Rev. Int. Mét. Num. Cálc. Dis. Ing.
Vol. 21, 2, 179-192 (2005)

Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería

Aplicaciones de la programación orientada a objetos y las colecciones al cálculo de estructuras

Miguel Fernández Ruiz y Marco Romera Corral

MC-2 Estudio de Ingeniería, S.L.
Víctor de la Serna 21, 28016 Madrid, España
Tel.: +34-91-519 74 77; Fax: +34-91-744 06 75
e-mail: miguel.fernandez@mc2.es, marco@mc2.es

Resumen

El desarrollo de los lenguajes de programación ha sufrido constantes transformaciones y mejoras aprovechando la experiencia acumulada en desarrollos anteriores así como las reflexiones que sobre el tema han ido surgiendo. En este marco se ha evolucionado desde lenguajes donde podía programarse sin ninguna estructura definida, a lenguajes estructurados y finalmente a lenguajes orientados a objetos, siendo cada uno de ellos una evolución natural de sus antecesores, superando sus deficiencias y aportando capacidades nuevas. Respecto al cálculo de estructuras, y a pesar del dominio de los lenguajes orientados a objetos en la mayoría de los campos de la programación, no se explotan todavía todas las posibilidades ofrecidas por las nuevas técnicas de programación, lo que se traduce en códigos poco flexibles en ciertos aspectos (principalmente en lo que a manejo de información se refiere) y con una cierta dificultad para ser mantenidos y ampliados. En este artículo se revisa la evolución de los lenguajes de programación, se apuntan y comentan las mejoras introducidas en este campo recientemente y se detallan posibles aplicaciones al cálculo de estructuras. Para ello se expone una base teórica de la programación orientada a objetos y se desarrolla su aplicación en un programa de cálculo estructural diseñado por los autores y basado en las ideas anteriores.

Palabras clave: *cálculo de estructuras, programación orientada a objetos, colecciones de objetos.*

APPLICATIONS OF THE OBJECT ORIENTED PROGRAMMING AND THE COLLECTIONS TO THE STRUCTURAL CALCULUS

Summary

The programming languages have suffered continuous changes and improvements based on the previous experiences as well as the theoretical ideas that have been developed. This way, the languages have evolved from a non–structured architecture to a structured one and finally to a full object oriented scheme, being each one a natural evolution of the formers, overcoming some of their deficiencies and incorporating new features. Concerning the structural calculus, and despite the object oriented languages massive use in the majority of the programming fields, the new possibilities offered by the current languages are not yet exploited. This situation is translated into less flexible codes concerning certain topics (mainly those of the information management) and with some difficulties to be updated. In this paper, the evolution of the programming languages is reviewed, highlighting the recent improvements in this field and pointing out their applications to the structural calculus. In order to do it, some theoretical aspects are commented and an application to the structural calculus, designed by the authors and based on the previous ideas, is explained.

Keywords: *structural calculus, object oriented programming, object collections.*

©Universitat Politècnica de Catalunya (España). ISSN: 0213–1315 Recibido: Julio 2004 Aceptado: Noviembre 2004

INTRODUCCIÓN Y MOTIVACIÓN DEL ESTUDIO

El cálculo de estructuras es una disciplina que se ha encontrado marcada desde la aparición de las primeras posibilidades del cálculo mecanizado por una sistematización y automatismo de las tareas, empleando para ello los recursos ofrecidos por las máquinas. Puede fijarse entorno a la década de 1970 cuando comenzó a disponerse con cierta normalidad de medios de cálculo suficientemente potentes, lo que se tradujo en la aparición –y evolución hasta hoy día– de numerosas formas de entender y resolver los diferentes problemas relativos al cálculo estructural.

En cuanto a los lenguajes de programación, vínculo entre el programador y la máquina, se desarrollaron desde temprano diferentes códigos, tanto ligados a determinadas máquinas como independientes de las mismas y, por lo tanto, permitiendo la reutilización de los algoritmos desarrollados, siendo este segundo tipo de lenguajes los que finalmente terminaron imponiéndose. Algunos ejemplos de lenguajes ligados a máquinas podrían ser, por ejemplo los desarrollados por HP, siendo los primeros lenguajes portables FORTRAN, BASIC y COBOL[14].

Todos estos lenguajes en un principio respondían a una estructura de programación denominada **de etiquetas** donde el código desarrollado iba saltando de línea en línea mediante la llamada a *etiquetas* o *marcadores*. Esta filosofía de programación permitía el desarrollo de algoritmos realmente muy compactos y eficientes desde el punto de vista computacional. Sin embargo, la desestructuración del código impedía un fácil entendimiento, corrección y posterior ampliación del mismo. Esta manera de programar, entrelazada y con numerosas secuencias de escape, fue bautizada como *programación espagueti*.

A la luz de estas deficiencias fueron desarrollándose nuevos lenguajes desde cero o bien evolucionando algunos de los antiguos hasta dar lugar a códigos de **programación estructurada**. Este tipo de programación se basa en la separación en bloques de código con cierta autonomía entre sí denominados funciones o subrutinas. Entre los principales exponentes de estos lenguajes cabe destacar el lenguaje C (desarrollado en 1969 en los laboratorios Bell por D. Ritchie[5]) y el FORTRAN77 (creado en 1977 a partir de una versión anterior de la década de 1940).

A pesar de las mejoras introducidas por esta filosofía de programación, una estructura de bloques de código se vuelve compleja de mantener y no dispone de la suficiente flexibilidad en su ampliación cuando los programas rebasan un cierto tamaño. (Según Schildt[13], el tamaño máximo podría fijarse entre las 25000 y las 100000 líneas de código.) Uno de los primeros lenguajes desarrollados para paliar estas deficiencias surgió como una variante del C, denominado en un principio “C con clases”, desarrollado por B. Stroustrup en 1979 nuevamente en el marco de los laboratorios Bell y que posteriormente pasaría a denominarse, haciendo referencia al operador incremental de dicho lenguaje, C++ en 1983. La irrupción del C++, y de la **programación orientada a objetos** que representaba, creó una nueva forma y filosofía de diseño de programas.

Como evolución natural de los lenguajes estructurados, en la programación orientada a objetos, el código y las diferentes variables se encuentran compartimentados en *objetos*, relacionándose entre sí mediante los denominados *métodos*. Diferentes lenguajes le siguieron al C++ en su filosofía incluyendo cada uno sus particularidades y desarrollándose algunos de los mismos desde cero como, por ejemplo, Eiffel[8] o Java en 1995. En general, la programación orientada a objetos permite unas mayores posibilidades de reutilización de código existente y aplica la filosofía de “dividir para vencer” a la hora de abordar un problema. Esta arquitectura permite además que diferentes equipos de programadores sean capaces de colaborar de una manera sencilla simultáneamente en un mismo proyecto, una vez que les haya sido definido su parcela de trabajo (*objeto* o agrupación de ellos en *paquetes* o *componentes*) y su *interfaz* (conjunto de métodos definidos sobre un objeto mediante

los cuales interactúa con los demás objetos del programa) con el resto de los objetos del programa. Actualmente, los lenguajes de programación dominantes se encuentran basados en esta filosofía y el resto de lenguajes se encuentra en mayor o menor grado evolucionando hacia este tipo de planteamientos. Baste como ejemplo el cambio sufrido por VisualBasic entre su versión 6.0 (basada en objetos) y su siguiente versión denominada .NET, donde se presenta un lenguaje completamente orientado a objetos[16] o el cambio de Pascal a Delphi.

En cuanto al cálculo de estructuras, y a pesar de la tendencia general seguida en los lenguajes de programación, existe un cierto decalaje temporal en la adopción de las nuevas posibilidades ofrecidas por los lenguajes de programación. Posiblemente, el lenguaje que ha dominado con una mayor intensidad en este campo ha sido el FORTRAN, debido fundamentalmente a su fuerte implantación cuando comenzaron a escribirse los grandes programas que actualmente dominan en el análisis de estructuras[6], en concreto el FORTRAN77, aunque ya existen desarrollos interesantes en FORTRAN90 orientado al trabajo con matrices. Entre algunos de los desarrollos en este lenguaje cabría destacar, por ejemplo, los realizados de Pastor y Mira[7]. Algunos ejemplos de programas de cálculo de estructuras escritos íntegramente en este lenguaje pueden ser, por ejemplo, los desarrollados en el CIMNE[10] o FEAP[15] siendo además su conocimiento necesario para programar subrutinas de usuario en ABAQUS o en el APDL de ANSYS. También existen, aunque con menor intensidad, programas de este estilo escritos en C (como, por ejemplo, el código IRIS[12], en desarrollo aún).

No obstante, a pesar de las aparentes ventajas de la programación orientada a objetos[17,2,3], existe un número muy inferior de desarrollos importantes en este campo. Tal vez, una posible razón del éxito de la programación estructurada reside en que su intensivo trabajo con matrices permite una rápida relación entre los algoritmos y datos almacenados. Es decir, almacenamiento y cálculo se basan en el empleo de la misma entidad (la matriz o tabla), lo que se traduce en algoritmos compactos basados en estructuras de bucles indexados. Como posteriormente se expondrá, en la programación orientada a objetos usualmente se trabaja con colecciones de objetos y bucles con *iteradores* no necesariamente indexados. Esta estructura permite abordar el problema de una manera más general, siendo además compatible con volcar en un determinado momento la información en matrices para trabajar con los datos de manera más compacta o eficiente. No obstante, el esquema estructurado presenta una serie de inconvenientes:

- La recuperación, usando numeración no secuencial, de un valor en un elemento dado del modelo de cálculo obliga al empleo de vectores de conectividad que ensucian el código y son fuente continua de errores.
- La correspondencia entre los índices de los vectores o matrices y sus propiedades físicas debe establecerse llevando el programador un control externo de la misma, lo que hace en ocasiones difícil seguir e interpretar el código y su significado físico. En general, puede decirse que la programación estructurada resulta en un código sencillo de escribir pero difícil de entender.
- Borrar un elemento que pertenezca a un modelo de cálculo o borrar completamente un modelo de cálculo o sus resultados se vuelve una tarea complicada –incluso peligrosa, ya que suele requerir preservar el resto de valores– al no disponerse de una gestión automática de la memoria. Este problema puede solventarse estableciendo valores nulos, pero esto nuevamente ensucia el código (debiendo introducirse además secuencias condicionales) y es también una posible fuente de errores.
- Añadir y reemplazar es una operación sencilla cuando se trata de valores aislados del sistema. En cambio, añadir o reemplazar entidades que agrupan otras (modelos

completos de cálculo o resultados), especialmente cuando puede haber cambiado por ejemplo el número de nodos o elementos del modelo resulta más compleja y nuevamente problemática.

- Otro problema intrínseco a la programación estructurada es que el acceso por parte de las diferentes subrutinas a las variables públicas (accesibles por diferentes partes del programa) permite que en ocasiones se realicen modificaciones no deseadas de las mismas.

- Lectura de datos. El almacenamiento en matrices de los datos exige definir la dimensión de dichas matrices antes de ser llenadas, lo que plantea una serie de alternativas para resolver el problema:

  - Indicar con un número antes de la lectura de un determinado grupo de datos cuántos valores van a leerse.
  - Realizar una doble lectura del fichero, almacenando en la primera pasada el número de datos de cada tipo y llenando las matrices en una segunda lectura.
  - Redimensionar en cada nueva lectura la matriz de datos modificada preservando los valores ya existentes. Esta opción, dependiendo del lenguaje, puede estar limitada o no disponible.
  - Emplear unas matrices auxiliares en la lectura de tamaño muy grande en comparación con el número de datos que se espera leer y volcar posteriormente su contenido y dimensiones en las matrices de almacenamiento.

  Aunque todos estos procedimientos son posibles, en realidad ninguno es plenamente satisfactorio y cada planteamiento tiene sus dificultades, lo que debe estudiarse a la hora de programar la lectura por fichero o interactiva.

Todos estos inconvenientes, derivados de la arquitectura de programación estructurada, son totalmente obviados por la programación orientada a objetos al funcionar bajo un principio completamente diferente como a continuación se presenta.

PROGRAMACIÓN ORIENTADA A OBJETOS

En esta sección se van a dar unas breves notas sobre la programación orientada a objetos y las colecciones de objetos, de forma que puedan entenderse los desarrollos y comparaciones presentados con otras arquitecturas de programación.

Debido a la complejidad del asunto, se hará un recorrido de lo particular a lo general, comenzando por los objetos y sus propiedades para posteriormente comentar las diferentes maneras de trabajar con colecciones de objetos. Finalmente se abordará el tema de la estructura modular de un programa como paso final en un proceso de programación orientada a objetos. En realidad, este recorrido se realiza para mejorar la claridad de la exposición, ya que es justamente al contrario de lo que debe plantearse a la hora de diseñar un programa, debiendo comenzarse por establecer las diferentes partes del mismo e ir detallando posteriormente sus componentes.

Estas nociones se particularizarán al final de cada apartado destacando sus posibles aportaciones en el cálculo de estructuras y sus ventajas respecto a un enfoque de programación estructurado.

Objetos

Un objeto es un componente de un programa que contiene una serie de variables así como diferentes métodos (funciones) para operar sobre las mismas y relacionarse con los demás objetos del programa. De esta forma, los objetos actúan como compartimentos estancos, como pequeños programas independientes que se encargan de realizar diferentes tareas cuando se les solicita. Por lo tanto, las variables que interesan para resolver un problema sólo son accesibles para las rutinas con las que están relacionadas. Esta propiedad se denomin *encapsulamiento* y limita el alcance o visión de las variables. En este sentido, es fundamental la correcta definición de los objetos y sus interfaces cuando se desea abordar un problema, estableciendo cuáles son las partes del mismo y cómo se relacionan entre sí.

Tal y como ya se ha comentado, la manera de relacionarse los diferentes objetos y de operar sobre sus propias variables es mediante los métodos que sobre el mismo se hayan definido. Dichos métodos (al menos los *públicos*, accesibles por otros objetos del programa) permiten definir la relación del objeto con el resto de los objetos del programa. Además, completando esta filosofía, existe una serie de variables (*públicas*) que en general son accesibles por los demás objetos del programa y otras internas del objeto (*privadas*) sobre las que no pueden actuar más que a través de los propios métodos del objeto. Es una buena práctica evitar el uso de variables públicas y obligar a que toda interacción con el objeto sea siempre a través de sus métodos. De esta forma se impide el cambio de las propiedades o variables que afectan a su estado sin conocimiento o demanda del propio objeto.

Estas ideas, con especial énfasis en el concepto de ámbito de las variables, se presentan en la Figura 1 donde pueden apreciarse las diferencias con un esquema de programación estructurada[1].

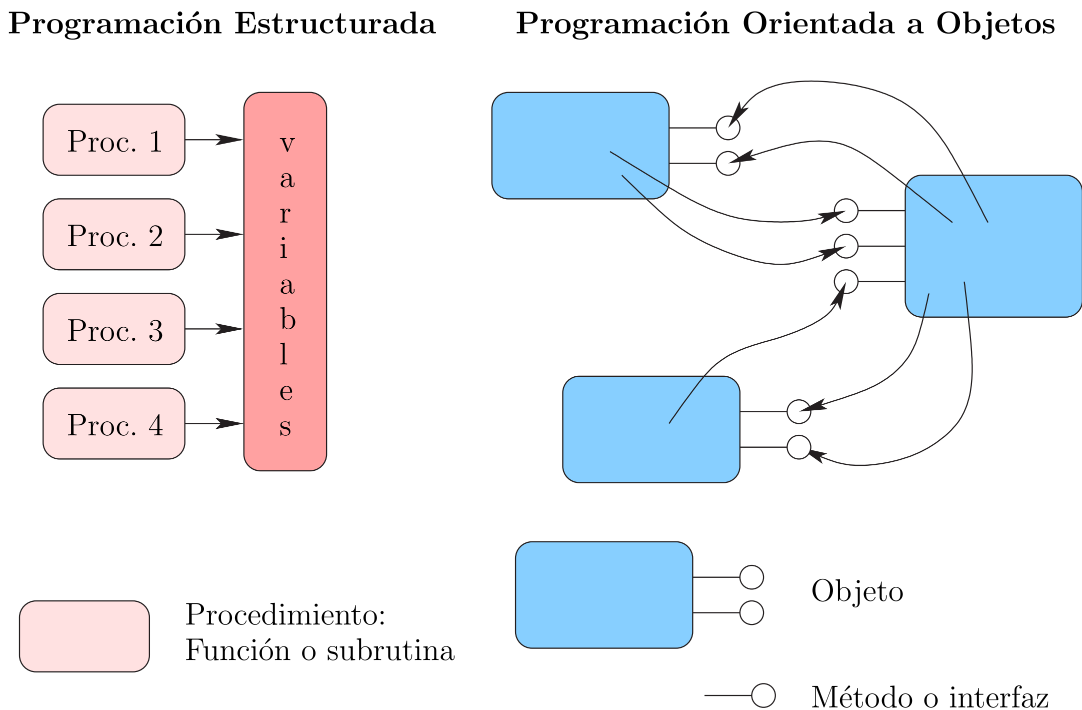

Figura 1. Comparación entre la programación estructurada y orientada a objetos

En cuanto al cálculo de estructuras, las ventajas del empleo de objetos para el tratamien-

to de la información respecto de un esquema estructurado son múltiples, entre las que cabría destacar:

- Los objetos definidos tienen un significado físico claro (nodo, elemento, carga, etc.) al igual que sus métodos (`nodo.x()` $\leftrightarrow$ coordenada x del nodo) escribiéndose un código mucho más sencillo de comprender y depurar y evitándose recurrir a tener una hoja con la correspondencia de las variables física con las matrices donde se almacenan los datos

- Ampliar las propiedades de los objetos (nuevas dimensiones, nuevas características) es inmediato y no ensucia el código existente ni obliga a realizar retoques en lo ya escrito.

- Al poderse definir métodos en los propios objetos, estos pueden efectuar ciertas tareas de manera automática en su construcción y avisar de posibles errores en su definición (cálculo de la longitud del elemento, conectividades, cargas, etc.).

- Todo el código y algoritmos aplicados al cálculo de estructuras que han sido desarrollados con anterioridad en programación estructurada pueden ser empleados sin tener que realizar adaptaciones mayores. El motivo se debe a que en la resolución de los modelos numéricos (especialmente una vez hecha la renumeración de nudos) la información puede volcarse en matrices, de hecho es recomendable, para que su manejo sea más eficiente. De esta forma la reutilización de código y algoritmos ya existentes es total no debiendo replantearse esa parte de los programas.

- El encapsulamiento de las variables dentro de los objetos y su acceso mediante métodos (que permiten establecer de esta manera un determinado control) evita el problema ya citado de la programación estructurada en el que pueden realizarse modificaciones no deseadas de algunas variables accesibles desde diferentes partes del programa.

Colecciones de objetos

Las colecciones son agrupaciones de objetos dentro de otro objeto. El desarrollo de las colecciones ha posibilitado el trabajo con grandes volúmenes de información de manera ordenada y abstracta. Uno de los más potentes exponentes de las colecciones de objetos se encuentra desarrollado en C++ en las Biblioteca de Plantillas Standard (STL)[4] y su definición y uso se han extendido prácticamente a todos los lenguajes orientados a objetos con mayor o menor intensidad.

Existen diferentes estructuras de agrupación de datos, por ejemplo en Java se dispone de *colecciones*, *listas*, *conjuntos* y *mapas*, teniendo cada una sus particularidades en cuanto a almacenamiento y recuperación de información, pero en general todas responden a la misma filosofía: agrupar objetos y poder recuperarlos, bien por uso de un *iterador* que recorre todos los elementos de la colección bien por la demanda de un objeto con una etiqueta determinada.

La potencia y flexibilidad de las colecciones se muestra muy superior al empleo de matrices de objetos que, a pesar de ser otra opción en el manejo de datos, no es tan general ni dispone de una batería tan amplia de posibilidades como las colecciones. Debe no obstante encontrarse un compromiso entre la claridad del código (uso de colecciones y objetos) y eficiencia (uso de matrices), ya que el óptimo no se encuentra en ninguno de los dos extremos.

Se reproduce en la Figura 2 un esquema de lo que podría ser una colección de objetos donde, además, la propia colección tiene una serie de métodos para recuperar o eliminar objetos e incorporar otros nuevos.

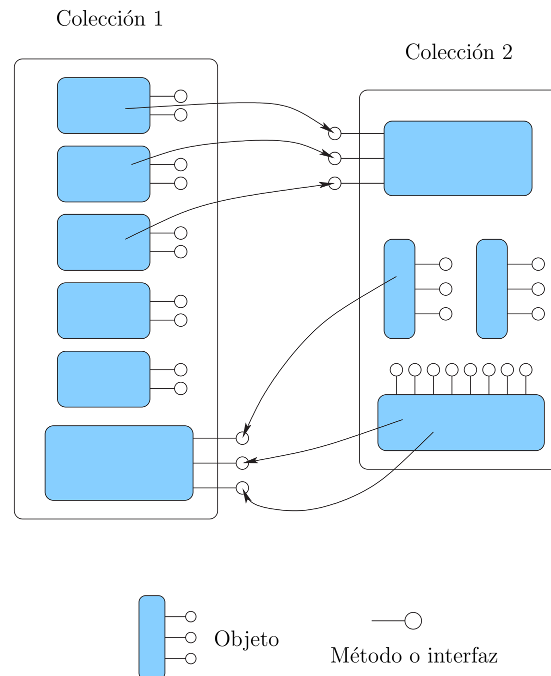

Figura 2. Colecciones de objetos formadas por objetos del mismo tipo (izquierda) y de diferentes tipos (derecha)

Es interesante destacar que las colecciones de objetos se emplean para el manejo de objetos, y por lo tanto, deben encontrarse contenidas dentro de otros objetos, siendo de esta forma una *agrupación por relación* entre objetos. La agrupación de clases (códigos que definen las variables y métodos que posteriormente tendrá un objeto creado a partir de la misma), concerniendo sus tareas se realiza en los denominados *módulos* o *paquetes*, que serán tratados en el apartado siguiente.

En cuanto a las ventajas que puede aportar el empleo de colecciones en un programa orientado a objetos en general y en uno de cálculo estructural en particular, cabría destacar las siguientes:

- Los datos pueden introducirse siguiendo cualquier orden y además intercalando grupos de datos (nodos, elementos, etc.) en la definición del modelo, lo que resulta especialmente valioso en la definición de diferentes modelos, aprovechando información ya introducida. Se evita así el redimensionado de matrices (operación no siempre satisfactoria o posible) y las dobles lecturas, introducción de dimensiones o empleo de matrices ficticias de gran tamaño en las lecturas por fichero.

- Eliminar un objeto de una colección es un procedimiento inmediato, siendo algo complejo (incluso peligroso) cuando los datos se encuentran almacenados en matrices.

- Se evita el empleo de matrices de conectividad, siempre problemáticas y que ensucian el código, pudiendo adoptarse de una manera libre la numeración de entrada debido a la recuperación mediante mapas que asocian un objeto a una clave. En definitiva, se deja que sea el propio lenguaje el que internamente considere las conectividades.

- Pueden efectuarse barridos de las colecciones de objetos por medio de *iteradores* o bien se puede tener acceso directo a datos puntuales habiendo definido previamente sus claves. Este aspecto permite plantear los algoritmos del programa de una manera muy general, no necesariamente indexada, actuando sobre entidades abstractas y realizando bucles sin importar *a priori* las dimensiones o propiedades de los objetos de las colecciones, definiendo operaciones genéricas (abstractas) sobre los mismos.

Programación modular

En ocasiones, ciertas tareas requieren la utilización de un conjunto de objetos para su resolución. Se pueden entonces agrupar los objetos formando paquetes que puedan además definir objetos con ámbito restringido a dicho paquete. Esta posibilidad permite también una libre elección de los nombres de las clases, sin miedo a que coincidan con los de otros paquetes, ya que cada uno queda debidamente identificado por la referencia a su paquete.

El conjunto de clases de un paquete forma una entidad que permite su reutilización desde otras partes del programa o incluso desde otros programas. Si se pretende que la reutilización se lleve a cabo desde otros programas escritos en lenguajes diferentes y a través de redes informáticas o entre programas que se ejecutan en procesos u ordenadores distintos, se puede recurrir a la creación de módulos.

Un módulo puede estar programado internamente utilizando una o varias clases de objetos, pero define una interfaz muy precisa a la hora de comunicarse con otras partes del programa u otros programas. Esta interfaz suele ser binaria, respetando todos los módulos un protocolo común para poder comunicarse. Existen diferentes protocolos o plataformas desarrolladas que incluyen toda la infraestructura necesaria para los lenguajes más extendidos, como, por ejemplo, CORBA (multiplataforma) o COM (exclusiva de MicroSoft).

La división de una tarea en sus diferentes partes lógicas, definiendo los diferentes paquetes y módulos y las interfaces de comunicación entre ellos es definitivamente la tarea más importante a la hora de abordar la programación de un problema. La solución no es única ni tampoco tiene por qué existir un óptimo, habrá maneras de orientar la programación más eficientes, otras más flexibles, otras más sencillas, etc. Es a esta parte del trabajo de planificación a la que deben dedicarse por lo tanto los mayores esfuerzos a la hora de diseñar un programa relativamente ambicioso.

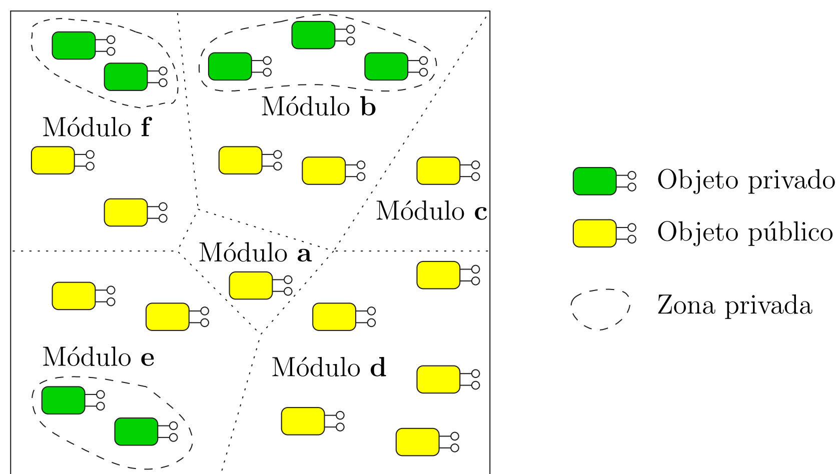

Figura 3. División de un programa en sus diferentes módulos e interfaces de comunicación

De esta forma puede trabajarse en el desarrollo de un programa mediante un equipo compuesto por diferentes personas una vez se hayan definido los módulos o paquetes a programar, sus objetos y los métodos de estos que serán accesibles por otras partes del programa. En general, esta filosofía de diseño entronca con la reutilización de código y el empleo de objetos desarrollados por otros equipos. No es necesario conocer la implementación interna de cada objeto ni módulo sino únicamente su interfaz con el resto del programa, lo que permite reemplazar además fácilmente objetos o módulos completos por otros más eficientes siempre que se mantenga su interfaz con el resto del programa. Se muestra a modo de esquema en la Figura 3 la división de un programa en módulos, cada uno de los cuales contiene sus respectivos objetos así como las posibles interfaces entre ellos.

Las ventajas de la programación modular son también fácilmente identificables en el cálculo de estructuras, ya que permite:

1. Reutilizar código ya existente. Incluso aunque haya sido escrito en otro lenguaje y se encuentre compilado a código máquina, puede ser empleado una vez definida su interfaz con el resto de objetos. Este aspecto es muy interesante en programas de cálculo estructural, dado que permite aprovechar todo el trabajo ya invertido en otros lenguajes.

2. Permite seccionar el trabajo y separar las diferentes partes de un programa. A este respecto, en un programa de cálculo de estructuras es interesante separar la interfaz del usuario (para preproceso y posproceso) del almacenamiento de modelos (de cálculo y resultados) y algoritmos de resolución.

3. Reemplazar código es inmediato siempre que se mantengan las interfaces de los diferentes paquetes o módulos. Esta propiedad permite que si en un futuro deba alterarse una parte del código por motivos de errores o para incorporar nuevas capacidades, pueda ensamblarse sin tener por ello que modificar lo ya existente en otras áreas del programa.

4. Ampliar código. Con esta filosofía es sencillo añadir nuevos módulos con otras capacidades de preproceso, posproceso o cálculo a los existentes sin tener que realizar modificaciones en lo ya escrito, siendo por lo tanto los programas de este estilo muy flexibles y fácilmente adaptables a realizar otras tareas adicionales.

EJEMPLO DE APLICACIÓN: JSTRUCT

A continuación se presenta una aplicación de las ideas anteriores en un programa diseñado por los autores para el análisis de estructuras reticuladas bidimensionales. (Los códigos fuentes del programa, así como la documentación de sus clases y el manual de usuario, pueden descargarse gratuitamente bajo licencia general pública GNU (pudiendo colaborarse además en las líneas en desarrollo que actualmente se encuentran abiertas) desde:
`http://www.cercis.net/jstruct/jstruct.html`.) En el desarrollo de esta herramienta se ha empleado el lenguaje de programación Java. La decisión ha sido adoptada por diferentes motivos: ser un lenguaje completamente orientado a objetos, independiente del sistema operativo y que permite una potente edición de interfaces gráficas (punto clave en el pre y posproceso de datos y resultados).

En contra del empleo de Java puede argumentarse que no es un lenguaje tan rápido como otros, también orientados a objetos, pero cuyo resultado se encuentra compilado a código nativo (como C++, por ejemplo). Para que su portabilidad sea posible, en vez de código máquina, Java genera *bytecode* que es interpretado por la máquina virtual de Java,

siendo esta a su vez la encargada de relacionarse con el sistema operativo en cuestión. No obstante, existen compiladores *JIT (Just In Time)* que compilan el *bytecode* a código máquina antes de ejecutarlo, siendo, por lo tanto, particularmente eficaz en la ejecución de bucles. Sin embargo, la adopción de algoritmos optimizados y la renumeración de nudos en el código escrito permiten que el tiempo de cálculo sea muy reducido. Además, una vez cargada una clase, la próxima vez que la misma sea llamada, el tiempo que tarda en ser ejecutada es muy inferior al estar ya en memoria.

También puede entenderse como una desventaja de `Java` respecto de otros lenguajes orientados a objetos la no inclusión de la programación genérica o con plantillas (muy usadas, por ejemplo, en C++). La razón de ello es que se ha preferido renunciar en `Java` a su flexibilidad para obtener una mayor claridad del código (uno de los objetivos de la programación orientada a objetos). No obstante, `Java` ha reservado una palabra clave (*generic*) para poder implementarlas en un futuro.

En cuanto a la estructura del programa, se ha optado por dividirlo en los siguientes paquetes:

1. `input` – Paquete que engloba las clases relativas al modelo de cálculo. Este paquete tiene una clase denominada `ModelCalc`, la cual almacena las colecciones de objetos que definen dicho modelo y trabaja con ellas. Los objetos de las colecciones se obtienen como instancias del resto de clases del paquete, definiendo nodos (`NodeCalc`); elementos (`ElemCalc`); cargas sobre nodos (`HnodCalc`) o elementos (`HeleCalc`); condiciones de borde tipo apoyo (`BounCalc`) o bien resorte elástico (`LsprCalc`) así como una clase con información del modelo (`InfoCalc`) con las unidades y título. Los objetos pertenecientes a este paquete se construyen como inmutables, es decir, una vez creados no pueden modificarse, lo que elimina posibilidades de error por alteraciones de los datos en el paquete de cálculo.

2. `output` – Paquete que engloba las clases relativas al modelo de resultados. Nuevamente existe una clase principal denominada `ModRes` que trabaja con colecciones de objetos donde se encuentran almacenados los resultados de movimientos en nodos (`NodeRes`) y esfuerzos en elementos (`ElemRes`). Finalmente el paquete se compone de dos clases auxiliares (`NodeData` y `ElemData`) que facilitan y sistematizan la tarea de almacenamiento. En este paquete se definen las clases también como inmutables para impedir modificaciones accidentales de los mismos en el posproceso de resultados. En definitiva, se ha adoptado una estrategia de *programación defensiva* para proteger los modelos de datos y resultados de manipulaciones accidentales.

3. `engine` – Paquete que agrupa los algoritmos de cálculo en una clase denominada `Solver`, la cual trabaja con una serie de clases auxiliares empleadas únicamente durante el cálculo y cuyos valores pueden modificarse (no siendo, por lo tanto, inmutables). Dichas clases son `NodeResForSolver`, `ElemResForSolver` y `ModResForSolver`. El objetivo de este paquete es en definitiva recibir un modelo de cálculo del paquete `input` y devolver un modelo de resultados del paquete `output` empleando para ello una serie de clases intermedias.

4. `ui` – Paquete que agrupa 25 clases relativas a la interfaz con el usuario: ventanas, gráficos, listados, lectura de ficheros, etc. Esta interfaz se dedica a producir y mostrar modelos de prueba (no inmutables) que una vez modificados y aceptados por el usuario se convierten en modelos de cálculo. También permite visualizar el contenido de los modelos de resultados. Esta interfaz puede además ser ampliada, sustituida o reemplazada por otra sin tener por ello que alterar el motor del programa.

La interfaz con el usuario (Figura 4) se ha trabajado de manera que existan diferentes formas de introducir los datos de los modelos así como de chequearlos y presentar los resultados gráficos. La presentación de datos y resultados puede realizarse mediante tablas, gráficos o listados.

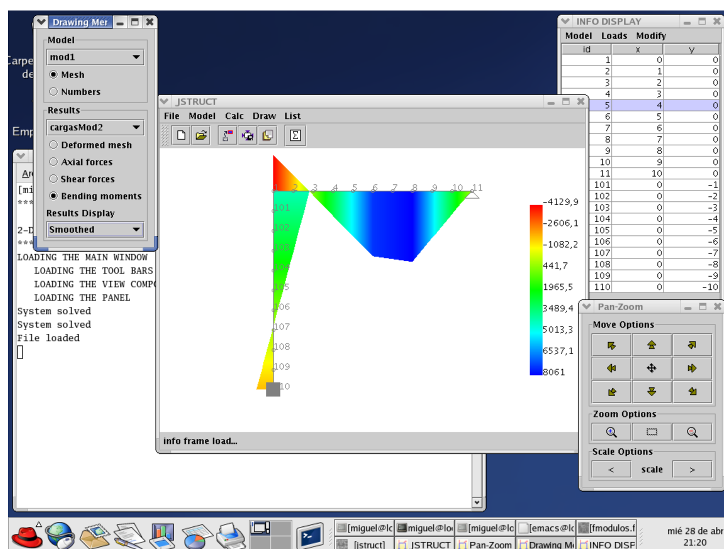

Figura 4. Interfaz gráfica desarrollada para el programa

La introducción de datos puede realizarse exclusivamente desde fichero o bien interactivamente (o mediante una combinación de ambos procedimientos). La salida de resultados puede solicitarse también bajo fichero o de manera interactiva, existiendo diferentes formas de hacerlo (Figura 5).

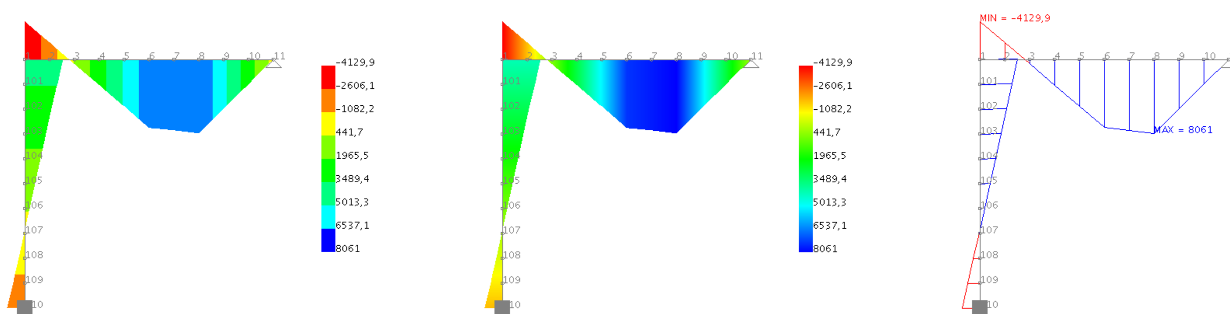

Figura 5. Ejemplo de posibilidades de presentación de resultados. Opciones de relleno de color (izquierda), suavizado (centro) y sin relleno (derecha)

Respecto a los algoritmos de cálculo contenidos en el paquete `engine` cabe destacar que se ha optimizado la resolución del modelo estructural adoptando los siguientes pasos:

- Renumeración de nudos. Antes de comenzar el cálculo se realiza una reordenación de los nudos de la estructura, minimizando el tamaño de la banda de la matriz de rigidez

almacenando exclusivamente el semiancho de banda más el término de la diagonal para optimizar la memoria y operaciones realizadas. Se ha implementado para ello un algoritmo de búsqueda en árbol, basado en una idea presentada en la referencia 11 pero habiendo replanteado y reescrito el algoritmo para pasarlo de programación de etiquetas a programación estructurada. El algoritmo original emplea continuamente la secuencia de dirección `GOTO`, construcción no válida en `Java`. La robustez y velocidad del algoritmo ha llevado a su elección frente a otras posibilidades (algoritmos de McCaulay o bien almacenamiento de tipo skyline de la matriz de rigidez[9]).

- El vector de cargas se forma por adición de las contribuciones en cada nodo de los diferentes tipos de carga que pueden ser introducidos. De esta forma se produce finalmente un único vector de cargas (aunque existan múltiples cargas aplicadas), lo que produce un único vector de desplazamientos.

- La resolución del sistema se realiza por triangularización de la banda almacenada y sustitución inversa según una estrategia iterativa de triple bucle.

- La introducción de muelles se realiza como factor de penalización relativo a la rigidez más alta de la matriz.

Finalmente es interesante comentar que el programa se completa con una documentación `html` que describe las diferentes clases con sus variables y métodos, generada empleando `javadoc`. Aparte de esta documentación, útil para poder ampliar y mejorar el código ya desarrollado, se ha escrito un manual de usuario donde se explica el funcionamiento del programa y sus diferentes posibilidades.

DESARROLLOS FUTUROS

Una vez definida una interfaz para los diferentes objetos, estos pueden reemplazarse, cambiarse o ampliarse de manera sencilla permitiendo además reciclar código y emplear objetos elaborado por otros programadores. De hecho, una clase del programa `ExampleFileFilter.java` se ha tomado directamente de la web de Sun (`http://www.java.sun.com`) y algunas ideas sobre la distribución de objetos en la ventana gráfica se han tomado de un interesante programa llamado `jbeam` de K.G. Schwebke (`http://www.schwebke.com`). Como muestra de esta filosofía, gracias a la estructura de colecciones de modelos de datos y resultados, existe una serie de ampliaciones sencillas de implementar sobre la estructura del programa desarrollado.

Dichas capacidades pueden incorporarse añadiendo nuevas propiedades y métodos a los objetos existentes (pudiendo definir, por ejemplo, problemas tridimensionales) o bien directamente programando nuevos objetos. Entre estas posibles ampliaciones cabría destacar que, gracias al trabajo con colecciones de modelos de resultados, puede abordarse de manera sencilla la generación de envolventes o bien la realización de cálculos dinámicos o no lineales geométricos.

CONCLUSIONES

En este artículo se ha expuesto la evolución que han sufrido los lenguajes de programación a lo largo del tiempo, desde la programación desestructurada hasta la programación completamente orientada a objetos. Esta última, con más de veinte años de existencia, ha mostrado tener una serie de ventajas que le han hecho ocupar una posición dominante dentro de los lenguajes de programación hoy en día, siendo de hecho imprescindible en el desarrollo de interfaces gráficas.

Respecto del cálculo de estructuras, la aplicación de esta filosofía de programación no se ha desarrollado con intensidad, siendo la programación estructurada la que se encuentra actualmente más extendida. Sin embargo, el empleo de la programación orientada a objetos permite evitar gran parte de los inconvenientes de la programación estructurada, siendo más flexible, potente y con mayores perspectivas de ampliación y reutilización en los códigos desarrollados. Además, las posibilidades de encapsulamiento y programación modular permiten que, mediante un correcto establecimiento de interfaces, puedan reutilizarse prácticamente todos los algoritmos y esquemas de resolución existentes incluso en otros lenguajes.

Entre las aplicaciones más ventajosas de este enfoque al cálculo de estructuras cabría destacar:

1. Definición de **objetos**. El trabajo en un programa con entes con significado físico o matemático claro en vez de con matrices, cuyo valor para un determinado índice deba ser establecido arbitrariamente, permite escribir, entender y depurar el código generado de una forma más sencilla. Además, el trabajo con objetos, previa definición de sus interfaces, logra unas mejores posibilidades de ampliación y reutilización de los códigos, bien sea por la persona que desarrolla un programa, bien sea empleando código escrito por otros programadores.

2. Trabajo con **colecciones de objetos**. El empleo de estos objetos (que agrupan a otros de forma que puedan ser recorridos y recuperados de manera sencilla) evita gran parte de los problemas de lectura bajo fichero, introducción o eliminación de nuevos datos en los modelos y el empleo de matrices de conectividad. Su uso es precisamente el que posibilita que un cálculo de estructuras pueda orientarse completamente a objetos, al lograr una eficaz gestión de los datos y resultados.

3. Empleo de la **programación modular**. Finalmente, la agrupación de código por tareas lógicas, nuevamente con un correcto diseño de interfaces, permite dividir el trabajo de forma clara y poder desarrollar y ampliar diferentes partes del programa de manera independiente. El diseño con componentes puede además ser empleado para reutilizar directamente código existente (incluso aunque esté escrito en un lenguaje estructurado), lo que evita partir desde cero a la hora de desarrollar nuevas aplicaciones.

Los autores han diseñado un programa de cálculo de estructuras reticuladas bidimensionales donde se han desarrollado las ideas expuestas a lo largo del artículo. Su código se deja público y expuesto al desarrollo, apuntándose diferentes líneas para ello. La ampliación del mismo no requiere además un conocimiento de las implementación de cada objeto, sino únicamente de las interfaces definidas (las cuales pueden consultarse en una documentación generada con dicho propósito).

En resumen, la programación orientada a objetos es una técnica con un gran potencial, madura en sus planteamientos y desarrollos y que puede ser aplicada en el diseño de programas de cálculo estructural. Su empleo permite superar gran parte de las deficiencias que presenta la programación estructurada pudiendo incluso incorporarse el trabajo que a modo de algoritmos y código ya ha sido realizado en la última. Finalmente, las posibilidades de ampliación, reutilización y separación de tareas de la programación orientada a objetos, permiten que los programas diseñados bajo esta arquitectura tengan un potencial muy superior al de la programación estructurada, siendo una alternativa más recomendable a la hora de abordar nuevos proyectos de cierta entidad.

REFERENCIAS

1 F.J. Ceballos, “Programación orientada a objetos en C++”, ed. ra–ma, (1993).

2 Y. Dubois-Pelerin y T. Zimmermann, “Object–oriented finite element programming: an effective implementation in C++”, *Computational Methods in Applied Mechanics and Engineering*, Vol. **108**, pp. 165–183, (1993).

3 B. Forde, “Object–oriented finite element analysis”, *Computers and Structures*, Vol. **34**, pp. 335–374, (1990).

4 N.M. Josuttis, “*The C++ Standard Library*”, Addison–Wesley, (1999).

5 B. Kernigan y D. Ritchie, “*The C Programming Language*”, Prentice–Hall, (1978).

6 A. Madan, “Object–oriented paradigm in programming for computer–aided analysis of structures”, *ASCE Journal of Computing in Civil Engineering*, Vol. **18**, N° 3, (2004).

7 P. Mira McWilliams, “Análisis por elementos finitos de problemas de rotura de geomateriales”, Tesis doctoral, ETSICCP–Universidad Politécnica de Madrid, (2001).

8 B. Meyer, “*Eiffel: the languaje*”, Prentice–Hall, (1992).

9 S. Muelas Medrano, “Curso básico de programación del método de los elementos finitos”, ETSICCP–Universidad Politécnica de Madrid, Madrid, (1999).

10 E. Oñate, “*Cálculo de estructuras por el método de los elementos finitos*”, Centro Internacional de Métodos Numéricos para la Ingeniería, Barcelona, (1995).

11 F.J. Pozo Vindel, “*ARTCAL–Programa de estructuras articuladas planas*”, ETSICCP–Universidad Politécnica de Madrid, (1995).

12 I. Romero, “*IRIS: User's manual*”, ETSICCP–Universidad Politécnica de Madrid, (2004).

13 H. Schildt,“*Java 2 Manual de referencia*, McGraw–Hill, Madrid, (2001).

14 H. Schildt, “*C Guía de autoenseñanza*”, McGraw–Hill, Madrid, (1994).

15 R.L. Taylor, “*FEAP–A finite element analysis program”*, University of California at Berkeley, Berkeley, (2000).

16 C. Utley, “*Visual basic.NET*”, SAMS, Indianapolis, (2001).

17 T. Zimmermann, Y. Dubois-Pelerin *et al.*, “Object–oriented finite element programming: I governing principles” *Computational Methods in Applied Mechanics and Engineering*, Vol. **98**, pp. 291–303, (1992).