Centro de Investigación Científica y de
Educación Superior de Ensenada

ORIENTACION A OBJETOS DEL METODO DE VORTICES EN CELDA APLICADO A FLUIDOS GEOFISICOS

TESIS
MAESTRIA EN CIENCIAS

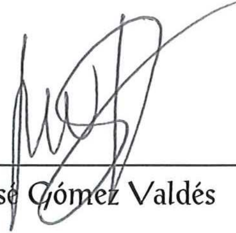
ALBERTO LEOPOLDO MORAN Y SOLARES

ENSENADA, BAJA CALIFORNIA, MEXICO. ENERO DE 1998.

TESIS DEFENDIDA POR

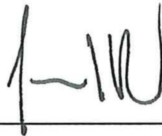
ALBERTO LEOPOLDO MORÁN Y SOLARES

Y APROBADA POR EL SIGUIENTE COMITE



Dr. José Gómez Valdés

Director del Comité



Dr. Jesús Favela Vara

Miembro del Comité

Dr. Oscar Uriel Velasco Fuentes

Miembro del Comité

Dr. José Luis Medina Monroy

Jefe del Departamento de Electrónica y Telecomunicaciones

Dr. Federico Graef Ziehl

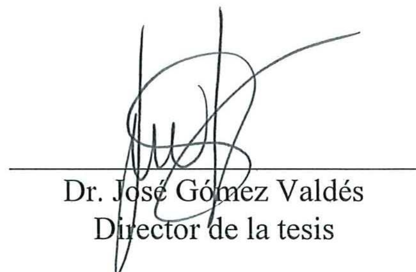
Director de Estudios de Posgrado

16 de enero de 1998

RESUMEN de la Tesis de Alberto Leopoldo Morán y Solares, presentada como requisito parcial para la obtención del grado de MAESTRO EN CIENCIAS en CIENCIAS DE LA COMPUTACIÓN. Ensenada, Baja California, México. Enero de 1998.

ORIENTACIÓN A OBJETOS DEL MÉTODO DE VÓRTICES EN CELDA APLICADO A FLUIDOS GEOFÍSICOS

Resumen aprobado por:



Dr. José Gómez Valdés
Director de la tesis

Se elabora un ambiente orientado a objetos del método de vórtices en celda aplicado a la dinámica de fluidos geofísicos. El trabajo se inicia aplicando el esquema de desarrollo por versiones sucesivas. Se parte de la versión Fortran de Velasco Fuentes (1994), se hace una versión en C completamente estructurada y se llega a una versión orientada a objetos en C++. Se usa la técnica OMT para realizar el análisis y diseño orientado a objetos, lo que permitió una mejor comunicación entre el desarrollador y los clientes. Se propone una arquitectura para el desarrollo de aplicaciones de simulación numérica, cuyas principales componentes son conjuntos de datos básicos, métodos numéricos básicos, conjuntos de datos de la simulación, métodos numéricos de la simulación y programa de simulación. Esto permitió mantener separados diferentes bloques de la arquitectura durante el desarrollo. Se generaron los modelos de objetos, dinámico y funcional para cada uno de los componentes y se usan envolturas de código y código legado durante la implantación. El código orientado a objetos resultante se puede portar a plataformas PC y estaciones de trabajo con los sistemas operativos usuales. La extensibilidad del código se puede probar para todas las clases. Se escogió una de las clases de los métodos numéricos básicos y se realizó una prueba que resultó exitosa. El modelo orientado a objetos así obtenido permite tener por separado jerarquías de conjuntos de datos y jerarquías de métodos numéricos, lo cual facilita la reutilización, extensión y mantenimiento de las aplicaciones numéricas. Sin embargo, se requiere abordar el problema de optimización del código, ya que el tiempo de ejecución se incrementó por un factor de dos.

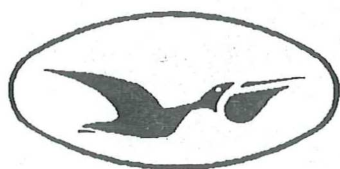
Palabras clave: *Orientación a objetos, OMT, método de vórtice en celda, dinámica de fluidos geofísicos.*

ABSTRACT of the thesis of Alberto Leopoldo Morán y Solares, presented as a partial requeriment to obtain the MASTER IN SCIENCES degree in COMPUTER SCIENCES. Ensenada, Baja California, México. January 1998.

AN OBJECT-ORIENTED APPROACH TO THE VORTEX-IN-CELL METHOD WITH APPLICATION TO GEOPHYSICAL FLUID DYNAMICS

An Object-Oriented framework is presented for the application of the vortex-in-cell method to geophysical fluid dynamics. First, the scheme of successive versions is applied, which starts from the Fortran version of Velasco Fuentes, then a fully structured C version is made and finally, an object-oriented version using C++ is developed. The OMT technique was used for analysis and design, which allowed for a better communication among developer and clients. A software architecture is proposed for the development of numerical simulation systems. The basic data sets, the basic numerical methods, the simulation data sets, the simulation numerical methods and the simulation program are the main components of that proposal. This approach allowed us to maintain separated architectural blocks during the development. Object, functional and dynamic models were developed for each component, and code wrappers and legacy code were used during the implementation. The resulting object-oriented code is portable to a wide range of PCs and workstations. The extensibility of the code can be tested for each class. The class of basic numerical methods was chosen to make the test, which was successful. The object-oriented model allowed us to have separated hierarchies of data sets and numerical methods, which facilitate reuse, extension and maintenance of numerical applications. However, due to an increase in the execution time by a factor two, it is necessary to address the code optimization problem.

Keywords: *Object-oriented, OMT, vortex-in-cell method, geophysical fluid dynamics.*



CENTRO DE INVESTIGACIÓN CIENTÍFICA Y DE EDUCACIÓN SUPERIOR DE ENSENADA

DIVISIÓN DE FÍSICA APLICADA

DEPARTAMENTO DE ELECTRÓNICA Y COMUNICACIONES

ÁREA: CIENCIAS DE LA COMPUTACIÓN

ORIENTACIÓN A OBJETOS DEL MÉTODO DE VÓRTICES EN CELDA APLICADO A FLUIDOS GEOFÍSICOS

TESIS

que para cubrir parcialmente los requisitos necesarios para obtener el grado

de MAESTRO EN CIENCIAS presenta:

ALBERTO LEOPOLDO MORÁN Y SOLARES

Ensenada, Baja California, México. Enero de 1998.

DEDICATORIA

A mis papás Ernestina y Leopoldo,

a mis hermanas Rosa Alicia, Luz Ernestina, Ana Lilia y Elisa Beth,

a mis sobrinos Blanca Milena, Regina Guadalupe y Leopoldo Francisco,

a mi esposa Laura Elena,

a mi abuelita Amparo y a mi cuñado Benito,

ya que por su amor, cuidados, apoyo, consejos y bendiciones estoy aquí.

A mi hermana Blanquita, mi abuelita Cruz y mis abuelitos Miguel y Felipe,

mis ángeles guardianes, donde quiera que estén.

Al resto de la familia, que no terminaría de nombrar.

L.Q.M.   A.L.M.Y.S.

AGRADECIMIENTOS

A Dios por permitirme llegar hasta aquí.

A mi director Dr. José Gómez Valdés por su apoyo, consejos, dedicación y paciencia al dirigir mi trabajo y por brindarme su amistad.

A los miembros de mi comité de tesis por sus valiosas aportaciones durante el desarrollo y revisión del trabajo.

A todos mis amigos, compañeros y maestros, porque con su apoyo he logrado dar un paso más en mi formación profesional.

Al Centro de Investigación Científica y de Estudios Superiores de Ensenada.

A la Facultad de Ciencias de la UABC.

Al Consejo Nacional de Ciencia y Tecnología.

CONTENIDO

I. INTRODUCCIÓN 1

II. EQUIPO DE CÓMPUTO Y HERRAMIENTAS BÁSICAS PARA EL DESARROLLO DEL TRABAJO 4

III. MÉTODOS 5

IV. MODELO MATEMÁTICO 7

IV.1 VÓRTICES PUNTUALES 7
IV.2 MÉTODO DE VÓRTICE EN CELDA. 8
IV.2.1 Discretización *8*
IV.2.2 Movimiento de los vórtices puntuales *9*
IV.2.2.1 Interpolación 10
IV.2.2.2 Función de corriente 12
IV.2.2.3 Velocidades 12
IV.2.2.4 Integración en el tiempo 13
IV.2.2.5 Nueva vorticidad relativa 13

V. DESARROLLO **14**

V.1 CONVERSIÓN DEL PROGRAMA FORTRAN AL PROGRAMA C COMPLETAMENTE ESTRUCTURADO. 14
V.1.1 Etapa de traducción. *15*
V.1.2 Etapa de depuración y ejecución. *16*
V.1.3 Etapa de optimización. *16*
V.1.4 Resultados obtenidos en la fase de conversión de la versión Fortran a la versión C completamente estructurada *17*
V.2 ANÁLISIS ORIENTADO A OBJETOS DEL MÉTODO DE VÓRTICE EN CELDA 18
V.2.1 Establecimiento del problema. *19*
V.2.2 Construcción del modelo de objetos. *19*
V.2.2.1 Identificación de clases de objetos en el método de vórtice en celda. 19
V.2.2.2 Asociaciones entre clases. 19
V.2.2.3 Atributos de los objetos. 20
V.2.2.4 Especificación de herencia y agregación. 20
V.2.3 Construcción del modelo dinámico. *22*
V.2.3.1 Preparación de escenarios típicos. 22
V.2.3.2 Construcción de trazos de eventos. 23
V.2.3.3 Construcción de diagramas de estados. 25
V.2.4 Construcción del modelo funcional. *26*
V.2.4.1 Identificación de valores de entradas y salidas 26
V.2.4.2 Los diagramas de flujo de datos. 27
V.2.5 Afinación del modelo de objetos. *29*
V.3 DISEÑO DE SISTEMAS 30

V.3.1 Identificación de subsistemas. 30
V.3.2 Selección de la estrategia básica para implantar los almacenamientos de datos. 32
V.3.3 Selección del esquema para implantar el control del software. 33
V.3.4 Establecimiento de prioridades de decisión sobre características deseables del programa. 33
V.4 DISEÑO DE OBJETOS 35
V.4.1 Obtención de las operaciones para el modelo de objetos a partir de los otros modelos. 35
V.4.2 Diseño de los algoritmos para implementar las operaciones. 42
V.4.3 Implementación del flujo de control del software. 43
V.4.4 Ajuste de la estructura de clases para incrementar herencia. 45
V.4.5 Diseño de la implementación de las asociaciones. 47
V.4.6 Determinación de la representación de los atributos de los objetos. 47
V.4.7 Empaquetamiento de las clases y las asociaciones en módulos. 49
V.5 IMPLEMENTACIÓN 50
V.5.1 Los tipos de datos básicos: Vector y Matriz. 50
V.5.2 Los objetos genéricos del programa de simulación. 51
V.5.3 Los tipos de datos de la simulación. 51
V.5.4 Clases de Métodos. 58
V.5.5 El programa de simulación. 64
VI. RESULTADOS 66
VI.1 ARQUITECTURA PARA PROGRAMAS DE SIMULACIÓN NUMÉRICA 66
VI.2 EL MODELADO ORIENTADO A OBJETOS 66
VI.3 EL PROGRAMA DE SIMULACIÓN ORIENTADO A OBJETOS EN C++ 66
VII. DISCUSIÓN 69
VIII. CONCLUSIONES 79
LITERATURA CITADA 80
APÉNDICE A. GLOSARIO 82
MECÁNICA DE FLUIDOS 82
PROGRAMACIÓN ORIENTADA A OBJETOS 84
APÉNDICE B. LA TÉCNICA DE MODELADO POR OBJETOS (OMT) COMO UNA METODOLOGÍA DE INGENIERÍA DE SOFTWARE. 86
ANÁLISIS. 86
DISEÑO DE SISTEMAS. 89
DISEÑO DE OBJETOS. 90
ELEMENTOS Y NOTACIÓN. 92
APÉNDICE C. CASOS DE PRUEBA. 103

LISTA DE FIGURAS

1	Modelo de desarrollo de aplicaciones por creación de prototipos.	5
2	Esquema de interpolación nube en celda.	11
3	Modelo de objetos de las clases identificadas para los datos de la simulación, mostrando herencia y agregación.	21
4	Trazo de eventos para el Escenario 1.	24
5	Trazo de eventos para el Escenario 2.	24
6	Diagrama de estados general para el método de vórtice en celda.	25
7	Diagrama de estados para "Calcula las Velocidades".	26
8	Diagrama de flujo de datos para los procesos detectados.	28
9	Diagrama a bloques de la organización del sistema en capas y particiones.	31
10	Diagrama de clases para Algoritmo Numérico.	37
11	Diagrama de clases para Conjunto de Datos.	38
12	Modelo dinámico general de un algoritmo numérico.	40
13	Modelo funcional general de un algoritmo numérico.	40
14	Diagrama de clases para el programa de simulación.	41
15	Diagrama de clases para los datos básicos y de la simulación.	41
16	Diagrama de clases para las operaciones de la simulación.	42
17	Pseudocódigo a primer nivel, generado a partir del modelo funcional.	44
18	Jerarquía propuesta para las clases de métodos básicos.	45
19	Diagrama de clases para las operaciones y métodos de la simulación aplicando la jerarquía de clases de métodos básicos.	46
20	Representación de la tira de vórtices puntuales mostrando los atributos.	48
21	Representación de la malla de vórtices mostrando los atributos.	48
22	Archivo de encabezado para la clase de los Objetos de la aplicación.	52

LISTA DE FIGURAS (Continuación)

23	Archivo de encabezado para la clase Datos genéricos, la cual se deriva de la clase de objetos genéricos.	53
24	Archivo de encabezado para la clase TiraVortices, la cual se deriva de la clase de Datos genéricos.	54
25	Archivo de encabezado para la clase MallaVortices, la cual se deriva de la clase de Datos genéricos.	56
26	Archivo de encabezado para la clase Métodos, la cual es la clase base para todos los métodos.	59
27	Archivo de encabezado para la clase MetodosCIC.	59
28	Archivo de encabezado para la clase PoissonSolver.	60
29	Archivo de encabezado para la clase IntegradorEDO.	62
30	Archivo de encabezado para la clase RungeKutta2.	62
31	Archivo de encabezado para la clase RungeKutta4.	63
32	Archivo fuente que muestra la implementación del método de vórtice en celda.	64
33	Notación OMT para clases, atributos y operaciones.	93
34	Notación OMT para asociaciones entre clases.	93
35	Notación para multiplicidad de asociaciones entre clases.	94
36	Notación para agregación entre clases componente y clases conjunto.	94
37	Notación OMT para mostrar generalización entre superclase y subclases.	95
38	Asociación ternaria.	95
39	Estados, eventos y transiciones.	96
40	Notación OMT para estados inicial y final.	97
41	Notación para una transición con condición.	97
42	Notación para una transición con una acción asociada.	98

LISTA DE FIGURAS (Continuación)

43	Notación para especificar una actividad en un estado.	98
44	Generalización de estados (anidamiento).	99
45	Notación para un proceso en un diagrama de flujo de datos.	100
46	Flujo de datos entre dos procesos.	100
47	Notación para representar actores empezando y terminando un flujo de datos.	101
48	Notación para representar un almacén de datos.	101
49	Notación para diferentes combinaciones de acceso y actualización a almacenes de datos.	102
50	Formato del archivo para los casos de prueba.	103
51	Caso INI_TEST, dipolo Lamb con anillo (sin vorticidad ambiental ni topografía).	103
52	Caso INI_BETA, dipolo Lamb con vorticidad ambiental (plano β).	104
53	Caso INI_GAMA, dipolo Lamb con vorticidad ambiental (plano γ).	104
54	Caso INI_TOP1, dipolo Lamb con topografía (caso 1).	104
55	Caso INI_TOP2, dipolo Lamb con topografía (caso 2).	105
56	Caso INI_TOP3, dipolo Lamb con topografía (caso 3).	105
57	Caso INI_TOP4, dipolo Lamb con topografía (caso 4).	105

LISTA DE TABLAS

I	Plataformas de cómputo utilizadas durante el desarrollo del trabajo.	4
II	Comparación en la cantidad de espacio de disco requerido por los programas C y Fortran.	17
III	Comparación de la cantidad de memoria primaria requerida por los programas.	18
IV	Comparación de la cantidad de tiempo de ejecución de ambos programas.	18
V	Clases identificadas con sus respectivos atributos.	20
VI	Operaciones detectadas con clases propietarias y parámetros.	29
VII	Algoritmos básicos identificados con el paso de la simulación que implementan.	43
VIII	Empaquetamiento de clases y asociaciones por unidades.	49
IX	Plataformas de cómputo sobre las que está funcionando la versión orientada a objetos del programa de simulación.	67
X	Tiempos de ejecución para el caso de prueba INI TEST.	67
XI	Como Tabla X para INI BETA.	67
XII	Tiempos de ejecución para el caso de prueba INI GAMA.	68
XIII	Como Tabla XII para INI TOP1.	68
XIV	Como Tabla XII para INI TOP2.	68
XV	Como Tabla XII para INI TOP3.	68
XVI	Como Tabla XII para INI TOP4.	68

ORIENTACIÓN A OBJETOS DEL MÉTODO DE VÓRTICES EN CELDA APLICADO A FLUIDOS GEOFÍSICOS

I. INTRODUCCIÓN

El método de partículas, donde las partículas interactúan entre sí, es uno de las herramientas de discretización para modelar y simular sistemas físicos con ayuda de la computadora. En la mayoría de sus aplicaciones, las partículas pueden ser identificadas directamente como objetos físicos (Hockney y Eastwood, 1988). Cada partícula tiene atributos, tales como masa, carga, posición o momentum. El estado del sistema físico está definido por los atributos de un conjunto finito de partículas y su evolución se determina por las leyes de interacción de las partículas. Según Hockney y Eastwood (1988) hay al menos tres tipos de métodos de partículas: 1) partícula-partícula (PP), 2) partícula-malla (PM) y 3) partícula-partícula-partícula-malla (P^3M). El uso de mallas es opcional. Cuando se usa una malla, las interacciones entre partículas se dan a través de campos transitorios (e.g. PM y P^3M); cuando no se usa una malla, las partículas interactúan directamente a través de leyes de fuerza conocidas (e.g. PP). El método a utilizar depende de la física del fenómeno que se quiere investigar y también del costo computacional de la aplicación.

Aparte, el modelado y diseño orientado a objetos dan un mejor entendimiento de las necesidades computacionales, proporciona mayor claridad a las partes del problema, y genera sistemas más flexibles y mantenibles (Rumbaugh *et al.*, 1991). Su meta principal es la identificación y organización de los conceptos del dominio del problema a resolver, en

vez de la representación final en algún lenguaje particular de programación. Abarca la primera parte del ciclo de desarrollo del software: el análisis, el diseño y la implementación. El elemento principal es el objeto, el cual combina tanto las estructuras de datos o atributos como el comportamiento u operaciones en una sola entidad. Así mismo, *identidad*, *clasificación*, *polimorfismo* y *herencia* son elementos esenciales en su desarrollo. Entre ellos, *identidad* significa que los datos y las operaciones se representan como objetos, los cuales a su vez pueden ser concretos o conceptuales. Por *clasificación* se entiende que los objetos con las mismas estructuras de datos y comportamiento se agrupan en una clase que los describe, donde un objeto es un elemento de una clase. *Polimorfismo* significa que la misma operación puede comportarse de manera diferente en diferentes clases de objetos, una operación es una acción o transformación que un objeto realiza o a la que está sujeto. *Herencia* es el compartimiento de atributos y operaciones entre clases basado en una relación jerárquica; una clase puede definirse de manera general y posteriormente refinarse por medio de subclases más especializadas, en donde cada subclase incorpora o hereda todas las propiedades de la superclase y agrega las propias.

La Técnica de Modelado por Objetos (OMT) es una de las metodologías orientadas a objetos más populares (Vayda, 1995; Rumbaugh, 1991), la cual consiste en construir un modelo del dominio de la aplicación agregando detalles de implementación a través de sus etapas de análisis, diseño de sistemas, diseño de objetos e implementación. Los tres modelos que utiliza permiten describir un sistema proporcionando vistas que se complementan y hacen referencia unas a otras. El modelo de objetos describe la estructura

de los objetos, así como las relaciones entre ellos. El modelo dinámico representa las interacciones que se dan entre los objetos. El modelo funcional describe las transformaciones de los datos del sistema. Varios autores (Schroeder *et al.*, 1996; Vayda, 1995; Spinelli *et al.*, 1994; Koved y Wooten, 1993) reportan beneficios al utilizar orientación a objetos en el desarrollo de aplicaciones científicas, entre los principales se encuentran: promover la reutilización futura del código, reducir el número de errores en las etapas finales y facilitar el mantenimiento.

En este trabajo se hace orientación a objetos del método de vórtice en celda. Se usa la metodología OMT de Rumbaugh (1991) para el análisis, diseño e implementación de un modelo orientado a objetos que proporciona los requerimientos de extensibilidad para incorporar diferentes algoritmos y representaciones de datos, así como el de flexibilidad para desarrollar con eficacia aplicaciones de propósito específico a partir de otras más generales. El método de vórtice en celda, clasificado como modelo PM, se aplica a la dinámica de fluidos geofísicos según Velasco Fuentes (1994).

II. EQUIPO DE CÓMPUTO Y HERRAMIENTAS BÁSICAS PARA EL DESARROLLO DEL TRABAJO

La Tabla I muestra las características de las computadoras y los compiladores utilizados. Las dos primeras máquinas (**en negrillas**) fueron utilizadas durante el desarrollo de los programas, el resto fue utilizado para pruebas de portabilidad y tiempo de ejecución.

Tabla I. Plataformas de cómputo utilizadas durante el desarrollo del trabajo.

Computadora	Memoria RAM	Sistema Operativo	Compilador(es)
Pentium 133 MHz	**32 MB**	**Windows 95**	**BorlandC++ 5.01**
Pentium 133 MHz	**32 MB**	**Unix FreeBSD 2.1.7**	**g++ 2.7.1 y f77**
Pentium pro 200 MHz	32 MB	Unix FreeBSD 2.2	g++ 2.7.2 y f77
HP-900, 715/80	64 MB	HP-UX 9.07	c++ hp
IBM PowerPC	16 MB	AIX 3.2	g++ 2.7.0

III. MÉTODOS

Siguiendo el modelo de construcción de prototipos o versiones sucesivas de Pressman (1992), mostrado en la Figura 1, se partió de una versión Fortran (programa de aplicación inicial), enseguida se generó una versión C completamente estructurada (primer prototipo), y posteriormente, con orientación a objetos se construyó una aplicación C++ (segundo prototipo o programa de aplicación final).

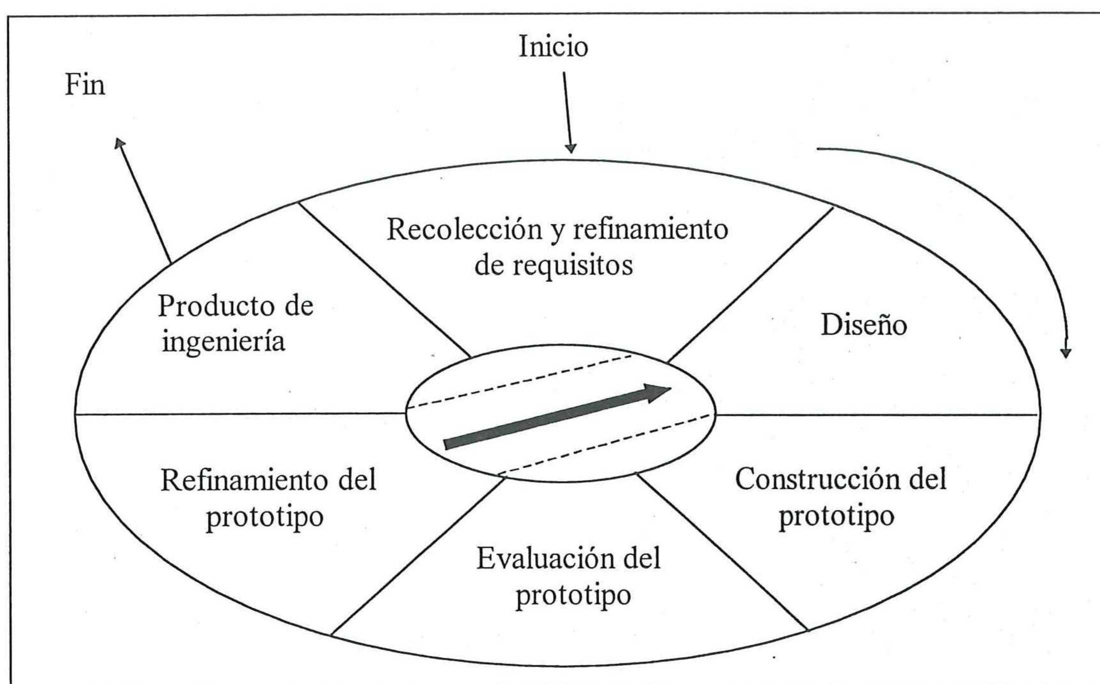


Figura 1. Modelo de desarrollo de aplicaciones por creación de prototipos.

Bajo este esquema es posible usar una metodología para análisis y diseño en cada paso de refinamiento, metodología interior, la cual puede ser diferente a la metodología de creación de prototipos o metodología exterior, de acuerdo a las necesidades de desarrollo que vayan surgiendo. Los pasos fueron los siguientes:

I. Programa de aplicación inicial:

   A. Método de vórtice en celda, según Velasco Fuentes (1994).

   B. Programa Fortran, tomado de Velasco Fuentes (1994).

II. Primer prototipo:

   A. Conversión del programa Fortran a un programa C completamente estructurado:

      1. Traducción.

      2. Ejecución y depuración.

      3. Optimización.

III. Segundo prototipo o programa de aplicación final:

   A. Análisis orientado a objetos, según Rumbaugh *et al.* (1991):

      1. Modelado de objetos.

      2. Modelado funcional.

      3. Modelado dinámico.

   B. Diseño orientado a objetos, según Rumbaugh *et al.* (1991):

      1. De sistemas.

      2. De objetos.

   C. Implementación orientada a objetos:

      1. Codificación C++.

IV. MODELO MATEMÁTICO

La base de esta sección es el trabajo de Velasco Fuentes (1994), del cual se hace un resumen de los capítulos referentes a la física de fluidos geofísicos y método de vórtices en celda.

IV.1 Vórtices Puntuales.

Los vórtices puntuales se definen como funciones delta en el plano. Un vórtice con fuerza κ_i ubicado en el punto (x_i,y_i) se mueve con una velocidad igual a la suma de las velocidades inducidas por los otros N-1 vórtices en el sistema. La evolución del conjunto de vórtices se obtiene resolviendo el conjunto de $2N$ ecuaciones diferenciales ordinarias

$$\frac{dx_i}{dt} = -\frac{1}{2\pi}\sum_{\substack{j=1 \\ j\neq i}}^{N} \kappa_j \frac{y_i - y_j}{r_{ij}^2}, \qquad (1)$$

$$\frac{dy_i}{dt} = \frac{1}{2\pi}\sum_{\substack{j=1 \\ j\neq i}}^{N} \kappa_j \frac{x_i - x_j}{r_{ij}^2}, \qquad (2)$$

donde r_{ij} es la distancia entre los vórtices i y j, para todos los valores de i desde 1 hasta N. Si un vórtice puntual representa un pequeño parche de vorticidad, entonces la circulación κ_i es igual a la vorticidad ω_i multiplicada por el área del parche a_i, esto es, $\kappa_i = \omega_i a_i$.

La conservación de *vorticidad potencial* implica que la *vorticidad relativa* ω de un vórtice tubular moviéndose en dirección meridional cambie en el plano β cumpliendo

$$\frac{D}{Dt}(\omega + \beta y) = 0, \qquad (3)$$

donde β es la variación latitudinal del parámetro de Coriolis y $\frac{D}{Dt} \equiv \frac{\partial}{\partial t} + (\bar{u} \cdot \nabla)$ es la derivada material en el plano (x, y), y en el plano-γ

$$\frac{D}{Dt}\left[\omega - \gamma\left(x^2 + y^2\right)\right] = 0 , \tag{4}$$

donde $\gamma = \frac{\Omega}{R^2}$, Ω es la velocidad angular y R es el radio de la Tierra.

Estas ecuaciones, además de la conservación de la masa, dan las ecuaciones de la modulación de la circulación en la forma

$$\kappa_j = \begin{cases} \kappa_{j0}, & \text{plano } f, \\ \kappa_{j0} + \beta a_j\left(y_{j0} - y_j\right), & \text{plano } \beta, \\ \kappa_{j0} - \gamma a_j\left(x_{j0}^2 - x_j^2 + y_{j0}^2 - y_j^2\right), & \text{plano } \gamma. \end{cases} \tag{5}$$

donde los subíndices '$j0$' denotan el valor inicial de alguna cantidad asociada al vórtice j, un solo subíndice denota una cantidad en cualquier otro tiempo, y las coordenadas (x,y) se pueden expresar como

$$x = R\theta \cos\phi_0, \qquad y = R(\phi - \phi_0). \tag{6}$$

donde θ es la longitud, ϕ es la latitud y ϕ_0 es la latitud de referencia.

IV.2 Método de vórtice en celda.

IV.2.1 Discretización.

El primer paso para calcular numéricamente la evolución de una distribución de vorticidad $\omega(x,y)$ con el método de vórtice en celda, es representarla por un conjunto finito de vórtices

puntuales. Supongamos que todo vórtice puntual representa un área a igual de fluido. Entonces, la *circulación* del vórtice se da por $\kappa_k = a\omega\,(x_k, y_k)$, donde (x_k, y_k) es la posición del vórtice. La determinación del número de vórtices y su ubicación inicial depende del fenómeno físico. Por ejemplo, si no hay gradientes de *vorticidad ambiental*, los vórtices puntuales se colocan en regiones del fluido con vorticidad diferente de cero, si es posible, se acomodan a lo largo de líneas de flujo. Por el contrario, si están presentes gradientes de vorticidad ambiental en el sistema, la vorticidad relativa puede ser generada sobre el dominio del flujo completo, aún si sólo una pequeña región contiene vorticidad inicialmente.

IV.2.2 Movimiento de los vórtices puntuales.

Una vez que se tiene el campo de vorticidad continua $\omega\,(x,y)$ representado por un conjunto finito de vórtices puntuales, la evolución temporal puede hacerse integrando numéricamente las Ecs. 1 - 2, siendo posible agregar la modulación de vórtices puntuales (Ec. 5). Sin embargo, cuando el número de vórtices N_p es grande, la integración de las ecuaciones de movimiento de este sistema consume mucho tiempo (el tiempo de computación es proporcional a N_p^2). Un esquema alternativo consiste en definir una malla donde el grupo de vórtices puntuales evoluciona y la vorticidad se obtiene por interpolación en los puntos de la malla; subsecuentemente la función de corriente y la velocidad se obtienen como en cualquier técnica estándar de malla. Por último, las velocidades de los vórtices puntuales se obtienen por interpolación del campo de velocidad. En este caso el

tiempo de computación es proporcional a N_p, si las dimensiones de la malla se mantienen constantes. A continuación se describen brevemente estos pasos para simular el movimiento de los vórtices puntuales en cada paso en el tiempo.

IV.2.2.1 Interpolación.

Supongamos que el conjunto de vórtices con vorticidad ω_k y coordenadas (x_k, y_k) cae dentro de una región rectangular cubierta por una malla cartesiana de dimensiones N_x x N_y y tamaño de celda Δx x Δy. Para obtener la vorticidad $\omega_{i,j}$ en el punto (i,j) de la malla, se puede usar el esquema de nube en celda ("cloud-in-cell") de cuatro puntos (CIC4), en el cual para asignar vorticidad a los cuatro puntos circundantes de la malla se supone que el vórtice puntual posee vorticidad uniforme ω_k dentro de un área rectangular $A = \Delta x$ x Δy. Cualquier punto alrededor recibe una cantidad de vorticidad proporcional al área del vórtice que cae dentro de esa celda, como se muestra en la Figura 2. Si las coordenadas de un vórtice puntual se dan como $x_k = i\Delta x + \delta x$ y $y_k = j\Delta y + \delta y$, los factores de interpolación de CIC4 serán

$$f_1 = (\Delta x - \delta x)(\Delta y - \delta y) / A \tag{7}$$

$$f_2 = \delta x (\Delta y - \delta y) / A, \tag{8}$$

$$f_3 = \delta x \, \delta y / A, \tag{9}$$

$$f_4 = (\Delta x - \delta x)\delta y / A, \tag{10}$$

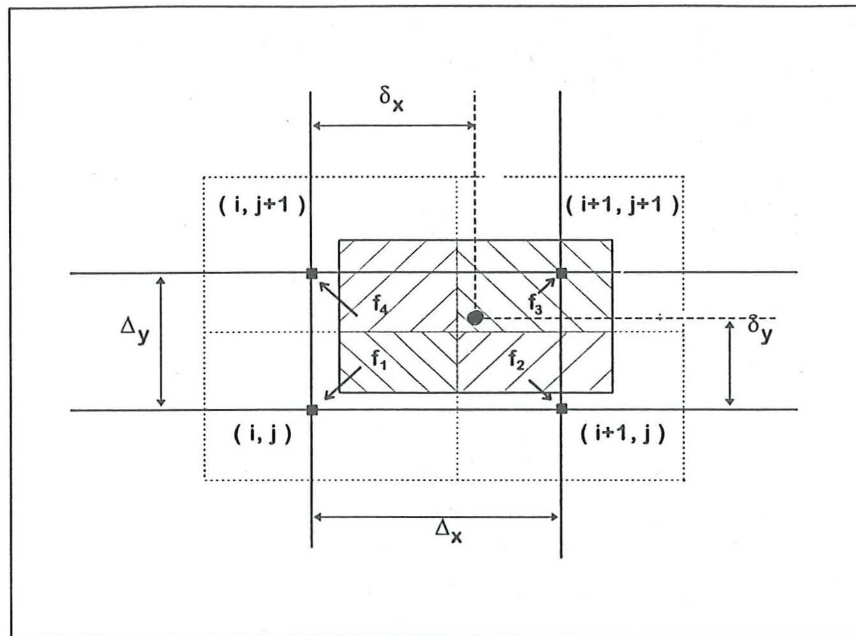


Figura 2. Esquema de interpolación nube en celda. El vórtice puntual (círculo relleno) representa un área igual a la celda rectangular D*x*D*y*. La intersección de estas áreas con las áreas que corresponden a los cuatro puntos más cercanos de la malla (cuadros con rayas), proporciona la fracción de vorticidad que se le atribuye a cada punto de la malla. Tomada de Velasco Fuentes, (1994).

y la vorticidad en la malla será

$$\omega_{i,j} = \frac{f_1 a}{A}\omega_k, \tag{11}$$

$$\omega_{i+1,j} = \frac{f_2 a}{A}\omega_k, \tag{12}$$

$$\omega_{i,j+1} = \frac{f_4 a}{A}\omega_k, \tag{13}$$

$$\omega_{i+1,j+1} = \frac{f_3 a}{A}\omega_k, \tag{14}$$

donde a es el área representada por cada vórtice puntual. La distribución de vorticidad sobre la malla se obtiene usando estas relaciones en cada vórtice (k = 1,...,N_p) del dominio de flujo.

IV.2.2.2 Función de corriente.

La función de corriente está relacionada con la vorticidad por la ecuación de Poisson $\nabla^2 \psi = -\omega$, la cual se discretiza como

$$\frac{\psi_{i,j+1} - 2\psi_{i,j} + \psi_{i,j-1}}{(\Delta y)^2} + \frac{\psi_{i+1,j} - 2\psi_{i,j} + \psi_{i-1,j}}{(\Delta x)^2} = -\omega_{i,j}, \tag{15}$$

y que puede resolverse con el método FACR agregando las condiciones de frontera apropiadas, asegurándose que las dimensiones de la malla sean potencias de 2, $N_x = 2^p$ y $N_y = 2^q$ donde p y q son números enteros. Este método permite los tres tipos de condiciones de frontera en cada dirección: (*i*) función prescrita ψ, (*ii*) ψ periódica y (*iii*) velocidad tangencial nula, pero velocidad normal diferente de cero.

IV.2.2.3 Velocidades.

El campo de velocidad, se evalúa a partir de la función de corriente, usando diferencias centradas en la forma

$$u_{i,j} = \frac{\psi_{i,j+1} - \psi_{i,j-1}}{2\Delta y}, \tag{16}$$

$$v_{i,j} = -\frac{\psi_{i+1,j} - \psi_{i-1,j}}{2\Delta x}. \tag{17}$$

La velocidad de cada vórtice puntual se determina a partir de las velocidades en los puntos más cercanos de la malla, que son los 4 puntos que definen la celda en la cual el vórtice puntual está ubicado.

Para calcular la velocidad en un punto se usan los mismos factores de interpolación con los cuales se obtuvo la vorticidad en la malla

$$u_k = f_1 u_{i,j} + f_2 u_{i+1,j} + f_3 u_{i+1,j+1} + f_4 u_{i,j+1}, \quad (18)$$

$$v_k = f_1 v_{i,j} + f_2 v_{i+1,j} + f_3 v_{i+1,j+1} + f_4 v_{i,j+1}. \quad (19)$$

IV.2.2.4 Integración en el tiempo.

Las nuevas posiciones de los vórtices puntuales se calculan resolviendo

$$x_k^{n+1} = x_k^{n-1} + 2\Delta t u_k^n, \quad (20)$$

$$y_k^{n+1} = y_k^{n-1} + 2\Delta t v_k^n, \quad (21)$$

donde el superíndice n denota el tiempo $n\Delta t$ y Δt es el paso en el tiempo.

IV.2.2.5 Nueva vorticidad relativa.

Conforme se va conociendo la nueva posición de los vórtices puntuales, se hace el cálculo de la nueva vorticidad relativa usando (5).

V. DESARROLLO

V.1 Conversión del programa Fortran al programa C completamente estructurado.

La asignación de memoria primaria en los programas generados por compiladores Fortran se hace bajo un esquema estático. Este hecho, aunque tiene ventajas, como la compartición de memoria entre subrutinas a través de bloques COMMON, impide que otros programas puedan utilizar la memoria que en ese momento no está siendo usada, y evita que el mismo programa pueda reasignar la memoria para darle un uso diferente del solicitado.

Otro esquema de asignación de memoria es el dinámico, el cual puede ser utilizado por programas generados en lenguaje C o C++, y a diferencia del esquema estático, permite solicitar la memoria conforme se va necesitando, así como la cantidad que se requiera, además se puede liberar la memoria una vez que se ha dejado de utilizar. El lenguaje C es más eficaz cuando se aplica al desarrollo de nuevos proyectos de software que utilizan un diseño estructurado. Sin embargo, muchos programas Fortran pueden beneficiarse con las extensiones de C o C++. Usar un compilador que soporte C++ al convertir un programa Fortran a C, permite que posteriormente los elementos de estructuramiento y orientación a objetos sean introducidos gradualmente.

Para iniciar la conversión se ha seleccionado la estrategia: "Primero hay que hacerlo funcionar, después hay que hacerlo que funcione mejor". Este proceso se puede dividir en tres etapas: *traducción:* se genera una primera versión en lenguaje C a partir del programa Fortran; *depuración y ejecución*: se genera una versión ejecutable libre de errores a partir de

la primera versión C; *optimización*: se genera una versión ejecutable optimizada para eficiencia en el uso de los recursos de cómputo.

V.1.1 Etapa de traducción.

Consiste en ir de expresiones Fortran a expresiones C, lo que se puede hacer de modo *directo* o *indirecto*. *Traducción directa* es cuando se elimina la sintaxis de un lenguaje y se aplica la del otro , así, si se tienen en Fortran las líneas

```
c   el comentario deseado
    IF(X.NE.Y)
    X=X+Y
    CALL SUBRUTINA
    ENDIF
```

se traducen a C como

```
/* el comentario deseado */
 if (x!=y) {
   x=x+y;
   subrutina( );
 }
```

Traducción indirecta es cuando se va más allá de la sintaxis del lenguaje; se trabaja con la estructura o flujo de ejecución del programa. Las líneas en Fortran

```
c       No tiene una estructura bien definida
        I=0
40      I=I+1
        IF(I.GT.N) GOTO 50
c       todo un bloque de expresiones (B1)
        GOTO 40
50      J=0
c       otro bloque (B2)
```

se traducen a C como

```
/* No es la única forma de darle la estructura */
 for (i=1;i<=N;i++) {
          /* todo el bloque B1 */
 }
 j=0;
/* el bloque B2 */
```

V.1.2 Etapa de depuración y ejecución.

La *depuración y la ejecución* forman un proceso cíclico. La *depuración* tiene dos fases, en la primera se eliminan todos los errores que permitan crear una versión ejecutable del programa y en la segunda, que involucra la ejecución del programa, se eliminan todos los errores al tiempo de ejecución. En la primera fase se tienen los errores de léxico, sintaxis y semántica, estos errores son relativamente fáciles de localizar; en cambio, en la segunda fase, la mayoría de los errores al tiempo de ejecución son más difíciles de localizar, ya que usualmente están en la lógica del programa.

V.1.3 Etapa de optimización.

La *optimización* consiste en usar las herramientas del lenguaje que permiten utilizar al máximo los recursos de cómputo. El lenguaje C cuenta con:

1) Apuntadores que permiten manipular las direcciones de memoria.
2) Funciones para el manejo dinámico de la memoria.
3) Llamadas a funciones optimizadas de biblioteca para realizar tareas comunes.

V.1.4 Resultados obtenidos en la fase de conversión de la versión Fortran a la versión C completamente estructurada.

La versión C optimizada utiliza asignación dinámica de memoria. Con ella, se generan los mismos archivos de salida que la versión Fortran, con los mismos nombres y datos. Las versiones C y Fortran presentan discrepancias entre algunos de los datos generados en la sexta cifra significativa, lo cual se atribuye a que en la versión C se utilizan variables de doble precisión y precisión sencilla para números flotantes, mientras que en la versión Fortran sólo se utilizan variables de precisión doble (real*8). En la Tabla II se muestra que los requerimientos de espacio en disco son menores en la versión C, siendo más notable esta diferencia para la versión ejecutable. También se puede notar que el tiempo de ejecución es menor para la versión Fortran. Así, con la traducción de Fortran a C se ha obtenido una mejora en la que respecta a la utilización de los recursos de almacenamiento del sistema (memoria y disco).

Tabla II. Comparación en la cantidad de espacio de disco requerido por los programas C y Fortran.

Espacio en disco	Versión Fortran	Versión C	Comparación
Programa(s) fuente	50844 bytes	42719 bytes	~ 16 % mejor
Programa ejecutable	245760 bytes	81920 bytes	~ 67 % mejor

En la Tabla III se muestra que es mejor usar arreglos dinámicos que estáticos. Esta comparación se hace con base en la estimación siguiente: la longitud de la tira de vórtices puntuales solicitada por el número de variables por el tamaño del tipo de dato (double) más

las dimensiones de la malla por el número de variables por el tamaño del tipo de dato (double). En este caso se toma una longitud máxima de 250,000 vórtices en la tira, 4,800 vórtices y una malla de 128 x 128.

Tabla III. Comparación de la cantidad de memoria primaria requerida por los programas.

Memoria Requerida	Versión Fortran Arreglos estáticos	Versión C Arreglos dinámicos	Comparación
tira	~13.7 Mb	~ 262.5 Kb	~ 98% mejor
malla	~512 Kb	~512 Kb	igual

En la tabla IV se encuentra una comparación de los tiempos de ejecución para el caso de prueba INI_TEST (ver apéndice C. Casos de Prueba). El tiempo de ejecución de la versión C es mayor que el tiempo de ejecución de la versión Fortran. Lo que indica que el compilador Fortran es mejor que el compilador C en la SPARC.

Tabla IV. Comparación de la cantidad de tiempo de ejecución de ambos programas.

Tiempo de ejecución	Versión Fortran	Versión C	Comparación
SPARC Server	10.28 seg.	11.25 seg.	~ 9.43 % más lento

V.2 Análisis orientado a objetos del método de vórtice en celda.

El análisis del método de vórtice en celda se realizó usando OMT de Rumbaugh *et al.* (1991) (véase apéndice B).

V.2.1 Establecimiento del problema.

Simular la evolución de vórtices bidimensionales con vorticidad ambiental aplicando el método de vórtice en celda en un ambiente orientado a objetos. El programa debe ejecutarse en varias plataformas de hardware (PC's, HP, SUN, IBM PowerPC) y software (MS-DOS, MS-Windows, UNIX) y generar los mismos resultados numéricos que el programa Fortran existente. Los archivos de entradas y salidas, así como sus formatos, deben ser totalmente compatibles con la versión Fortran existente. El programa resultante debe ser extensible y flexible tanto para modificación como para mantenimiento.

V.2.2 Construcción del modelo de objetos.

V.2.2.1 Identificación de clases de objetos en el método de vórtice en celda.

Las clases detectadas fueron las siguientes: vórtice puntual, tira de vórtices puntuales, malla, celda de malla y vórtice en punto de malla.

V.2.2.2 Asociaciones entre clases.

Se identifican y agregan las asociaciones entre las clases. Las *asociaciones* detectadas fueron las siguientes: el vórtice puntual *cae-dentro-de* una celda de malla, el vórtice puntual *proporciona-vorticidad-a-los-vórtices-en-punto-de-malla* en la celda de malla, las celdas de malla *generan-la-función-de-corriente* para los vórtices en los puntos de malla, las celdas de malla *proporcionan-velocidad* a los vórtices en los puntos de malla, el vórtice puntual *recibe-velocidades* de los vórtices en punto de malla de la celda específica, el vórtice

puntual *proporciona-la-nueva-posición* de los vórtices puntuales en la tira de vórtices puntuales y el vórtice puntual *proporcionan-la-nueva-vorticidad-relativa* a los vórtices puntuales en la tira.

V.2.2.3 Atributos de los objetos.

En seguida se deben de identificar los atributos de cada una de las clases, la Tabla V muestra estos atributos.

Tabla V. Clases identificadas con sus atributos respectivos.

Clase	**Atributos**
vórtice puntual	*vorticidad inicial, posición inicial, vorticidad, posición, velocidad.*
tira de vórtices puntuales	*longitud.*
malla	*dimensión en x (nΔx), dimensión en y (mΔy).*
celda de malla	*tamaño en x (Δx), tamaño en y (Δy).*
vórtice en punto de malla	*vorticidad, función de corriente, campo de velocidad.*

V.2.2.4 Especificación de herencia y agregación.

El siguiente paso consiste en organizar las clases para el diagrama de objetos y simplificar éste usando herencia y agregación. Al usar herencia, se obtiene la superclase *datos de la aplicación* (clase base) y dos clases derivadas: la subclase *tira de vórtices puntuales* y la subclase *malla*. Al usar agregación, se obtiene que la *tira de vórtices puntuales* está formada por un conjunto de *vórtices puntuales* y la *malla* está formada por un conjunto de *celdas*, que a su vez está formada por un conjunto de 4 *puntos de malla*.

La Figura 3 muestra el primer diagrama de objetos generado durante esta etapa.

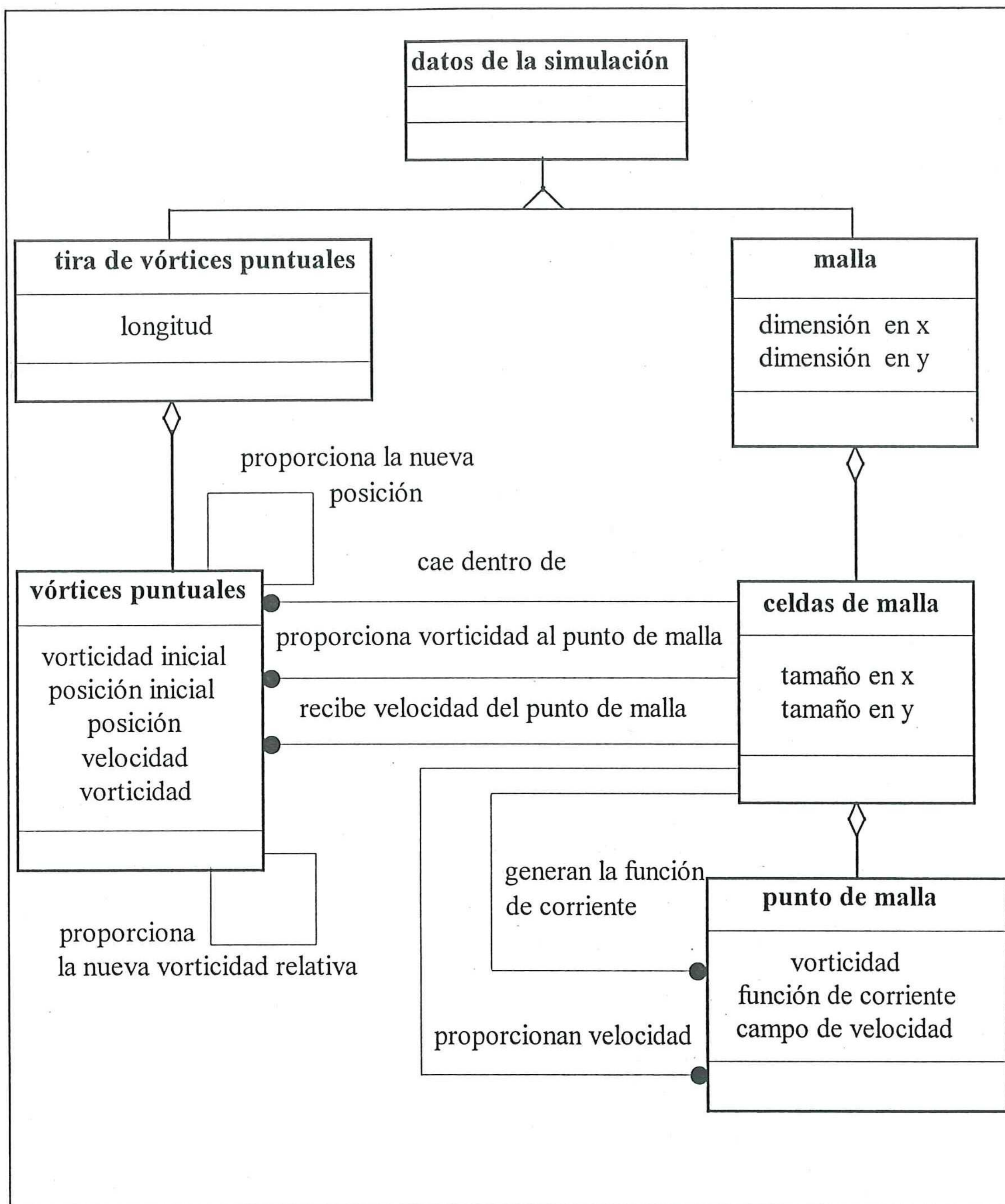


Figura 3. Modelo de objetos de las clases identificadas para los datos de la simulación, mostrando herencia y agregación.

V.2.3 Construcción del modelo dinámico.

V.2.3.1 Preparación de escenarios típicos.

El primer paso en la construcción de un modelo dinámico es la elaboración de escenarios para las secuencias de interacciones típicas.

A continuación se muestra un escenario típico sin errores.

Escenario 1 (sin condición de error)

1. El usuario proporciona las condiciones de inicialización (se dan la tira de vórtices, la malla, la modulación y la topografía).
2. Se inicializa la tira de vórtices puntuales (se dan la vorticidad inicial y la posición inicial).
3. Se inicializa la malla (se dan las dimensiones y el tamaño de la celda).
4. Se verifica el dominio de los vórtices puntuales sobre la malla.
5. La tira reparte la vorticidad a la malla.
6. La malla genera la función de corriente de los puntos de malla.
7. Las celdas proporcionan velocidad a los puntos de malla.
8. Las celdas proporcionan velocidad a los vórtices puntuales.
9. Se actualiza la posición de los vórtices puntuales.
10. Se actualiza la vorticidad de los vórtices puntuales.
11. Se calcula el nuevo paso.
12. Se escriben los resultados.

A continuación se muestra un escenario típico con error debido a partículas fuera del dominio computacional.

Escenario 2 (con condición de error)

1. El usuario proporciona las condiciones de inicialización (se dan la tira de vórtices, la malla, la modulación, la topografía).
2. Se inicializa la tira de vórtices puntuales (se dan la vorticidad inicial y la posición inicial).
3. Se inicializa la malla (se dan las dimensiones y el tamaño de la celda).
4. Se verifica el dominio de los vórtices puntuales sobre la malla.
5. Resulta una partícula fuera del dominio.
6. Se despliega el mensaje de error (partícula fuera de dominio computacional).
7. Termina el experimento.

V.2.3.2 Construcción de trazos de eventos.

El segundo paso consiste en la creación de trazos de eventos para cada uno de los escenarios. El trazo de eventos para el Escenario 1 se muestra en la Figura 4 y el trazo de eventos para el Escenario 2 se muestra en la Figura 5.

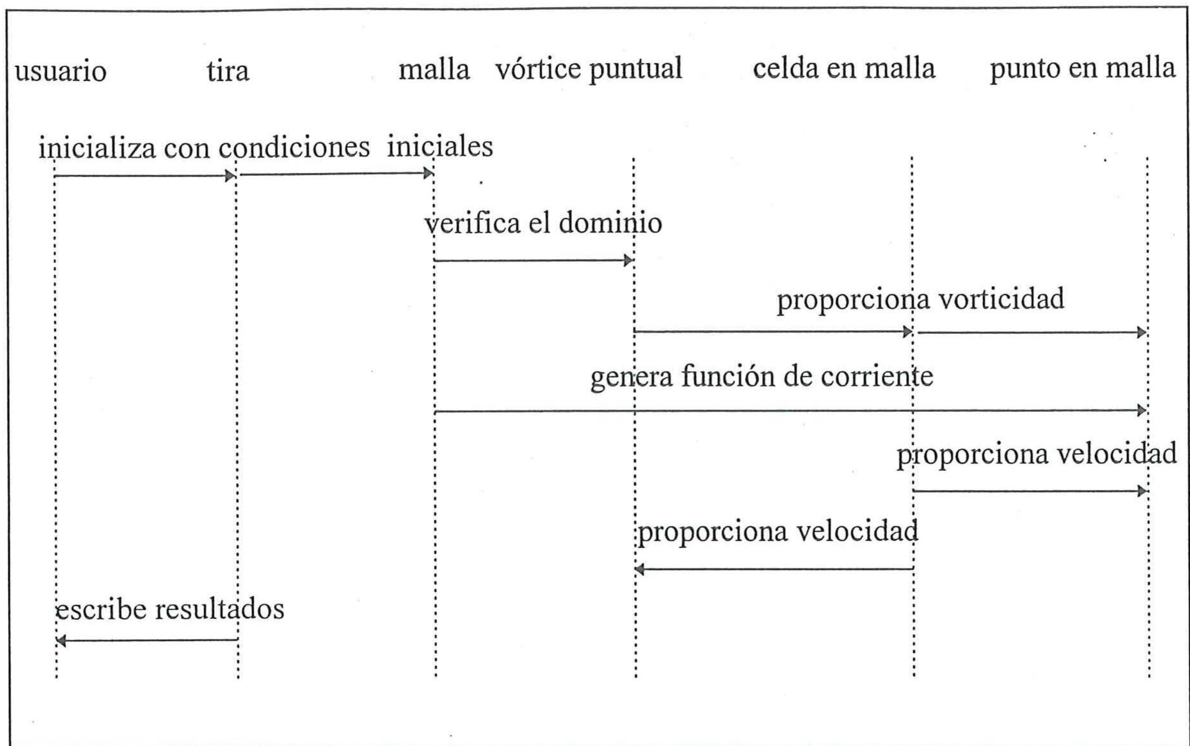


Figura 4. Trazo de eventos para el Escenario 1.

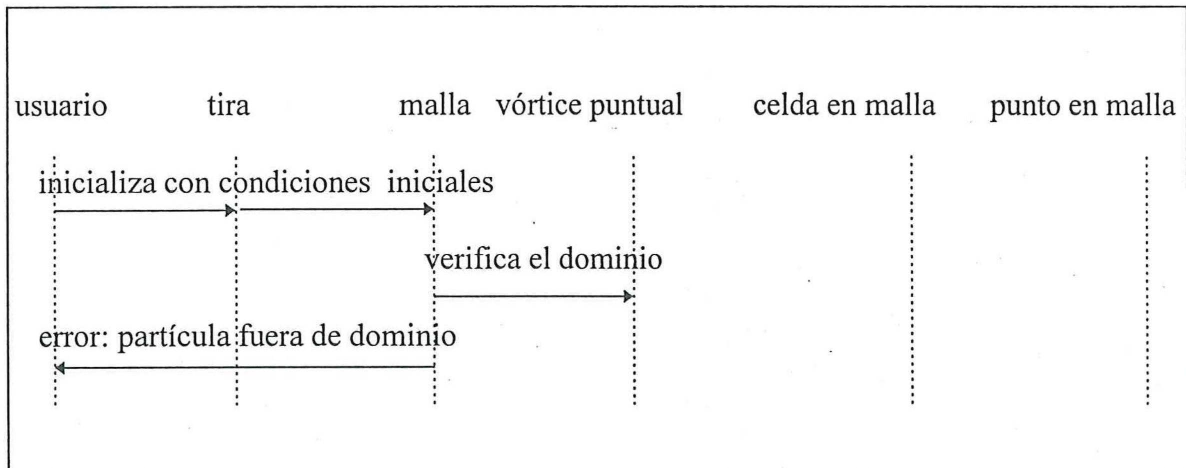


Figura 5. Trazo de eventos para el Escenario 2.

V.2.3.3 Construcción de diagramas de estados.

El último paso en el desarrollo del modelo dinámico es la representación del comportamiento a través de diagramas de estados. El diagrama de estados, así como el subdiagrama de estados para *Calcula las Velocidades* se muestran en las Figuras 6 y 7 respectivamente.

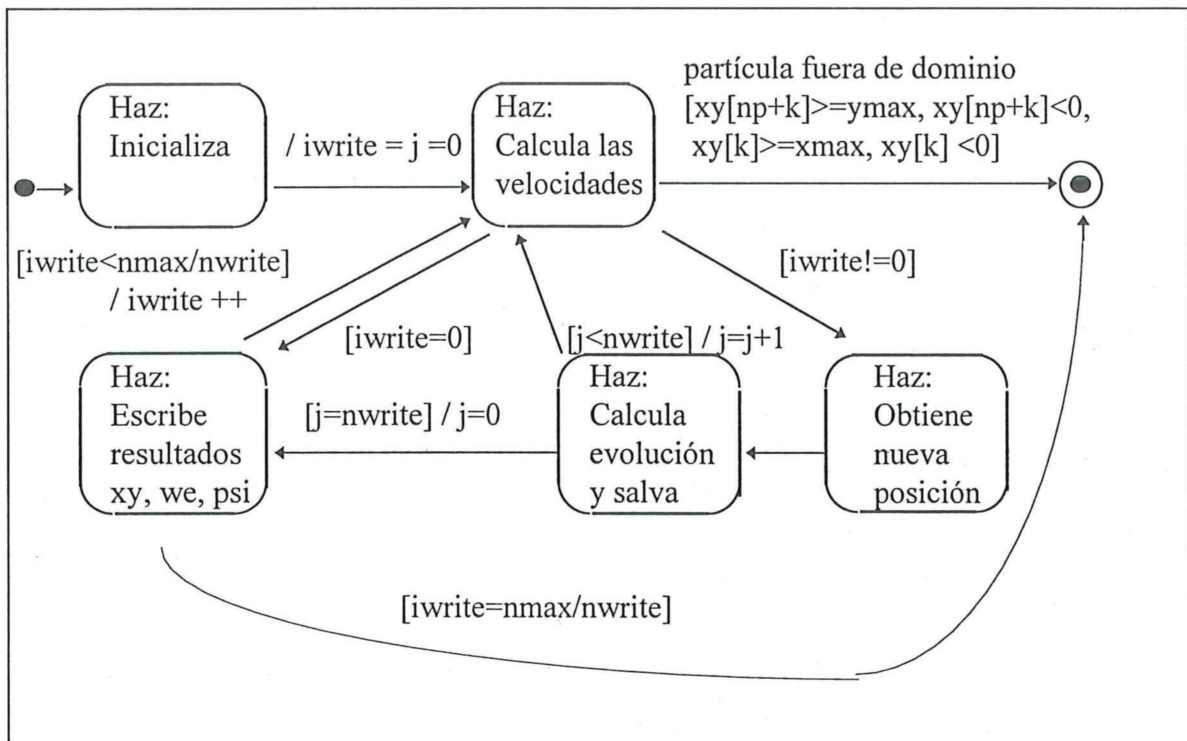


Figura 6. Diagrama de estados general para el método de vórtice en celda.

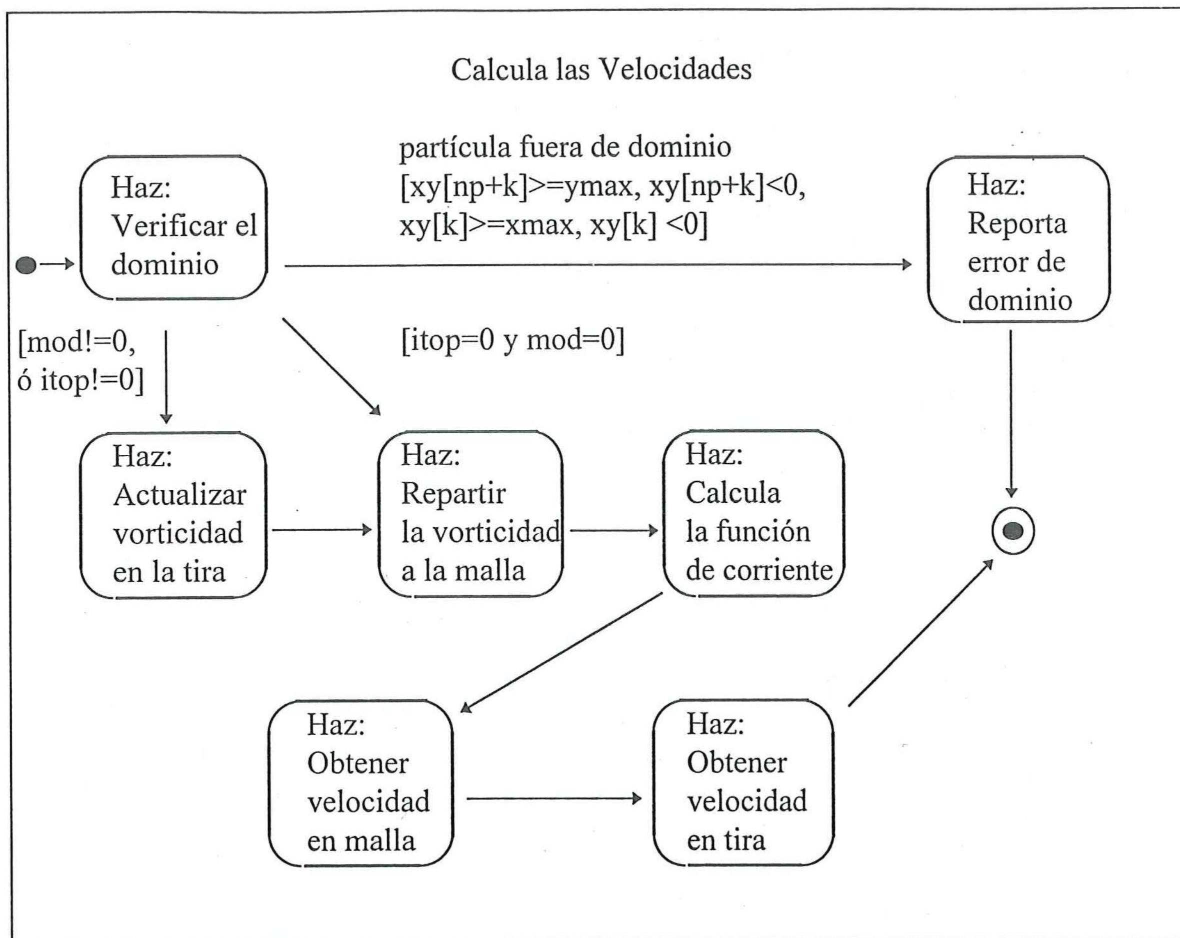


Figura 7. Diagrama de estados para "Calcula las Velocidades".

V.2.4 Construcción del modelo funcional.

V.2.4.1 Identificación de valores de entradas y salidas.

El primer paso en la construcción de un modelo funcional consiste en identificar los valores de entrada y de salida. Los valores identificados son:

a) Entradas. Son aquellos datos que son proporcionados, a través de eventos, al programa desde el exterior. Aquí, los valores de entrada identificados son:

Condiciones Iniciales:

- número de iteraciones para la simulación: (nmax),
- incremento en el tiempo: (dt),
- tamaño de la malla: ($n\Delta x$, $m\Delta y$),
- condiciones de frontera: (ibcx, ibcy),
- modulación y topografía: (mod, itop),
- datos del dipolo: (velocidad, dirección, centro y radio).

b) Salidas. Valores finales del programa de simulación hacia el exterior. Los valores de salida identificados son:

- vorticidad de los vórtices puntuales,
- posiciones de los vórtices puntuales: (x, y),
- vorticidad de los vórtices en punto de malla,
- función de corriente de los vórtices en punto de malla.

V.2.4.2 Los diagramas de flujo de datos.

Los diagramas de flujo de datos muestran cómo los valores de salida se calculan a partir de los valores de entrada. Los procesos detectados son los siguientes:

- Inicializar.
- Verificar el dominio.
- Actualizar la vorticidad.
- Repartir la vorticidad.
- Calcular la función de corriente.
- Obtener la velocidad en la malla.
- Obtener la velocidad en la tira.
- Calcular la nueva posición en la tira.

La Figura 8 muestra el diagrama de flujo de datos generado.

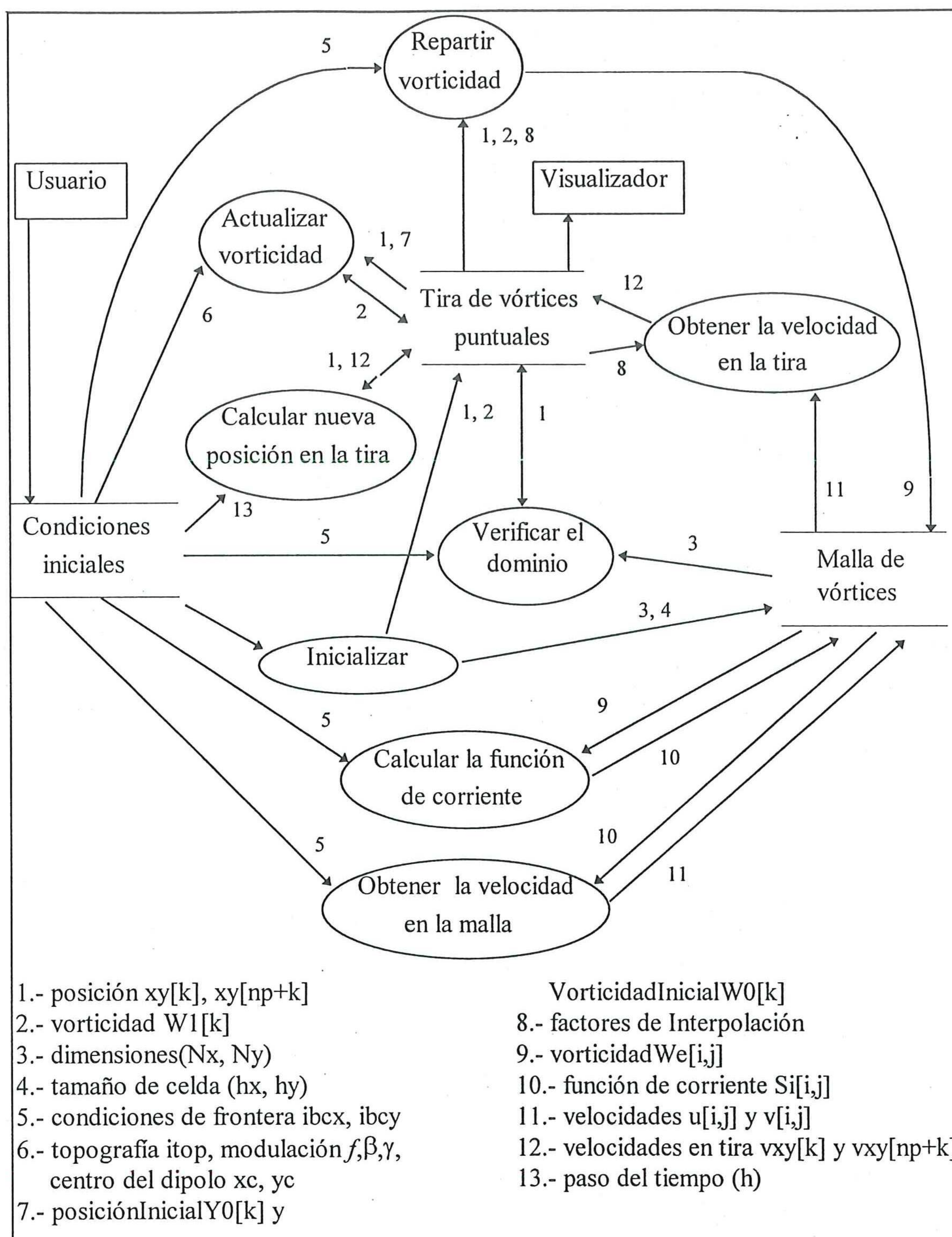


Figura 8. Diagrama de flujo de datos para los procesos detectados.

V.2.5 Afinación del modelo de objetos.

Una vez construida una versión preliminar de los modelos de objetos, dinámico y funcional, se pasa a la última fase del análisis, la cual consiste en iterar y afinar los tres modelos. Para ello es necesario agregar las operaciones identificadas (Tabla VI) durante la preparación del modelo funcional y verificar que los tres modelos sean consistentes.

Tabla VI. Operaciones detectadas con clases propietarias y parámetros.

Operación	Clase propietaria	Clase parámetro
Inicializa	tira	ninguna
Inicializa	malla	ninguna
Verifica el dominio	tira	malla
Actualiza la vorticidad	tira	ninguna
Reparte la vorticidad	tira	malla
Calcula función de corriente	malla	ninguna
Calcula velocidad	malla	ninguna
Obtiene velocidad	malla	tira
Calcula nueva posición	tira	ninguna

Las operaciones se deben agregar en el modelo de objetos. Sin embargo, se considera que las clases: vórtice puntual, tira de vórtices puntuales, malla, celda de malla y vórtice en punto de malla (conjuntos de datos) y las operaciones (métodos numéricos) pueden encapsularse por separado, como objetos individuales. Esta decisión se debe a consideraciones de implantación y se discutirá durante el diseño de objetos.

V.3 Diseño de sistemas.

El segundo paso en OMT es el diseño de sistemas, en el cual se define la estructura de alto nivel (arquitectura) del programa de simulación (sistema). En el caso presente, una parte de la arquitectura ya está definida implícitamente en el programa Fortran existente (requerimiento de compatibilidad total entre versiones), por lo que se procede a identificar y definir el resto de los elementos que conformarán la arquitectura completa.

V.3.1 Identificación de subsistemas.

La estructura propuesta divide el programa de simulación en 5 subsistemas:

1) Subsistema de la simulación (vórtice en celda).
2) Subsistema de los pasos y operaciones (métodos numéricos) de la simulación (repartir vorticidad, calcular función de corriente, obtener nueva posición del vórtice puntual, ...).
3) Subsistema de los datos (conjuntos de datos) de la simulación (tira de vórtices, malla).
4) Subsistema de las operaciones (métodos numéricos) básicos (interpolación bilineal, FACR, Runge-Kutta2, ...).
5) Subsistema de los datos (conjuntos de datos) básicos (escalar, vector, matriz).

Estos subsistemas se relacionan bajo el esquema cliente-proveedor, es decir, el subsistema cliente llama al proveedor, el cual ejecuta algún servicio sobre o para el cliente y devuelve un resultado.

El sistema está organizado en un esquema mixto de capas y particiones como se muestra en la Figura 9.

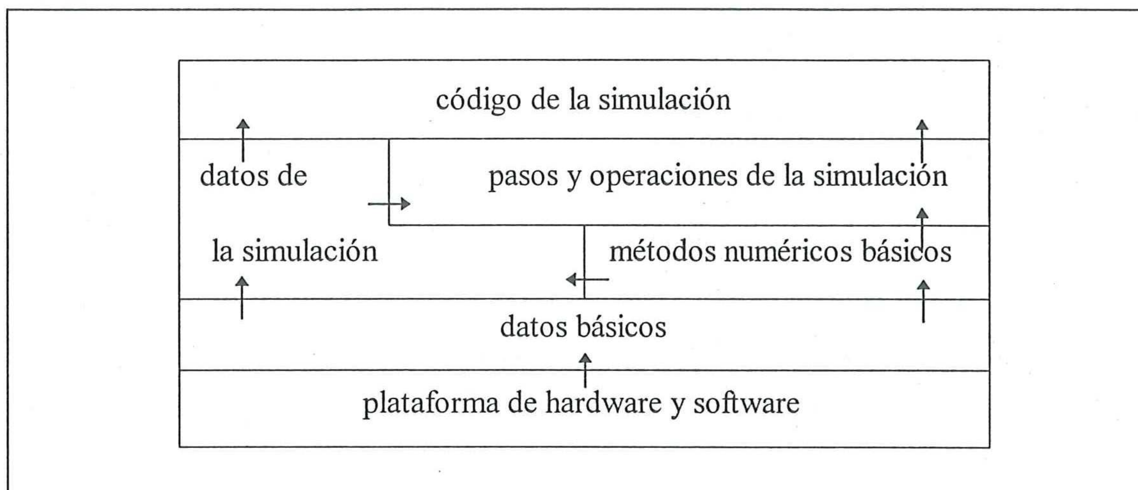


Figura 9. Diagrama a bloques de la organización del sistema en capas y particiones, las flechas denotan la relación cliente-proveedor.

Las capas o divisiones horizontales representan niveles, cada una de las cuales se construye con base en las capas debajo de ellas y proporciona las bases para la construcción de los que están por encima de ellas. Las particiones o divisiones verticales dividen verticalmente un sistema en varias partes independientes o débilmente acopladas, cada una de ellas proporcionando un tipo de servicio. Las relaciones cliente-proveedor entre capas son denotadas por flechas verticales y las relaciones cliente-proveedor entre particiones son denotadas por flechas horizontales. Las flechas parten del subsistema proveedor del servicio y las cabezas señalan al subsistema cliente.

V.3.2 Selección de la estrategia básica para implantar los almacenamientos de datos.

La selección de la estrategia básica para implantar el almacenamiento de los datos del programa de simulación es muy importante, ya que proporciona puntos de separación entre los subsistemas con interfaces bien definidas. La estrategia puede hacer uso de las estructuras de datos, los archivos y las bases de datos en forma combinada, para implantar cada uno de los almacenes de datos requeridos. Los almacenes de datos detectados en este trabajo requieren de dos esquemas de almacenamiento, cada uno con las siguientes características:

1. El primer esquema, necesario para almacenar los valores de la tira de vórtices puntuales y la malla usados en cada paso de la simulación, requiere que los datos se encuentren disponibles rápidamente, debido a la cantidad de accesos que estos tienen mientras se realizan los cálculos en la simulación. Este almacenamiento de datos puede ser reutilizado conforme se avanza en el proceso, es decir, aún cuando los valores sean desechados conforme se calcula el siguiente paso en la simulación, el espacio puede ser reutilizado por los nuevos valores calculados. Para este esquema se utilizan estructuras de datos en memoria primaria.
2. El segundo esquema, necesario para almacenar los resultados de cada paso para visualización posterior, requiere que puedan ser almacenados una gran cantidad de datos de manera permanente, los datos a almacenar son aquellos que dejan de ser utilizados al momento de ser reemplazados por los valores obtenidos en el paso siguiente de la simulación. A diferencia del primer esquema, el espacio de este

almacenamiento no puede ser reutilizado durante una misma simulación, ya que es necesario almacenar gradualmente los valores obtenidos conforme el proceso de simulación avanza, para este esquema se utilizan archivos de datos convencionales.

V.3.3 Selección del esquema para implantar el control del software.

Para este programa de simulación se ha seleccionado el esquema secuencial dirigido por los procedimientos, bajo el cual el control reside dentro del código del programa. Los eventos funcionan como operaciones alternadas de entrada y salida entre objetos (conjuntos de datos y algoritmos numéricos), por lo general después de cada par de operaciones de entrada y salida se obtienen nuevos resultados. Esta decisión depende de la naturaleza del modelo dinámico que se generó durante el análisis, ya que está dominado principalmente por las actividades que se realizan en cada uno de los estados, mientras que la mayoría de los eventos son de transición automática, es decir, se disparan al momento de concluir la actividad realizada en el estado.

V.3.4 Establecimiento de prioridades de decisión sobre características deseables del programa.

En esta etapa, se establecieron los criterios para definir las prioridades de las características deseables durante el desarrollo del programa de simulación contra las características deseables de la versión final ejecutable del mismo. Estos criterios de decisión incluyeron:

- optimización en el uso de los recursos contra rapidez de ejecución de la versión final ejecutable;

- funcionalidad del programa de simulación contra liberación del programa a tiempo;
- extensibilidad, portabilidad y claridad del diseño contra rapidez en la codificación de la primera versión.

Las características detectadas de mayor prioridad a partir del enunciado del problema incluyen:

1. Portabilidad: debe poder ejecutarse en una variedad de plataformas de hardware (PC's, HP, SUN, IBM PowerPC) y software (DOS, Windows, UNIX).
2. Buen comportamiento numérico: debe generar resultados correctos con base en la solución analítica del problema (cuando estos se conocen) o producir los mismos resultados correctos que la aplicación Fortran existente (probada con anterioridad).
3. Compatibilidad: los archivos de entradas y salidas, así como sus formatos, deben ser totalmente compatibles con la versión Fortran existente.
4. Facilidad de desarrollo y mantenimiento: el programa resultante debe ser extensible y flexible para modificación y mantenimiento.

Asimismo, un criterio que no debe perderse de vista en el desarrollo de aplicaciones de simulación, es la rapidez de ejecución, por lo que es importante identificar cual es el nivel aceptable de sacrificio en velocidad de ejecución, que puede ser compensado por ganancias en características como portabilidad, buen comportamiento numérico, compatibilidad y facilidad de desarrollo y mantenimiento.

V.4 Diseño de objetos.

En esta etapa se detalla el modelo de análisis, lo cual proporciona la base detallada a utilizar durante la implementación práctica en la computadora sin entrar en las particularidades de un lenguaje de programación.

V.4.1 Obtención de las operaciones para el modelo de objetos a partir de los otros modelos.

El primer paso en la obtención consiste en agregar al modelo de objetos las operaciones detectadas durante el modelado dinámico y funcional. Con base en el trabajo de Schroeder *et al.* (1996) se hacen las siguientes propuestas de diseño:

- El programa de simulación a desarrollar se trata como un conjunto de métodos o algoritmos numéricos que trabajan sobre conjuntos de datos del mismo.
- Se maneja un esquema donde las implementaciones de los algoritmos numéricos y los conjuntos de datos, ambos clasificados en básicos y de la simulación, se encapsulan por separado, como objetos individuales.
- Los métodos y datos de la simulación se refieren a los que están en el dominio de la misma, están formados por uno o más métodos o conjuntos de datos básicos.

En consecuencia, se requiere:

- Tener una clase base que encapsule los objetos genéricos que conforman el programa de simulación.

- Tener dos subclases, derivadas de los objetos genéricos, las cuales serán la base para las clases que se encarguen de implementar de manera general las operaciones o algoritmos numéricos y el manejo de los conjuntos de datos respectivamente.
- Crear objetos reutilizables con interfaces bien definidas con base en los conjuntos de datos básicos, que permitan desarrollar programas de simulación numérica específicos, flexibles, mantenibles y extensibles.

Lo anterior parece que va en contra de la programación orientada a objetos. Las consideraciones que llevan a proponer este esquema son:

- Los programas de simulación en cómputo numérico requieren de algoritmos complejos, los que, sí son combinados con los conjuntos de datos, darían como resultado objetos excesivamente grandes. Esto podría comprometer la simplicidad y la modularidad del diseño.
- Combinar los algoritmos numéricos y los conjuntos de datos daría como resultado una gran cantidad de código repetido, ya que la implementación de un algoritmo para diferentes tipos de datos usualmente difiere sólo en las regiones donde accesan los datos.
- De manera natural, los usuarios ven a los algoritmos numéricos como objetos que operan sobre otros objetos que son los conjuntos de datos.
- Los algoritmos o métodos y los datos pueden ser vistos como objetos individuales, ambos tienen características (atributos) y comportamiento (operaciones) propios que pueden ser encapsulados.

La jerarquía de algoritmos numéricos los clasifica con base en el comportamiento común, i.e., tipo de problema numérico que resuelve, y a los atributos y argumentos comunes, i.e., el modelado de datos que representa el problema o simulación numérica. La clase base de los métodos o algoritmos numéricos se muestra en la Figura 10.

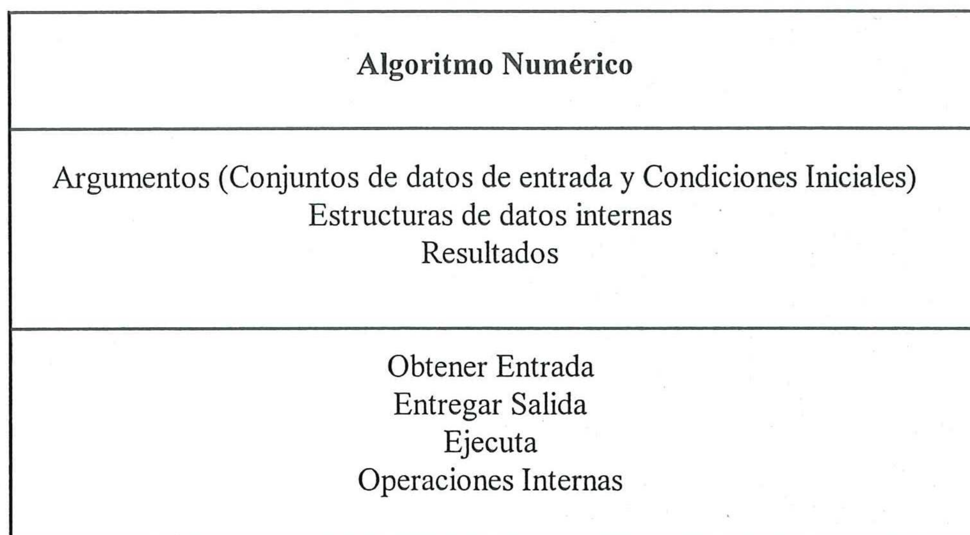


Figura 10. Diagrama de clase para Algoritmo Numérico.

Los atributos de los algoritmos numéricos son:

- Argumentos (interfaz estándar): condiciones iniciales del método y los conjuntos de datos de entrada que representan el modelado de datos del problema sobre el cual se aplica el algoritmo.
- Estructuras de datos internas: datos que son utilizados por el algoritmo para realizar su trabajo, usualmente son un superconjunto de los conjuntos de datos de entrada.
- Resultados (interfaz estándar): conjuntos de datos de salida, después de que el algoritmo numérico ha sido aplicado.

Las operaciones de los algoritmos numéricos son:

- ObtenerEntrada: operación de entrada de argumentos y verificación, opcionalmente puede convertirlos al formato requerido por la implementación del algoritmo.
- EntregarSalida: operación que entrega los resultados, opcionalmente puede convertirlos a algún formato requerido por el programa de simulación.
- Ejecuta: operación, que como su nombre lo indica, lleva la ejecución del algoritmo.
- OperacionesInternas: operaciones utilizadas internamente en la ejecución del algoritmo.

Los conjuntos de datos se clasifican de acuerdo al tipo de datos que se va a representar, es decir, va más allá de las representaciones clásicas de escalares, vectores y matrices. La clase base de los conjuntos de datos se muestra en la Figura 11.

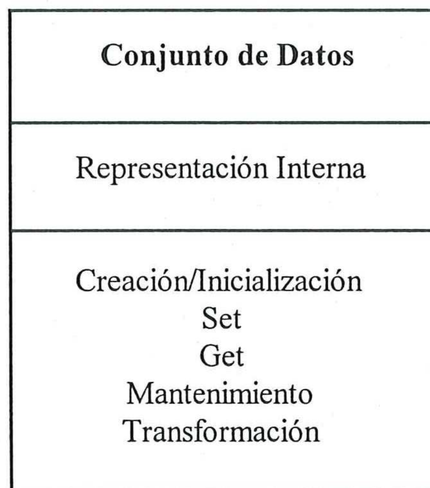


Figura 11. Diagrama de clase para Conjunto de Datos.

El atributo del conjunto de datos es:

- Representación Interna: comprende los elementos de datos que se utilizan internamente para la representación, usualmente son una combinación de los datos básicos y de otros conjuntos de datos.

Las operaciones de los conjuntos de datos son:

- Creación/Inicialización: operación que construye el conjunto de datos de acuerdo a la representación interna, puede asignar valores iniciales o por omisión.
- Set: operaciones que reciben los valores de los datos para ponerlos en la representación interna.
- Get: operaciones que entregan los valores de los datos de acuerdo a la representación interna.
- Transformación: operaciones que convierten de/hacia la representación interna (entrada/salida).
- Mantenimiento: el resto de las operaciones utilizadas en el manejo de la representación interna.

De acuerdo con lo anterior, el programa de simulación se construye con base en módulos con interfaces de entrada y de salida bien definidas (conjuntos de datos) y estándares para cada tipo de algoritmo y conjunto de datos. De esta forma, el modelo dinámico de un algoritmo numérico se muestra en la Figura 12.

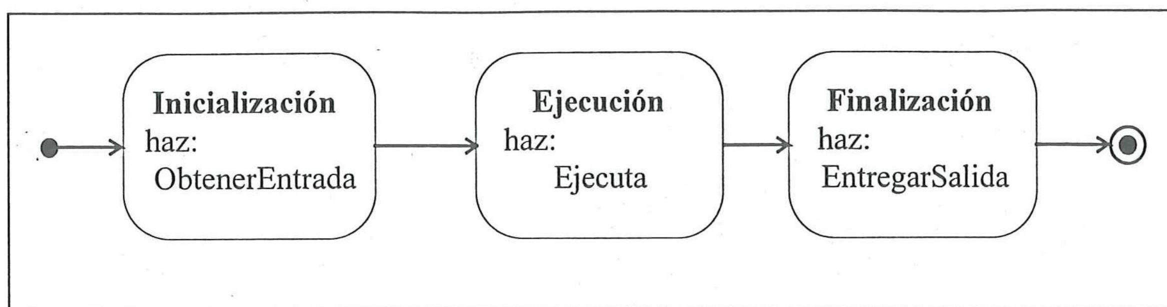


Figura 12. Modelo dinámico general de un algoritmo numérico.

Finalmente, el modelo funcional de un algoritmo numérico se muestra en la Figura 13.

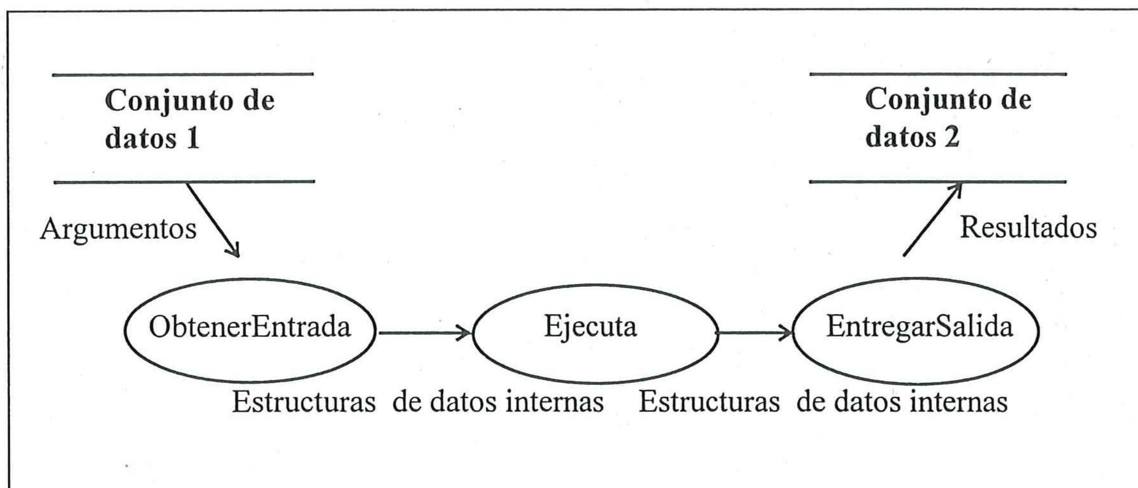


Figura 13. Modelo funcional general de un algoritmo numérico.

Con base en las propuestas de diseño planteadas, el modelo de objetos descrito en la Figura 3 debe ser modificado. El modelo de objetos resultante se muestra en las Figuras 14, 15 y 16.

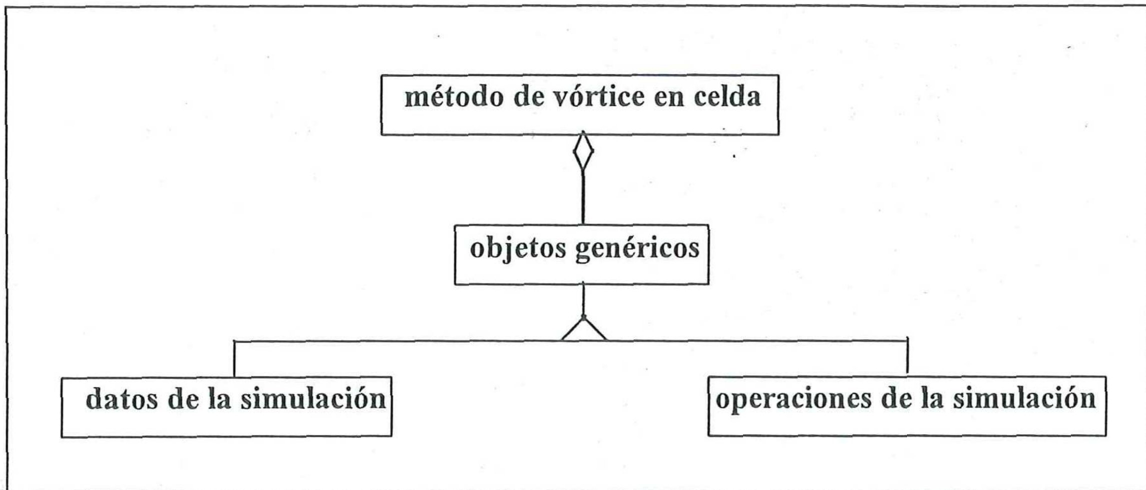


Figura 14. Diagrama de clases para el programa de simulación.

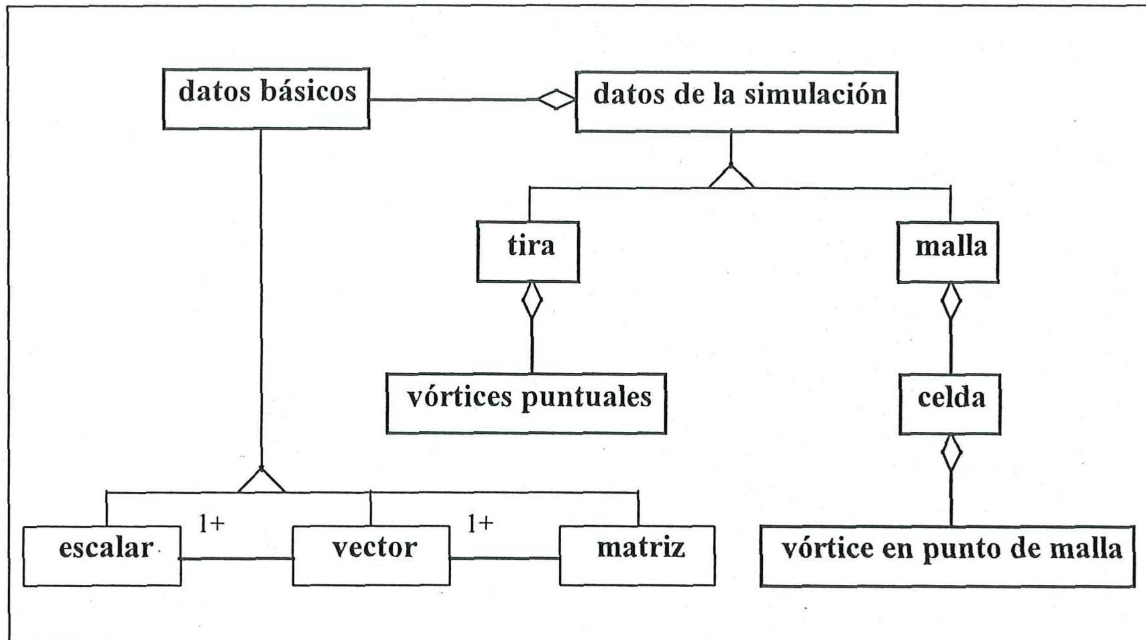


Figura 15. Diagrama de clases para los datos básicos y de la simulación.

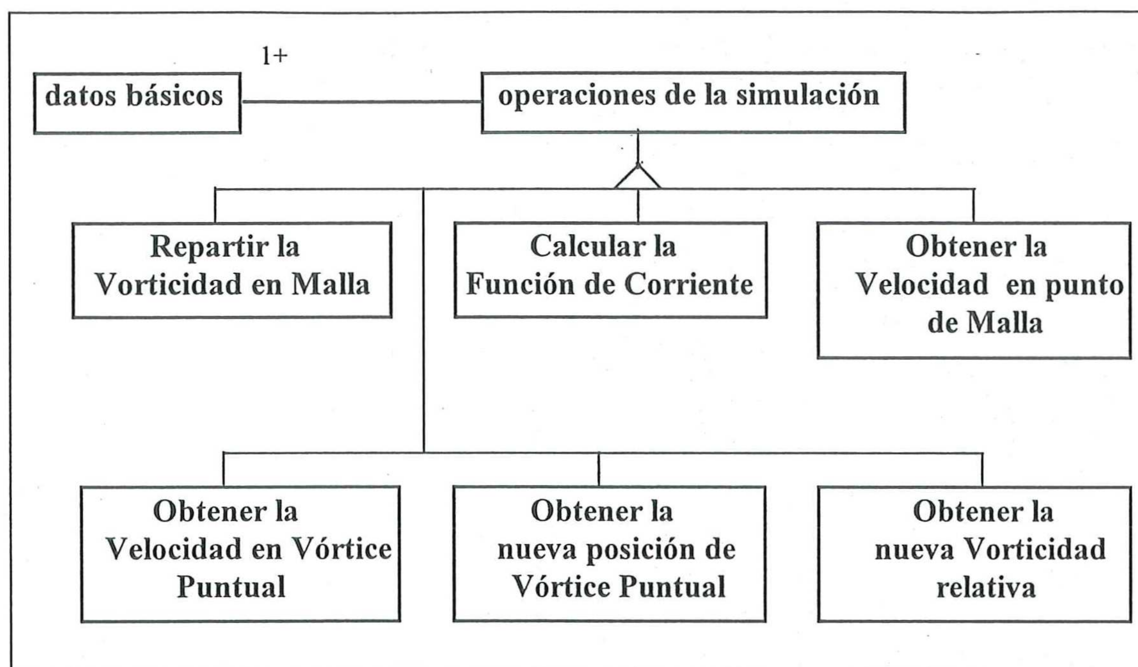


Figura 16. Diagrama de clases para las operaciones de la simulación.

V.4.2 Diseño de los algoritmos para implementar las operaciones.

Esta etapa consistió en identificar y seleccionar los algoritmos básicos con los cuales se construyen las pasos u operaciones de la simulación. Estos algoritmos se listan en la Tabla VII, ordenados con base en el paso específico que implementan.

Tabla VII. Algoritmos básicos identificados con el paso de la simulación que implementan.

Paso u operación de la simulación	Algoritmo básico especificado
Repartir la Vorticidad en la Malla	Interpolación bilineal
Calcular la función de corriente	Resolución de la Ec. de Poisson por el método FACR
Obtener la velocidad en la malla	Diferencias centradas
Obtener la velocidad en el vórtice puntual	Interpolación bilineal
Obtener la nueva posición del vórtice puntual	Runge-Kutta
Obtención de nueva vorticidad relativa	Método aplicando modulación y topografía

Dos observaciones importantes que deben considerarse al especificar estos algoritmos son: (1) que las estructuras de datos sobre las cuales trabajan se definen como arreglos en una y dos dimensiones y (2) que el nivel de abstracción en la especificación de los algoritmos es diferente, es decir, algunas se refieren a aplicación de algoritmos o variantes particulares como Runge-Kutta y FACR, mientras que otros se refieren a especificación de algoritmos numéricos generales como interpolación.

V.4.3 Implementación del flujo de control del software.

El pseudocódigo a primer nivel, que se obtiene directamente a partir del modelo funcional de la aplicación, se presenta en la Figura 17.

```
1. Inicializa (Entre otras cosas: número de escrituras (nwrite), número máximo de
   pasos (nmax))
2. iwrite = -1
3. Repite
      4. iwrite = iwrite + 1
      5. j =-1
      6. Repite
            7. j = j + 1
            8. Calcula las velocidades
            9. Si existe condición de error "Partícula fuera de dominio
               computacional" entonces
                  10. termina
            11. Si no
                  12. Si (iwrite = 0)
                        13. Escribe resultados
                        14. iwrite = iwrite + 1
                  15. Si no
                        16. Obtén nueva posición
                        17. Calcula evolución
      18. Hasta que j = nwrite
      19. Escribe resultados
20. Hasta que iwrite = nmax / nwrite
21. Termina
```

Figura 17. Pseudocódigo a primer nivel, generado a partir del modelo funcional.

Cabe mencionar que el pseudocódigo descrito en la Figura 17 no es la única manera de convertir el diagrama de flujo de datos mencionado, ya que pueden existir otras maneras

optimizadas de hacerlo. Sin embargo, se hace así con la finalidad de tener mayor claridad en el ejemplo.

V.4.4 Ajuste de la estructura de clases para incrementar herencia.

Con las propuestas de diseño, se generaron las nuevas clases: objetos genéricos, datos básicos y métodos básicos, que abstraen los comportamientos a un nivel más alto, para ello se identificaron atributos y operaciones comunes en las clases, de tal forma que la reestructuración permitiera incrementar la herencia. Sin embargo, también es necesario agregar clases para que los niveles de abstracción en las jerarquías de herencia resulten consistentes por niveles.

Se propuso tener jerarquías de herencia de los métodos básicos cuya superclase fuera el método básico y las subclases fueran las diferentes aproximaciones de solución (variantes) del mismo. La Figura 18 muestra esta jerarquía de manera general.

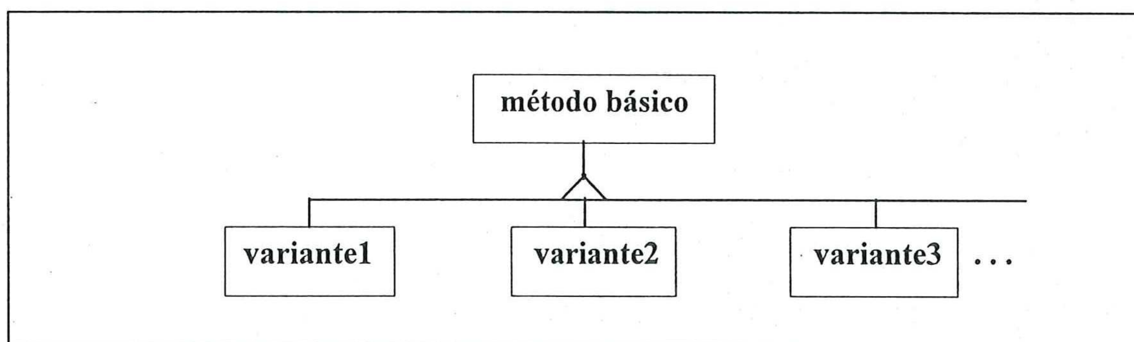


Figura 18. Jerarquía propuesta para las clases de métodos básicos.

La Figura 19 muestra el diagrama de clases resultante para las operaciones y métodos de la simulación aplicando la propuesta anterior.

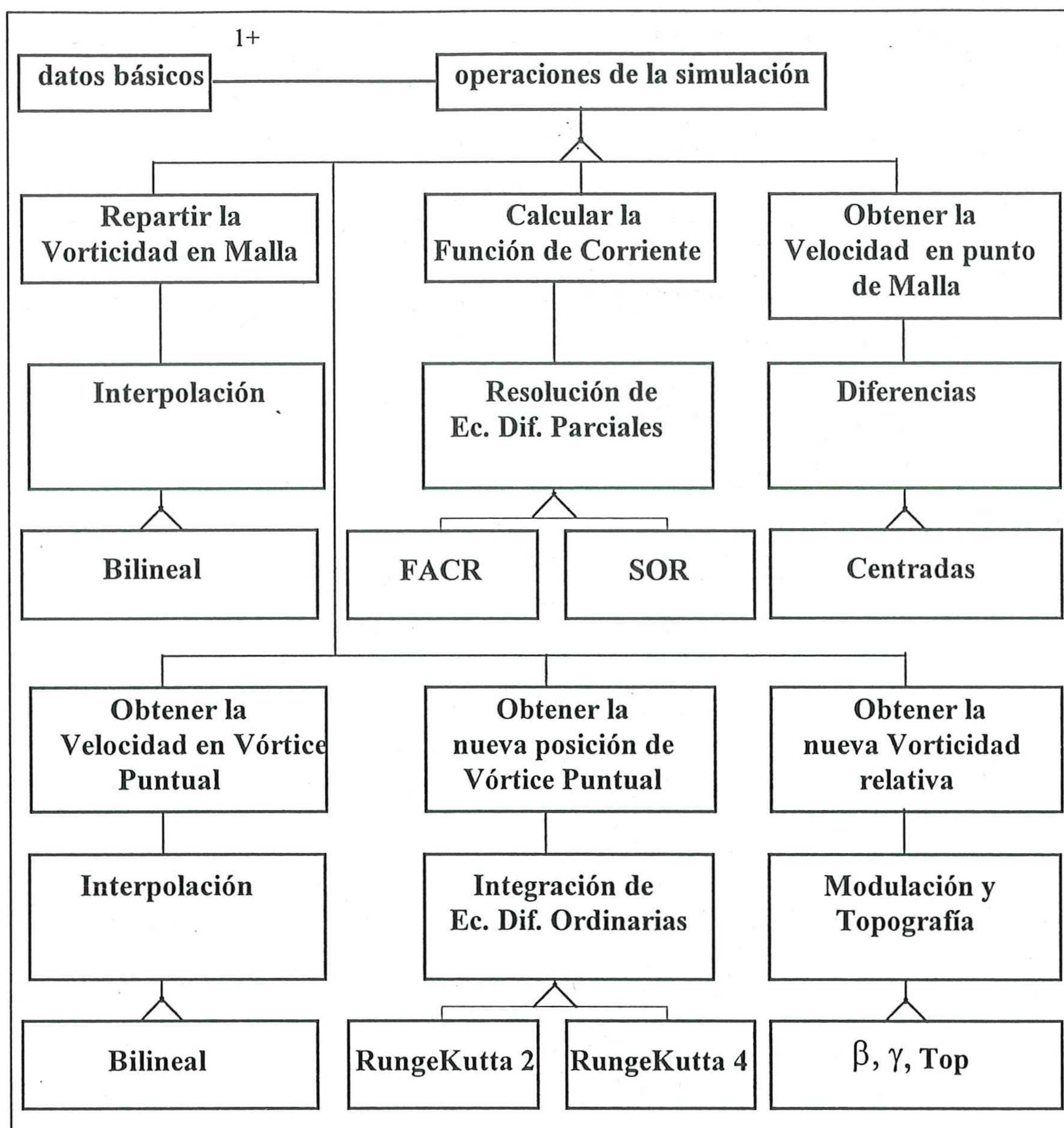


Figura 19. Diagrama de clases para las operaciones y métodos de la simulación aplicando la jerarquía de clases de métodos básicos.

V.4.5 Diseño de la implementación de las asociaciones.

Para implementar las asociaciones, éstas se clasificaron de acuerdo a la multiplicidad en sus extremos:

- *Asociaciones x-a-uno*: Estas asociaciones se implementaron como atributos en las clases de donde parte la asociación, a través del uso de apuntadores o referencias a instancias individuales de la otra clase.
- *Asociaciones x-a-muchos*: Este tipo de asociación requirió que en la clase que asocia al extremo muchos se tuviera un conjunto de apuntadores o referencias (incluso arreglos y listas).

La mayoría de las asociaciones que se detectaron en la etapa de análisis, se modelaron como clases de métodos.

V.4.6 Determinación de la representación de los atributos de los objetos.

Las clases de los conjuntos de datos para la simulación se reducen a Tira de Vórtices Puntuales y a Malla de Vórtices, debido a:

- Los algoritmos trabajan en el dominio de arreglos 1D y 2D (un tipo de dato estructurado proporcionando directamente por los lenguajes de programación).
- Por cuestiones de eficiencia y claridad del código, el acceso a elementos individuales es más rápido en una clase con arreglos internos, que en un arreglo de estructuras o clases.
- Esto permite usar vectores y matrices (tipo de dato estructurado predefinido o clases) directamente para representar cada uno de los conjuntos de datos.

Las Figuras 20 y 21 muestran la representación de los atributos de la tira de vórtices puntuales y la malla de vórtices respectivamente.

TiraVortices	
longitud:	entero
vorticidad inicial:	vector
posición inicial:	vector
posición x:	vector
posición y:	vector
velocidad x:	vector
velocidad y:	vector
vorticidad:	vector

Figura 20. Representación de la tira de vórtices puntuales mostrando los atributos.

MallaVórtices	
dimensión en x:	entero
dimensión en y:	entero
vorticidad:	matriz
función de corriente:	matriz
campo de velocidad x:	matriz
campo de velocidad y:	matriz

Figura 21. Representación de la malla de vórtices mostrando los atributos.

V.4.7 Empaquetamiento de las clases y las asociaciones en módulos.

Para finalizar con el diseño, se definió el empaquetamiento de clases y asociaciones, el cual se basa en la estructura propuesta para las clases de Algoritmo Numérico y Conjunto de Datos de las Figuras 10 y 11 respectivamente, y las asociaciones se empaquetan de acuerdo a la propuesta del diseño de la implementación de las asociaciones. El empaquetamiento externo por unidades o programas fuente resultante, se enlista en la Tabla VIII.

Tabla VIII. Empaquetamiento de clases y asociaciones por unidades.

Programa	Contenido
Aplicaci[h, cpp]	Código del programa de simulación del método de vórtice en celda.
Datosgen[h, cpp]	Clase base de datos genéricos (Datos)
Integedo[h]	Clase base para la jerarquía de integradores de ecuaciones diferenciales ordinarias
Mallavor[h, cpp]	Clase malla de vórtices (MallaVortices)
Metodcic[h, cpp]	Clases de métodos de aplicación CIC (MetodosCIC)
Metodgen[h, cpp]	Clase base de métodos genéricos (Métodos)
Matriz[h, cpp]	Clase matriz (Matriz)
Objetgen[h, cpp]	Clase base para los objetos de la aplicación (Objetos)
Poissol[h, cpp]	Clase de método para resolución de Ecuación de Poisson (PoissonSolver)
Rungeku2[h, cpp]	Clase para el integrador Runge-Kutta orden 2, derivada de IntegradorEDO.
Rungeku4[h, cpp]	Clase para el integrador Runge-Kutta orden 4, derivada de IntegradorEDO.
Tiravort[h, cpp]	Clase tira de vórtices puntuales(TiraVortices)
Vector[h, cpp]	Clase vector (Vector)

V.5 Implementación.

V.5.1 Los tipos de datos básicos: Vector y Matriz.

Estas clases representan la interfaz común (estándar) sobre la cual los métodos básicos (genéricos) trabajan. Con el fin de tener una biblioteca con las siguientes características:

1. Contar con un conjunto de datos básicos (vector y matriz, además de escalar) previamente probado, portable y extensible.
2. Probar la factibilidad de construir los conjuntos de datos del programa de simulación con base en conjuntos de datos básicos predefinidos.
3. Probar la fortaleza del modelo de objetos para los conjuntos de datos del programa de simulación.

se seleccionó la biblioteca Newmath, versión 09, de Davies (1997). Las características que motivaron la selección de la misma incluyen:

1. Condiciones de uso y disponibilidad de código fuente: el autor da su consentimiento para uso y distribución libre de costo. La biblioteca se encuentra disponible a través de Internet, en su versión de código fuente.
2. Variedad de tipos de matrices: soporta matrices de tipo rectangular, triangular superior, triangular inferior, diagonal, simétrica, banda, banda superior, banda inferior, banda simétrica, vector renglón y vector columna.
3. Variedad en los tipos de datos: soporta componentes reales de precisión sencilla y doble precisión.

4. Variedad de operaciones: incluye operaciones de multiplicación, suma, resta, concatenación, transpuesta, conversión entre tipos, submatrices, determinantes, descomposición de Choleski, triangularización QR, descomposición de valor singular, eigenvalores de una matriz simétrica, ordenamiento, transformada rápida de Fourier, impresión e interfaz para códigos del "Numerical Recipes in C" (Press *et al*, 1988).
5. Portabilidad: funciona sobre diversas plataformas de hardware con los siguientes compiladores, AT&T C++, Borland C++, GNU G++, HPUX CC, Microsoft C++, SUN CC, Watcom C++ y Zortech C++.
6. Documentación: Incluye una muy buena documentación, tanto en formato de texto, como en formato HTML.

V.5.2 Los objetos genéricos del programa de simulación.

La implementación de la clase base para la jerarquía de objetos del programa de simulación se muestra a través del archivo de encabezado *objetgen.h* en la Figura 22.

V.5.3 Los tipos de datos de la simulación.

Los conjuntos de datos que se requieren son:

- tira de vórtices puntuales,
- malla de vórtices en punto de malla.

Ambas clases se implementaron con los conjuntos de datos básicos vector y matriz. En la Figura 23 se muestran porciones del archivo de encabezado para la clase Datos, mientras

que en las Figuras 24 y 25 se muestran porciones de los archivos de encabezado para las clases TiraVortices y MallaVortices respectivamente.

```
// ---------------------------------------------------------------------------
// Nombre del Archivo: objetgen.h
// Descripción: Definición de clase de Objetos Genéricos para la aplicación
// Autor: ALMYS 11/03/97
// ---------------------------------------------------------------------------

#ifndef OBJETOS_GENERICOS_H
#define OBJETOS_GENERICOS_H

#include <iostream.h>
#include <stdio.h>

// ---------------------------------------------------------------------------
// Nombre de la clase: Objetos
// Descripción: Clase base para encapsular los diferentes conjuntos de objetos
//              (Datos y Métodos) para modelar la aplicación
// ---------------------------------------------------------------------------
class Objetos {
protected:
  char nombre[40];
  public:
  Objetos(const char* name);
  ~Objetos();
  void identifica() { cout << nombre << endl; }
};

#endif
```

Figura 22. Archivo de encabezado para la clase de los Objetos de la aplicación.

```
// ---------------------------------------------------------------------------
// Nombre del Archivo: datosgen.h
// Descripción: Definición de clase de Datos Genéricos
// Autor: ALMYS 11/03/97
// ---------------------------------------------------------------------------

#ifndef DATOS_GENERICOS_H
#define DATOS_GENERICOS_H

#include "objetgen.h"

// ---------------------------------------------------------------------------
// Nombre de la clase: Datos
// Descripción: Clase base para encapsular los diferentes conjuntos de datos
//              a utilizar en la aplicación : Derivada de Objetos
// ---------------------------------------------------------------------------
class Datos : public Objetos {
 public:
 Datos(const char* name);
 ~Datos();
 void identifica();
 friend class Métodos;            // Para permitir el acceso directo desde los métodos
};

#endif
// Fin de datosgen.h
```

Figura 23. Archivo de encabezado para la clase Datos genéricos, la cual se deriva de la clase de Objetos genéricos.

```
// ----------------------------------------------------------
// Nombre del Archivo: tiravort.h
// Autor: ALMYS '97, CICESE - UABC
// ----------------------------------------------------------
#ifndef TIRAVORT_H
#define TIRAVORT_H
// ----------------------------------------------------------
//        Nombre de la clase : TiraVortices
//
//        Descripción :
// ----------------------------------------------------------
class TiraVortices : public Datos {
 private:
// ----------------------------------------------------------
// Representación interna de los datos
// ----------------------------------------------------------
   int longitud;
   RowVector vorticidadInicialW0;
   RowVector posicionInicialY0;
   RowVector posicionX;
   RowVector posicionY;
   RowVector velocidadX;
   RowVector velocidadY;
   RowVector vorticidadW1;
   RowVector hb;
// ----------------------------------------------------------
// Datos compartidos por todas las tiras
// ----------------------------------------------------------
   int npmax;      // Longitud máxima de la tira, 250000L
// ----------------------------------------------------------
 public:
// ----------------------------------------------------------
// Operaciones
// ----------------------------------------------------------
// ----------------------------------------------------------
```

Continúa en la siguiente página

Figura 24. Archivo de encabezado para la clase TiraVortices, la cual se deriva de la clase de Datos genéricos.

```
// constructores                                    Viene de la página anterior
//-----------------------------------------------------------
   TiraVortices(const char* name = "tira de vórtices");
   TiraVortices(int lon,const char* name = "tira de vórtices");
   TiraVortices(const TiraVortices& tira);
//-----------------------------------------------------------
// destructor
//-----------------------------------------------------------
   ~TiraVortices();
//-----------------------------------------------------------
// misceláneas
//-----------------------------------------------------------
   void Redimensiona(int lon);
   void imprimepXpYV1(const char* fname, int ancho, int presicion);
   friend ostream& operator << (ostream& os, TiraVortices& tira);
//-----------------------------------------------------------
// set y get
//-----------------------------------------------------------
  int GetNpMax() { return npmax; }
  int GetLongitud() { return longitud; }
  RowVector& GetVectorVorticidadInicialW0();
  RowVector& GetVectorPosicionInicialY0();
  RowVector& GetVectorPosicionX();
  RowVector& GetVectorPosicionY();
  RowVector& GetVectorVelocidadX();
  RowVector& GetVectorVelocidadY();
  RowVector& GetVectorVorticidadW1();
//-----------------------------------------------------------
// Clases friend para permitir el acceso a los datos de la clase
//-----------------------------------------------------------
  friend class MetodosCIC;
  friend class RungeKutta2;
  friend class RungeKutta4;
//-----------------------------------------------------------
};
#endif
// Fin del bloque de tiravort.h
```

Figura 24 (*cont.*). Archivo de encabezado para la clase TiraVortices, la cual se deriva de la clase de Datos genéricos.

```
// ----------------------------------------------------------
// Nombre del Archivo: mallavor.h
// Autor: ALMYS '97, CICESE - UABC
// ----------------------------------------------------------
#ifndef __MALLAVOR_H__
#define __MALLAVOR_H__
//----------------------------------------------------------
//        Nombre de la clase : MallaVortices
//
//        Descripción :
//----------------------------------------------------------
class MallaVortices : public Datos {
 private:
// ----------------------------------------------------------
// Representación interna de los datos
// ----------------------------------------------------------
   int dimensionNx;
   int dimensionNy;
   Matrix vorticidadWe;
   Matrix funcionCorrienteS;
   Matrix campoVelocidadU;
   Matrix campoVelocidadV;
// ----------------------------------------------------------
// Datos para todas las mallas
// ----------------------------------------------------------
  int nxmax;                    // Dimensión máxima en x
  int nymax;                    // Dimensión máxima en y
 public:

// ----------------------------------------------------------
// Operaciones
// ----------------------------------------------------------
// ----------------------------------------------------------
// constructores
// ----------------------------------------------------------
  MallaVortices(const char* name = "malla de vórtices");
```

Continúa en la siguiente página

Figura 25. Archivo de encabezado para la clase MallaVortices, la cual se deriva de la clase de Datos genéricos.

Viene de la página anterior

```
  MallaVortices(int dimNx, int dimNy, const char* name = "malla de vórtices");
  MallaVortices(const MallaVortices& malla);
// ------------------------------------------------------------
// destructor
// ------------------------------------------------------------
  ~MallaVortices();

// ------------------------------------------------------------
// misceláneas
// ------------------------------------------------------------
  void Redimensiona(int dimNx, int dimNy);
  friend ostream& operator << (ostream& os, MallaVortices& malla);
  void ImprimeVorticidad(const char* fname, int ancho, int precision);
  void ImprimeFuncionCorriente(const char* fname, int ancho, int precision);
// ------------------------------------------------------------
// set y get
// ------------------------------------------------------------
  int GetNxMax() { return nxmax; }
  int GetNyMax() { return nymax; }
  int GetDimensionNx() { return dimensionNx; }
  int GetDimensionNy() { return dimensionNy; }
  Matrix& GetMatrizVorticidadWe();
  Matrix& GetMatrizFuncionCorrienteS();
// ------------------------------------------------------------
// Clases friend para permitir el acceso a los datos de la clase
// ------------------------------------------------------------
  friend class MetodosCIC;
  friend class RungeKutta2;
  friend class RungeKutta4;
// ------------------------------------------------------------
};
#endif
// Fin del bloque de mallavor.h
```

Figura 25 (*cont.)*. Archivo de encabezado para la clase MallaVortices, la cual se deriva de la clase de Datos genéricos.

V.5.4 Clases de Métodos.

Debido a que ya se contaba con todos los métodos codificados en C ("legacy code"), el esquema seguido para la implantación de los métodos fue envolturas de código ("code wrapper"). De tal forma que únicamente fue necesario envolver el código existente para que pudiera ser incrustado en la nueva implementación del programa.

El código agregado corresponde a las operaciones ObtenerEntrada y EntregarSalida del modelo de objetos de los métodos o algoritmos numéricos. Es decir, se le agregó el comportamiento necesario para que pudiera obtener los conjuntos de datos de la simulación, ponerlos en el formato requerido por su implementación particular y para convertirlos nuevamente al formato requerido por el programa después de haberlos procesado.

Los métodos están agrupados en la clase Métodos (de los métodos genéricos) de la cual se derivan las clases MetodosCIC , PoissonSolver e IntegradorEDO. De este último, a su vez, se derivan las clases RungeKutta2 y RungeKutta4.

A continuación, las Figuras 26 a la 31, muestran fragmentos de los archivos de encabezado para cada una de las clases.

```
// ---------------------------------------------------------------------------
// Nombre del Archivo: metodgen.h
// Descripción: Definición de clase de Métodos Genéricos
// Autor: ALMYS 11/03/97
// ---------------------------------------------------------------------------
#ifndef __METODOS_GENERICOS_H__
#define __METODOS_GENERICOS_H__
// ---------------------------------------------------------------------------
// Nombre de la clase: Métodos
// Descripción: Clase base para encapsular los diferentes Métodos
// a utilizar en la aplicación : Derivada de Objetos
// ---------------------------------------------------------------------------
class Metodos : public Objetos {
 public:
  Metodos(const char* name);
  ~Metodos();
  void identifica();
};
#endif
// Fin de bloque metodgen.h
```

Figura 26. Archivo de encabezado para la clase Métodos, la cual es la clase base para todos los métodos.

```
// ---------------------------------------------------------------------------
// Nombre del Archivo: metodcic.h
// Descripción: Definición de clase de Métodos de CIC
// Autor: ALMYS Marzo '97
// ---------------------------------------------------------------------------
#ifndef __METODOS_CIC_H__
#define __METODOS_CIC_H__
// ---------------------------------------------------------------------------
// Nombre de la clase: MetodosCIC
// Descripción: Clase base para encapsular los diferentes Metodos
// a utilizar en la aplicación CIC, derivada de Metodos
```

Continúa en la siguiente página

Figura 27. Archivo de encabezado para la clase MetodosCIC.

Viene de la página anterior

```
// ---------------------------------------------------------------------------
class MetodosCIC : public Metodos {
 protected:
  PoissonSolver* CalculaFuncionCorriente;          // Para resolver la ecuación de
                                                   // Poisson
 public:
  MetodosCIC(MallaVortices& malla, const char* name);
  ~MetodosCIC();
  void identifica();                               // Métodos conocidos fuera de la clase
  void reading(NombresArchivos& nombres, MallaVortices& malla);
  void initial(TiraVortices& tira, MallaVortices& malla, Real dt);
  void velocity(TiraVortices& tira, MallaVortices& malla);
  void writing(TiraVortices& tira, MallaVortices& malla);
  void evolution (TiraVortices& tira, MallaVortices& malla, Real dtot);
 protected:                                 // Métodos conocidos sólo dentro de la clase
  void check_domain(TiraVortices& tira);
  void update_vorticity(TiraVortices& tira);
  void spread_vorticity(TiraVortices& tira, MallaVortices& malla);
  void cic_factors(Real *f, Real x, Real y);
  void eulerian_velocity(MallaVortices& malla);
  void lagrangian_velocity (TiraVortices& tira, MallaVortices& malla);
  static Real h1(Real x, Real y);
  static Real h2(Real x, Real y);
  static Real h3(Real x, Real y);
  static Real h4(Real x, Real y);
};
#endif
```

Figura 27 (*cont.*). Archivo de encabezado para la clase MetodosCIC.

```
// ---------------------------------------------------------------------------
// Nombre del Archivo: poisssol.h
// Descripción: Definición de clase de Método PoissonSolver
// Autor: ALMYS Marzo '97
// ---------------------------------------------------------------------------
#ifndef POISSON_SOLVER_H
#define POISSON_SOLVER_H
```

Continúa en la siguiente página

Figura 28. Archivo de encabezado para la clase PoissonSolver.

Viene de la página anterior

```
// ----------------------------------------------------------------------------
// Nombre de la clase: PoissonSolver
// Descripción: Clase para encapsular el método para resolver la ecuación de Poisson,
// derivada de Métodos.
// ----------------------------------------------------------------------------
class PoissonSolver : public Metodos {
 private:
 Bound bound;
 F67com f67com;
 RowVector z;
 RowVector yy;
 RowVector core;
 RowVector akx;
 public:
 PoissonSolver(MallaVortices& malla, const char* name = "Poisson Solver");
 ~PoissonSolver();
 void identifica();                          // Métodos conocidos fuera de la clase
 void InicializaParametros(MallaVortices& malla);
 void poisson_solver(MallaVortices& malla);          // equivalente a Ejecuta()
 private:                                 // Métodos conocidos sólo dentro de la clase
 void setpt1();
 void setf67();
 void pot1();
 void rhse(int ibc, Real f1,Real f2,Real b,int j1,int j2);
 void fetchx(int k);
 void fourier(int ibc1, int iq1);
 void tfold(int is,int l,RowVector& z);
 void zero(int l);
 void kfold();
 void neg(int i1,int i3,int i2);
 void tfold1();
 void revneg();
 void storex(int k);
 void cred(int ibc,int l,int m,Real a,int ip1);
 void rhso(Real f1,Real f2,int j2);
};
#endif
// Fin de poisssol.h
```

Figura 28 (cont). Archivo de encabezado para la clase PoissonSolver.

```
// --------------------------------------------------------------------------------
// Nombre del Archivo: integedo.h
// Descripción: Definición de clase base de Integrador de EDO
// Autor: ALMYS Abril '97
// --------------------------------------------------------------------------------
#ifndef INTEGRADOR_EDO_H
#define INTEGRADOR_EDO_H
// --------------------------------------------------------------------------------
// Nombre de la clase: IntegradorEDO
// Descripción: Clase base para encapsular los métodos para resolver EDO's
// derivada de Métodos.
// --------------------------------------------------------------------------------
class IntegradorEDO : public Metodos {
public:
  IntegradorEDO(): Metodos("Integrador") {}
  ~IntegradorEDO(){}
  virtual void identifica(){}
  virtual void ejecuta(TiraVortices& tira, MallaVortices& malla, MetodosCIC& cic){}
};
#endif
// Fin de integedo.h
```

Figura 29. Archivo de encabezado para la clase IntegradorEDO.

```
// --------------------------------------------------------------------------------
// Nombre del Archivo: rungeku2.h
// Descripción: Definición de clase del Integrador Runge-Kutta orden 2
// Autor: ALMYS Abril '97
// --------------------------------------------------------------------------------
#ifndef RUNGE_KUTTA_2_H
#define RUNGE_KUTTA_2_H
// --------------------------------------------------------------------------------
// Nombre de la clase: RungeKutta2
// Descripción: Clase base para encapsular el método Runge-Kutta de orden 2
// derivada de IntegradorEDO.
// --------------------------------------------------------------------------------
```

Continúa en la siguiente página

Figura 30. Archivo de encabezado para la clase RungeKutta2.

Viene de la página anterior

```
class RungeKutta2 : public IntegradorEDO {
protected:
  int i;
  Real h;
  Real h2;
public:
  RungeKutta2(){}
  ~RungeKutta2() {}
  void identifica();
  void ejecuta(TiraVortices& tira, MallaVortices& malla, MetodosCIC& cic);
};
#endif
// Fin de rungeku2.h
```

Figura 30 (*cont.*). Archivo de encabezado para la clase RungeKutta2.

```
// ---------------------------------------------------------------------------
// Nombre del Archivo: rungeku4.h
// Descripción: Definición de clase del Integrador Runge-Kutta orden 4
// Autor: ALMYS Abril '97
// ---------------------------------------------------------------------------
#ifndef RUNGE_KUTTA_4_H
#define RUNGE_KUTTA_4_H
class RungeKutta4 : public IntegradorEDO {
protected:
  int i;
  Real h, hh, h6;
public:
  RungeKutta4() {}
  ~RungeKutta4() {}
  void identifica();
  void ejecuta(TiraVortices& tira, MallaVortices& malla, MetodosCIC& cic);
};
#endif
// Fin de rungeku4.h
```

Figura 31. Archivo de encabezado para la clase RungeKutta4.

V.5.5 El programa de simulación.

En la Figura 32 se muestra una porción del código que implanta el programa del método de vórtice en celda.

```
// --------------------------------------------------------------------------
// Nombre del Archivo: aplicaci.cpp
// Descripcion: Programa de aplicación del método de vórtice en celda (CIC4)
// Autor: ALMYS Marzo '97
// --------------------------------------------------------------------------
//      Nombre de la función : main()
//      Descripcion : Contiene el código de la aplicación del método de  vórtice en celda
//      (CIC4)
//      Entrada : Condiciones iniciales varias a través de inputfile
//      Salida : vorticidad, posición X y Y, ... a través de varios archivos (nombres)
// --------------------------------------------------------------------------
main()
{
 Real dtot = 0.0;
 TiraVortices tira;                                          // Tira de Vórtices
 MallaVortices malla;                                        // Malla de Vórtices
 MetodosCIC cic4(malla,"Cic4");                              // Los métodos
 IntegradorEDO* integradorEDO = new RungeKutta2();           // otro método

 cic4.reading(nombres, malla);                               // inicializar la aplicación
 malla.Redimensiona(potcom.mx,potcom.my);
 facdim.dimfac(potcom.hx, global.dt);
 inicializaEvolucion(malla);
 escribeBase(malla);
 cic4.initial(tira, malla, global.dt);                       // iniciar ejecución del método
 escribeEvolucion(dipolo);
 cic4.velocity(tira, malla);
 cic4.writing(tira, malla);
```

Continúa en la siguiente página

Figura 32. Archivo fuente que muestra la implementación del método de vórtice en celda.

Viene de la página anterior

```
for (int iwrite=1;iwrite<=global.nmax; iwrite++) {       // ciclo de ejecución del método
  dtot += global.dt;
  cic4.velocity(tira, malla);
  integradorEDO->ejecuta(tira, malla, cic4);
  cic4.evolution (tira, malla, dtot);
  if ((iwrite%global.nwrite)==0)
    cic4.writing(tira, malla);                            // escribe los resultados
 }
 fevolucion.Cierra();
}
// Fin de aplicaci.cpp
```

Figura 32 (*cont.*). Archivo fuente que muestra la implantación del método de vórtice en celda.

Para más información, el código de la aplicación implementada se encuentra disponible en la siguiente dirección de Internet: *http://fciencias.ens.uabc.mx/~amoran/tesis/fuentes/cplusplus.html*.

VI. RESULTADOS

VI.1 Arquitectura para programas de simulación numérica.

La arquitectura de un sistema de software define la organización global de sus componentes, dividiendo el sistema (programa) en subsistemas, los cuales pueden trabajarse de manera independiente, proporcionando así la posibilidad de hacer la implementación de cada uno de ellos por separado. En este caso, se propone una arquitectura abierta por capas y particiones, la cual fue mostrada en la Figura 9.

VI.2 El modelado orientado a objetos.

El modelo orientado a objetos desarrollado del programa de simulación se puede dividir en dos bloques principales, uno incluye la jerarquía de clases de los conjuntos de datos y el otro la jerarquía de clases de los métodos. Los diagramas de objetos, de estados y de flujo de datos resultantes para el caso general fueron mostrados en las Figs. 10 - 13 y 18 y para el caso particular en las Figs. 3, 6 - 8, 14 - 16 y 18 - 21.

VI.3 El programa de simulación orientado a objetos en C++.

El programa de simulación orientado a objetos implementado está funcionando correctamente sobre las plataformas que se enlistan en la Tabla IX.

Se tienen probados los mismos siete casos de prueba que la versión C (véase apéndice C). Los archivos de datos de entrada y de salida son completamente compatibles con las versiones Fortran y C del mismo.

Tabla IX. Plataformas de cómputo sobre las que está funcionando la versión orientada a objetos del programa de simulación.

Máquina	Tipo	Memoria RAM	Sistema Operativo	Compilador
1	Intel Pentium 133 MHz	32 MB	MS-Windows95	Borland C++ v.5.01 (32 bits)
2	Intel Pentium Pro 200 MHz	32 MB	UNIX FreeBSD v.2.2	GNU g++ v.2.7.2
3	Intel Pentium 133 MHz	32 MB	UNIX FreeBSD v.2.1.7	GNU g++ v.2.7.1
4	HP-9000 715/80	64 MB	HP-UX v.9.07	C++ HP.
5	IBM PowerPC	16 MB	AIX v.3	GNU g++ v.2.7.0

A Continuación las Tablas X - XVI muestran los tiempos de ejecución del código en las plataformas de la Tabla IX, para los casos de prueba del apéndice C.

Tabla X. Tiempos de ejecución para el caso de prueba INI TEST.

INI_TEST					
Máquina	1	2	3	4	5
versión C	No aplica	3	11	7	11
versión C++	9.5	5.25	16	14.5	35.5
comparación	No aplica	75% +	45% +	107% +	222% +

Tabla XI. Como Tabla X para INI BETA.

INI_BETA					
Máquina	1	2	3	4	5
versión C	No aplica	102	217	199	279
versión C++	313.5	196	506.5	424.5	863.5
comparación	No aplica	92% +	133% +	113% +	209% +

Tabla XII. Tiempos de ejecución para el caso de prueba INI GAMA.

INI_GAMA					
Máquina	1	2	3	4	5
versión C	No aplica	105	221	202	286
versión C++	316	199.5	510.5	428	871.5
comparación	No aplica	90% +	131% +	112% +	205% +

Tabla XIII. Como Tabla XII para INI TOP1.

INI_TOP1					
Máquina	1	2	3	4	5
versión C	No aplica	105	223	214	291
versión C++	322.25	201.5	514.5	439.5	877.25
comparación	No aplica	92% +	131% +	105% +	201% +

Tabla XIV. Como Tabla XII para INI TOP2.

INI_TOP2					
Máquina	1	2	3	4	5
versión C	No aplica	108	230	216	296
versión C++	323	203	516	441.25	884.5
comparación	No aplica	88% +	124% +	104% +	199% +

Tabla XV. Como Tabla XII para INI TOP3.

INI_TOP3					
Máquina	1	2	3	4	5
versión C	No aplica	112	235	221	308
versión C++	328	207.5	520.25	449.5	891.5
comparación	No aplica	85% +	121% +	103% +	189% +

Tabla XVI. Como Tabla XII para INI TOP4.

INI_TOP4					
Máquina	1	2	3	4	5
versión C	No aplica	104	233	210	288
versión C++	320	199	514	437.25	871
comparación	No aplica	91% +	120% +	108% +	202% +

VII. DISCUSIÓN

El desarrollo de software, a través de la cual se genera una solución por computadora para un problema en particular, es una actividad intelectual compleja. El desarrollo de aplicaciones (programas) científicas no es la excepción. Tradicionalmente más del 50% del esfuerzo dedicado a desarrollo se concentra en la segunda etapa del ciclo, es decir, en las etapas de adaptación, extensión y mantenimiento (Fairley, 1994; Pressman, 1992), debido principalmente a un enfoque en los temas relacionados con la implementación de la solución, descuidando, por otro lado, la parte inicial de análisis y diseño. La orientación a objetos promueve la identificación y organización de los conceptos del dominio de la aplicación, basándose en abstracciones que existen en el mundo real. Comprende la parte inicial del ciclo de desarrollo de software: análisis, diseño e implementación y para hacer la representación de las abstracciones utiliza objetos, i.e., entidades que incluyen tanto a los atributos como a sus comportamientos. Si bien el esfuerzo requerido para completar por primera vez el programa es aproximadamente el mismo, en ocasiones un poco más, que el realizado con otros esquemas convencionales, iteraciones subsecuentes en el desarrollo orientado a objetos son más fáciles y rápidas que con el desarrollo convencional, debido a que las revisiones son más localizadas.

Los métodos de partículas son métodos de discretización muy utilizados, junto con los métodos de diferencias finitas y elementos finitos, para modelar y simular sistemas físicos con ayuda de la computadora (Hockney y Eastwood, 1988). Se seleccionó el método de vórtice en celda por la analogía que existe entre partícula y objeto (Hockney y Eastwood,

1988), la disponibilidad de un modelo en producción (Velasco Fuentes, 1994), así como por la posibilidad de contar con asesoría por parte de un experto en el método.

El uso de una arquitectura para el desarrollo de software resuelve problemas de diseño al nivel de sistemas, excluyendo los algoritmos y estructuras de datos que caen dentro del módulo o componente en el nivel bajo y excluyendo también lo relacionado con la funcionalidad que es requerida por el cliente en el nivel superior (Shaw *et al.*, 1993b). La arquitectura propuesta con base en capas y particiones, es necesario que sea abierta, ya que esto permite que una capa utilice directamente características de otras capas de nivel inferior a cualquier profundidad, aún cuando no se explota el ocultamiento de información entre niveles se reduce el número de operaciones entre capas, lo que resulta en un código más eficiente y compacto. Los componentes o subsistemas identificados (Fig. 9) provienen directamente del dominio de la simulación, además de que corresponden directamente a los identificados como objetos genéricos principales en el modelado orientado a objetos. El entendimiento de la arquitectura da a los desarrolladores una terminología común cuando discuten su trabajo y les permite obtener puntos genéricos de diseño a partir de lo específico de la aplicación, lo cual eventualmente conduce a disponer de componentes reutilizables para soportar requerimientos específicos de la aplicación (Shaw *et al.*, 1993a).

Los tipos de datos básicos, a diferencia de los tipos de datos de las aplicaciones, tienen operaciones o comportamientos que son aplicables fuera del contexto de cualquier aplicación, digamos, la operación multiplicación es aplicable a vectores y matrices, así como es aplicable a enteros y reales, sin embargo la operación $A{\cdot}x = y$ puede representar el

planteamiento para la resolución de un sistema de ecuaciones, lo cual no está precisamente dentro del dominio del concepto (nivel de abstracción) básico matriz-vector. El primer tipo de operaciones tienen sentido fuera de cualquier contexto de aplicación, mientras que el segundo tipo requiere de estar asociado con algún concepto de aplicación (solución de un sistema de ecuaciones).

Aquellos conceptos del dominio del objeto como atributos y operaciones, deben estar encapsulados por la clase del objeto. Mientras que aquellos conceptos del dominio de la utilización o aplicación del objeto como atributos, operaciones y clases, deben ser encapsulados por un meta-concepto, es decir, en el siguiente nivel de abstracción. En el primer caso se definen los atributos y las operaciones del objeto, mientras que en el segundo caso se define como aplicarlos (utilizarlos) en (para) algo en un dominio o nivel de abstracción más alto. Esto es, a menor nivel de abstracción, se encapsula de manera tradicional (atributos y comportamientos en clases), mientras que a mayor nivel de abstracción, los conjuntos de datos y los métodos se conjuntan por agregación (composición de objetos).

La mayoría de los códigos numéricos orientados a objetos han encapsulado juntos atributos (datos) y comportamientos (métodos) tal como lo indican las reglas de la orientación a objetos (Schroeder *et al.*, 1996), sin embargo, la aplicación de esos métodos sigue estando atada a la representación específica de los datos de la aplicación. Una excepción podría ser el código de vectores y matrices. Sin embargo, en el resto de los casos, al momento de querer volver a utilizar los métodos en situaciones donde la representación de los datos de

la aplicación difiere considerablemente, hay que volver a codificarlos para el modelado de datos particular de la nueva aplicación (programa). Entonces, ¿qué tan eficiente es que el supuesto "mismo" método esté siendo forzado para trabajar sobre un conjunto de datos diferentes?. ¿Por qué no hacer lo contrario, como sucede en otras áreas del desarrollo tecnológico - electrónica por ejemplo - donde los datos se convierten al formato - usualmente estándar - sobre el cual trabaja el método (componente) deseado?. En el mejor (o peor) de los casos se puede redundar y tener varias versiones del mismo método (componente), cada una de las cuales tenga un cierto tipo de especialización particular (variantes) para trabajar con representaciones de datos diferentes, sin embargo, la tarea (comportamiento) del método (componente) deberá seguir siendo la misma y la manipulación de los datos se hará de manera transparente para el usuario, no así para el desarrollador. De lo anterior resulta:

- Existen métodos que pueden verse como objetos por sí mismos, ya que tienen una entidad propia, incluso fuera de cualquier dominio de aplicación (en el siguiente nivel de abstracción).
- Los datos básicos son utilizados para formar los datos de la aplicación y los métodos básicos son utilizados para construir las operaciones o métodos de la aplicación.
- Los métodos básicos trabajan sobre los tipos de datos básicos (interfaz estándar de entrada y salida).
- Las operaciones o métodos de aplicación (programa de simulación) agrupan los métodos básicos que trabajan sobre los datos de la aplicación, por ello, estos deben ser

transformados de entrada al dominio de los datos básicos y de salida nuevamente al dominio de los datos de la aplicación.

Los códigos legados son códigos que han evolucionado a través de los años y son considerados irremplazables, su uso es común en sistemas en los cuales el volver a implementar su función es considerablemente caro o ya sea por la confiabilidad por parte de los usuarios (Dietrich *et al.*, 1989). Sin embargo, debido a su "edad", estos códigos fueron desarrollados usando técnicas de análisis y diseño y lenguajes de programación limitados en el uso de abstracción o encapsulamiento, lo cual complica su reutilización, extensión y mantenimiento. Una de manera de aprovechar estos códigos con orientación a objetos es a través del uso de envolturas de código, la cual es una forma extremadamente efectiva de trabajar códigos legados (Vayda, 1995). De manera resumida, los pasos seguidos para construir la envoltura de código involucraron: desarrollar un modelo sólido de requerimientos del código legado, diseñar una buena solución orientada a objetos para esos requerimientos, e implementar las operaciones orientadas a objetos incrustando las funciones del código legado y agregando el comportamiento necesario para poder trabajar directamente con los datos. Dos de las principales ventajas de este esquema son que soporta que el código legado vaya siendo reemplazado gradualmente con las operaciones del nuevo sistema y que proporciona independencia de las plataformas.

Para el desarrollo de una aplicación científica es importante y necesario identificar cual es el nivel aceptable de sacrificio en rapidez de ejecución que puede ser compensado por otras

ganancias. En este trabajo la versión C++ fue más lenta que las versiones Fortran y C, pero se obtuvo:

- Portabilidad: la aplicación puede ejecutarse sobre 5 plataformas (máquina/compilador) diferentes sin necesidad de realizar cambios en el código.
- Buen comportamiento numérico: los resultados obtenidos sobre cada una de las 5 plataformas distintas son exactamente los mismos con la versión C++ orientada a objetos.
- Compatibilidad entre versiones: los archivos de datos de entrada y salida de la aplicación son totalmente compatibles en cualquiera de las 5 plataformas y hacia la plataforma de la versión original en Fortran y la versión C.
- Facilidad de desarrollo y mantenimiento: se tienen probadas, integradas y funcionando 2 variantes para el método de integración de ecuaciones diferenciales ordinarias, clases RungeKutta2 y RungeKutta4, está todo listo para agregar otras variantes a través de la clase base IntegradorEDO y para agregar otro esquema de solución para la resolución de la Ecuación de Poisson (clase PoissonSolver).

Estas y otras características permiten al investigador dedicar menos tiempo al desarrollo de su laboratorio experimental (los programas) y le dejan más tiempo para trabajar en sus experimentos (corrida o ejecución de los programas). Esto es, se tarda un poco más en realizar un experimento, correr un programa, pero tarda menos en preparar otros experimentos, escribir o extender otros programas.

Cada uno de los lenguajes de programación con base en las características que soporta, representa una etapa en la evolución de los lenguajes como herramientas de programación. Fortran, década de los 60's, soporta la programación no estructurada; C ,década de los 70's, soporta la programación estructurada; C++, década de los 80's, soporta la programación orientada a objetos. Cada uno de los paradigmas ha tratado de mejorar el anterior, es decir, se observan las características buenas y malas y se desarrolla un nuevo paradigma o modelo de programación que amplíe las ventajas y elimine las desventajas. El desarrollar una aplicación no sólo involucra el uso de un lenguaje de programación, sino que debe estar sustentado en los modelos y técnicas para el desarrollo de software. Una parte es la metodología utilizada y otra es el lenguaje utilizado, el lenguaje no necesariamente es un indicativo de la utilización de un paradigma en particular, pero puede definir características que son soportadas o no soportadas al momento de la implementación.

Una de las ventajas más importantes con las que cuenta Fortran, es que ha evolucionado (Fortran, Fortran IV, Fortran 66, Fortran 77, Fortran 90, HP Fortran), razón principal por la que todavía no desaparece o cae en desuso. Sin embargo, considero que su mayor desventaja es la de seguir soportando la programación no estructurada, que en algunos casos produce programas que se ejecutan con mayor rapidez, reduciendo el número de instrucciones que se tienen que ejecutar, por ello, no se puede garantizar que el código sea fácilmente portado, modificado o extendido, es decir adolece del problema de "*rapidez de ejecución a cambio de problemas al tiempo de mantenimiento y extensión*". La mayoría del código desarrollado en lenguajes no estructurados, aunque no se excluyen algunos escritos

en otros lenguajes, fueron desarrollados únicamente con la mentalidad de resolver el problema, es decir, se uso el paradigma "problem solving" y no con la mentalidad de desarrollar una herramienta que pueda ser utilizada posteriormente, planeada para reutilizarse.

C (Kernigan y Ritchie, 1988), al estar asociado muy estrechamente con UNIX, es el lenguaje de programación de sistemas por excelencia, esto le permite utilizar los recursos del sistema a bajo nivel, además de que soporta la programación estructurada.

C++ (Stroustrup, 1986) desarrollado con base en C, extiende este popular lenguaje agregando soporte para construcciones orientadas a objetos, que incluyen abstracción, encapsulamiento, polimorfismo, herencia, generalidad y verificación estática de tipos (Stroustrup, 1995).

En este trabajo, se selecciona C++ como el lenguaje para implementación, debido a que soporta las características deseadas, a la disponibilidad de los compiladores en las plataformas de hardware de la Tabla IX, y a que varios autores (Schroeder *et al.*, 1996; Spinelli *et al.*, 1994; Koved y Wooten, 1993; Rumbaugh *et al.*, 1991) han reportado implementaciones orientadas a objetos exitosas utilizando este lenguaje.

De los tiempos de ejecución obtenidos (Tablas X a la XVI), resaltan algunas características importantes:

- Consistentemente, entre las máquinas 2 - 4, existe una relación del orden 2 a 1 en los tiempos de ejecución de las versiones C y C++. En el caso de la máquina 1 no existe un punto de comparación para la versión C++.

- La máquina 5, que cuenta con la menor cantidad de memoria primaria (16 MB), es la que tiene el desempeño más pobre.
- Las máquinas 1-3, basadas en sistemas personales, dedicadas a la ejecución del programa, tienen en general dos de los mejores tiempos de ejecución.

Con base en lo anterior se tienen las siguientes observaciones:

- Para esta aplicación se tiene que el nivel de sacrificio en rapidez de ejecución es de 2 a 1, lo cual puede ser aceptable debido a que el tiempo de ejecución no es muy grande y a la relación sacrificio-ganancia que se obtenga al desarrollar bajo el esquema orientado a objetos.
- El efecto de una baja cantidad de memoria primaria en el sistema es grande, debido a la cantidad de este tipo de recurso que es requerido para la ejecución del programa. Aún cuando el sistema puede satisfacer la demanda del recurso a través de esquemas como memoria virtual, la pérdida en desempeño puede ser más alta, de 3 a 1.
- El uso de plataformas de cómputo dedicadas, basadas en computadoras personales y en sistemas comerciales y de libre distribución, proporcionan una capacidad o poder de procesamiento similar e incluso mejor que algunos sistemas compartidos basados en estaciones de trabajo, mejorando así la relación costo-rendimiento.

El presente trabajo involucró dos enfoques principales, la implementación específica del método de vórtices en celda y el enfoque general para el desarrollo de aplicaciones científicas. Sin embargo, todavía hay mucho por realizar, a continuación se enlistan algunos de los puntos sobre los cuales se debe continuar trabajando:

- Iterar sobre el modelo orientado a objetos con la finalidad de refinar los detalles en el modelo de objetos, planteando nuevos algoritmos y estructuras de datos además de usar código legado y envolturas de código. También, explorar esquemas de solución alternativos al uso de código legado y envolturas de código, para encontrar otras opciones que optimicen la implementación actual.
- Reemplazar/reescribir gradualmente módulos (clases, objetos) de esta solución inicial, que aplica códigos legados y envolturas de código, con fragmentos que utilicen un esquema alternativo equivalente aprovechando el enfoque orientado a objetos.
- Implementar más variantes de los métodos numéricos y de los conjuntos de datos aprovechando la característica de extensibilidad del enfoque orientado a objetos.
- Implementar una plataforma de conjuntos de datos básicos y métodos básicos que siga los lineamientos especificados por el modelado y la arquitectura en lugar de utilizar unos ya existentes que no los sigan.
- Probar la productividad, robustez y corrección de la arquitectura y el modelo desarrollando más aplicaciones sobre éstos y analizando sus resultados.
- Promover la utilización de la arquitectura y del modelo orientado a objetos propuestos entre la comunidad científica.

VIII. CONCLUSIONES

1. Se diseñó exitosamente un modelo orientado a objetos del método de vórtices en celda aplicado a fluidos geofísicos. Del paradigma orientado a objetos se explotaron sus características de portabilidad, buen comportamiento numérico, compatibilidad entre versiones y facilidad de extensión y mantenimiento. El modelado y la codificación se hicieron con base en los conceptos del modelo matemático, lo que permitió a un modelador externo trabajar con expertos geofísicos, para generar un código más entendible y adecuado a las necesidades del investigador. La cantidad de memoria primaria necesaria se redujo para la versión orientada a objetos, pero el tiempo de ejecución se incrementó.
2. Se encontró que para desarrollar sistemas de cómputo científico orientados a objetos es necesario usar una metodología de ingeniería de software. En particular, OMT produjo un modelo orientado a objetos en términos del dominio de la dinámica de fluidos geofísicos y permitió la propuesta y uso de una arquitectura para desarrollo por capas y particiones con interfaces bien definidas. El código legado y la técnica de envolturas de código fueron adecuadas para la implementación, proporcionando un código verificado y confiable, y facilitando la integración del código. El uso de C++ fue conveniente en la codificación de la versión orientada a objetos debido al esquema seguido (partir de una versión Fortran, pasar por una versión C y llegar a una versión C++).

LITERATURA CITADA

Davies, R. B. 1997. "Newmath, version 0.9", Copyright (C) 1991-1997. http://nz.com/webnz/robert/.

Dietrich, W. C., L. R. Nackman y F. Gracer. 1989. "Saving legacy with objects". Proceedings of the OOPSLA'89 Conference. ACM-SIGPLAN. New Orleans, Louisiana. (OOPSLA CD-ROM Compendium 1986-1996).

Fairley, R. 1994. "Ingeniería de Software". McGraw-Hill/Interamericana. Segunda edición. México, D.F. 390 pp.

Hockney, R.W. y J.W. Eastwood. 1988. "Computer simulation using particles". IOP Publishing LTD. Primera edición. Philadelphia, PA. 540 pp.

Kernigan B. W. y D. M. Ritchie. 1988. "The C programming language". Prentice Hall. Segunda edición. Englewood, NJ. 272 pp.

Koved L. y W. L. Wooten. 1993. "GROOP: An object-oriented toolkit for animated 3D graphics". Proceedings of the OOPSLA'93 Conference. ACM-SIGPLAN. Washington, D.C. (OOPSLA CD-ROM Compendium 1986-1996).

Press H. W., B. P. Flannery, S. A. Teukolsky y W. T. Vetterling. 1988. "Numerical Recipes in C". Primera edición. Cambridge University Press. 735 pp.

Pressman, R.S. 1992. "Software engineering. A practitioner's approach". McGraw-Hill. Inc. Tercera edición. New York. 824 pp.

Rumbaugh J., M. Blaha, W. Premerlani, F. Eddy y W. Lorensen. 1991. "Object-oriented modeling and design". Prentice Hall, Inc. Primera edición. Englewood, NJ. 500 pp.

Schroeder, W., K. Martin y B. Lorensen. 1996. "The visualization toolkit, an object-oriented approach to 3D graphics". Prentice Hall. ISBN 0-13-199837-4.

Shaw M., L. Best y K. Beck. 1993a. "Panel: Software architecture: The next step for object technology". Proceedings of the OOPSLA'93 Conference. ACM-SIGPLAN. Washington, D.C. (OOPSLA CD-ROM Compendium 1986-1996).

Shaw M., L. Best y K. Beck. 1993b. "Panel: Software architecture: The next step for object technology". Addendum to the proceedings of the OOPSLA'93 Conference. ACM-SIGPLAN. Washington, D.C. (OOPSLA CD-ROM Compendium 1986-1996).

Spinelli A., P. Salvaneschi, M. Cadei y M. Rocca. 1994. "MI - An object-oriented environment for integration of scientific applications". Proceedings of the OOPSLA'94 Conference. ACM-SIGPLAN. Portland, OR. (OOPSLA CD-ROM Compendium 1986-1996).

Stroustrup B. 1986. "The C++ programming language". Addison-Wesley. Primera edición. Reading, MA. 327 pp.

Stroustrup B. 1995. "Why C++ is not just an object-oriented programming language". Addendum to the proceedings of the OOPSLA'95 Conference. ACM-SIGPLAN. Austin, TX. (OOPSLA CD-ROM Compendium 1986-1996).

Vayda, T. P. 1995. "Lessons from the battlefield". Proceedings of the OOPSLA'95 Conference. ACM-SIGPLAN. Austin, TX. (OOPSLA CD-ROM Compendium 1986-1996).

Velasco Fuentes, O. U. 1994. "Two-dimensional vortices with background vorticity". Ph. D. Thesis, Eindhoven University of Technology. 165 pp.

APÉNDICE A

GLOSARIO

Mecánica de fluidos.

Aproximación de plano f: cuando el fenómeno que se estudia se extiende latitudinalmente cientos de kilómetros, el dominio se considera plano y la rotación uniforme. $f = f_0$ (ver parámetro de Coriolis).

Aproximación de plano β : cuando el fenómeno que se estudia tiene una extensión latitudinal entre mil y diez mil kilómetros, el dominio se considera plano y la rotación varía linealmente con la latitud. $f = f_0 + \beta y$, donde $\beta = \frac{2\Omega \cos \phi_0}{R}$, y es la distancia latitudinal, Ω es la velocidad angular, ϕ es la latitud geográfica y R es el radio de la Tierra.

Aproximación de plano γ : para fenómenos que ocurren en escalas similares a las consideradas en la aproximación plano β, pero en regiones polares. $f = f_0 - \gamma r^2$, donde $\gamma = \frac{\Omega}{R^2}$ y r es la distancia al polo.

Circulación: la integral de la componente tangencial de la velocidad a lo largo de una curva cerrada.

Conservación de la masa: la razón de incremento de masa en un volumen fijo es igual a la razón de entrada de masa a través de las fronteras.

Conservación de la vorticidad potencial: en la teoría de agua somera, un incremento en la profundidad (h) de la columna de agua, debe de ir acompañado de un incremento en la vorticidad absoluta.

Corrientes geostróficas: corrientes derivadas del balance horizontal entre el gradiente de presión y la fuerza de Coriolis.

Equivalentes dinámicos de los planos β y γ: un plano inclinado causa un gradiente uniforme, siendo equivalente al plano β. Una topografía parabólica, como la superficie libre de un fluido en rotación, es equivalente al plano γ. Somero en el caso topográfico es equivalente a norte en los planos β y γ.

Parámetro de Coriolis: $f = 2\Omega \operatorname{sen}\phi$, donde Ω es la razón de la rotación de la Tierra y ϕ es la latitud.

Vórtice: fluido arrastrado en movimiento giratorio.

Vórtice dipolar o dipolo: está formado por dos remolinos monopolares estrechamente acoplados que rotan en direcciones opuestas.

Vórtice monopolar o monopolo: consiste de una región de fluido que rota alrededor de un punto común, es el tipo de vórtice que ocurre con más frecuencia.

Vorticidad: rotacional del campo de velocidad. Igual a dos veces la velocidad angular local.

Vorticidad absoluta: vorticidad relativa mas la vorticidad planetaria.

Vorticidad potencial: para el caso de un fluido homogéneo, vorticidad absoluta entre profundidad.

Programación orientada a objetos.

Agregación: relación componente-conjunto en la cual los objetos que representan los componentes de algo se asocian con un objeto que representa al conjunto completo.

Asociación: describe un grupo de ligas con estructura y significado comunes.

Atributo: propiedad de los objetos de una clase.

Atributo de liga: propiedad de las ligas de una asociación.

Calificador: reduce la multiplicidad efectiva de una asociación al seleccionar de entre los conjuntos de objetos en cada extremo "muchos" de la asociación.

Clase: describe un grupo de objetos con atributos, operaciones y semántica comunes.

Delegación: mecanismo de implantación en el cual un objeto responde a una operación sobre él mismo, pasando la llamada a una operación sobre otro objeto. Los métodos pueden ser asociados directamente a instancias y donde la resolución de los métodos se realiza buscando en una cadena de apuntadores a instancias, en lugar de buscar en una jerarquía de clases.

Generalización: proporciona el medio para refinar una superclase en una o más subclases. La superclase contiene características comunes a todas las clases; las subclases contienen características específicas a cada clase.

Herencia: se refiere al mecanismo de compartir atributos y operaciones usando la relación de generalización.

Liga: conecta a dos o más objetos.

Multiplicidad: especifica cuantas instancias de una clase pueden relacionarse con cada instancia de otra clase.

Objeto: abstracción que combina las estructuras de datos o atributos y el comportamiento u operaciones en una sola entidad. Todos los objetos tienen identidad y son distinguibles.

Operación: acción que puede aplicarse o realizarse por los objetos de una clase.

Orientación a objetos: significa que se organiza el software como una colección de objetos discretos que incorporan tanto la estructura de datos como el comportamiento.

Polimorfismo: significa que la misma operación puede comportarse de manera diferente sobre diferentes clases.

APÉNDICE B

LA TÉCNICA DE MODELADO POR OBJETOS (OMT) COMO UNA METODOLOGÍA DE INGENIERÍA DE SOFTWARE

La ingeniería de software es un proceso para la producción ordenada de software, usando una colección de técnicas y convenciones de notación. Una metodología es una serie de pasos, con técnicas y notación asociadas a cada paso. Los conceptos y notación que soportan la metodología OMT se presentan en el modelado de objetos, dinámico y funcional. Los pasos para la producción de software son organizados en un ciclo de vida que consisten de varias fases de desarrollo. OMT abarca sólo la primera parte de ellos, es decir, análisis, diseño e implantación. A continuación se describen de manera resumida los pasos que forman la metodología.

Análisis.

El objetivo del análisis es desarrollar un modelo del funcionamiento del sistema. El modelo se expresa en términos de objetos y relaciones, el control dinámico de flujo y las transformaciones funcionales. El proceso de capturar los requerimientos y consultar con el solicitante debe ser continuo a través del análisis. A saber:

1. Contar con una descripción inicial del problema (enunciado del problema).
2. Construir un **modelo de objetos**, cuyas fases son:
   - Identificar las clases de objetos.

- Iniciar un diccionario de datos que contenga descripciones de clases, atributos y asociaciones.
- Agregar asociaciones entre clases.
- Agregar atributos a objetos y ligas.
- Organizar y simplificar las clases de objetos usando herencia.
- Probar las rutas de acceso usando escenarios e iterar los pasos anteriores según sea necesario.
- Agrupar las clases en módulos, basándose en "acoplamiento cercano" y función relacionada.

En resumen:

Modelo de objetos = diagramas del modelo de objetos + diccionario de datos.

3. Desarrollar un **modelo dinámico** que incluya:

- Preparar escenarios para las secuencias de interacción típicas.
- Identificar eventos entre objetos y preparar trazos de eventos para cada escenario.
- Preparar un diagrama de flujo de eventos para el sistema.
- Desarrollar un diagrama de estados para cada clase que tenga un comportamiento dinámico importante.
- Verificar que los eventos compartidos entre diagramas de estado sean consistentes y correctos.

En resumen:

Modelo dinámico = diagramas de estado + diagrama global de flujo de eventos.

4. Construir un **modelo funcional** que incluya:

   - Identificar valores de entrada y salida.
   - Usar diagramas de flujo de datos para mostrar dependencias funcionales.
   - Describir las funciones.
   - Identificar restricciones.
   - Especificar criterios de optimización.

   En resumen:

   Modelo funcional = diagramas de flujo de datos + restricciones.

5. Verificar, iterar y refinar los tres modelos:

   - Agregar al **modelo de objetos** operaciones clave que sean descubiertas durante la preparación del **modelo funcional**. No deben mostrarse todas las operaciones durante el análisis, sólo las más importantes.
   - Verificar que las clases, asociaciones, atributos y operaciones sean consistentes y completos al nivel seleccionado de abstracción. Comparar los tres modelos con el enunciado del problema y el conocimiento relevante al dominio y probar los modelos usando varios escenarios.
   - Desarrollar escenarios más detallados (incluyendo condiciones de error) como variaciones de los escenarios básicos, para verificar aún más los tres modelos.
   - Iterar los pasos anteriores según sea necesario para completar el análisis.

   En resumen:

Documento de análisis = enunciado del problema + modelo de objetos + modelo dinámico + modelo funcional.

Diseño de sistemas.

Durante el diseño de sistemas, se selecciona la estructura de alto nivel del sistema. Existen varias arquitecturas canónicas que pueden servir como un punto de inicio adecuado. El paradigma orientado a objetos no introduce vistas especiales en el diseño del sistema, pero se incluye para tener una cobertura completa del proceso de desarrollo de software. Los pasos son:

1. Organizar el sistema en subsistemas.
2. Identificar la concurrencia inherente al problema.
3. Asignar subsistemas a procesadores y tareas.
4. Escoger la estrategia básica para implantar los almacenamientos de datos en términos de estructuras de datos, archivos y bases de datos.
5. Identificar recursos globales y determinar los mecanismos para controlar su acceso.
6. Seleccionar un esquema para implantar el control del software:
   - Usar la ubicación dentro del programa para mantener el estado,
   - o implantar directamente una máquina de estado,
   - o usar tareas concurrentes.
7. Considerar las condiciones de frontera.
8. Establecer prioridades de decisión sobre características deseables del producto de software.

En resumen:

Documento de diseño de sistemas = estructura de la arquitectura básica del sistema + las decisiones de estrategias de alto nivel.

Diseño de objetos.

Durante el diseño de objetos se detalla el modelo de análisis y se proporciona una base detallada para la implementación. Se toman las decisiones necesarias para realizar un sistema sin entrar en las particularidades de un lenguaje o base de datos específico. El diseño de objetos inicia un corrimiento en el enfoque de la orientación del mundo real del modelo de análisis hacia la orientación en la computadora requerida para una implantación práctica. Los pasos son:

1. Obtener las operaciones para el **modelo de objetos** a partir de los otros modelos:
   - Encontrar una operación para cada proceso en el **modelo funcional**.
   - Definir una operación para cada evento en el **modelo dinámico**, dependiendo de la implantación del control.
2. Diseñar los algoritmos para implantar las operaciones:
   - Escoger los algoritmos que minimicen el costo de implementación de las operaciones.
   - Seleccionar las estructuras de datos apropiadas para los algoritmos.
   - Definir clases internas y operaciones nuevas según sea necesario.
   - Asignar las responsabilidades para las operaciones que no están asociadas claramente con una sola clase.

3. Optimizar las rutas de acceso a los datos:
   - Agregar asociaciones redundantes para minimizar los costos de acceso y maximizar la conveniencia.
   - Reacomodar los cálculos para una mayor eficiencia.
   - Guardar los valores derivados para evitar recalcular expresiones complicadas.
4. Implantar el control del software introduciendo el esquema seleccionado durante el diseño de sistemas.
5. Ajustar la estructura de clases para incrementar la herencia:
   - Reacomodar y ajustar las clases y las operaciones para incrementar la herencia.
   - Abstraer el comportamiento común de los grupos de clases.
   - Usar delegación para compartir comportamiento donde la herencia sea semánticamente inválida.
6. Diseñar la implantación de las asociaciones:
   - Analizar las travesías de las asociaciones.
   - Implantar cada asociación como un objeto distinto o agregando atributos objeto-valor a una o ambas clases en la asociación.
7. Determinar la representación de los atributos de los objetos.
8. Empaquetar las clases y las asociaciones en módulos.

En resumen:

Documento de diseño = modelo de objetos detallado + **modelo dinámico** detallado + **modelo funcional** detallado.

A continuación se describe de manera resumida la notación y elementos de OMT utilizados para el modelado en el presente trabajo.

Elementos y notación.

Modelos de objetos, dinámico y funcional.

Los tres modelos que utiliza OMT para describir un sistema son aplicables durante todas las etapas de desarrollo y adquieren detalles de implantación conforme avanza el mismo. Una descripción completa de un sistema requiere el uso de los tres modelos.

El modelo de objetos.

El modelo de objetos describe la estructura estática de los objetos en un sistema y sus relaciones; utiliza el diagrama de objetos, el cual es un grafo cuyos nodos son clases de objetos y cuyos arcos son las relaciones entre las clases. Sus elementos son: objetos, clases, atributos y operaciones. Un *objeto* es un concepto, abstracción o cosa con límites bien marcados y significativo para la aplicación. Todos los objetos tienen identidad y son distinguibles. Una *clase de objetos* describe un grupo de objetos con atributos, operaciones y semántica comunes. Un *atributo* es una propiedad de los objetos de una clase. Una *operación* es una acción que puede ser aplicada a o realizada por los objetos de una clase. La notación empleada para representar gráficamente una clase se muestra en la Figura 33.

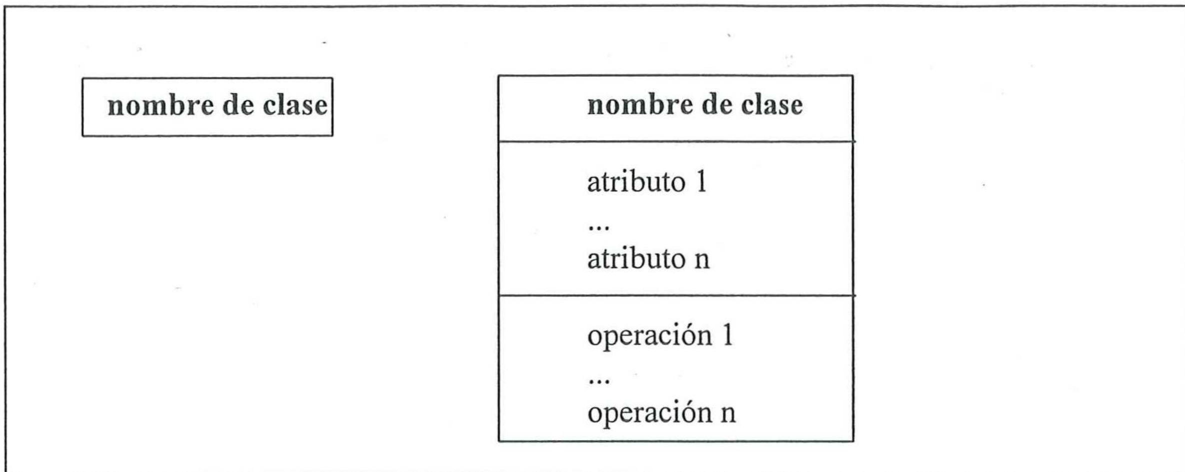


Figura 33. Notación OMT para clases, atributos y operaciones.

Asociaciones y ligas.

Las *asociaciones* establecen relaciones entre clases. Una *liga* conecta a dos o más objetos. Una asociación describe un grupo de ligas con estructura y significado comunes. La notación se muestra en la Figura 34.

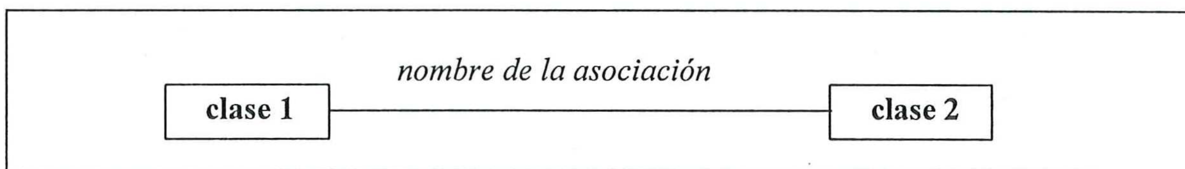


Figura 34. Notación OMT para asociaciones entre clases.

Multiplicidad de asociaciones.

La *multiplicidad* especifica cuantas instancias de una clase pueden relacionarse a cada instancia de otra clase. La notación se muestra en la Figura 35.

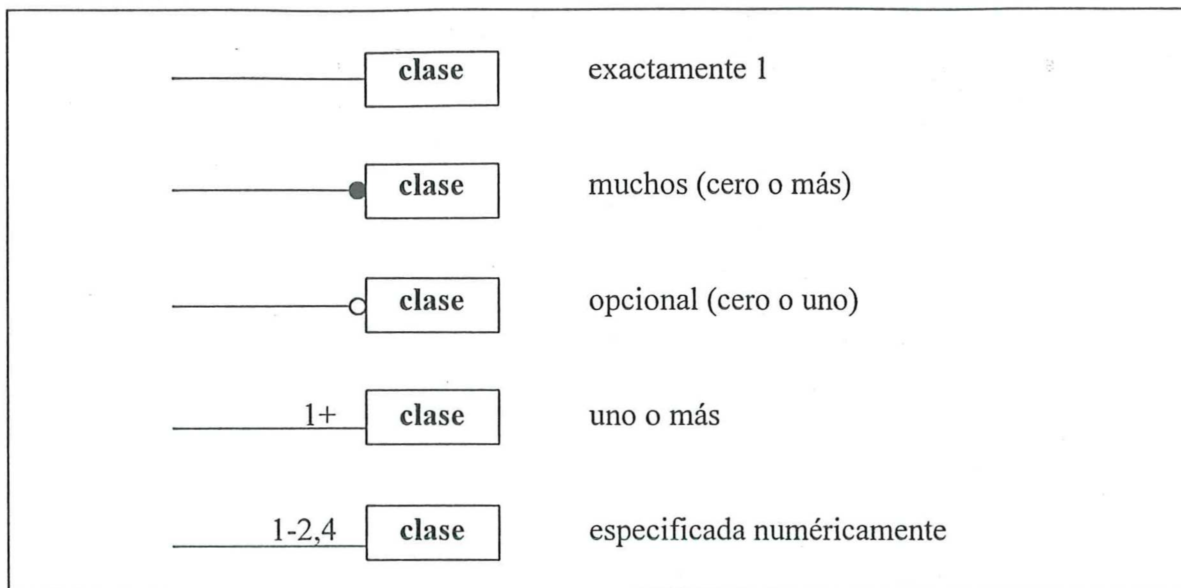


Figura 35. Notación para multiplicidad de asociaciones entre clases.

Agregación.

Una construcción adicional para modelar asociaciones es la *agregación*. La agregación es una relación componente-conjunto en la cual las clases que representan a los componentes se asocian con una clase que representa al conjunto completo. Su notación se presenta en la Figura 36.

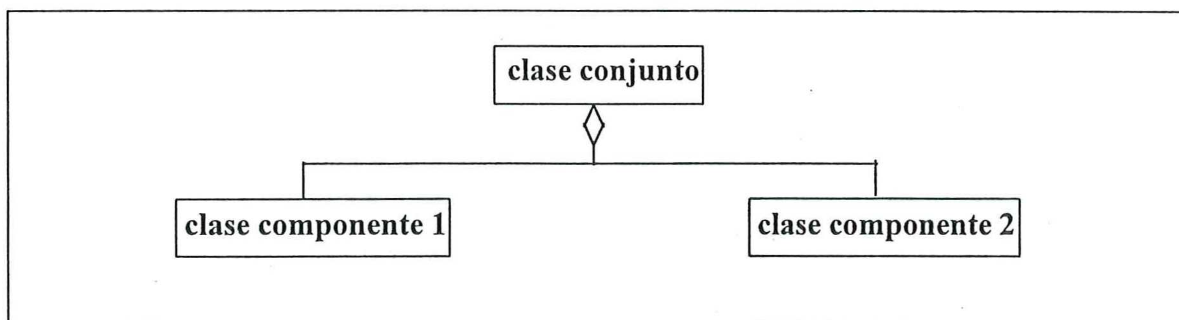


Figura 36. Notación para agregación entre clases componente y clase conjunto.

Generalización y herencia.

La *generalización* proporciona el medio para refinar un superclase en una o más subclases. La superclase contiene las características comunes de todas las clases; las subclases contienen las características específicas de cada clase. *Herencia* se refiere al mecanismo de compartir atributos y operaciones usando la relación de generalización. La Figura 37 muestra la notación empleada para representar la generalización.

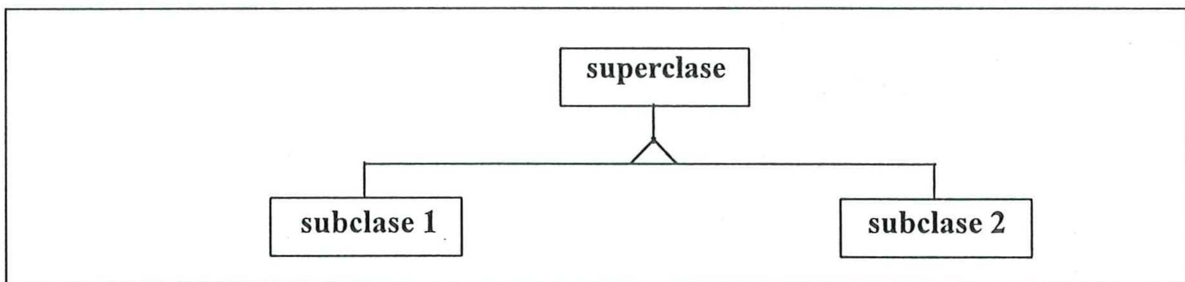


Figura 37. Notación OMT para mostrar una generalización entre superclase y subclases.

Asociaciones n-arias.

Las asociaciones pueden involucrar a más de 2 clases, es decir, pueden existir asociaciones n-arias, donde n es el número de clases involucradas en la asociación. La Figura 38 muestra una *asociación ternaria.*

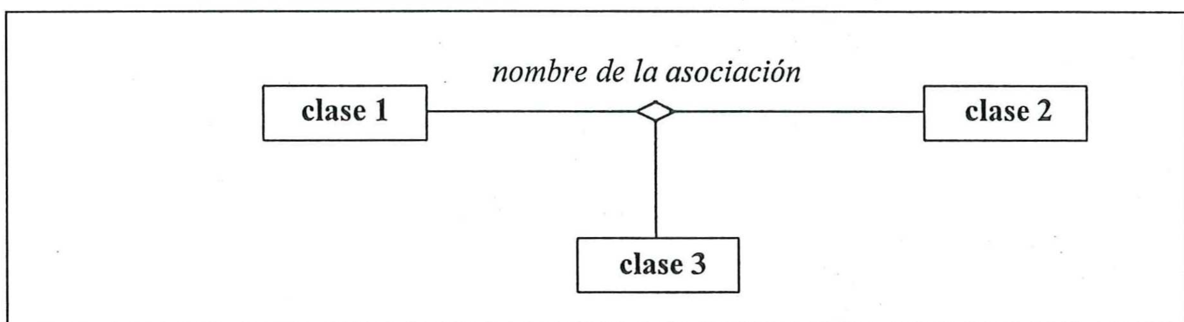


Figura 38. Asociación ternaria.

El modelo dinámico.

El modelo dinámico describe los aspectos del sistema que cambian conforme pasa el tiempo. Se usa para especificar e implantar los aspectos de control del sistema. Contiene diagramas de estados, los cuales son grafos cuyos nodos son estados y cuyos arcos son transiciones entre los estados, las cuales son causadas por los eventos. Sus elementos son: estados, transiciones y eventos. El *estado* de un objeto se determina por los valores de sus atributos y sus ligas en un momento específico. Las *transiciones* denotan los cambios en el estado de un objeto. Los *eventos* son los que disparan las transiciones de un estado a otro. La notación para ellos se muestra en la Figura 39.

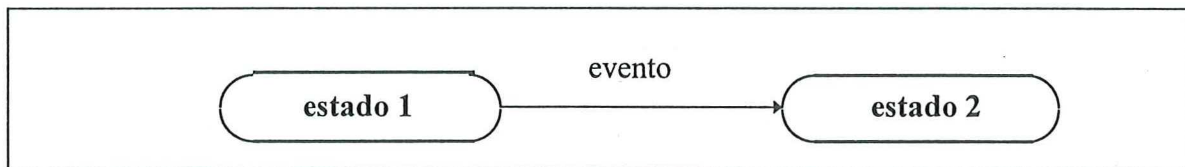


Figura 39. Estados, transiciones y eventos.

Estados inicial y final.

Los diagramas de estados pueden representar ciclos continuos en los cuales no se sabe ni donde empiezan, ni donde terminan. También se pueden especificar diagramas que contienen *estados de inicio y fin*. Los diagramas que especifican *sus estados inicial y final* son llamados "de un solo ciclo de vida". Su notación se da en la Figura 40.

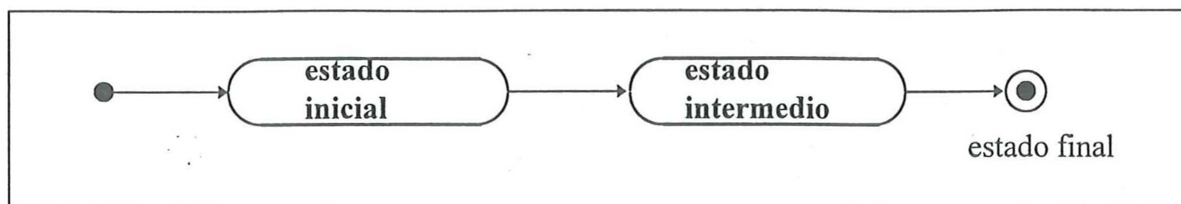


Figura 40. Notación OMT para estados inicial y final.

Condiciones.

Una condición es una función booleana sobre valores del objeto, es válida sobre un intervalo de tiempo específico. Su notación se da en la Figura 41.

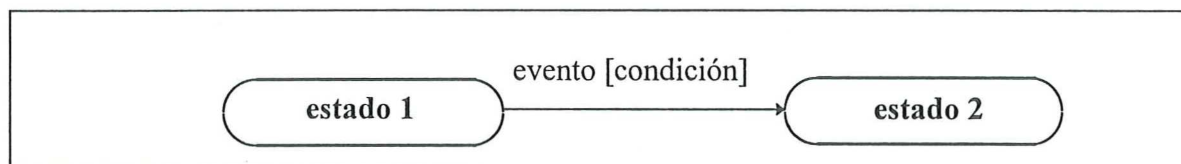


Figura 41. Notación para una transición con condición.

Operaciones.

Las operaciones que se presentan en el modelo dinámico son de dos tipos: acciones y actividades. La principal diferencia entre ellas es el tiempo de duración, usualmente las acciones son asociadas a eventos, mientras que las actividades son asociadas a estados.

Acciones.

Una *acción* es una operación instantánea, la cual se lleva a cabo al momento de que ocurre un evento. Su notación se muestra en la Figura 42.

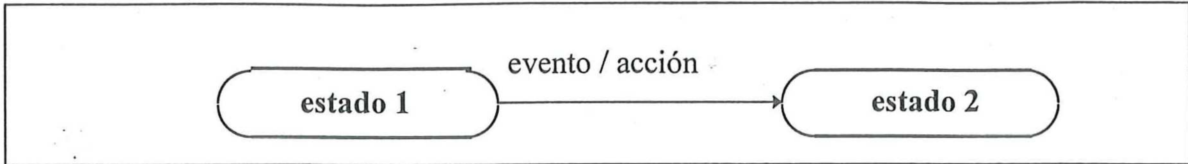


Figura 42. Notación para una transición con una acción asociada.

Actividades.

Una *actividad* es una operación que ocupa un lapso para completarse. Cuando existen transiciones que parten de un estado, las cuales no están etiquetadas con eventos, se disparan automáticamente al completarse la actividad en el estado. Su notación se muestra en la Figura 43.

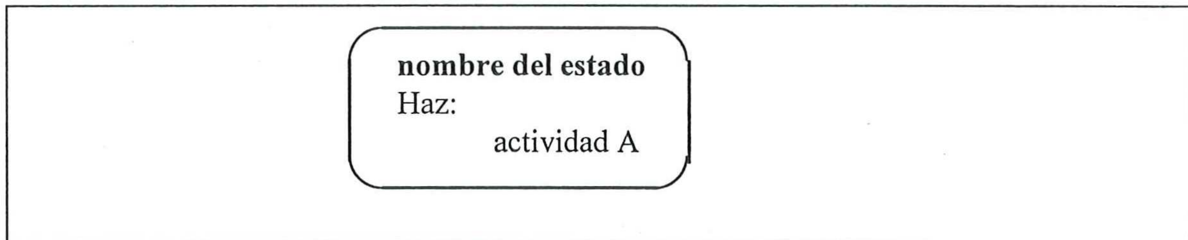


Figura 43. Notación para especificar una actividad en un estado.

Generalización de estados.

Una actividad en un estado puede ser expandida como un diagrama de más bajo nivel (subdiagrama), donde cada uno de los estados en el subdiagrama representa un paso de esa actividad. Las actividades anidadas son diagramas de un solo ciclo de vida con transiciones de entrada y salida. Lo anterior significa que estos subdiagramas deben especificar explícitamente su estado inicial (el estado final es representado implícitamente). Su notación se presenta en la Figura 44.

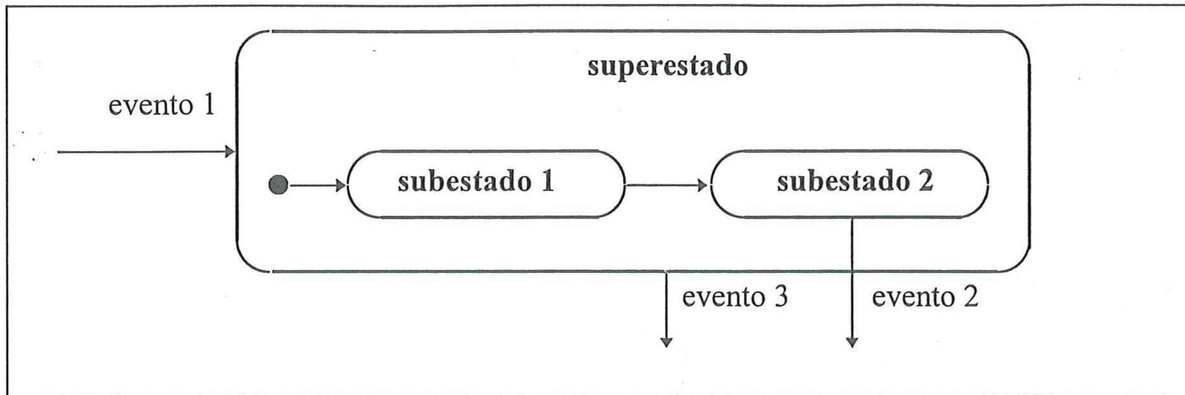


Figura 44. Generalización de estados (anidamiento).

El modelo funcional.

El modelo funcional describe las transformaciones de los valores de los datos dentro de un sistema. Contiene diagramas de flujo de datos, los cuales representan cálculos o procesamientos. Un diagrama de flujo de datos es un grafo cuyos nodos son procesos y cuyos arcos son flujos de datos. Los elementos del modelo funcional son:

Procesos.

Un *proceso* representa una transformación de los valores de los datos. Un diagrama de flujo de datos completo representa un proceso de alto nivel, el cual ha sido expandido. Los procesos en este diagrama de flujo pueden ser expandidos a su vez en procesos de bajo nivel. Eventualmente, esta recursión termina al llegar a procesos atómicos, los cuales son procesos triviales, tales como el acceso simple a una variable. Su notación se muestra en la Figura 45.

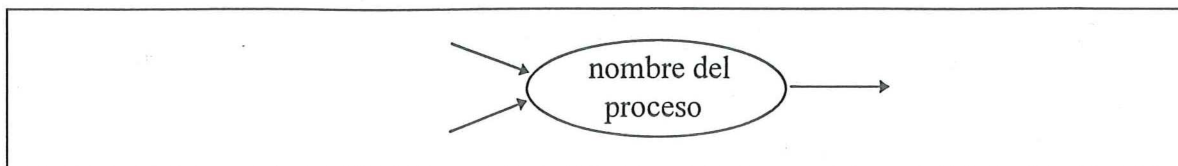


Figura 45. Notación para un proceso en un diagrama de flujo de datos.

Flujo de datos.

Un *flujo de datos* interconecta la salida de un objeto o proceso con la entrada de otro, representa los valores de datos intermedios dentro de un cálculo, los valores no son modificados por el flujo de datos. La Figura 46 muestra la notación utilizada para su representación.

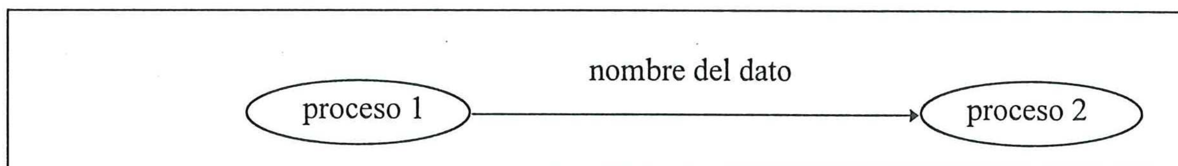


Figura 46. Flujo de datos entre dos procesos.

Actores.

Un *actor* es un objeto activo que conduce el grafo de flujo de datos, ya sea consumiendo o produciendo valores. Los actores están asociados con las entradas y salidas de flujo de datos en las fronteras de un diagrama, por ello, algunas veces se llaman terminadores. La Figura 47 muestra su representación en la notación.

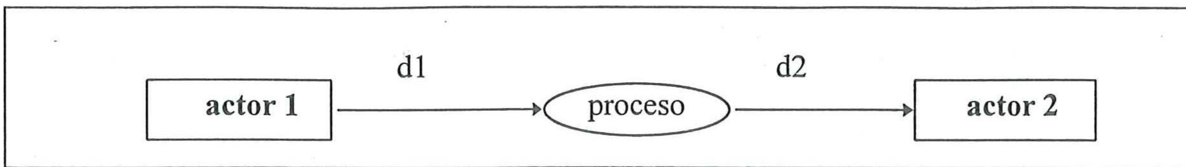


Figura 47. Notación para representar actores empezando y terminando un flujo de datos.

Almacenamiento de datos.

Un *almacén de datos* es una objeto pasivo dentro de un diagrama de flujo de datos. Este objeto, como su nombre lo indica, almacena los datos para acceso posterior. Usualmente permite que los datos puedan ser accesados de manera diferente de la que fueron generados. En la Figura 48 se muestra la notación utilizada para su representación. La Figura 49 muestra la notación para los diferentes tipos de acceso y actualización a los almacenamientos de datos.

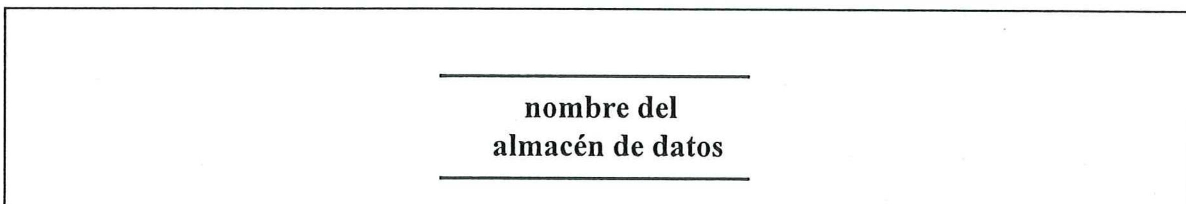


Figura 48. Notación para representar un almacén de datos.

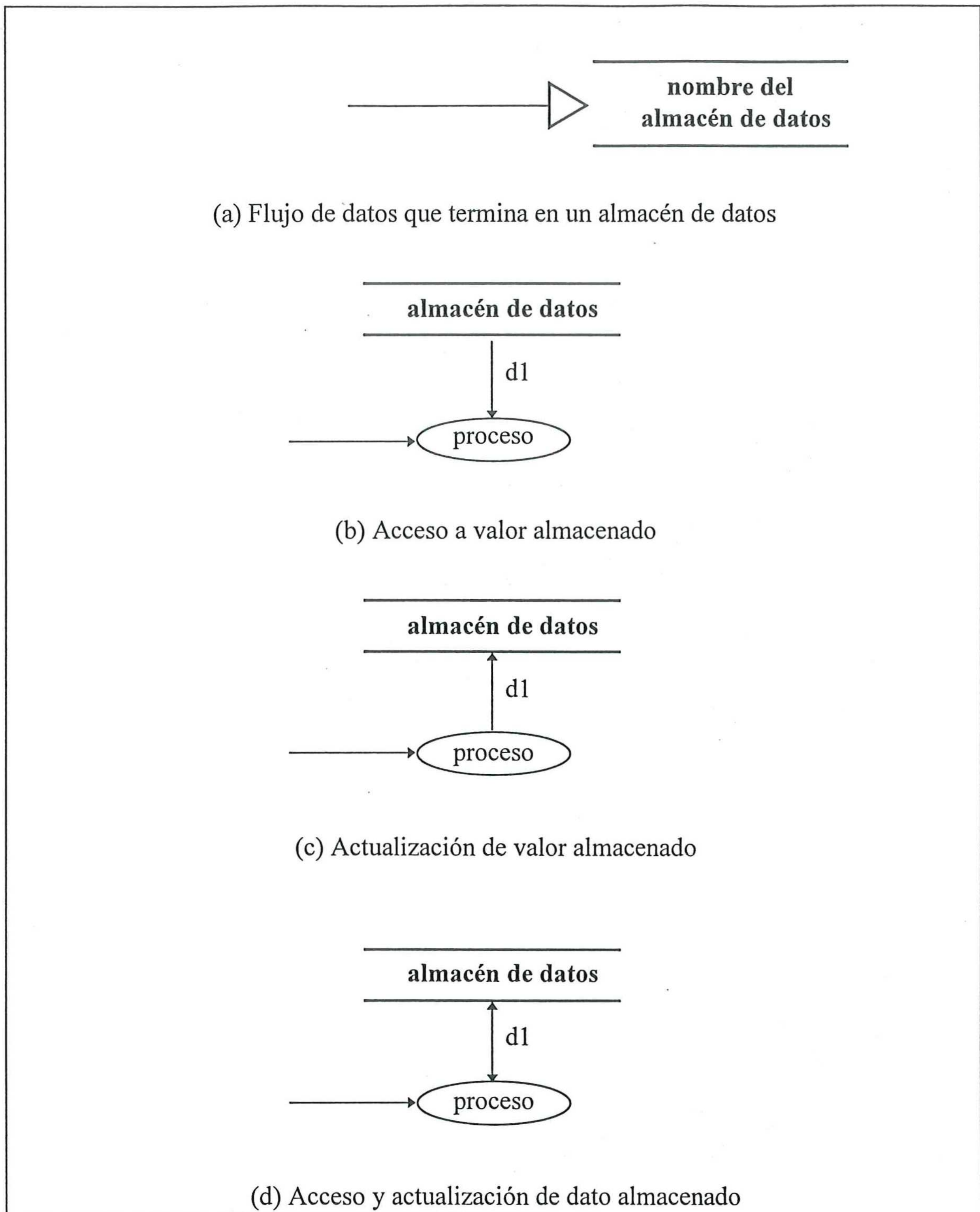


Figura 49. Notación para diferentes combinaciones de acceso y actualización a almacenes de datos.

APÉNDICE C

CASOS DE PRUEBA

En total se utilizaron siete casos de prueba para verificar los resultados que se generan con diferentes versiones del código. Cada caso está definido en un archivo de inicialización para la aplicación. El formato y el contenido de los archivos se muestran en las Figuras 50 - 57.

```
headtext                                  //encabezado
base                                      //archivo
nmax  dt  nwrite                          // iteraciones , incremento, escrituras
nx  ny  ibcx  ibcy                        // tamaño de la malla, condiciones de frontera
mod  itop                                 // modulación beta o gama y topografía
(be,ga, f) ( ,xc,h0) ( ,yc,a) ( , ,b) ( , ,xc) ( , ,yc)
u  ang                                    // velocidad, dirección, centro y radio
xcv  ycv  a                               // del dipolo
```

Figura 50. Formato del archivo para los casos de prueba.

```
Un dipolo de Lamb con anillo
test
20  .01  10
64  64  3  3
0  0
5  0
32  32  5
```

Figura 51. Caso INI_TEST, dipolo de Lamb con anillo (sin vorticidad ambiental ni topografía).

```
Un dipolo de Lamb con vorticidad ambiental
beta
1000 .01 100
64 64 3 3
1 0
0.1 32 32
5 0
32 32 5
```

Figura 52. Caso INI_BETA, dipolo Lamb con vorticidad ambiental (plano β).

```
Un dipolo de Lamb con vorticidad ambiental
gama
1000 .01 100
64 64 3 3
2 0
0.1 32 32
5 0
32 32 5
```

Figura 53. Caso INI_GAMA, dipolo Lamb con vorticidad ambiental (plano γ).

```
Un dipolo de Lamb con topografía
top1
1000 .01 100
64 64 3 3
0 1
1 1 0.1 5 32 32
5 0
16 32 5
```

Figura 54. Caso INI_TOP1, dipolo Lamb con topografía (caso 1).

```
Un dipolo de Lamb con topografía
top2
1000 .01 100
64 64 3 3
0 2
1 1 0.1 5 32 32
5 0
16 32 5
```

Figura 55. Caso INI_TOP2, dipolo Lamb con topografía (caso 2).

```
Un dipolo de Lamb con topografía
top3
1000 .01 100
64 64 3 3
0 3
1 1 0.1 5 32 32
5 0
16 32 5
```

Figura 56. Caso INI_TOP3, dipolo Lamb con topografía (caso 3).

```
Un dipolo de Lamb con topografía
top4
1000 .01 100
64 64 3 3
0 4
1 1 0.1 5 32 32
5 0
16 32 5
```

Figura 57. Caso INI_TOP4, dipolo Lamb con topografía (caso 4).