Asociación Argentina
de Mecánica Computacional

Mecánica Computacional Vol XXXV, págs. 1391-1409 (artículo completo)
Martín I. Idiart, Ana E. Scarabino y Mario A. Storti (Eds.)
La Plata, 7-10 Noviembre 2017

IMPLEMENTACIÓN COMPUTACIONAL DEL MÉTODO DE RED DE VÓRTICES INESTACIONARIO: UNA VERSIÓN BASADA EN LOS PARADIGMAS DE PROGRAMACIÓN ORIENTADA A OBJETOS Y CO-SIMULACIÓN

Martín E. Pérez Segura[a,b], **Mauro S. Maza**[b] **y Sergio Preidikman**[a,b]

[a]*Instituto de Estudios Avanzados en Ingeniería y Tecnología, IDIT UNC-CONICET.*

[b]*Dpto. de Estructuras, Facultad de Ciencias Exactas, Físicas y Naturales, Universidad Nacional de Córdoba, Córdoba, Argentina*

Palabras Clave: Multi-física computacional, Método de Red de Vórtices Inestacionario y No Lineal, Programación Orientada a Objetos, Computación de Alto Desempeño.

Resumen. La multi-física computacional ha adquirido un papel preponderante a medida que los avances en el ámbito de la investigación científica en ingeniería impulsaron la aparición de modelos cada vez más complejos y que exceden los métodos de análisis tradicionales. En este contexto, el presente desarrollo intenta proporcionar un marco generalizado de simulación computacional para análisis fluido-dinámico basado en el método de red de vórtices inestacionario y no lineal, enfatizando aspectos tales como: *i*) el desarrollo de software fundado en el paradigma de la Programación Orientada a Objetos, *ii*) la utilización de técnicas de Computación de Alto Desempeño; y, *iii*) el diseño de software conducente a la vinculación con otras herramientas computacionales. El desarrollo siguiendo el paradigma de la Programación Orientada a Objetos facilita la implementación, la lectura, el mantenimiento y la extensibilidad del código, garantizando la expansibilidad necesaria para su utilización eficiente por distintos grupos de investigación. Por otro lado, se incorporaron rutinas de pre-procesamiento que requieren cantidades mínimas de información como datos de entrada. Esto brinda una gran flexibilidad frente a otras herramientas que tienen importantes restricciones respecto a la definición topológica de las mallas utilizadas. Se admiten también múltiples cuerpos rígidos y/o flexibles, definidos a partir de grillas independientes, con gran versatilidad en cuanto a: la elección de las zonas de convección de estelas, los tipos de elementos a utilizar, y otras características tales como condiciones de permeabilidad, y determinación de velocidades en puntos arbitrarios inmersos en el seno del fluido. Aún con una estructura de programación secuencial, el método de red de vórtices posee una excelente relación entre aplicabilidad y costo de cálculo. Sin embargo, se explotan varias de las posibilidades que ofrece el método para paralelizar el cómputo. El objetivo general de este esfuerzo es construir una herramienta funcional que combine la aceptación y aplicabilidad del método de red de vórtices, con la confiabilidad y versatilidad de un código modular bajo la metodología de la Programación Orientada a Objetos. Además, se explora la generación de estructuras computacionales de co-simulación que permitan atacar problemas gobernados por una aerodinámica inherentemente inestacionaria y no lineal.

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

1 INTRODUCCIÓN

Actualmente, la investigación y el desarrollo en el ámbito de la ingeniería y la tecnología han incorporado la simulación computacional como práctica habitual para alcanzar resultados. En este contexto, se han impuesto metas cada vez más ambiciosas en cuanto a la representatividad de los modelos computacionales, obligando la sofisticación de las técnicas de análisis en pos del detalle en la descripción de los fenómenos estudiados. En este sentido, la multi-física es la disciplina computacional que ha permitido la implementación de un enfoque particionado de fenómenos complejos, involucrando diversas áreas de la física en un proceso de co-simulación, basado en la combinación de modelos disímiles altamente especializados.

En lo que concierne a este trabajo, la co-simulación se enmarca en la aeroelasticidad computacional (CAE) como método de análisis de la interacción entre fluido y estructuras (FSI). Conceptualmente, se abordan por separado el problema aerodinámico y el problema estructural-dinámico, con modelos apropiados para cada uno de ellos, requiriendo de la incorporación de un método de interacción entre ambos. Bajo estas inferencias generales, se reduce el enfoque al desarrollo del modelo aerodinámico, implementando el Método de Red de Vórtices Inestacionario (UVLM).

La construcción de un modelo de investigación computacional no puede concebirse de manera aislada, sino íntimamente relacionada con un lenguaje y estilo de programación, y un entorno de desarrollo. Luego, fijar el alcance de la implementación y, en consecuencia, definir un paradigma como guía resulta preponderante. En muchos aspectos, y a pesar de la generalidad alcanzada por la programación procedural, para los grupos de investigación resulta necesario optar por un estilo más adecuado. De entre las posibilidades que cuentan con probada aceptación, el presente desarrollo adopta los paradigmas de la Programación Orientada a Objetos (OOP) y la co-simulación, en el entorno de la investigación científica, para fijar las pautas que guían la implementación del UVLM. Por lo tanto, el objetivo general de este esfuerzo es construir una herramienta funcional que combine la aceptación y aplicabilidad del UVLM, con la confiabilidad y versatilidad de un código modular bajo la metodología de la OOP.

En particular, se desarrolla un software de simulación fluido-dinámica desde el punto de vista del entorno de investigación. Esto es, con el fin de facilitar la lectura, la comprensión y el mantenimiento del código, explotando en la mayor medida posible las ventajas que la OOP proporciona a este respecto. Del mismo modo, se favorece la flexibilidad y expansibilidad para futuros desarrollos, el intercambio y divulgación de módulos, y la vinculación con otros modelos, sin la necesidad de modificaciones sustanciales.

Como características adicionales, se acentúa la robustez y la generalidad incluyendo mecanismos de pre-proceso que garantizan la correcta ejecución del método, reduciendo las exigencias y restricciones impuestas al usuario en cuanto a la definición de parámetros de simulación. Así también, con la adaptabilidad como premisa, se reduce al mínimo la información a proporcionar como datos de entrada, a los fines de admitir la construcción de simulaciones con independencia respecto de las características de los mismos. Igualmente, se permite desestructurar los datos de salida dando mutabilidad para generar visualizaciones, pos-procesamientos, nuevas iteraciones, etcétera. En cuanto al desempeño, se prioriza la reducción de tiempos de ejecución implementando técnicas de Computación de Alto Desempeño (HPC) y aprovechando las posibilidades de ejecución en paralelo. Todo el desarrollo se realiza utilizando FORTRAN 2008 como lenguaje de programación, siguiendo con la tendencia del contexto de trabajo y debido a su versatilidad y condiciones tanto para la OOP como para técnicas de HPC. No obstante, todas las conclusiones son independientes del lenguaje utilizado y, por ende, extrapolables a otros, siempre que soporten el paradigma de la OOP.

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

2 PARADIGMAS DE PROGRAMACIÓN

La implementación computacional de un modelo fluido-dinámico, al igual que con cualquier modelo físico de simulación, requiere un exhaustivo proceso de análisis y diseño previo a la implementación propiamente dicha. Aquí intervienen diversos niveles de abstracción que se concatenan progresivamente desde el fenómeno físico que se pretende representar hasta el software desarrollado como herramienta de simulación. Este proceso define lo que se conoce como Paradigma de Programación. Es decir, representa un enfoque particular que define los conceptos, los sistemas de abstracción y los procedimientos que integran la solución del problema en análisis, en el marco del desarrollo computacional. Un diagrama representativo de estos conceptos se muestra en la Figura 1.

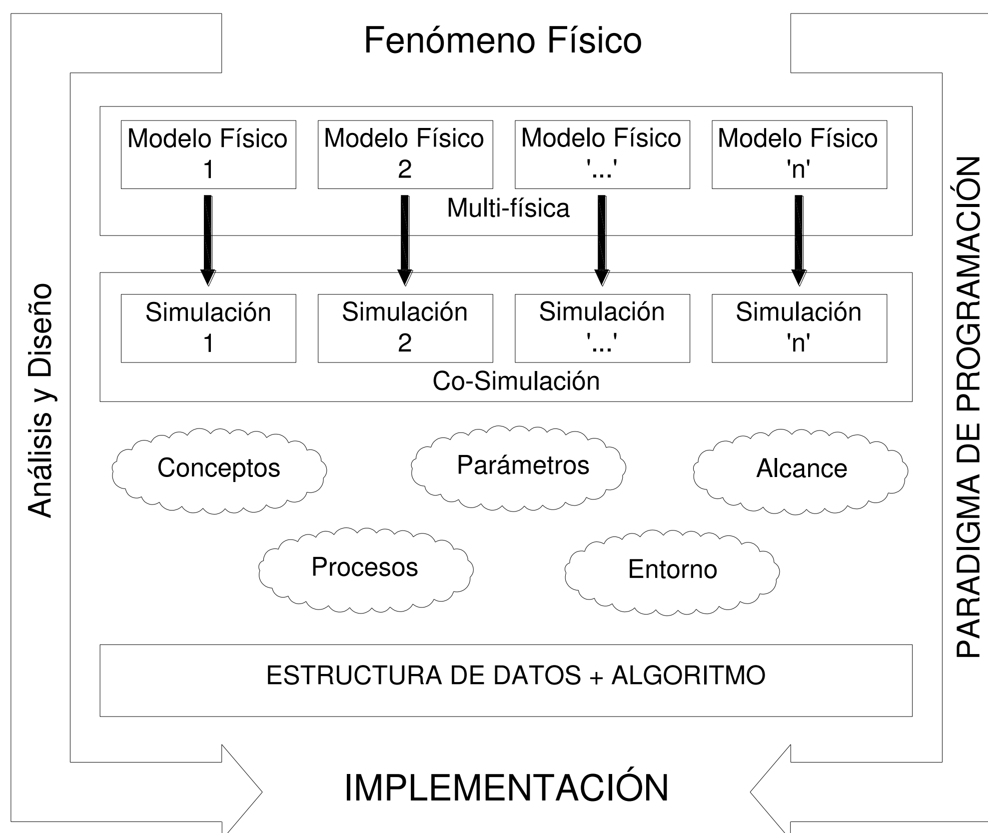

Figura 1: Paradigma de Programación, análisis y diseño.

En términos generales, se puede concebir un software como la combinación de dos partes: una estructura de datos y un algoritmo que opera sobre ésta. El paradigma adoptado para el desarrollo guarda una estrecha relación con ambas, determina cómo se construye cada una y cómo se vinculan entre sí. Evidentemente el contexto y la funcionalidad del código en desarrollo definen la factibilidad de adoptar un determinado paradigma, al igual que el lenguaje de programación a utilizar. Mientras que la aceptación y el uso dentro de la comunidad de investigación también resultan influyentes. De ahí la importancia de adoptar un paradigma para estructurar y unificar los desarrollos.

Existen diversos paradigmas que han surgido y mutado en función de los cambiantes requerimientos en el desarrollo de softwares. Por lo tanto, la bibliografía especializada suele ser difusa a la hora de establecer los límites de cada uno de ellos. No obstante, y en vista del alcance del presente trabajo, los paradigmas pueden asociarse en dos grandes grupos. Por un lado,

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

aquellos relacionados con la programación imperativa y, por otro, los relacionados con la programación declarativa.

En su primer nivel de abstracción, la programación imperativa consiste en definir el estado del programa y ejecutar sentencias que progresivamente modifican dicho estado para alcanzar la solución buscada. En otras palabras, la programación imperativa establece cuidadosamente cómo se desarrolla una tarea específica y conduce al programa a través de las instrucciones necesarias para completar dicha tarea.

La programación declarativa, en cambio, se basa en definir condiciones que describen el problema y la búsqueda de la solución se realiza a través de mecanismos de control. No es central definir cómo realizar una tarea sino detallar minuciosamente qué tarea se quiere ejecutar y dar paso a mecanismos de inferencia de información a partir de la misma.

La clasificación precedente puede parecer imprecisa, pero constituye la base para identificar y calificar un paradigma. La programación científica en ingeniería es en buena parte imperativa (al menos de manera conceptual), y es dentro de este grupo que se encuentran los paradigmas que constituyen opciones factibles a la hora de la implementación computacional de modelos multi-físicos. El ejemplo más ampliamente difundido (o el más naturalizado) de programación imperativa es, probablemente, el Paradigma Procedural. Este paradigma se deriva o incluye dentro del paradigma de programación estructurada y resulta dificultoso describir uno sin el otro.

El Paradigma de Programación Procedural se caracteriza por estructurar el código en bloques (o procedimientos) reutilizables a los que puede recurrirse de manera arbitraria durante la ejecución. Por lo tanto, el proceso de diseño de código procedural se basa en identificar secuencias de instrucciones concretas que puedan aislarse y estructurarse para ejecutarse cada vez que sean necesarias. Luego, se definen entornos locales para cada procedimiento que se vinculan, a través de un punto de entrada y un punto de salida, con el entorno principal. Durante la ejecución de cada procedimiento las variables (o estados) del programa se modifican con una finalidad específica en función de los datos disponibles en el punto de entrada. Cada uno de estos procedimientos se identifica con rutinas, subrutinas, funciones, etcétera, dependiendo del lenguaje utilizado.

L. J. Aguilar (Aguilar, L. J., 1996) expone algunas de las desventajas de la Programación Procedural. Hace referencia a la cantidad de recursos que se consumen en propagar una modificación de estado a través de todos los procedimientos involucrados, la criticidad del medio en el que se almacenan dichos estados, la dificultad para concebir abstracciones como procedimientos y las limitaciones en la intercambiabilidad de código dentro del grupo de desarrollo. En resumen, la Programación Procedural permite optimizar el desarrollo de procedimientos (o algoritmos) pero presenta significativas debilidades a la hora de construir estructuras de datos.

Otra alternativa, que ha cobrado importancia últimamente y a la que el presente trabajo toma como eje, la constituye el Paradigma de la Programación Orientada a Objetos. A pesar de ser una variante de la programación imperativa, la OOP requiere una abstracción algo diferente y una estructura diseñada desde otro enfoque. Un análisis más amplio se presentará más adelante.

Si bien, desde el punto de vista de la abstracción teórica, el diseño de un software puede encuadrarse dentro de un paradigma definido, cuando se trata con la realización los límites no son tan claros y múltiples paradigmas pueden solaparse. Se da lugar entonces a lo que se conoce como programación multi-paradigma. Este estilo combinado fija como guía la metodología de un paradigma en función del objetivo general, pero incluye atributos de otros paradigmas para perseguir los objetivos particulares del proyecto. Evidentemente, utilizar más de un estilo de programación restringe el universo de lenguajes utilizables puesto que no todos son aptos para la multiplicidad de paradigmas.

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

3 MODELO AERODINÁMICO: UVLM

El modelo aerodinámico se concibe a partir de un método que asume un flujo irrotacional y no viscoso, basado en la distribución de singularidades. En particular, las singularidades usadas son vórtices cuyas intensidades se obtienen de resolver ecuaciones integrales con determinadas condiciones de contorno sobre las superficies sólidas y en el infinito.

Se introduce entonces el campo de vorticidad en el modelo cuyo comportamiento dinámico queda caracterizado por la convección de la misma desde las superficies sustentadoras a la velocidad local de las partículas de fluido, mientras se desprecie la difusión. Luego, las zonas con vorticidad convectada en el seno del fluido conforman las estelas. Según expone M. S. Maza (Maza, M. S., 2013), para números de Reynolds crecientes, la distribución de vorticidad se vuelve lo suficientemente compacta como para considerar el dominio dividido en dos zonas: una pequeña porción del espacio ocupada por fluido rotacional (estelas y capa límite) y la porción restante, donde el fluido se asume irrotacional.

En forma general, si se verifican ciertas hipótesis para que las capas límites y estelas sean lo suficientemente compactas pueden representarse con sábanas vorticosas. Las capas límites se modelan con una superficie vorticosa adherida al sólido que se compone de paneles formados por una red o grilla de segmentos vorticosos que condensan o discretizan la vorticidad distribuida sobre ellas. Mientras que, para las estelas, la sábana vorticosa se compone sólo de segmentos que definen una grilla y se mueven libremente en el seno del fluido. Además, cada panel de la grilla adherida cuenta con un punto de control donde se calculan las cargas aerodinámicas y se imponen las condiciones de contorno.

El esquema de la Figura 2 muestra las entidades básicas involucradas en el modelo que se corresponden con lo antes descrito. En primer lugar, la grilla ("GRID") representa una superficie sustentadora y su cinemática está definida por el cuerpo al cual está adherida. En ella, se identifican paneles, construidos por nodos y segmentos, que cuentan con un punto de control y un versor normal que define su orientación. Cada segmento vorticoso está definido, a su vez, por dos nodos y discretiza una porción de la vorticidad de la grilla. Además, se identifica una línea de convección, formada por un grupo de segmentos consecutivos, que permite determinar a priori el lugar desde el que se convecta la estela. En cuanto a la estela ("WAKE"), está representada por una sábana vorticosa libre, es decir que se mueve con el fluido. En este caso, no se definen paneles sino solo segmentos vorticosos que condensan la vorticidad y están definidos por nodos, al igual que antes.

Con estos elementos se calculan las variaciones de coeficiente de presión ΔC_p sobre cada panel, que se relacionan con las cargas aerodinámicas sobre la superficie y es el principal resultado del método, definido como:

$$\Delta C_p = C_p\big|_L - C_p\big|_U = \left[\frac{p - p_\infty}{q}\right]\Bigg|_L - \left[\frac{p - p_\infty}{q}\right]\Bigg|_U \quad (1)$$

Donde p representa la presión sobre la superficie del panel, p_∞, la presión de referencia, C_p, el coeficiente de presión, y los subíndices L y U indican posiciones inmediatamente por encima y por debajo de la superficie del panel, respectivamente.

La deducción formal del método excede el alcance de este trabajo, por lo que para mayores precisiones al respecto se refiere al lector al trabajo de Preidikman S. (1998).

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

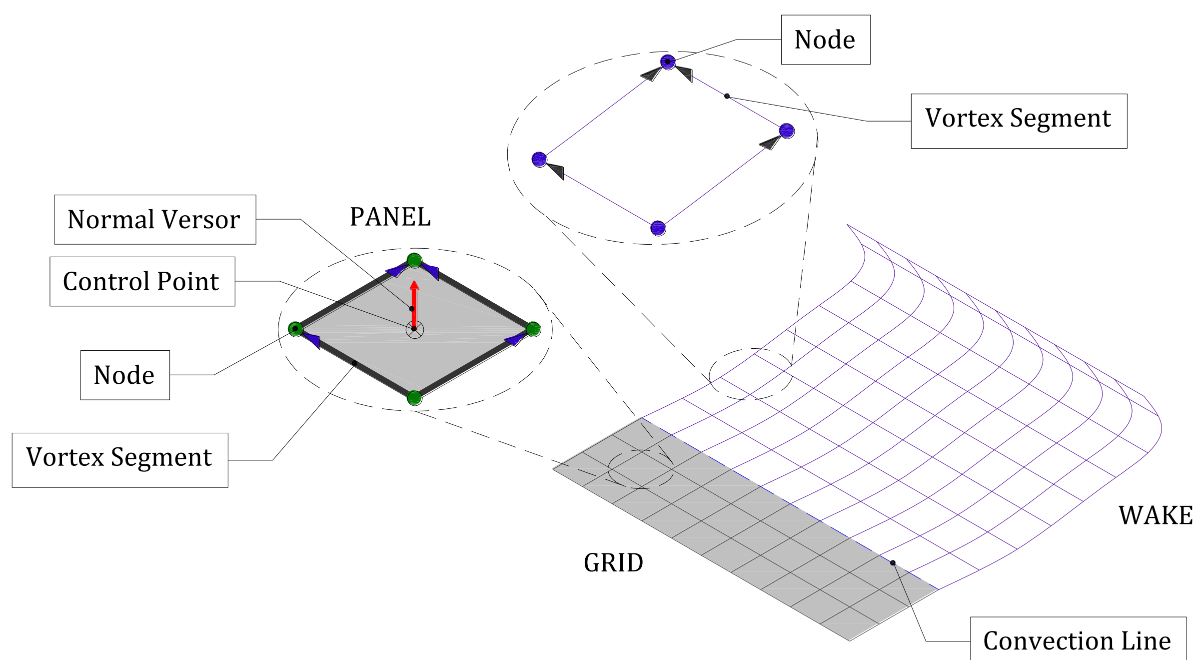

Figura 2: UVLM, descripción del modelo.

4 PROGRAMACIÓN ORIENTADA A OBJETOS: IMPLEMENTACIÓN

La bibliografía dedicada a exponer y analizar los conceptos acerca de la OOP suele agruparse en dos extremos diametralmente opuestos. Por un lado, se tienen obras sobre arquitectura de software que son sumamente especializadas y exceden el alcance del presente trabajo. Por otro, aquellas que abordan los conceptos con ejemplos imprácticos que dan nociones generales y simplificadas, pero detienen el estudio considerablemente antes de enfocarse en la aplicación. Por ende, en este trabajo se explora un tratamiento intermedio utilizando la implementación del UVLM para describir la OOP, en cuanto al diseño y análisis.

El concepto de OOP fue formalmente introducido en la década de 1960 a partir de estudios realizados en el Centro Noruego de Cómputos por O. J. Dahl y K. Nygaard. Conceptualmente, G. Booch (Booch, G., 2007) define la OOP como: “un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representan una instancia de alguna clase, y cuyas clases son todas miembros de una jerarquía de clases unidas mediante relaciones de herencia”. Asimismo, más adelante en su libro sugiere que los pilares fundamentales de la OOP son la *abstracción*, la *encapsulación*, la *modularidad* y la *jerarquía* o *herencia*, a riesgo de no poder definir un modelo como orientado a objetos ante la falta de alguno de ellos. Sin embargo, un análisis más moderno incluye al *polimorfismo* como el quinto pilar, además de las *clases* y *objetos* como unidades fundamentales del diseño.

De manera general, la OOP permite una representación de los fenómenos en un modo fácil y natural, reduciendo el proceso de abstracción que comprende el paso de las especificaciones en términos del usuario a los requisitos del sistema en términos del código. Este estilo satisface una necesidad actual del desarrollo de softwares: es capaz de manipular diversos tamaños de sistemas de manera fiable, dándoles flexibilidad, mantenibilidad y la capacidad de mutar ante cambios en los requerimientos.

4.1 Abstracción: clases y objetos

La abstracción se entiende como el proceso de enfrentarse a la complejidad inherente de un software. Se centra en la representación de las características externas de una entidad del

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

modelo para describirlo desde el punto de vista de la implementación y en utilizar esta descripción en el código.

En OOP una *clase*, también llamada tipo abstracto de datos (ADT), representa el concepto de abstracción en el programa. Es la estructura fundamental y se identifica con una entidad del modelo a representar. En la implementación se utiliza el Principio de Responsabilidad Simple (SRP) para definir las clases que intervendrán en el código. Esto es, se asocia una clase a cada entidad del modelo que tenga una función específica. La Figura 3 presenta un *diagrama de clases* a través del *Lenguaje Unificado de Modelado* (UML) (Fowler. M., Kendall, S., 1999), en él se exponen todas las clases presentes en el código desarrollado y las relaciones que se construyen entre ellas.

Cada ADT se compone de un *nombre* que lo identifica, *atributos* que lo caracterizan y *procedimientos* que pueden actuar sobre éste. En FORTRAN 2008, se utilizan módulos para implementar los ADT. Esto es, cada clase se define dentro de un módulo que luego será incorporado en las *dependencias* de otro si fuera necesario y, a su vez, se relacionará con otros a través de sus propias dependencias. Si bien, la interdependencia de módulos es un aspecto específico del lenguaje utilizado y no constituye una característica de la OOP, su utilización es imprescindible para relacionar subrutinas. Al respecto, la Figura 4 muestra el módulo de implementación de la clase "`Panel`" y se identifican cada uno de sus componentes.

Cabe destacar aquí que, si bien una clase por sí misma representa una abstracción, no es capaz de aportar a la implementación sin que se generen instancias de ella. Equivalentemente, un tipo de dato no es útil, en la práctica, si no existe un dato particular de ese tipo.

Un *objeto* se define como una instancia, ejemplo o caso de una clase, y permite la aplicación práctica de ésta. Es decir, una clase representa un tipo de dato genérico y un objeto es un ente específico de dicho tipo de dato. Por ejemplo, el panel con índice número 9 es un objeto de la clase panel y tiene a su vez atributos específicos que lo caracterizan, tal y como se muestra en la Figura 5.

4.2 Encapsulación y modularidad

La encapsulación representa quizás uno de los aspectos menos tangibles de la OOP, pero es, en términos de las intenciones del presente trabajo, uno de los más importantes. También conocida como "ocultamiento de información", la encapsulación es la inclusión de todos los recursos que necesita un objeto para su funcionalidad dentro del mismo objeto. Está fuertemente relacionada con la creación de clases y garantiza que estos recursos estén ocultos para el resto de los objetos y sean sólo accesibles a través de mecanismos de vinculación.

Regresando a la Figura 4 y a modo de ejemplo, todos los procedimientos enlistados en el bloque *`CONTAINS`* son específicos de la clase "`Panel`" y, si bien pueden relacionarse con otras clases, su contenido está restringido a cada objeto definido como "`Panel`". En particular para FORTRAN 2008, el uso de los atributos "`Public`" y "`Private`" reflejan el nivel de ocultamiento de cada procedimiento y atributo, en cuanto al acceso a través de mecanismos de vinculación (Metcalf, M., Reid, J., Cohen, M., 2011).

Por otro lado, por modularidad se entiende a la propiedad de implementar, valga la redundancia, módulos. Esto es, la posibilidad de subdividir el código en partes independientes en funcionalidad pero que pueden vincularse, aunque esta relación es, como lo indica L. J. Aguilar (Aguilar, L. J., 1996), débil. En términos computacionales, y en particular en FORTRAN 2008, un módulo es una unidad que puede compilarse independientemente sin necesidad de definir a priori sus relaciones con otras. Así se consigue una construcción progresiva del software que, junto con la encapsulación aplicada de manera apropiada, optimiza los desarrollos en grupos de investigación.

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

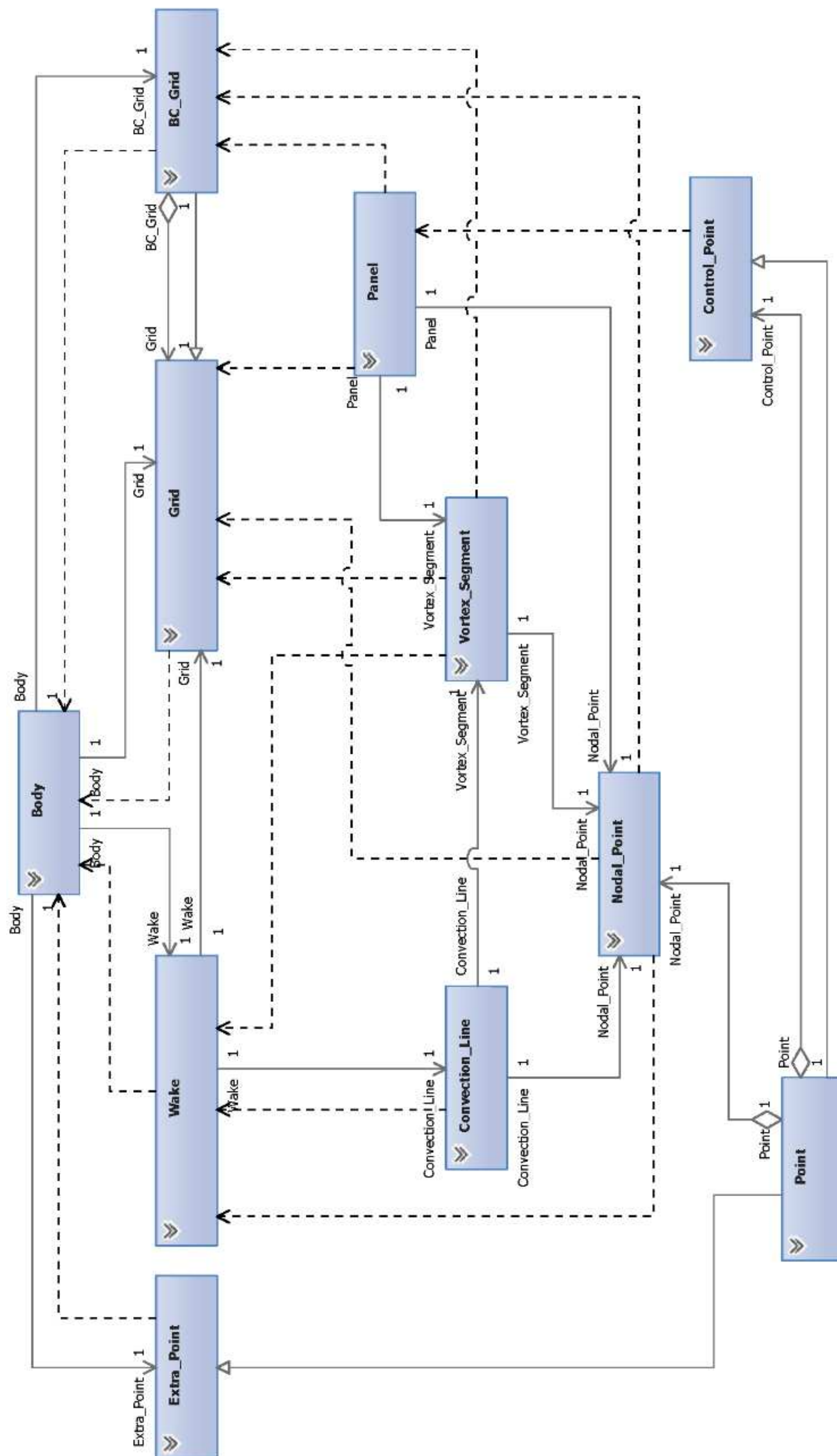

Figura 3: Diagrama de Clases (UML).

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

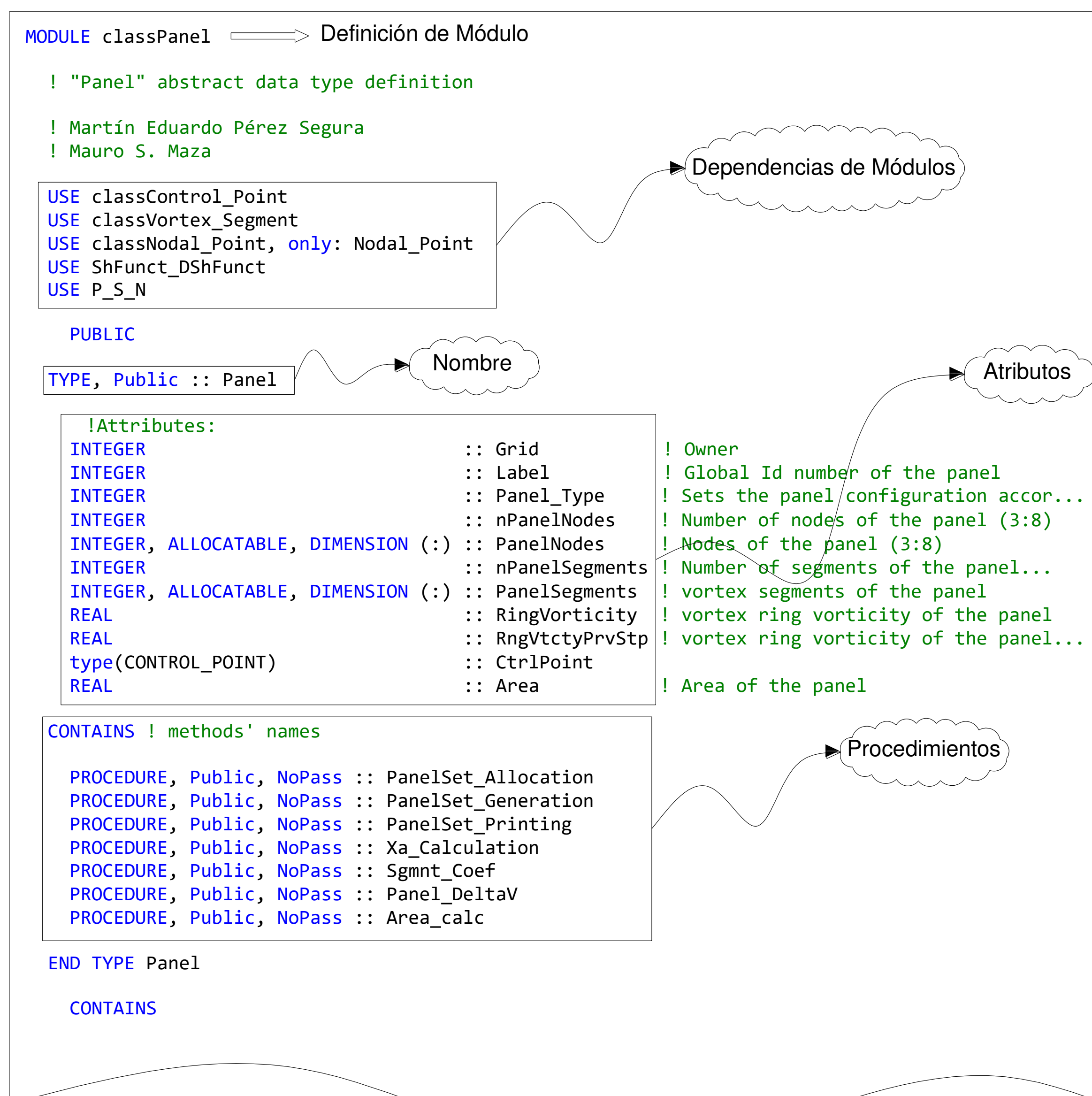

Figura 4: Módulo de implementación de clase “Panel”.

4.3 Jerarquía o herencia

El verdadero potencial del OOP se vuelve evidente cuando se analizan las relaciones entre clases. Estas relaciones ordenan las abstracciones dentro del diseño del código y pueden ser de diversa naturaleza. Las más difundidas e importantes son aquellas que refieren a la generalización/especialización y a la agregación. El primer caso se conoce como “*Is-a*”, por su nombre en inglés, mientras que el segundo, por el mismo motivo, se nombra “*Has-a*”.

La relación de jerarquía de clases o “*Is-a*” (también llamada herencia) se concibe a partir de derivar una nueva clase de otra existente. Luego, la segunda “hereda” las características, atributos y procedimientos de la primera. Adicionalmente, es posible incluir nuevas características exclusivas de la clase derivada. Dependiendo del sentido de análisis, este proceso puede considerarse una generalización o una especialización. En FORTRAN 2008, las relaciones de herencia se especifican con el atributo “`Extend`” como se muestra en el ejemplo de la Figura 6.

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

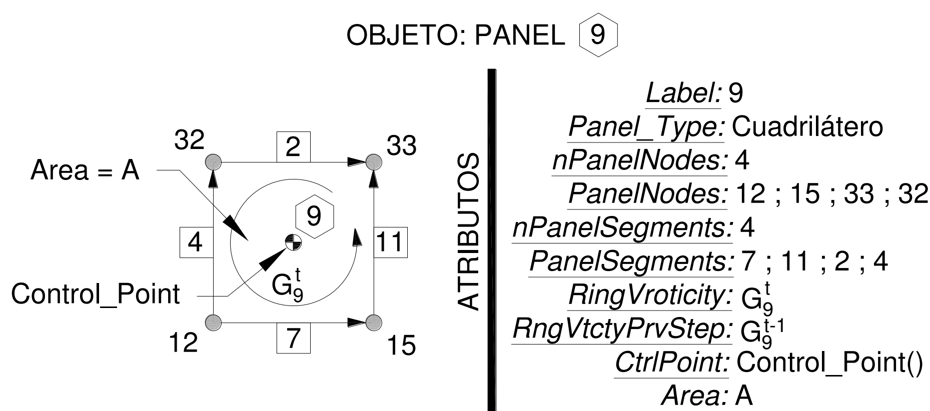

Figura 5: Objeto, Panel Nº 9.

Por otro lado, la relación "*Has-a*" (o de composición de objetos) permite reducir la complejidad agrupando objetos individuales en grupos o dentro de otros objetos. Es decir, se utiliza un objeto de una clase como atributo de otro objeto de una clase distinta. Esto permite, por ejemplo y como se muestra en la Figura 7, que un objeto de la clase "`Grid`" contenga un conjunto de objetos de la clase "`Nodal_Points`" para representar los nodos que la componen y, de igual modo, con paneles ("`Panel`") y segmentos vorticosos ("`Vortex_Segment`").

4.4 Polimorfismo

Otra característica que ilustra el potencial de la OOP es el Polimorfismo, aunque desde algunos puntos de vista más ortodoxos no se considere fundamental. De acuerdo a lo que intuitivamente se prevé, el Polimorfismo refiere a la posibilidad de que una entidad, una clase en este caso, tome diversas formas. En un sentido más específico, una variable polimórfica es aquella cuyo tipo de dato puede variar en tiempo de ejecución.

En FORTRAN 2008, existen dos formas principales de polimorfismo. En primer lugar, el llamado *de tipo compatible* ("*type-compatible*") que se representa con la clave "`class`" y es una versión más restringida. En este caso, el objeto polimórfico sólo puede mutar a una clase compatible con su tipo, es decir, a una clase extendida (en generalización/especialización) de ella. Por ejemplo, la Figura 8 muestra un procedimiento (subrutina) donde la variable de ingreso se declara con el atributo "`CLASS()`". Este tipo de declaración de variable permite que el procedimiento sea transparente a todas las clases, o mejor dicho a los objetos de todas las clases, extendidas de aquella indicada en el argumento del atributo, en particular "`Grid`". Luego, las instrucciones serán válidas para los objetos de clases "`Grid`" o "`BC_Grid`" (ver Figura 3.) mientras sean de tipo compatible. Ésta última condición hace referencia a que, si alguna instrucción utiliza algún atributo exclusivo de una de las clases compatibles, deberá explicitarse para evitar errores, lo que se logra utilizando el constructor "`Select Type`" (Metcalf, M., Reid, J., Cohen, M., 2011).

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

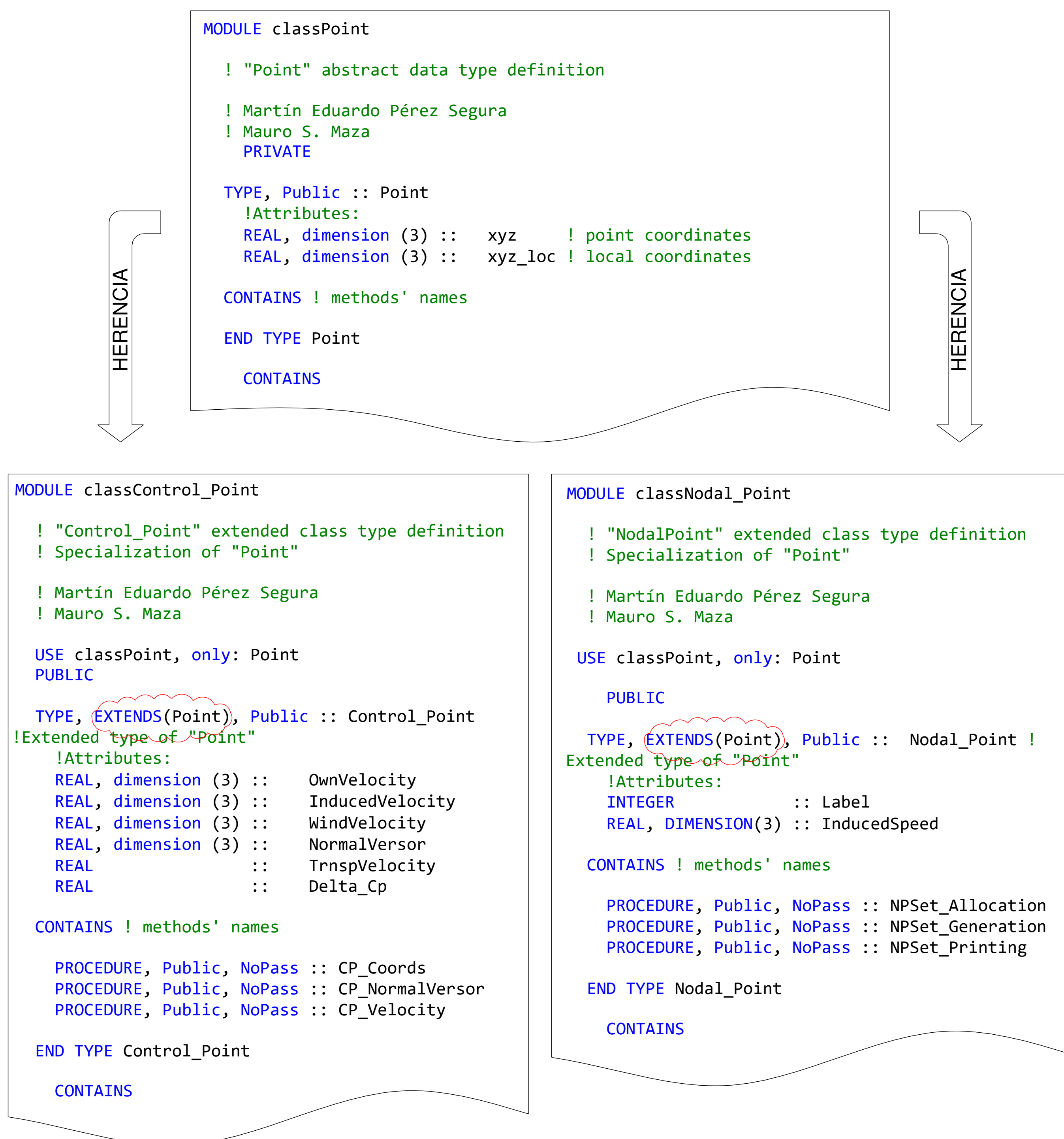

Figura 6: Ejemplo de relación “Is-a” (o herencia de clases).

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

```
MODULE classGrid

  ! GRID abstract data type definition

  ! Martín Eduardo Pérez Segura
  ! Mauro S. Maza

  USE classVortex_Segment
  USE classNodal_Point
  USE classPanel
  USE P_S_N
  USE Input_ReadLoad

   PUBLIC

  TYPE, Public :: Grid

  CHARACTER (len=15)                                :: GridName
  INTEGER                                           :: GridLabel ! Grid ID.
  CHARACTER (len=15)                                :: ExtraData  ! for miscellaneous purposes
  INTEGER                                           :: KinCondition ! Kinematic condition of the grid.
  INTEGER                                           :: nWakes ! Number of wakes shedding from the grid.
  INTEGER, ALLOCATABLE, DIMENSION(:)                :: WakesLabels ! Labels of the nWakes.
  INTEGER                                           :: nNodalPoints ! number of nodal points
  type(NODAL_POINT), ALLOCATABLE, DIMENSION(:)      :: NodalPointSet !  Nodal points of the grid
  INTEGER                                           :: nVortexSegments ! number of vortex segments
  type(VORTEX_SEGMENT), ALLOCATABLE, DIMENSION(:)   :: VortexSegmentSet ! Vortex segments of the grid
  INTEGER                                           :: nPanels ! number of panels
  type(PANEL), ALLOCATABLE, DIMENSION(:)            :: PanelSet  ! Panels of the grid
  INTEGER                                           :: AMatLoc ! AeroMatrix position index.

  CONTAINS ! methods' names

    PROCEDURE, Public, NoPass :: GridLoading
    PROCEDURE, Public, NoPass :: GridPrinting
    PROCEDURE, Public, NoPass :: InGrid_Xa
    PROCEDURE, Public, NoPass :: Panel_Coef
    PROCEDURE, Public, NoPass :: RingVrtct_2_SgmntCirc
    PROCEDURE, Public, NoPass :: PanelSet_RingVtct_Loading
    PROCEDURE, Public, NoPass :: GridReading

  END TYPE Grid

    CONTAINS
```

Figura 7: Ejemplo de la relación “Has-a” (o composición de clases).

La segunda forma, es bastante más general y se conoce como polimorfismo ilimitado. A pesar de que la denominación pueda resultar enérgica, también posee restricciones. Las entidades polimórficas ilimitadas satisfacen la necesidad de contar con un puntero que pueda referir no sólo a objetos de clases extendidas sino a objetos de cualquier tipo, tanto ADT como tipos intrínsecos. En este caso el atributo se expresa “`CLASS(*)`” y, aunque no fue utilizado en la implementación en análisis, posee elevado potencial y merece mención (Metcalf, M., Reid, J., Cohen, M., 2011).

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

```
SUBROUTINE Panel_Coef (GRD)
    ! USES POLYMORPHIC VARIABLES SO AS TO PROCESS GRIDS AND BC_GRIDS.

!Input Variable
CLASS(Grid),INTENT (INOUT)    :: GRD

!Local Variables
INTEGER      :: i,j,k    !Loop indexes
INTEGER      :: nShPnls, nPnlSgmnts  ! Number of sharing panels, number of panel segments.
INTEGER      :: PanelID ! Panel Label. Extraction variable
INTEGER      :: SGN     ! Sign of the label. Extraction variable

DO i=1,GRD%nVortexSegments
    nShPnls = GRD%VortexSegmentSet(i)%nSharingPanels

    DO j=1,nShPnls
        PanelID = GRD%VortexSegmentSet(i)%SharingPanels(j)
        nPnlSgmnts = GRD%PanelSet(PanelID)%nPanelSegments
        DO k=1,nPnlSgmnts
            if (abs(GRD%PanelSet(PanelID)%PanelSegments(k))==GRD%VortexSegmentSet(i)%Label) then
                SGN = sign(1,GRD%PanelSet(PanelID)%PanelSegments(k))
            end if
        END DO
        GRD%VortexSegmentSet(i)%SharingPanels(j) = SGN * GRD%VortexSegmentSet(i)%SharingPanels(j)
    END DO
END DO

END SUBROUTINE Panel_Coef
```

Figura 8: Polimorfismo de tipo compatible.

4.5 Programación genérica

Una última consideración sobre la OOP debe hacerse sobre la Programación Genérica. De acuerdo con M. Glasser (Glasser, M., 2009), la OOP proporciona facilidades para separar los aspectos de un software y tratar cada uno de manera independiente. En consecuencia, se pretende que cada aspecto sea representado una única vez para evitar multiplicidad de código. Sin embargo, también es deseable que cada representación pueda aplicarse a una gran variedad de situaciones. Estas intenciones combinadas desembocan en la necesidad de recurrir a la programación genérica que produce código general e intensamente parametrizado que puede ser utilizado, sin mayores modificaciones, en numerosos contextos. Intuitivamente se asocian los conceptos de *encapsulación*, *herencia* y *polimorfismo* a la *programación genérica*, de ahí otra ventaja importante de la OOP.

5 RESULTADOS Y VALIDACIÓN

Efectivizar completamente una implementación de simulación implica utilizarla para obtener resultados y, evidentemente, todo el desarrollo carece de sentido si dichos resultados no son correctos. En este sentido, verificar que las salidas del software sean adecuadas y contrastarlas contra otras versiones o métodos similares es fundamental.

Para testear el código desarrollado se recurre, en primer lugar, al análisis de una placa plana de gran alargamiento en arranque impulsivo. Este caso es fácilmente comparable con algunas otras versiones del UVLM e incluso con teorías de otra naturaleza. En particular, se analiza una placa plana con un alargamiento $\varLambda=20$, un ángulo de ataque $\alpha=10^{o}$, y para una corriente libre de velocidad $V_{\infty}=3\ m/s$. Para la cual se analiza la evolución del coeficiente de sustentación, C_L, en función del tiempo, hasta alcanzar aproximadamente un estado estacionario.

Para estas condiciones, se comparan los resultados con la teoría de perfiles delgados, corregida por alargamiento, para la cual se tiene:

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

$$C_L = \left(\frac{2\pi}{1 + \frac{2}{\Lambda}} \right) \alpha \tag{2}$$

Como parte del análisis también se incorporan los resultados una versión bidimensional del UVLM, donde se consideraron cuatro y seis paneles sobre la cuerda.

Todo esto, puesto en contraste contra tres casos simulados con el código en desarrollo: cuatro paneles cuadriláteros sobre la cuerda, ocho paneles cuadriláteros sobre la cuerda, y ocho paneles triangulares sobre la cuerda. Las curvas de resultados se muestran en la Figura 9 y, con mayor detalle, en la Figura 10, donde *P* indica la cantidad de paneles sobre la cuerda.

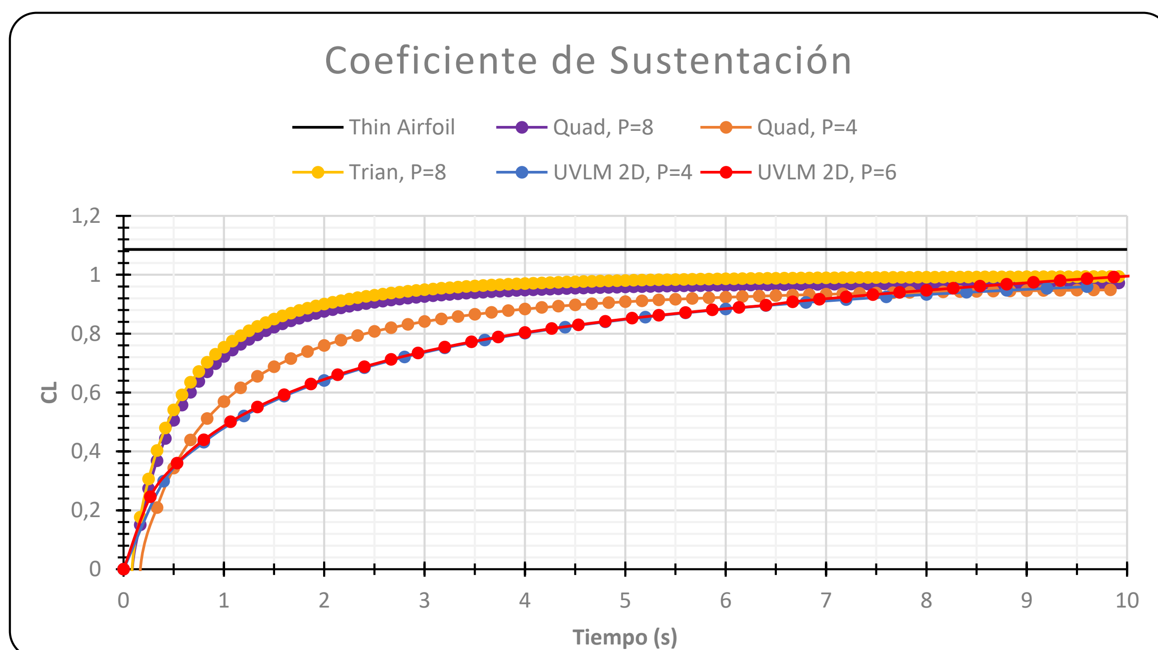

Figura 9: Coeficiente de sustentación para distintos modelos de placa plana en arranque impulsivo, curva completa.

Adicionalmente, se presentan capturas de las simulaciones para cada uno de los casos mostrados en los gráficos anteriores (Figura 11, Figura 12 y Figura 13).

Siguiendo con el análisis en cuanto a resultados y validación se considera el siguiente caso: una placa plana con un movimiento oscilatorio impuesto. Aquí, la placa cuenta con seis paneles en la cuerda y un alargamiento $\Lambda=10$. El movimiento oscilatorio se impone en dos coordenadas, un movimiento de cabeceo que varía el ángulo de ataque y otro que modifica la posición vertical de la placa. Además, se implementa el corte de estela a una distancia de seis cuerdas detrás del borde de fuga. Una secuencia de la simulación se muestra en la Figura 14.

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

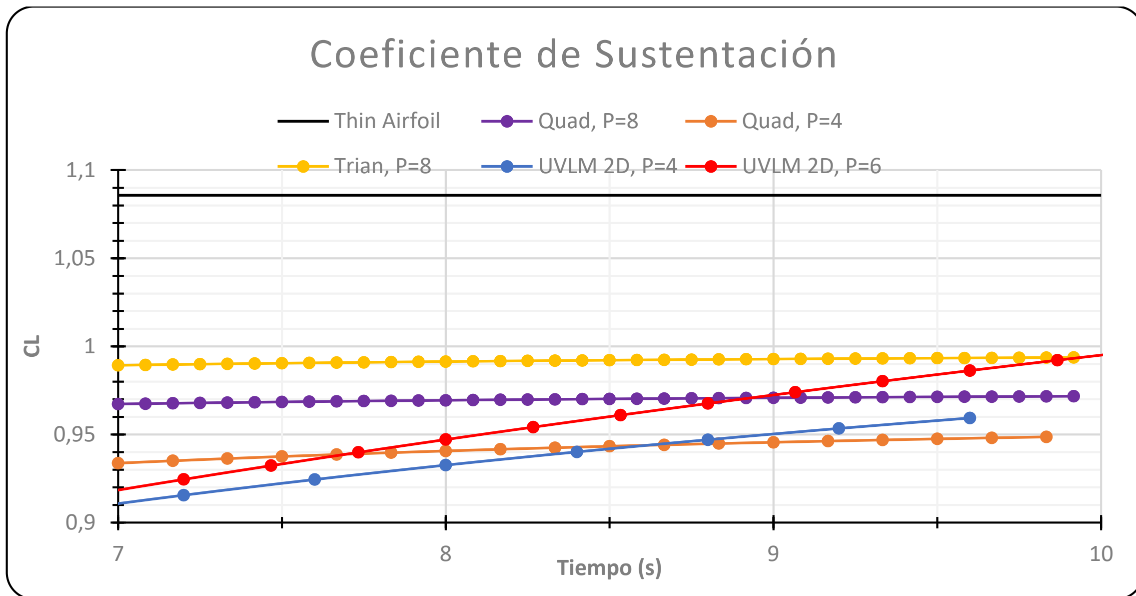

Figura 10: Coeficiente de sustentación para distintos modelos de placa plana en arranque impulsivo.

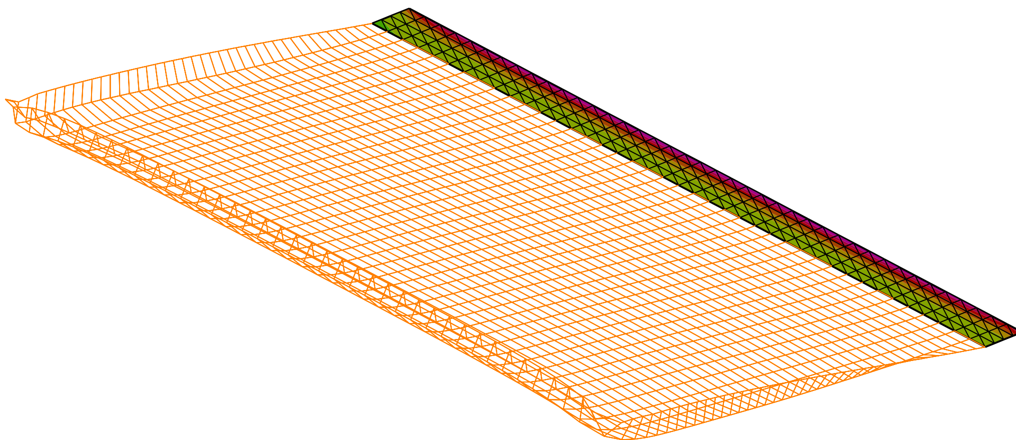

Figura 11: Simulación de placa plana con 8 paneles triangulares en la cuerda.

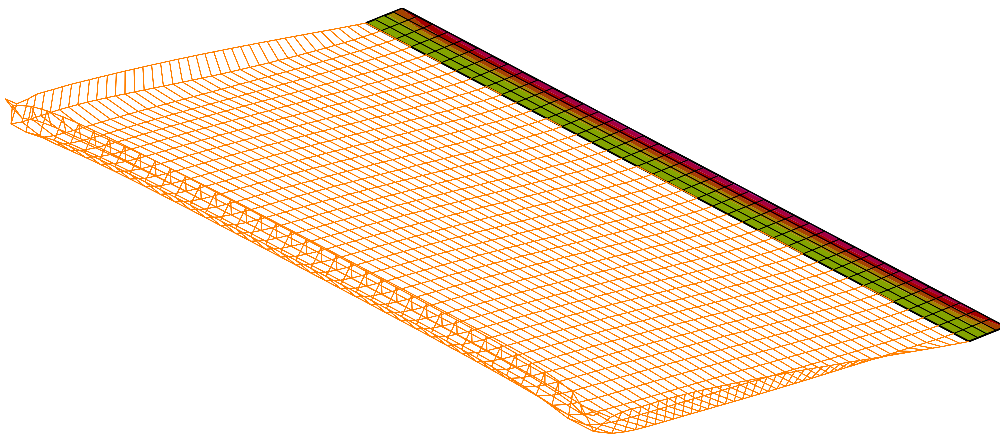

Figura 12: Simulación de placa plana con 4 paneles cuadriláteros en la cuerda.

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

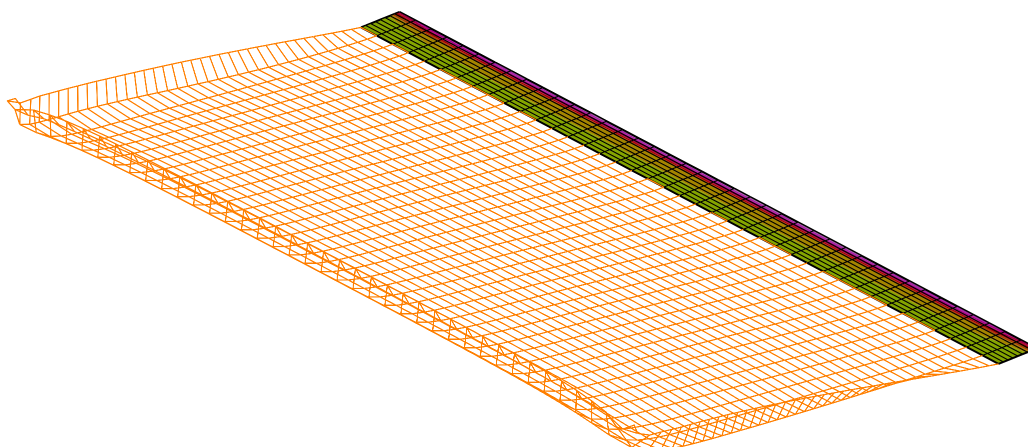

Figura 13: Simulación de placa plana con 8 paneles cuadriláteros en la cuerda.

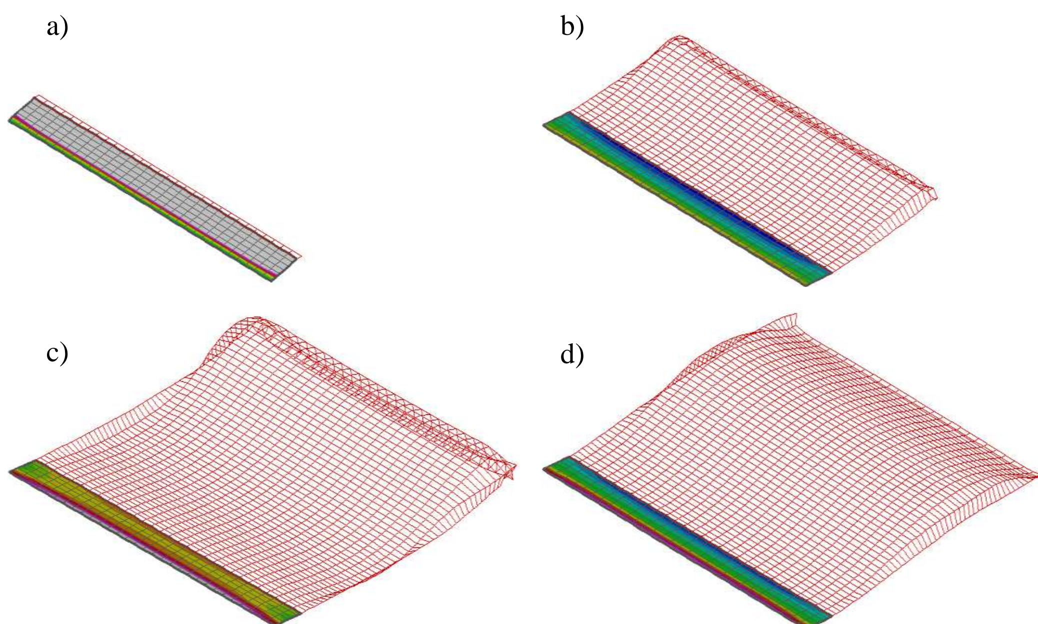

Figura 14: Arranque impulsivo de placa plana con movimiento oscilatorio impuesto: a) 1 paso de tiempo, b) 25 pasos de tiempo, c) 50 pasos de tiempo, d) 100 pasos de tiempo.

Estos resultados se comparan con el trabajo de Verstraete, M. L. (2016) en cuanto a la evolución del coeficiente de sustentación en función del tiempo y se representan ambos en la Figura 15.

El último caso a considerar aquí cuenta con un sentido práctico algo más notorio. Se analiza un modelo de "*Sensor-Craft*" concebido como una aeronave no tripulada (UAV) con configuración de alas unidas. En esta configuración se tienen cuatro superficies sustentadoras: dos alas delanteras y dos alas traseras. Además, el fuselaje y el empenaje vertical son superficies de contorno donde se exige la no penetración del flujo. Se simula un arranque impulsivo con una velocidad constante de corriente libre de módulo $V_\infty=3\ m/s$, ángulo de ataque $\alpha=10^o$, y ángulo de deslizamiento $\beta=20^o$. En la Figura 16 se muestra una imagen de la simulación para $t=45\ s$.

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

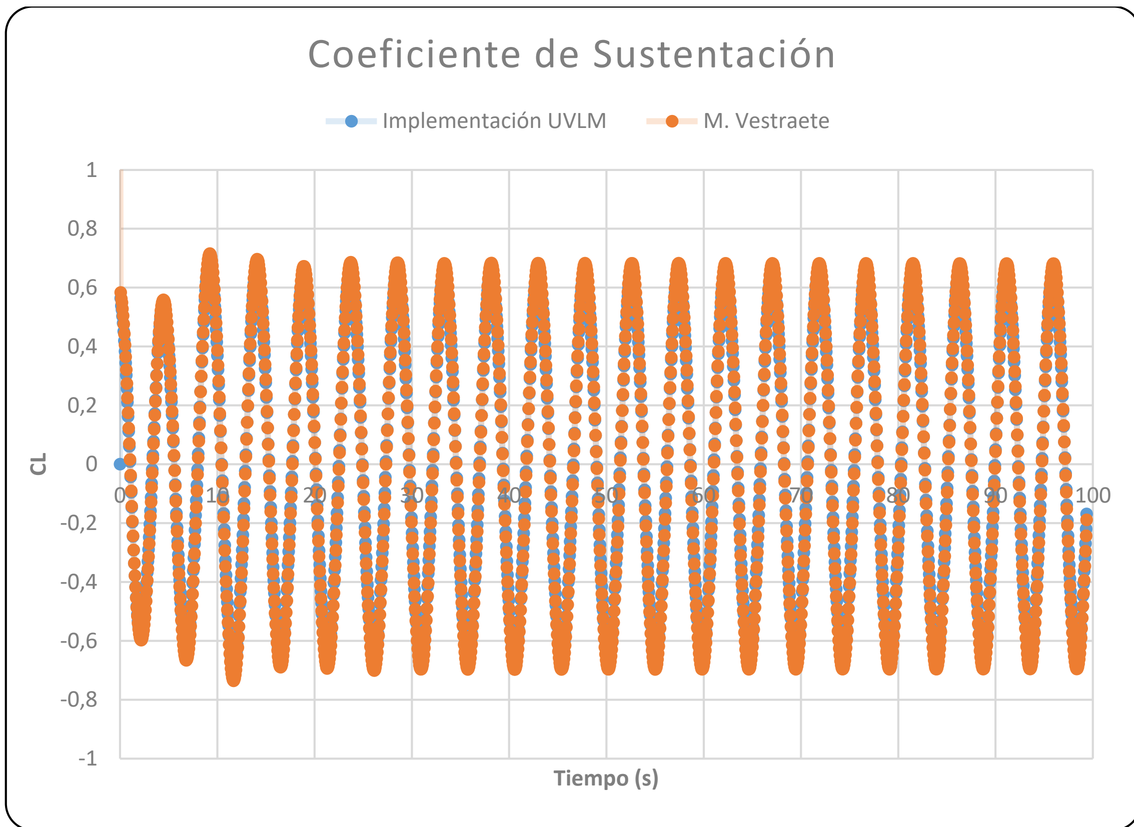

Figura 15: Coeficientes de sustentación para placa plana con movimiento oscilatorio impuesto.

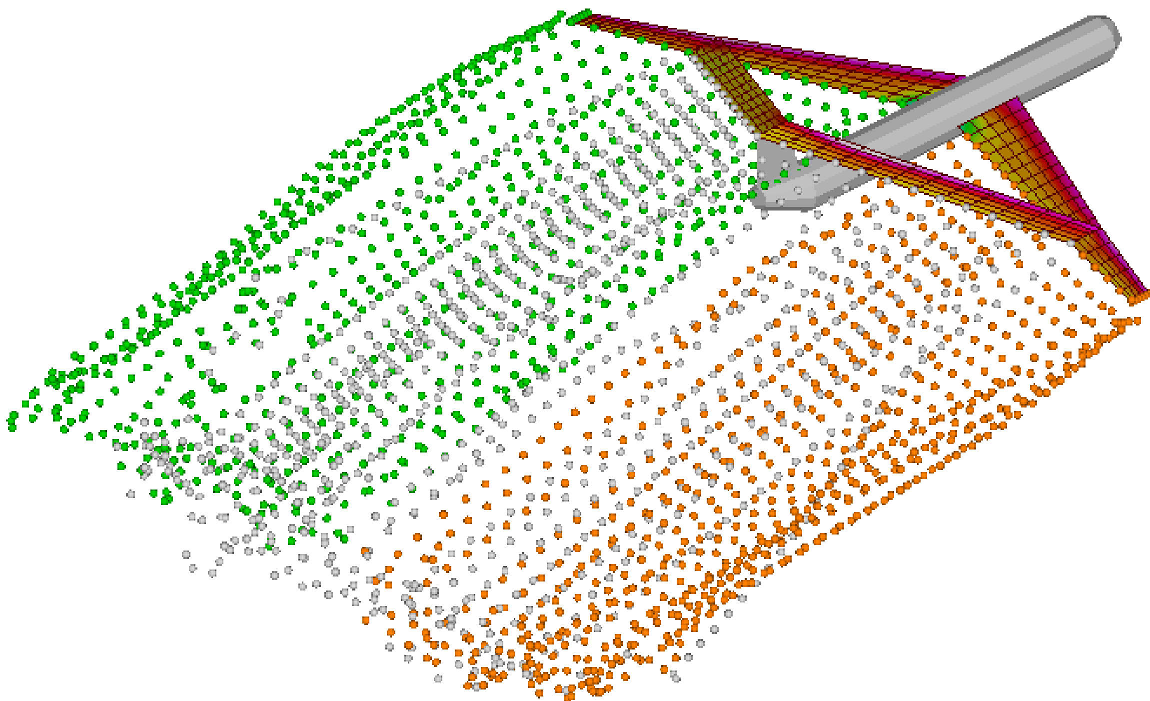

Figura 16: Simulación de UAV “Sensor-Craft”.

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

6 CONCLUSIONES Y TRABAJOS FUTUROS

A lo largo del presente esfuerzo se construyó una herramienta computacional de simulación fluido-dinámica enmarcada en el paradigma de la OOP, concebida para el uso eficiente en un grupo de investigación ingenieril, adaptada para el acoplamiento con otros códigos de simulación, y haciendo énfasis en la performance y la validación. En este proceso se exploró una alternativa a la clásica programación procedural, procurando vencer las deficiencias que este paradigma impone para el desarrollo de colaborativo y la reutilización de código en grupos de investigación.

El resultado general es una implementación de UVLM que permite evaluar las ventajas de la OOP. Éstas ventajas se hacen evidentes a la hora de incorporar nuevas características al código, modificar funcionalidades y agregar elementos de simulación. En particular, la inclusión de nuevos tipos de paneles, la adaptabilidad a distintos formatos de datos de entrada, el cambio de formato de datos de salida, la variabilidad en las propiedades de las superficies, la definición de puntos auxiliares para cálculo de velocidades inducidas, la versatilidad en la descripción de las líneas de convección, son algunas de las características que se vieron fuertemente favorecidas con el estilo de OOP. En definitiva, la generación de una estructura de datos base que permite sumar diversos desarrollos de acuerdo a los variantes requerimientos del grupo de investigación requiere un paradigma más adecuado y encuentra en la OOP soluciones a muchos de sus problemas.

En general, estos conceptos son extrapolables a cualquier fenómeno físico que pueda encasillarse en la co-simulación, bajo condiciones similares a las expuestas en este trabajo. Esto es, cualquier método con aplicabilidad probada puede someterse a un proceso de análisis y diseño que permita implementarlo con el paradigma de la OOP, aprovechando las ventajas de éste.

Todas estas consideraciones ponen en evidencia la vital importancia de la elección de un paradigma de programación adecuado. Este paradigma debe articularse apropiadamente con la fuerte relación que existe entre el enfoque utilizado para abordar la solución de un problema a través de la co-simulación y la dinámica de trabajo del grupo de investigación que la desarrolla. En este contexto, una etapa previa de análisis de alternativas y pre-diseño de soluciones representa una disminución de la carga de trabajo asociada a la adaptación, modificación, expansión o especificación del código y a las validaciones correspondientes.

Por otro lado, el presente trabajo representa un punto de partida tanto para desarrollos en el área de la FSI como en aquellas relacionadas al desarrollo de herramientas computacionales, en cuanto al análisis de paradigmas. De aquí se desprenden varias líneas de trabajo que pueden seguirse, entre ellas se destacan:

- La implementación de variantes al UVLM, como estelas de partículas, densificación localizada de grillas, convección de paneles triangulares, convección condicionada, etc.
- Evaluar otras estructuras de datos que coloquen otros elementos (por ejemplo, paneles) como núcleos de la definición de clases.
- Utilizar otros criterios para la definición de clases y comprobar sus ventajas frente al SRP.
- Indagar en los paradigmas orientados a aspectos y orientados a componentes para probar su adaptabilidad a las necesidades en consideración.
- Extender los análisis de validación y aplicabilidad para detectar posibles incompatibilidades o errores.
- Analizar el balance entre el nivel de parametrización del código y su performance.
- Considerar la utilización de otros lenguajes de programación.

En resumen, el resultado obtenido es alentador y cumplimenta los objetivos propuestos, los

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar

cuales se materializan en una implementación del UVLM diseñada y construida desde el punto de vista del desarrollo colaborativo de software de simulación.

REFERENCIAS

Aguilar, L. J., Programación Orientada a Objetos, McGraw Hill, España, 1996.

Booch, G., Object-Oriented Analysis and Design with Application, Third Edition, Addison-Wesley, 2007.

Fowler. M., Kendall, S., UML Gota a Gota, Pearson, Méjico, 1999.

Glasser, M., Open Verification Methodology Cookbook, Springer, USA, 2009.

Maza, M. S., Desarrollo de Herramientas Numéricas para la Simulación de la Interacción de Estructuras con un Fluido a Elevado Número de Reynolds, Tesis de MSC, Facultad de Ingeniería, Universidad Nacional de Río Cuarto, 2013.

Metcalf, M., Reid, J., Cohen, M., Modern Fortran Explained, Oxford, USA, 2011.

Preidikman S., Numerical Simulations of Interactions Among Aerodynamics, Structural Dynamics, and Control Systems, PhD thesis, Department of Engineering Science and Mechanics, Virginia Polytechnic Institute and State University, Blacksburg, VA, USA, 1998.

Verstraete, M. L., Simulaciones Numéricas del Comportamiento Aeroelástico de Vehículos Aéreos No Tripulados con Alas que Cambian de Forma, Tesis Doctoral, Facultad de Ingeniería, Universidad Nacional de Río Cuarto, Córdoba, Argentina, 2016.

Copyright © 2017 Asociación Argentina de Mecánica Computacional http://www.amcaonline.org.ar