo y nombre de sus parámetros encerrados en paréntesis y separados por comas y el tipo del resultado de la operación, si existe.
- La descripción de la operación escrita entre llaves.
- Las precondiciones y poscondiciones, si existen.

Siguiendo los principios establecidos por la orientación por objetos, los tipos de operaciones definidas sobre una clase son:

- Constructor: 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.
- Observador: operación que no modifica el estado del objeto y eventualmente puede mostrar el contenido de alguno de sus atributos.
- Transformador: operación que modifica el estado del objeto y que eventualmente, puede realizar algún cálculo y regresar algún o varios valores que indique si el cambio se efectuó exitosamente.
- Convertidor: función que crea un objeto nuevo a partir de otro objeto que pertenece a otra clase diferente de la clase del objeto nuevo.
- Destructor: operación que destruye el objeto.
- El tipo de acceso se especifica como privado, protegido o público.

Las precondiciones son condiciones que deben ser cubiertas en algunas variables para considerar válida la entrada al módulo (restricciones de entrada), las cuales se expresan como aserciones. Las poscondiciones son las aserciones de salida del módulo o efecto del mismo.

A continuación del encabezado se escribe el algoritmo con su respectiva numeración y opcionalmente la documentación de variables, archivos, funciones y/o procedimientos utilizados en las columnas correspondientes. Para el diseño del algoritmo se utilizan las ecuaciones definidas en la tabla 2.

A esta tabla se le agrega una nueva fila para especificar los casos de prueba del método u operación, en caso de que sea necesario, ya que algunas de las operaciones son tan simples que es no es necesario el diseño de casos de prueba para las mismas. Cada caso de prueba se enumera, a continuación se escribe la descripción formal del caso de prueba bajo la forma *entradas* $\Rightarrow$ *salidas* siguiendo el formato $(X_1=c_1, ..., X_n=c_n) \Rightarrow (X_i=c_i,...,X_p=c_p)$ y su correspondiente descripción informal.

En las tablas 6, 7, 8, 9, 10 y 11 se muestran los enunciados Ep correspondientes a las soluciones algorítmicas que permiten crear un objeto de la clase DosNumeros (crearNumeros), determinar si un número es mayor o igual que cero (positivoNumero), determinar el mayor de dos números (mayorNumeros), determinar el menor de dos números (menorNumeros), calcular el máximo común divisor entre los números (mcdNumeros) y destruir un objeto de la clase DosNumeros (destruirNumeros). Dichas soluciones algorítmicas resultan de un proceso de refinamiento previamente realizado en el cual los algoritmos generales se van refinando hasta obtener la solución algorítmica al nivel de detalle deseado.

Tabla 6. Especificación formal de la operación crearNumeros

E_p:

31/01/07		Versión 1.0
1,1 (Constructor, Público) crearNumeros( ): DosNumeros **{Crear un objeto de la clase DosNumeros}**		
{pre: ∃ suficiente memoria}		{pos: Se crea un objeto de la clase DosNumeros }
1	crearNumeros = crearObjeto(DosNumeros)	*crearObjeto( )*: Crea un objeto de una clase determinada por el parámetro.
	crearNumeros.n = 0	
2	crearNumeros.d = 0	
3		
1	Llamar al constructor.	Se crea un objeto de la clase DosNumeros inicializado en los valores especificados.

Tabla 7. Especificación formal de la operación positivoNumero

E_p:

31/01/07	Versión 1.0
1,2 (Observador, Privado) positivoNumero(Entero numero): Logico **{Determinar si el número dado es mayor o igual que cero}**	
{pre: ∃ número}	{pos: cierto ∨ falso }
1 Si numero ≥ 0 devuelva(cierto) SiNo Devuelva(falso) FinSi	

Tabla 8. Especificación formal de la operación positivoNumero (continuación)

1	(numero = 2) => positivoNumero = cierto)	Número válido.
2	(numero = 0) => positivoNumero = cierto)	Número válido.
3	(numero = -3) => positivoNumero = falso)	Número inválido.

Tabla 9. Especificación formal de la operación mayorNumeros

E_p:

31/01/07	**Versión 1.0**
1,3 (Observador, Público) **mayorNumeros(DosNumeros num): Entero** **{Determinar el mayor de dos números dados}**	
{pre: ∃ objeto DosNumeros}	{pos: Número mayor ∨ NULO (-1) }
1 Si positivoNumero(num.n) ≥ 0 ∧ positivoNumero(num.d) ≥ 0 Si num.n ≥ num.d devuelva(num.n) SiNo devuelva(num.d) FinSi SiNo devuelva(-1) FinSi	*positivoNumero()*: Función que determina si un número es mayor o igual que cero.
1 (num.n = 2, num.d = 2) => mayorNumeros = 2)	Devuelve el mayor de los números, en este caso ambos números son iguales.
2 (num.n = 4, num.d = 2) => mayorNumeros = 4)	Devuelve el mayor de los números.
3 (num.n = 1, num.d = 10) => mayorNumeros = 10)	Devuelve el mayor de los números.
4 (num.n = -4, num.d = 2) => mayorNumeros = -1)	Operación inválida. Uno de los dos números es negativo.
5 (num.n = 4, num.d = -3) => mayorNumeros = -1)	Operación inválida. Uno de los dos números es negativo.
6 (num.n = -1, num.d = -2) => mayorNumeros = -1)	Operación inválida. Ambos números son negativos

Tabla0 10. Especificación formal de la operación menorNumeros

E_p:

31/01/07	Versión 1.0
1,4 (Observador, Público) menorNumeros(DosNumeros num): Entero **{Determinar el menor de dos números dados}**	
{pre: ∃ objeto DosNumeros}	{pos: Número menor ∨ NULO (-1) }
1 Si positivoNumero(num.n) ≥ 0 ∧ positivoNumero(num.d) ≥ 0 Si num.n ≤ num.d devuelva(num.n) SiNo devuelva(num.d) FinSi SiNo devuelva(-1) FinSi	*positivoNumero()*: Función que determina si un número es mayor o igual que cero.
1 (num.n = 2, num.d = 2) => menorNumeros = 2)	Devuelve el menor de los números, en este caso ambos números son iguales.
2 (num.n = 4, num.d = 2) => menorNumeros = 2)	Devuelve el menor de los números.
3 (num.n = 1, num.d = 10) => menorNumeros = 1)	Devuelve el menor de los números.
4 (num.n = -4, num.d = 2) => menorNumeros = -1)	Operación inválida. Uno de los dos números es negativo.

Tabla 11. Especificación formal de la operación menorNumeros (continuación)

5	(num.n = 4, num.d = -3) => menorNumeros = -1)	Operación inválida. Uno de los dos números es negativo.
6	(num.n = -1, num.d = -2) => menorNumeros = -1)	Operación inválida. Uno de los dos números es negativo. Operación inválida. Ambos números son negativos.

Tabla 12. Especificación formal de la operación mcdNumeros

E_p:

31/01/07	**Versión 1.0**
1,5 (Transformador, Público) mcdNumeros(DosNumeros num): Entero {Calcular el máximo común divisor de dos números}	
{pre: num.n ≥ 0 ∧ num.d ≥ 0}	{pos: máximo común divisor de n y d}
1 (num.d ≠ 0)[t, num.n, num.d = num.n mod num.d, num.d, t] 2 devuelva (num.n)	*mod()*: Función que regresa el resto de n entre d. *t*: Variable tipo entero utilizada para el intercambio de valores.
1 (num.n = 945, num.d = 651) ⇒ (mcdNumeros = 21)	El máximo común divisor de 945 y 651 es 21.

Tabla 13. Especificación formal de la operación destruirNumeros

E_p:

31/01/07	Versión 1.0
1,6 (Destructor, Público) destruirNumeros( ) {Libera el espacio de memoria ocupada por un objeto de la clase DosNumeros.}	
{pre: ∃ objeto DosNumeros }	{pos: }
1 Liberar (n, d)	
1 Llamar al destructor.	Se destruye un objeto de la clase DosNumeros.

Los algoritmos diseñados pueden ser tan detallados que equivalen a un código de programa. El nivel de detalle de los algoritmos dependerá del diseñador y de las personas encargadas de la implementación del sistema programado. Para una persona con experiencia en la resolución de problemas, le sería suficiente contar con enunciados menos refinados para expresarlos en código fuente escrito en algún lenguaje de programación. Sin embargo, para una persona no experta en la resolución de problemas, sería conveniente suministrarle algoritmos bien detallados a fin de que pueda realizar la codificación sin alterar la funcionalidad del sistema programado.

4 Conclusiones

La técnica para el desarrollo de sistemas de objetos aquí presentada permite modelar las propiedades estáticas (atributos y relaciones) y dinámicas (operaciones) de los objetos involucrados en un sistema programado, manteniendo la descomposición modular-jerárquica de los problemas y el refinamiento paso a paso. 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 la omisión de algún detalle importante.

Tal y como está concebida, TDSO se caracteriza por ser una técnica bastante flexible, permitiendo cierta libertad a la hora de incluir detalles obviados en alguna operación o método, así como la libertad de comenzar el desarrollo de los módulos sin ningún orden o jerarquía, para una vez finalizado, hacer la secuenciación de las ecuaciones o estructuras ya definidas.

Otra ventaja de TDSO reside en que la durante la solución del problema se construye, a medida que se analiza el problema, la documentación y las pruebas (no incluidas en MEDEE) del mismo, en forma clara, modular y jerárquica. Esto hace de TDSO una herramienta bastante apropiada en aquellos casos en los cuales el grupo de diseñadores del sistema programado es diferente al grupo encargado de su implementación, y, además, facilita el mantenimiento del sistema programado. Los diseñadores indican en TDSO las características del sistema de objetos deseado y los programadores implementan en el lenguaje de programación seleccionado, la solución diseñada en TDSO.

Es importante destacar que aun cuando la especificación utilizando TDSO se hace con miras a la utilización de herramientas de programación orientadas por objetos, esto no imposibilita el uso de aquellas que no lo sean. Así mismo, permite abordar soluciones tanto iterativas como recursivas.

Referencias

Aguilar J, Besembel I, Cerrada M, Hidrobo F y Narciso F, 2008, Una Metodología para el Modelado de Sistemas de Ingeniería Orientado a Agentes, Revista Iberoamericana de Inteligencia Artificial, Vol 12, No. 38, pp. 39-60.

Aguilar J, León L, Ríos A, Cerrada M, Hidrobo F, Narciso F, Hernández D y Besembel I, 2005a, Informe 5: Documento de Ingeniería de Implementación de los Niveles Superiores del Sistema de Gestión, Optimización y Control del Net-DAS, Informe Técnico, Fundacite-ULA, Mérida, Venezuela.

Aguilar J, León L, Ríos A, Cerrada M, Hidrobo F, Narciso F, Mousalli G, Hernández D y Besembel I, 2005b, Informe 3: Documento de Ingeniería de Implantación del Medio de Gestión de Servicios, Informe Técnico, Fundacite-ULA, Mérida, Venezuela.

Ávila M, Bastidas T, Bislick M, Narciso F, Páez D, Parra R, Ramírez MA, Rodríguez C, y Valero G, 2008, Conceptualización, análisis y diseño del agente de mantenimiento, Informe Técnico, Universidad de Los Andes, Escuela de Ingeniería de Sistemas, Departamento de Computación, Mérida, Venezuela.

Besembel I, Díaz G, Hernández D, Hidrobo F, Narciso F, Ramírez V y Rivero D, 2008, Informe Final (Octavo Informe), Informe Técnico, PDVSA-INTEVEP-ULA, Mérida, Venezuela.

Besembel I, Díaz G, Hernández D, Hidrobo F, Narciso F, Ramírez V y Rivero D, 2007, Diseño de Módulos de Prioridad Dos (Sexto Informe), Informe Técnico, PDVSA-INTEVEP-ULA, Mérida, Venezuela.

Besembel I, Díaz G, Hernández D, Hidrobo F, Narciso F, Ramírez V y Rivero D, 2006a, Diseño de Módulos de Prioridad Uno (Segundo Informe), Informe Técnico, PDVSA-INTEVEP-ULA, Mérida, Venezuela.

Besembel I, Díaz G, Hernández D, Hidrobo F, Narciso F, Ramírez V y Rivero D, 2006b, Diseño de Módulos de Prioridad Uno: Regiones de Oriente y Occidente (Cuarto Informe), Informe Técnico, PDVSA-INTEVEP-ULA, Mérida, Venezuela.

Besembel I, 2003, Técnica de Desarrollo de Sistemas de Objetos (TDSO), Guía Técnica, Grupo de Investigación en Ingeniería de Datos y Conocimiento, Universidad de Los Andes, Mérida, Venezuela.

Besembel I y Rivero D, 2000, Estructuras lineales de datos, Disponible en: http://sistemas.ing.ula.ve/ pr3/unidad_3/ tema6/pdf/estructuras_nolinealesdatos.pdf.

Booch G, Jacobson Y y Rumbaugh J, 1999, The Unified Modeling Language User Guide, Addison-Wesley.

Dahl O, 1972, Structured Programming, Academic Press.

Dufourd J, 1988, Vers un cadre unique pour spécifier et

construire des programmes. Une expérience avec la méthode déductive et les types abstraits algébriques. TSI. Technique et science informatiques, Vol. 7, No. 3, pp. 281-293.
Guttag J y Horning J, 1978, The Algebraic Specification of Abstract Data Types, Acta Informatica, Vol. 10, pp. 27-52.
Guttag J, 1977, Abstract Data Types and the Development of Data Structures. Communications of the ACM, Vol. 20, No. 6, pp. 396 - 404.
Hoare C, 1987, An Overview of Some Formal Methods for Program Design, IEEE Computer, Vol. 20, No. 9, pp. 85-91.
Hoare C, 1969, An axiomatic basis for computer programming. Communications of the ACM, Vol. 12, No. 10, pp. 576-580.
Jorgensen P y Erickson C, 1994, Object-Oriented Integration Testing, Communication of the ACM, Vol. 37, No. 9, pp. 30-38.
Joyanes L, 1999, Programación Orientada a Objetos, McGraw-Hill.
Joyanes L, 1987, Metodología de la Programación. Diagramas de Flujo, Algoritmos y Programación Estructurada. McGraw-Hill.
McGregor J y Korson T, 1994, Integrating Object-Oriented Testing and Development Processes, Communications of the ACM, Vol. 37, No. 9, pp. 59-77.
Montilva, J, 2004, Desarrollo de Aplicaciones Empresariales, El Método Watch, Informe Técnico, Grupo de Investigación en Ingeniería de Datos y Cnocimiento, Universidad de Los Andes, Mérida, Venezuela.
Montilva J, Hamzam K y Gharawi H, 2000, The watch model for developing business software in small and midsize organizations, Actas de la IV Multiconferencia en Sistemas, Cibernética e Informática – SCI'2000, Orlando, USA.
Montilva J, 1990, Desarrollo de Sistemas de Información, Editorial Universidad de Los Andes, Mérida, Venezuela.
Narciso F, Ríos-Bolívar A, Avila M y Páez D, 2008a, Diseño de un Sistema Multiagentes de Mantenimiento: Agente Manejador de Fallas, Aceptado para su publicación en las Actas del XIII Congreso Latinoamericano de Control Automático/VI Congreso Venezolano de Automatización y Control, Mérida, Venezuela.
Narciso F, Ríos-Bolívar A, Rodríguez C y Valero G, 2008b, Diseño de un Sistema Multiagentes de Mantenimiento: Agente Planificador, Aceptado para su publicación en las Actas del XIII Congreso Latinoamericano de Control Automático/VI Congreso Venezolano de Automatización y Control, Mérida, Venezuela.
Poston R, 1994, Automated Testing from Object Models, Communications of the ACM, Vol. 37, No. 9, pp. 48-58.
Rumbaugh J, Jacobson I y Booch G, 2004, Unified Modeling Language Reference Manual, Second Edition, Addison Wesley.
Rumbaugh J, Jacobson I y Booch G, 1998, The Unified Modeling Language Reference Manual, Addison-Wesley.
Rumbaugh J, 1991, Object Oriented Modeling and Design, Prentice Hall.
Stay J, 1976, HIPO and integrated program design, IBM Systems Journal, Vol. 15, No. 2, pp. 143-154.
Stevens W, Myers G. y Constantine L, 1974, Structured design, IBM Systems Journal, Vol. 13, No. 2, pp. 115-139.
Wirth N, 1971, Program Development by Stepwise Refinement, Communications of the ACM, Vol. 14, No. 4, pp. 221-227