UNIVERSIDAD Rafael Belloso Chacín

Revista Electrónica de Estudios Telemáticos

TÉLÉMATIQUE

NASPOO: UNA NOTACIÓN ALGORÍTMICA ESTÁNDAR PARA PROGRAMACIÓN ORIENTADA A OBJETOS

(NASPOO: an Algorithmic Standard Notation for Object-Oriented Programming)

Recibido: 09/04/2011 **Aceptado** 25/06/2011

Ottogalli Fernández, Kiara Alexandra
Universidad de Carabobo, Venezuela
kottogal@uc.edu.ve

Martínez Morales, Amadís Antonio
Universidad Simón Bolívar, Venezuela
aamartin@uc.edu.ve

León Guzmán, Luis Guillermo
Universidad de Carabobo, Venezuela
lleon@uc.edu.ve

RESUMEN

En ciencias de la computación existen diferentes enfoques para el desarrollo de software, los cuales han evolucionado para convertirse en diferentes paradigmas de programación, entre ellos se encuentra el paradigma de Programación Orientada a Objetos (POO). Este paradigma ha demostrado ser de gran importancia en la actualidad para el diseño y desarrollo de aplicaciones, haciéndose fundamental en el proceso de enseñanza de la ciencia de la computación. Al enseñar POO se desea que el estudiante maneje sus conceptos básicos, independientemente de los lenguajes de programación, razón por la cual se recomienda el uso de un lenguaje pseudoformal o pseudocódigo. Sin embargo, al no existir un lenguaje pseudoformal estándar, el estudiante se enfoca más en el aprendizaje de las diferentes notaciones para representar las estructuras algorítmicas que en las estructuras algorítmicas en sí. Esto además conlleva a errores de lectura e interpretación entre diferentes programadores que usen notaciones distintas para representar los algoritmos. Para solucionar este problema se presenta una propuesta de Notación Algorítmica Estándar para la enseñanza de la Programación Orientada a Objetos (NASPOO). NASPOO complementa la propuesta de Notación Algorítmica Estándar para la enseñanza de Programación Imperativa (NASPI) y ofrece estructuras que soportan las características esenciales del modelo orientado a objetos como lo son la abstracción, el encapsulamiento, la modularidad y la jerarquía.

Palabras claves: Algoritmos, Programación Orientada a Objetos, Pseudocódigo.

ABSTRACT

In computer science many approaches to develop software exist and have evolved to become different programming paradigms, including the Object-Oriented Programming paradigm (OOP). This paradigm has proved to be important today for the design and development of applications, making it fundamental to the process of teaching computer

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

science. When teaching OOP is desirable that the student knows the basic concepts, independently of the programming languages, reason why is recommended the use of a pseudoformal language or pseudocode. However, the absence of a standard pseudoformal language has made the student to focus more on learning different notations to represent the algorithmic structures than learning the structures themselves. Moreover, this leads reading and interpretation errors among different programmers that use different notations to represent algorithms. To solve this problem, a proposal of an algorithmic standard notation for teaching object-oriented programming (NASPOO) is presented. NASPOO complements the proposal of algorithmic standard notation for teaching imperative programming (NASPI), and offers structures that support the essential characteristics of the object-oriented model: abstraction, encapsulation, modularity and hierarchy.

Keywords: Algorithms, Object-Oriented Programming, Pseudocode.

INTRODUCCIÓN

En la ciencia de la computación existen diferentes enfoques para el desarrollo de software, los cuales permiten descomponer un sistema complejo en unidades más pequeñas, manejables para el entendimiento del programador.

Estos enfoques han evolucionado en diferentes paradigmas de programación, que permiten al programador ver un problema de diferentes maneras. Entre los paradigmas de programación más conocidos se encuentran: el paradigma imperativo, el paradigma lógico, el paradigma funcional y el paradigma orientado a objetos (OO) (Joyanes, 1996).

El enfoque OO toma como elemento básico al objeto, el cual es un ente real o abstracto con ciertas características y funcionalidades que lo definen. Mediante este enfoque, un sistema complejo puede ser descompuesto en una serie de entes autónomos que colaboran entre sí para lograr un propósito.

Dentro de las tendencias actuales de la ingeniería del software, el paradigma de Programación Orientada a Objetos (POO) se revela como el más adecuado para la elaboración del diseño y desarrollo de aplicaciones por diferentes razones:

Es apropiado para la realización de prototipos y simulaciones, permiten un alto grado de reutilización de código, soluciona limitaciones de los modelos tradicionales en el entorno de las bases de datos, el uso de interfaces de usuario gráficos y visuales, y el aumento de lenguajes de programación orientados a objetos, entre otras (Joyanes, 1996). Esto lo hace fundamental en el proceso de enseñanza de la ciencia de la computación.

Cuando se enseña la POO, se desea que el estudiante aprenda sus bases, independientemente del lenguaje en el cual se implementarán sus estructuras, por lo cual es recomendable el uso de un lenguaje pseudoformal o pseudocódigo.

El pseudocódigo puede variar de un programador a otro, lo cual pudiese llevar a diferencias sintácticas y semánticas que, posteriormente, pueden producir errores de lectura e interpretación (Martínez y Rosquete, 2009). Esto constituye un problema en la

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

enseñanza de la programación, ya que obliga al estudiante a aprender diferentes notaciones para representar las mismas estructuras, desviando su atención del aprendizaje de las estructuras en sí.

Este problema se presenta debido a la falta de un pseudocódigo estándar que permita representar todas las estructuras necesarias para el aprendizaje de la programación orientada a objetos.

Existen diferentes propuestas de pseudocódigo, pero muchas de ellas están basadas en algún lenguaje de programación, no cuentan con soporte para nuevas tecnologías de programación, no han tenido suficiente difusión (Martínez y Rosquete, 2009) o simplemente no abarcan el paradigma de POO.

Con miras a resolver este problema se creó una propuesta de Notación Algorítmica Estándar para Programación Imperativa llamada NASPI (Martínez y Rosquete, 2009) y en este trabajo se presenta una propuesta de Notación Algorítmica Estándar para la enseñanza del paradigma de Programación Orientada a Objetos (NASPOO), la cual complementa NASPI con la finalidad de mejorar el proceso de enseñanza de la programación orientada a objetos y la comunicación entre los participantes de proyectos de desarrollo de software.

Este artículo fue estructurado en diez secciones incluyendo la introducción. En la sección 2 se define NASPOO. En la sección 3, se define una clase y se muestra su declaración, se define la sintaxis para la accesibilidad y referencia a miembros de una clase, y la declaración de miembros estáticos y constantes. También se mencionan las clases primitivas soportadas por NASPOO.

En la sección 4 se introduce la sintaxis utilizada por NASPOO para los elementos estructurales de la clase: atributos y métodos. En la sección 5 se describe cómo son declarados, creados y referenciados los objetos de una clase. En la Sección 6 se desarrolla el concepto de polimorfismo y como es trabajado en NASPOO.

En la sección 7 se habla de la herencia de clases y se desarrolla el concepto de clases abstractas y métodos virtuales. En la sección 8 se muestra la estructura general de un algoritmo en NASPOO. En la sección 9 se da un breve ejemplo de uso de NASPOO. Finalmente en la sección 10 se presentan las conclusiones del trabajo.

NASPOO: LA NOTACIÓN ALGORÍTMICA

NASPOO (Notación Algorítmica Estándar para Programación Orientada a Objetos) es un lenguaje pseudoformal que define las estructuras necesarias para la enseñanza de la Programación Orientada a Objetos (POO) independientemente de los lenguajes de programación.

Esta propuesta nace de la necesidad de desligar la enseñanza de los conceptos fundamentales de la POO de las características específicas de los lenguajes de programación, permitiendo a los estudiantes enfocarse en el modelo de programación más que en la notación que se utiliza. Mediante esta notación se desea mejorar el

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

proceso de enseñanza/aprendizaje y además evitar problemas de comunicación entre los programadores, inherentes al uso de notaciones distintas en el diseño de los algoritmos.

NASPOO se fundamenta en NASPI (Martínez y Rosquete, 2009), UPSAM (Joyanes, 2008) y Lenguaje Pseudoformal (Coto, 2002), los cuales extiende para representar conceptos de la POO tanto básicos como avanzados, proporcionando así estructuras generales que soportan las características esenciales de un modelo orientado a objetos: abstracción, encapsulamiento, modularidad y jerarquía (Booch y otros, 2007).

Vale la pena destacar que esta propuesta de lenguaje pseudoformal se basa principalmente en el manejo de referencias, pero sin descartar la posibilidad de incorporar también a los apuntadores, ya que ambas herramientas son importantes para el desarrollo de un programador.

Como lenguaje pseudoformal NASPOO provee: (1) estructura de una clase, (2) declaración de los atributos de una clase, (3) declaración y definición de los métodos de una clase, (4) niveles de accesibilidad a los miembros de una clase: privado, protegido y público, (5) miembros constantes o protegidos, (6) declaración e inicialización de objetos, (7) referencia a los miembros de una clase, (8) referencia al objeto receptor de un mensaje, (9) polimorfismo (estático y dinámico), (10) herencia (simple y múltiple), (11) acceso a los métodos de una superclase, (12) clases abstractas y (13) métodos virtuales.

CLASE

Una clase es una descripción de la organización y las acciones compartidas por uno o más objetos similares (Armstrong, 1996). De acuerdo a esta definición, una clase está estructurada por la abstracción de las características y funcionalidades relevantes compartidas por un conjunto de objetos que representan parte de un sistema. Las características comunes a todos los objetos se conocen como atributos de la clase, y las funcionalidades se conocen como los métodos de la clase.

La estructura general de una clase es la siguiente:

```
clase Nombre_de_clase
    atributos:
            // Declaración de los atributos de la clase
    métodos:
            // Declaración de los métodos de la clase
fFclase
```

Donde Nombre_de_clase debe ser un identificador válido para la clase. Un identificador válido debe contar con las siguientes características:

- Debe ser una única palabra, es decir, no debe contener espacios.

- Debe estar compuesto únicamente por letras (mayúsculas o minúsculas), números, o los símbolos guión (‘-’) o piso (‘_’).

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

- No puede comenzar por un número o un símbolo.
- No puede coincidir con ninguna palabra reservada del lenguaje pseudoformal.

Nótese que además de las características previamente mencionadas, por convención, la primera letra del identificador de una clase se debe colocar en mayúsculas. En NASPOO todos los elementos son objetos derivados de una clase y todas las clases se derivan de una única clase base llamada **Objeto.**

CLASES PREDEFINIDAS

Se asume la existencia de una serie de clases predefinidas que corresponden a los tipos de datos definidos previamente en NASPI (Martínez y Rosquete, 2009). Estas clases predefinidas son:

- Primitivas o Elementales: **Ordinal**, **Entero**, **Real**, **Lógico**, **Símbolo**.
- Compuestas: **Cadena**, **Apuntador**, **Arreglo**, **Enumerado**, **Intervalo, Archivo_s** (archivo de acceso secuencial), **Archivo_d** (Archivo de acceso directo).

En la Tabla 1 se presenta un resumen de los operadores y métodos definidos en NASPOO sobre cada una de las clases. Las clases **Entero**, **Lógico** y **Símbolo** son subclases de la clase **Ordinal**, por lo tanto comparten sus operadores y métodos; se colocarán de forma explícita en la tabla. Todas las clases cuentan con el operador de asignación (←).

Tabla 1. Operadores y métodos de las clases predefinidas

Clase	Operadores y Métodos
Ordinal	ord() (valor ordinal), pred() (predecesor), suc() (sucesor),=, <, >, ≤, ≥, ≠
Entero	ord() (valor ordinal), pred() (predecesor), suc() (sucesor), =, <, >, ≤, ≥, ≠, - (menos unario), –, +, *, div (división entera), mod (resto de la división entera), ** (exponenciación)
Real	=, <, >, ≤, ≥, ≠, - (menos unario), –, +, *, /, ** (exponenciación)
Lógico	ord() (valor ordinal), pred() (predecesor), suc() (sucesor),=, <, >, ≤, ≥, ≠, ¬ (negación), ∧ (conjunción), ∨ (disyunción)
Símbolo	ord() (valor ordinal), pred() (predecesor), suc() (sucesor),=, <, >, ≤, ≥, ≠
Cadena	=, + (concatenación), long() (longitud), construir(), copiar(**Cadena:** c), destruir(), símbolo(**Entero:** posicion)
Apuntador	=, β (referenciación), ↑ (desreferenciación), construir(), destruir(), **a** Nombre_de_clase (crear apuntador a un objeto de la clase Nombre_de_Clase), copiar(**Apuntador:** ptr)
Arreglo	[**Intervalo:** i_1 [, i_2, …, i_n]] **de** Nombre_de_clase (crear arreglo) construir(**Objeto:** o_1 [, o_2, …, o_n]), copiar(**Arreglo:** a), destruir(), [**Entero:** índice] (referencia al elemento en índice)
Enumerado	construir(identificador$_1$, …, identificador$_n$), destruir(), ord() (valor ordinal), pred() (predecesor), suc() (sucesor), =, <, >, ≤, ≥, ≠
Intervalo	construir(**Ordinal:** ord_1, ord_2), destruir(), ord() (valor ordinal), pred() (predecesor), suc() (sucesor), =, <, >, ≤, ≥, ≠
Archivo_s	abrir(**Caracter:** modo; **Cadena:** nombre), leer(**Objeto** var), escribir(expresión), cerrar(), fda() (Fin de Archivo)
Archivo_d	abrir(**Caracter:** modo; **Cadena:** nombre, tipo), leer(**Entero:** pos; **Objeto:** var), escribir(**Entero:** pos, expresión), cerrar(), fda() (Fin de Archivo)

Fuente: elaboración propia.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

ACCESIBILIDAD A LOS MIEMBROS DE UNA CLASE

Los miembros de una clase son los atributos y los métodos, y su nivel de protección determina quién puede acceder a los mismos (Ceballos, 2003). NASPOO define tres modificadores de control de acceso para los miembros de una clase: **privado**, **protegido** y **público**.

Un miembro declarado como Privado es accesible solamente por los métodos de su propia clase, un miembro declarado como Protegido es accesible por los métodos de su propia clase y los de sus subclases, y un miembro de una clase que haya sido declarado como Público es accesible por los métodos de su propia clase y por los métodos de cualquier otra clase o subclase.

Los miembros de una clase se declaran luego de la identificación de la sección a la cual pertenecen (atributos o métodos), en grupos encabezados por su modificador de acceso seguido de dos puntos, como se muestra a continuación:

privado:
$miembro_1$
$miembro_2$
⋮
$miembro_n$
protegido:
$miembro_{n+1}$
$miembro_{n+2}$
⋮
$miembro_{n+m}$
público:
$miembro_{n+m+1}$
$miembro_{n+m+2}$
⋮
$miembro_{n+m+k}$

Donde $miembro_i$ ($1 \leq i \leq n+m+k$) es un atributo o un método de la clase.

En la declaración de una clase, el modificador de control de acceso privado puede ser omitido en caso de que no se declaren miembros privados. El mismo criterio se aplica para los modificadores protegido y público.

Opcionalmente, con el modificador de control de acceso protegido se puede extender el conjunto de clases que pueden acceder a ciertos atributos o métodos de la clase que se está declarando. Para ello, se debe agregar, antes de los dos puntos, una lista con los identificadores de las clases a las cuales se quiere dar acceso, de la siguiente forma:

protegido Nombre_de_clase$_1$, …, Nombre_de_clase$_n$**:**

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

De esta manera las clases cuyos identificadores se encuentran en la lista y sus subclases contarán con acceso protegido a los miembros declarados en el bloque encabezado por el modificador.

REFERENCIA A LOS MIEMBROS DE UNA CLASE

Para hacer referencia a los atributos y métodos de una clase se utiliza el operador de selección de miembros, punto (**.**).

Para hacer referencia a un atributo de la clase, la sintaxis es la siguiente:

objeto.atributo

Para hacer referencia a un método de la clase, la sintaxis es la siguiente:

objeto.metodo([A_1; A_2; …; A_n])

Donde:

- objeto es el identificador de un objeto.
- atributo es el identificador de un atributo de la clase a la cual pertenece el objeto.
- método es el identificador de un método de la clase a la cual pertenece el objeto.
- Cada A_i ($1 \leq i \leq n$) es un grupo de parámetros actuales, a_1, a_2, …, a_m, donde a_j ($1 \leq j \leq m$) es el identificador del parámetro actual correspondiente al j-ésimo parámetro formal del i-ésimo grupo.

MIEMBROS ESTÁTICOS

En una clase se pueden declarar miembros estáticos mediante el uso del modificador estático, el cual se coloca al principio de la declaración. Cuando se crea un objeto de una clase, éste guarda una copia de cada uno de sus atributos.

Un atributo declarado como estático es un atributo de la clase, el cual almacena información común a todos los objetos de la clase, pero no pertenece a ninguno, y existe aunque no existan objetos de la clase.

Si el valor de un atributo es común a todos los miembros de la clase, es conveniente declararlo como estático para evitar copias innecesarias en cada uno de los objetos de dicha clase.

Un método declarado como estático es un método de la clase. Este tipo de método no es ejecutado para un objeto en particular, y por lo tanto no puede acceder a los atributos no estáticos de un objeto.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

MIEMBROS CONSTANTES

Para declarar atributos y métodos como constantes se utiliza el modificador **const**, el cual se coloca al principio de la declaración, luego del modificador estático si éste también fuera parte de la declaración.

Al declarar un atributo como constante se evita que éste pueda ser modificado accidentalmente por algún método. Cuando se declara un método como constante se evita que este pueda ser redefinido, por ejemplo, por una subclase. Esto implica que todo método llamado de forma directa o indirecta desde un método de tipo constante, debe ser a su vez constante.

ESTRUCTURA DE UNA CLASE

Como se había mencionado anteriormente, una clase está formada por atributos y métodos. En esta sección se describe como estos elementos son manejados por NASPOO.

ATRIBUTOS

Los atributos constituyen la estructura interna de los objetos de una clase (Ceballos, 2003); son un conjunto de variables que describen el estado de un objeto (Joyanes, 1996).

DECLARACIÓN DE UN ATRIBUTO

Para declarar uno o varios atributos de una clase, se coloca el nombre de la clase seguido de dos puntos (:) y la lista de identificadores pertenecientes a dicha clase separados por comas, de la siguiente forma:

Nombre_de_clase: $atributo_1$, $atributo_2$, …, $atributo_n$

Donde:

- Nombre_de_clase es el nombre de la clase a la cual pertenecen los atributos.
- Cada $atributo_i$ ($1 \leq i \leq n$) es un atributo de la clase Nombre_de_clase.

MÉTODOS

Los objetos se comunican unos con otros mediante el envío de mensajes, los cuales son llamadas a métodos, hechas por un objeto emisor hacia un objeto receptor. Un objeto responde a un mensaje mediante la ejecución de un método (Poo, Kiong y Swarnalatha, 2008).

Según lo expresado anteriormente, un método es una colección de sentencias que ejecutan una tarea específica, se implementa en una clase y determina cómo debe actuar un objeto cuando recibe un mensaje vinculado con ese método (Ceballos, 2003).

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

DEFINICIÓN DE UN MÉTODO

Para definir un método, se aplica la misma sintaxis definida para NASPI (Martínez y Rosquete, 2009). La definición de una función tiene la siguiente sintaxis:

[**estático** | **const** | **virtual**] **func** metodo ([F_1; …; F_n]) : Nombre_de_clase
[// Declaraciones locales]
inicio
    // Instrucciones
    retornar(expr)
ffunc

La definición de un procedimiento tiene la siguiente sintaxis:

[**estático** | **const** | **virtual**] **proc** metodo ([F_1; …; F_n])
[// Declaraciones locales]
inicio
    // Instrucciones
fproc

Donde:

- método es el identificador de un método (función o procedimiento).

- Cada F_i ($1 \leq i \leq n$) es un grupo de parámetros formales definidos de la siguiente forma: {**val** | **ref**} $Nombre\_de\_Clase_i$: p_1, p_2, …, p_m, donde:

- **val** o **ref** indican si el paso de parámetros se realiza por valor o por referencia, respectivamente.
- $Nombre\_de\_clase_i$ es la clase de los parámetros formales que forman el i-ésimo grupo.
- p_j ($1 \leq j \leq m$) es el identificador del j-ésimo parámetro formal del i-ésimo grupo.

Además, en el caso de una función:

- Nombre_de_clase es la clase del objeto que retorna la función.

- expr es el valor de retorno de la función. El valor resultante de expr debe pertenecer a la clase Nombre_de_clase.

OPERADOR DE RESOLUCIÓN DE ALCANCE BINARIO

Un método se puede definir luego de su correspondiente declaración de clase, en cuyo caso se utiliza un operador llamado operador de resolución de alcance binario (**::**). Para realizar la definición de un método bajo esta modalidad se coloca dentro de la clase solamente la declaración del método, y fuera de la clase se coloca su definición. En la

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

definición, el nombre del método es antecedido por el nombre de la clase a la cual pertenece, seguido del operador de resolución de alcance de la siguiente forma:

Para una función:

func Nombre_de_clase_pert::metodo ([F_1; …; F_k]): Nombre_de_clase

Para un procedimiento:

proc Nombre_de_clase_pert::metodo ([F_1; …; F_k])

Donde Nombre_de_clase_pert, es el nombre de la clase a la cual pertenece el método.

Dado que diferentes clases pueden tener los mismos nombres de miembros, el operador de resolución de alcance "une" el nombre de miembro con el nombre de la clase, para fijar de forma única las funciones miembro de una clase en particular (Deitel y Deitel, 1995).

CONSTRUCTORES

Un constructor es un método especial cuya función principal es inicializar nuevos objetos de su clase (Ceballos, 2003). En NASPOO el nombre predeterminado para el constructor de una clase es "construir".

En cada clase se debe declarar al menos un método constructor, y de ser necesario, en una misma clase pueden declararse otros métodos constructores, siempre y cuando sus parámetros difieran en tipo, cantidad o posición. La declaración de un constructor tiene la siguiente sintaxis:

proc construir ([F_1; …; F_k])

CONSTRUCTOR COPIA

El constructor copia de una clase es un constructor que se invoca para iniciar un nuevo objeto creado a partir de otro existente (Ceballos, 2003). Este tipo especial de constructor acepta un único argumento, el cual es una referencia al objeto que se quiere copiar, para después asignar miembro a miembro ese objeto al nuevo objeto construido. En NASPOO el nombre predeterminado para el constructor copia de una clase es "copiar". La declaración de un constructor copia tiene la siguiente sintaxis:

proc copiar (**ref** Nombre_de_clase: p)

Donde:

- copiar es el nombre predeterminado del constructor copia de una clase.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

- Nombre_de_clase es el nombre de la clase a la cual pertenece el objeto que se quiere copiar, que debe ser de la misma clase a la cual pertenece el constructor copia.

- p es el nombre del objeto que se quiere copiar, el cual es el único parámetro formal del constructor copia.

Se realizará una llamada implícita al constructor copia de una clase automáticamente cada vez que alguna función retorne un objeto de dicha clase o, cuando se realice una llamada a un método que posea entre sus parámetros por valor un objeto de la misma.

DESTRUCTORES

Un destructor es un método especial de una clase, el cual es llamado cuando se destruye un objeto. Este no recibe parámetros ni retorna valor alguno (Deitel y Deitel, 2009). En NASPOO el nombre predeterminado para el método destructor de una clase es "destruir". Su sintaxis es la siguiente: **proc** destruir(). Es importante acotar que cada clase puede tener únicamente un destructor.

OBJETO

Un objeto es un elemento individual e identificable, ya sea real o abstracto, el cual contiene una serie de datos que lo definen y las descripciones de cómo pueden ser manipulados dichos datos (Armstrong, 1996). Un objeto también es llamado instancia de una clase.

DECLARACIÓN DE UN OBJETO

Uno o varios objetos pueden ser declarados como sigue:

Nombre_de_clase: $objeto_1$, $objeto_2$, …, $objeto_n$

Donde:

- Nombre_de_clase es el identificador de la clase a la cual pertenecen los objetos declarados.

- $objeto_i$ ($1 \leq i \leq n$) es un identificador válido para un objeto perteneciente a la clase.

CREACIÓN DE UN OBJETO

La creación de un objeto consiste en asignarle al mismo un nuevo espacio en memoria para ser almacenado. En NASPOO esta creación se realiza de forma implícita al declarar un objeto, mediante un método predefinido llamado "crear", el cual no tiene parámetros y retorna una referencia al objeto creado. Tiene la siguiente sintaxis:

func crear(): Nombre_de_clase

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

Este método debe ser usado explícitamente para asignar memoria a un objeto de tipo Apuntador para asignar memoria al objeto al cual apunta.

INICIALIZACIÓN DE UN OBJETO

La inicialización de un objeto consiste en asignar valores iniciales válidos a los atributos de un objeto después de haber sido creado. Esta acción es realizada por el constructor de la clase a la cual pertenece el objeto. Para inicializar un objeto de una clase es posible utilizar el constructor de la clase ("construir") o el constructor copia de la clase ("copiar"). La sintaxis es la siguiente:

Cuando se invoca al constructor de la clase:

objeto.construir([A_1; A_2; …; A_n])

Cuando se invoca al constructor copia de la clase:

$objeto_2$.copiar(objeto)

Donde:

- objeto es el identificador de un objeto.
- construir es el identificador por defecto del constructor de la clase a la cual pertenece el objeto.
- Cada A_i ($1 \leq i \leq n$) es un grupo de parámetros actuales, a_1, a_2, …, a_m, donde a_j ($1 \leq j \leq m$) es el identificador del parámetro actual correspondiente al j-ésimo parámetro formal del i-ésimo grupo.
- $objeto_2$ es el identificador de un objeto de la misma clase que "objeto".

REFERENCIA AL OBJETO RECEPTOR DE UN MENSAJE

Para que un método conozca la identidad del objeto para el cual fue invocado se utiliza la palabra reservada instancia, que es una referencia a dicho objeto.

POLIMORFISMO

En POO, el polimorfismo es la habilidad que tienen diferentes objetos para invocar el método apropiado en respuesta a un mismo mensaje (Poo, Kiong y Swarnalatha, 2008). Existen dos tipos de polimorfismo: estático y dinámico. El polimorfismo estático y el polimorfismo dinámico son soportados por la ligadura estática y dinámica de métodos respectivamente. Ambos tipos de polimorfismo son permitidos por NASPOO.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

LIGADURA ESTÁTICA Y DINÁMICA

Los lenguajes imperativos utilizan ligadura estática, esto significa que los tipos de variables, expresiones y funciones se conocen en tiempo de compilación (Joyanes, 1996). Por el contrario, los lenguajes orientados a objetos utilizan ligadura dinámica, la cual permite determinar en tiempo de ejecución el método a llamar para un objeto particular de una subclase dependiendo del argumento (Puntambekar, 2010). Esta característica, propia de los lenguajes orientados a objetos, hace posible el polimorfismo. NASPOO considera la utilización de la ligadura dinámica.

POLIMORFISMO ESTÁTICO

Cuando se aplica el polimorfismo estático, el compilador resuelve en tiempo de compilación que implementación debe llamar para métodos que comparten el mismo nombre (Sarang, 2009). Este tipo de polimorfismo usa ligadura estática y se implementa utilizando técnicas de sobrecarga: (1) de métodos y (2) de operadores.

1. Sobrecarga de Métodos

La sobrecarga permite definir diferentes métodos con un mismo nombre, siempre y cuando tengan signaturas diferentes (Deitel y Deitel, 2009). Cuando un método es sobrecargado, el compilador decide que implementación utilizar al examinar el número, tipo y orden de los parámetros en la llamada al método.

2. Sobrecarga de Operadores

La sobrecarga de operadores permite la redefinición del significado asignado por defecto a los operadores regulares en un contexto determinado (Sarang, 2009). En NASPOO un operador sobrecargado tiene la misma sintaxis definida anteriormente para un método, sin embargo se debe especificar que se está sobrecargando un operador mediante la palabra reservada **operador**, la cual se coloca justo antes del identificador del método, que en este caso es el operador a sobrecargar. La signatura de un operador sobrecargado es la siguiente:

func operador #([F_1; …; F_k]): Nombre_de_clase

Donde # es el operador que se desea sobrecargar.

POLIMORFISMO DINÁMICO

Cuando se aplica polimorfismo dinámico, el compilador retrasa hasta el tiempo de ejecución la decisión sobre cual implementación se debe llamar para un método (Sarang, 2009). El polimorfismo dinámico es implementado mediante la creación de métodos con el mismo nombre a través de la jerarquía de clases, por lo cual, no necesitan tener signaturas diferentes. En este caso el método de la superclase es sobrescrito por el método de la subclase.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

HERENCIA

La herencia permite que las propiedades de la superclase se propaguen a las subclases en una jerarquía de clases (Poo, Kiong y Swarnalatha, 2008), por lo tanto, es un mecanismo que permite a una clase de objetos compartir la representación y los métodos de otra clase de objetos (LaLonde y Pugh, 1990). Es una forma de reutilización de software, en la cual para crear una nueva clase se absorben los datos y comportamientos de una clase existente y se mejoran con capacidades nuevas (Deitel y Deitel, 2009). Existen dos tipos de herencia: simple y múltiple.

1. Herencia Simple

La herencia simple se da cuando una clase se deriva de una sola clase base (Deitel y Deitel, 2009). Para representar la herencia en NASPOO se utiliza la palabra reservada **hereda_de** luego del identificador de la clase derivada, seguido del identificador de la clase base de la siguiente forma:

clase Subclase
    hereda_de Superclase
    // Declaración de los atributos y métodos de la clase
fclase

Donde:

- Subclase es el identificador de la clase derivada.
- Superclase es el identificador de la clase base.

2. Herencia Múltiple

La herencia múltiple se da cuando una clase derivada hereda de varias clases base (posiblemente no relacionadas) (Deitel y Deitel, 2009). Este tipo de herencia no es soportado por todos los lenguajes de programación orientados a objetos (por ejemplo, Java), sin embargo teóricamente es importante conocer este mecanismo, su uso, ventajas y desventajas.

En NASPOO se representa la herencia múltiple colocando la palabra reservada **hereda_de** luego del identificador de la clase derivada, seguido de los identificadores de las clases base separadas por comas como se muestra a continuación:

clase Subclase
    hereda_de Superclase_1, Superclase_2, …, Superclase_n
    // Declaración de los atributos y métodos de la clase
fclase

Donde:

- Subclase es el identificador de la clase derivada.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

- Superclase$_i$ ($1 \leq i \leq n$), es el identificador de una de las clases base.

HERENCIA SELECTIVA

La herencia selectiva se usa para prevenir que ciertos atributos o métodos sean heredados por una subclase (Shoval, 2007). NASPOO soporta la herencia selectiva mediante la utilización de la palabra reservada **excepto**, la cual se coloca luego del nombre de la superclase a la cual pertenece el miembro que se desea descartar, seguido de los identificadores de los atributos o métodos a descartar, entre llaves. Su sintaxis es la siguiente:

clase Subclase
    hereda_de Superclase **excepto** miembro$_1$, miembro$_2$, …, miembro$_n$
    // Declaración de los atributos y métodos de la clase
Fclase

ACCESO A LOS MÉTODOS DE UNA SUPERCLASE

Es posible acceder a los métodos de una superclase en la definición de una clase derivada mediante la variable **super**. Esta variable provee acceso a los métodos en la cadena de superclases aun si el método ha sido redefinido en la subclase (LaLonde y Pugh, 1990). Su sintaxis es la siguiente:

super.metodo()

CLASES ABSTRACTAS

Una clase que especifica un protocolo pero es incapaz de implementarlo completamente porque sus subclases pueden tener representaciones diferentes se denomina clase abstracta (LaLonde y Pugh, 1990). Las clases abstractas no pueden ser instanciadas, su finalidad es definir miembros comunes que heredan sus subclases, también llamadas clases concretas. Para declarar una clase abstracta se utiliza la palabra reservada **abstracto**.

La sintaxis de una clase abstracta es la siguiente:

abstracto clase Nombre_de_clase
    // Declaración de los atributos y métodos de la clase
Fclase

Los métodos definidos en la clase abstracta pueden ser redefinidos (sobrescritos) en sus subclases. Esto es posible gracias al polimorfismo dinámico.

MÉTODOS VIRTUALES Y VIRTUALES PUROS

Un método virtual es un método que se declara dentro de la clase base y es redefinido por las clases derivadas (ISRD Group, 2007). Para declarar un método virtual se utiliza la

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

palabra reservada **virtual**, la cual se coloca justo antes del modificador que indica si el método es una función o es un procedimiento. Su sintaxis es la siguiente:

Para una función:
virtual func metodo ([F_1; …; F_k]): Nombre_de_clase

Para un procedimiento:
virtual proc método ([F_1; …; F_k])

En las clases derivadas debe colocarse cuales de los métodos declarados como virtuales en la clase base van a ser redefinidos, para ello se utiliza la palabra reservada **sobrescribe** de la siguiente forma:

clase Subclase
hereda_de $Superclase_1$, $Superclase_2$, …, $Superclase_n$
sobrescribe $metodo_1$, $metodo_2$, …, $metodo_m$
// Declaración de los atributos y métodos de la clase
fclase

Donde:

- Subclase es el identificador de la clase derivada.
- $Superclase_i$ ($1 \leq i \leq n$), es el identificador de una de las clases base.
- $metodo_i$ ($1 \leq i \leq m$), es el identificador de un método sobrescrito en la subclase.

Vale la pena destacar que cuando se sobrescribe un método, puede cambiarse su modificador de control de acceso para restringir aún más el acceso a dicho método. Esto se logra colocando el método a redefinir en la sección correspondiente a un modificador más restrictivo. Si sólo se desea cambiar el modificador de acceso, mas no la implementación del método solo se coloca la nueva signatura sin su definición.

Existen métodos que son comunes a todas las subclases, pero su implementación no es común o puede ser muy general, por lo cual solamente se declaran en la clase abstracta, mas no se definen. Estos métodos son llamados métodos virtuales puros o métodos abstractos, y no tienen cuerpo (Ceballos, 2003). Es responsabilidad de las clases concretas definir cada método virtual puro. Para estos métodos solo es necesaria su declaración en la clase abstracta.

ESTRUCTURA DE UN ALGORITMO

NASPOO define una clase especial llamada **Algoritmo** que contendrá un método público especial, propio de la clase, llamado “principal”, en el cual estarán todas las operaciones que deben ser llevadas a cabo. Este método debe ser declarado y definido únicamente dentro de la clase **Algoritmo**. Además, pueden existir otros métodos

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

(privados o protegidos) para uso exclusivo de la clase **Algoritmo**. La estructura general de un algoritmo es:

```
clase Algoritmo {Nombre del Algoritmo}
    atributos:
        [ // Declaraciones de constantes y atributos ]
    métodos:
        [ // Métodos de clase ]
        público:
            estático proc principal()
            [ // Declaraciones locales ]
            inicio
                // Instrucciones
            fproc
fclase
```

Ejemplo

En el ejemplo se utilizará NASPOO para mostrar parte de la implementación de los tipos abstractos Lista y Pila (como herencia de Lista), haciendo uso de estructuras dinámicas.

```
clase Nodo
    atributos:
        privado:
            Real: elemento
            Nodo: proximo
    métodos:
        público:
            proc construir()
            inicio
                instancia.elemento ← 0.0
                instancia.proximo ← nulo
            fproc

            proc modificarElemento(Real: elem)
            inicio
                instancia.elemento ← elem
            fproc

            proc modificarProximo(Nodo: prox)
            inicio
                instancia.proximo ← prox
            fproc

            func proximo(): Nodo
                retornar(instancia.proximo)
```

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

```
                ffunc
                …
fclase

clase Lista
    atributos:
        privado:
            Nodo: primero, ultimo
            Entero: cantidad
    métodos:
        público:
            proc construir()
            func es_vacia(): Lógico
            virtual proc insertar(Real: elem; Entero: pos)
            virtual proc eliminar(Entero: pos)
            virtual func consultar(Entero: pos): Real
            func longitud(): Entero
fclase

proc Lista::construir()
inicio
    instancia.primero ← nulo
    instancia.ultimo ← nulo
    instancia.cantidad ← 0
fproc

func Lista::es_vacia(): Lógico
inicio
    retornar(instancia.cantidad = 0)
ffunc

proc Lista::insertar(Real: elem; Entero: pos)
var
    Nodo n, ant, sig
inicio
    n.construir()
    n.modificarElemento(elem)
    si(instancia.cantidad = 0) entonces
        instancia.primero ← n
        instancia.ultimo ← n
    sino
        si(pos = 1) entonces // Insertar al principio
            n.modificarProximo(instancia.primero)
            instancia.primero ← n
        sino                          // Insertar al final
            si(pos = instancia.cantidad + 1) entonces
```

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

```
                    instancia.ultimo.modificarProximo(n)
                    instancia.ultimo ← n
                sino                    // Insertar en el centro
                    ant ← instancia.primero
                    sig ← ant.proximo()
                    para i ← 2 hasta pos – 1 hacer
                        ant ← sig
                        sig ← sig.proximo()
                    fpara
                    ant.modificarProximo(n)
                    n.modificarProximo(sig)
                fsi
            fsi
        fsi
        instancia.cantidad ← instancia.cantidad + 1
    fproc
    …

    clase Pila
        hereda_de Lista excepto longitud
        sobrescribe insertar, eliminar, consultar
        métodos:
            privado:
                proc insertar(Real: elem; Entero: pos)
                proc eliminar(Entero: pos)
                func consultar(Entero: pos): Real
            público:
                func construir(): Pila
                func es_vacia(): Lógico
                proc apilar(Real: elem)
                proc desapilar()
                func tope(): Real
    fclase

    proc Pila::construir()
    inicio
        super.construir()
    fproc

    func Pila::es_vacia(): Lógico
    inicio
        retornar(super.es_vacia())
    ffunc

    proc Pila::apilar(Real: elem)
    inicio
```

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

```
    instancia.insertar(elem, 1)
fproc

proc Pila::desapilar()
inicio
    instancia.eliminar(1)
fproc

func Pila::tope(): Real
inicio
    retornar(instancia.consultar(1))
ffunc
// Algoritmo Principal
clase Algoritmo {Manejo de Pilas}
    métodos:
        público:
            estático proc principal()
            var
                // Definición y creación de objetos
                Pila: P
                Real: x
                Entero: i
            inicio
                // Inicialización de la pila
                P.construir()
                // Llenado de la pila
                x ← 2.5
                para i ← 1 hasta 10 hacer
                    P.apilar(i*x)
                fpara
                // Consulta del tope de la pila
                escribir("El tope de la pila es: ", P.tope())
            fproc
fclase
```

CONCLUSIONES Y TRABAJO FUTURO

En este trabajo se presentó NASPOO, una propuesta de notación algorítmica estándar para programación orientada a objetos, como complemento a la propuesta de notación algorítmica estándar para programación imperativa llamada NASPI (Martínez y Rosquete, 2009).

Esta propuesta se basó en NASPI (Martínez y Rosquete, 2009), UPSAM (Joyanes, 2008) y Lenguaje Pseudoformal (Coto, 2002). Además, se tomaron en cuenta las características más importantes de los lenguajes de programación orientada a objetos, para lograr una notación robusta y general para la enseñanza de la POO.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

NASPOO provee las estructuras necesarias para abarcar los conceptos fundamentales del modelo orientado a objetos (abstracción, encapsulamiento, modularidad y jerarquía), lo cual permite realizar el diseño de algoritmos que puedan ser traducidos de forma sencilla a cualquier lenguaje de POO.

Como trabajo futuro se propone culminar el desarrollo de la plantilla para la escritura de algoritmos en LaTeX, basada en algorithm2e (Martínez y Rosquete, 2009), abarcando también las estructuras presentadas en NASPOO.

REFERENCIAS BIBLIOGRÁFICAS

Armstrong, D. (1996). The Quarks of Object-Oriented Development. Communications of the ACM. 49(2). Pp. 123 – 128.

Booch, G.; Maksimchuk, R.; Engle, M.; Young, B.; Conallen, J. y Houston, K. (2007). Object-Oriented Analysis and Design with Applications (Tercera ed.). USA. Pearson Education Inc.

Ceballos, F. (2003). Java 2: Curso de Programación (Segunda ed.). México. Alfaomega Grupo Editor.

Coto, E. (2002). Lenguaje Pseudoformal para la Construcción de Algoritmos. (Tech. Rep. ND 2002-08). Venezuela. Fondo Editorial de la UCV.

Deitel, H. y Deitel P. (1995). Cómo Programar en C/C++ (Segunda ed.). México. Prentice-Hall Hispanoamericana, S.A.

Deitel, P. y Deitel, H. (2009). Cómo Programar en C++ (Sexta ed.). México. Pearson Educación de México, S.A. de C.V.

Instructional Software Research and Development (ISRD) Group. (2007). Introduction to Object Oriented Programming and C++. India. Tata McGraw Hill Publishing Company Limited.

Joyanes, L. (1996). Programación Orientada a Objetos (Segunda ed.). España. McGraw Hill Interamericana de España S.A.

Joyanes, L. (2008). Fundamentos de Programación (Cuarta ed.). España. McGraw Hill Interamerica S.A.

LaLonde, W. y Pugh, J. (1990). Inside Smalltalk, Volume I. USA. Prentice-Hall, Inc.

Martínez, A. y Rosquete, D. (2009). NASPI: Una Notación Algorítmica Estándar para la Programación Imperativa. Télématique. 8(3). Pp. 55 – 74.

Poo, D.; Kiong, D. y Swarnalatha, A. (2008). Object-Oriented Programming and Java (Segunda ed.). Inglaterra. Springer.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011

Puntambekar, A. (2010). Data Structures (Primera ed.). India. Technical Publications.

Sarang, P. (2009). Object-Oriented Programming with C++ (Segunda ed.). India. Prentice-Hall of India Private Limited.

Shoval, P. (2007). Functional and Object-Oriented Analysis and Design: An Integrated Methodology (Primera ed.). USA. Idea Group Inc.

Depósito Legal: PPX200002ZU2142 / ISSN:1856-4194. Volumen 10 Edición No 1 – Año 2011