Capítulo 7:

LA PROGRAMACIÓN ORIENTADA A OBJETO Y EL MÉTODO DE LOS ELEMENTOS FINITOS

El objeto de este capítulo es introducir la evolución que ha seguido la programación del método de los elementos finitos en consonancia con las estrategias y lenguajes de programación que han ido apareciendo. A continuación, definir someramente las propiedades de los lenguajes orientados a objeto y, finalmente, describir las implementaciones de tipo práctico, en el marco de un código orientado a objeto, que se han hecho en esta tesis para poner en funcionamiento los conceptos teóricos desarrollados en los capítulos anteriores.

7.1 EVOLUCIÓN DE LA PROGRAMACIÓN Y EL MEF

Los sistemas informáticos de hoy en día, abarcan un grupo de diferentes disciplinas, la correcta conjunción de las cuales determina el éxito en el uso de los ordenadores. En este sentido, es necesaria la adecuada combinación de hardware, sistema operativo y software para conseguir el máximo rendimiento del sistema[1]. En este entorno, se entiende por programación, la generación de un código con la finalidad de que el ordenador ejecute alguna tarea, es decir, la programación es la creación de software.

La programación contempla tres fases básicas: en primer lugar, el análisis del problema y el diseño de la solución informática. En segundo lugar, el desarrollo de la implementación donde se contemplan actividades como la escritura, compilación y depuración del código. Y finalmente, el rediseño de partes del código a partir del uso real del programa. El concepto de generación de software ha evolucionado a lo largo de estos años, inicialmente se asociaba a la existencia de un programa que hacía algo. 'El programa' había sido desarrollado básicamente por un único programador que se encargaba de todas las fases. Sin embargo, hoy en día, el software se suele asociar al concepto de: programa hecho a trozos o grupo de programas. Esto quiere decir, que el desarrollo es el fruto de un equipo organizado.

[1] 'This may involve developing a new hardware system, a new software system or a system with both new hardware and new software.' Winder (1993) [W1].

De todas maneras, y a pesar de la creciente profesionalización de los programadores, se puede afirmar que el mundo de la programación es, ¡por suerte!, absolutamente libre y heterogéneo por varias razones:

- El ordenador ejecutará aquella aplicación que se le programe, desde una contabilidad doméstica hasta un diseño gráfico comp licado.
- Existe una gran diversidad de máquinas y sistemas operativos bajo los que pueden correr las distintas aplicaciones informáticas.
- Los códigos se pueden escribir en una infinidad de lenguajes, y sus versiones de versiones.
- La formación de los programadores es variada.

Todo ello permite, que en la programación, sigan coexistiendo la persona aislada que genera el pequeño programa para cubrir sus necesidades y el gran equipo multidisciplinar que compite en el mercado. Lógicamente, como los objetivos de producto final y calidad para uno y para otro grupo son diferentes, el enfoque que dan a la programación también lo es. Sin embargo, todos los programadores, sea cual sea su naturaleza y entorno de trabajo en el que se encuentren, tienen algunos puntos en común, esto es:

1. La necesaria visión del programa desde el punto de vista del usuario final. En este caso la bondad del software vendría avalada por:
   - *Facilidad de uso*, en este concepto se incluyen aspectos tan diversos como la rapidez de ejecución, la existencia de ayudas y documentación, una presentación cuidada, la portabilidad y uso en distintas máquinas, etc.
   - *Funcionamiento correcto*, debe resolver el problema planteado.
   - *Robustez*, debe funcionar incluso en condiciones anómalas.
2. La visión desde el punto de vista del desarrollador de código. Y aquí se debe cuidar especialmente:
   - *Reusabilidad*, el programador debe intentar que el código sea de fácil lectura evitando sentencias complejas y preocupándose de un flujo lógico, pensando incluso en una posterior revisión que él mismo puede llegar a hacer. También debe prever el potencial de crecimiento del programa, y en consecuencia, dejar algunas puertas abiertas para que se pueda ampliar sin demasiadas dificultades. Y finalmente, debe esforzarse en escribir el código ciñéndose a los estándares del lenguaje que haya escogido.
   - *Eficiencia*, en la medida de lo posible y sin perjuicio para la claridad, debe economizar el tiempo de ejecución y el uso de la memoria.

Desde la primera aparición de los ordenadores hasta nuestros días, la concepción de la programación ha evolucionado mucho. De hecho, la aparición de nuevos lenguajes muchas veces va asociada a las nuevas necesidades conceptuales de los desarrolladores. Históricamente, los primeros programas estaban escritos en lenguajes de bajo nivel, tipo ensamblador, que a la máquina

le eran fáciles de entender. Los algoritmos eran de tipo *secuencial* y el ordenador se limitaba a hacer operaciones según unos pasos predeterminados y en un cierto orden. El tipo de problemas que podían resolver era limitado, y al aparecer los primeros lenguajes y volverse los algoritmos más ambiciosos, nació en la década de los 60-70 la llamada programación *estructurada*. En su día representó una revolución conceptual, en el sentido que todo programa debía hacer algo, y ese algo, se podía dividir en partes más pequeñas y así sucesivamente hasta llegar a unas pequeñas partes programables e indivisibles. Esto dió lugar a la regla, 'decide lo que quieres hacer y utiliza los mejores algoritmos que encuentres'.[2]
El método de los elementos finitos se adapta perfectamente a esta filosofía, la existencia de un flujo principal de programa y de unos cálculos que se repiten muchas veces para distintos elementos de la malla. Por lo tanto, no es de extrañar que alrededor de esos años surgieran los primeros códigos de elementos finitos.

Más tarde, se observó que el crecimiento del código provocaba efectos colaterales en las variables, por ejemplo cambios bruscos de valores de variables en tiempo de ejecución sin razón aparente, o la existencia de variables globales que van cambiando de valor a lo largo de varias subrutinas, y a las que hay que seguirles la pista muy de cerca. Todo esto provocó que en los años 70-80 apareciese un nuevo concepto: defender que en las partes divididas, los módulos, era mejor ocultar algunos datos de manera que al resto del código no le importara lo que hacían ciertas variables, llamadas locales. La regla sería, 'decide que módulos necesitas y divide el programa en partes de manera que los datos se oculten en los módulos'.[3] Esto facilitó que los códigos de elementos finitos, principalmente desarrollados en FORTRAN, crecieran con la incorporación de cálculo no lineal, diferentes elementos, ecuaciones constitutivas, etc. Básicamente este es el nivel de desarrollo de programación en el que se ha quedado el método de los elementos finitos.

Algunos lenguajes posteriores como C o PASCAL permiten crear variables especiales, llamadas tipos de datos. Es decir, definir algo que para el programa se comporta como una variable, y que contiene a otras variables estándares del lenguaje. Los tipos corresponden a la abstracción que el programador hace de los datos reales del problema. Nótese, que todos los lenguajes utilizan un mínimo de definiciones de variables, y que estas definiciones ya son en realidad una abstracción de los datos. Por ejemplo, en el FORTRAN clásico la instrucción *real*8* es la abstracción de un número real, de manera que al definir en el código una instrucción como: *real*8 test* el compilador entenderá que la variable *test* contendrá un número real en tiempo de ejecución. La diferencia, respecto a los tipos de datos, reside en que estos últimos representan la creación por parte del

[2] The original -and probably still most common- programming paradigm is: "Decide which procedures you want; use the best algorithms you can find".' [S2]
[3] 'A set of related procedures and the data they manipulate is often called a module. The programming paradigm is: "Decide which modules you want; partition the program so that data is hidden in modules".' [S2]

usuario de las variables del programa, mientras que los de FORTRAN venían impuestos por el propio lenguaje. Con la idea de la abstracción de tipos se puede imaginar, a modo de ejemplo, que los nodos de un elemento finito podría ser una variable tipo, y en este caso el tipo nodo contendría tres variables reales que serían sus coordenadas. Por lo tanto, para este nivel de programación la regla sería, 'decide que tipos quieres y crea unas operaciones para trabajar con ellos'.[4] Hay que destacar que estos nuevos lenguajes que permiten otro enfoque del tratamiento de los datos y sus operaciones, no fueron especialmente importantes en cambiar la filosofía de programación de los elementos finitos. En todo caso, sólo se aprovechó de ellos algunas ventajas como la generación dinámica de memoria o el trabajo con punteros.

Finalmente, en los años 90 se ha definido la OOP programación orientada a objeto como una evolución natural de la abstracción de datos. La OOP incorpora algunos conceptos nuevos como la herencia, pero esto y más se detallará en el siguiente apartado.

Por lo tanto, la concepción de la programación ha evolucionado en un sentido evidente, la preocupación inicial era conseguir que el programa hiciera algo (¿qué operaciones?) mientras que en los últimos tiempos la preocupación ha sido la organización de la información (¿qué datos?). Las razones de esta evolución subyacen en la esencia de las cosas, los programas se basan en algoritmos y en estructura de datos. Por lo tanto elegir e implementar una estructura de datos puede ser tan importante como las rutinas que la manipulan, incluso más, porque la forma de organizar la información y de acceder a ella está necesariamente determinada por la naturaleza del problema a que se enfrenta el programador. Por otro lado el lenguaje de programación tiene un doble propósito: proporcionar unos conceptos que sirvan para representar el problema y proporcionar un medio para que el programador pueda ordenar cosas a la máquina. La primera consideración potencia la existencia de un lenguaje cercano al problema, mientras que la segunda requiere un lenguaje cercano a la máquina. Lógicamente, la máquina es la que debe adaptarse al lenguaje a través del compilador, el lenguaje debe organizar la información de forma natural y la información surge frente a la reflexión abstracta del problema, sin duda esta concepción de la programación es la que da libertad al desarrollador de código. En cambio, estas ideas tan razonables no se han impuesto en la programación de códigos de elementos finitos, y la gran mayoría de programas de elementos finitos todavía siguen organizándose en base a una programación secuencial-estructurada.

[4] 'Such a type is often called an abstract data type, although I prefer to call it a user-defined type. The programming paradigm becomes: "Decide wich types you want; provide a full set of operations for each type".' [S2]

7.2 CONCEPTOS BÁSICOS DE PROGRAMACIÓN ORIENTADA A OBJETO (OOP) y C++

La creación de un código con una programación orientada a objeto contiene una filosofía completamente distinta. Lo cual no quiere decir ni mejor ni peor, simplemente distinta; con unas ventajas y unos inconvenientes que se analizarán, en los próximos apartados, para el caso concreto de los elementos finitos.

Existe mucha literatura sobre la programación orientada a objeto, y en particular sobre el C++, algunas referencias ya han sido nombradas en el apartado anterior, pero sería interesante destacar Stroupstrup (1993) [S1], Devis (1993) [D1] y Hernández *et al*. (1993) [H1]. La intención de este apartado es situar el marco donde se mueve la programación orientada a objeto, centrando las propiedades y conceptos que se consideran de interés y sin entrar, en ningún momento, en el detalle.

Tal vez sería interesante empezar con un enfoque distinto al que aparece en casi toda la literatura y destacar que la OOP (Programación Orientada a Objeto) está relacionada con la teoría del conocimiento[5] enunciada por Bartlett (1932) y Minsky (1975) citados en Durkin (1994) [D1]. Los autores mencionados, consideran que los humanos tienden a almacenar la información según unos esquemas mentales predeterminados llamados *schemas* o *frames* que se han creado a lo largo del tiempo mediante el cúmulo de experiencias vitales de la persona. De manera, que ante nuevas situaciones, el hombre siempre intenta hacer una referencia a los conocimientos previos que se encuentran almacenados en unidades de información donde hay: datos, actitudes, sensaciones, etc. Por ejemplo, ante la lectura fuera de contexto de la palabra cuadrilátero, el especialista en elementos finitos pensará en los elementos de una malla que tienen asociados conceptos como número de nudos, funciones de forma, etc. En cambio, un boxeador lo asociará a un lugar de combate, al dolor físico, a los aplausos, etc.. Esto, hace particularmente interesante la OOP, porque puede representar el conocimiento, bien de tipo heurístico, o bien de tipo teórico, a partir de los esquemas mentales del individuo.

La base de la programación orientada al objeto son las *Clases.* Una Clase no es más que un esquema mental, una abstracción de un conjunto de datos y de su forma de relacionarse con el mundo. A modo de ejemplo ilustrativo, piénsese que se podría definir la Clase Vehículo, y todo el mundo seguro que la asociaría con una serie de características físicas: la presencia de ruedas, un volante, una fuerza tractora, etc. Y con una forma de relacionarse en el mundo: ser propiedad de

[5] Aquí se produce el enlace conceptual con la nota al pié núm 5 del capítulo 6.

alguien, estar estacionado, frenar, etc. Por lo tanto, la abstracción contiene dos tipos de conocimientos:

1. *Datos o variables* que caracterizan el esquema mental. Por ejemplo, número de ruedas en un vehículo.
2. *Métodos, funciones o procedimientos* que relacionan el esquema mental con el resto de esquemas. Por ejemplo, atropellar a la Clase Hombre.

Nótese que al definir la Clase Vehículo, no se está hablando de un coche, un tractor o un autocar en particular, se está hablando en términos generales, en términos de conceptos abstractos. Por consiguiente, se puede decir que la Clase pretende ser la inducción de aquellas cosas o conceptos que caracterizan un grupo de realidades del problema.

El hecho de asociar la Clase, la idea abstracta, a un elemento real, físico, se llama instanciación de un Objeto: crear un Objeto. En el acto de instanciación, las variables o datos de la Clase tomarán unos valores particulares, y los métodos o funciones, se inicializarán. En el caso que se comentaba anteriromente, se puede definir el Objeto Moto de la Clase Vehículos, y dicho Objeto puede caracterizarse por tener un 2 en la variable *número-de-ruedas*; mientras que el Objeto Coche, va a tener el valor 4 en esa misma variable. Objetos y Clases tienen pues una relación directa, y por ello, casi todos los autores suelen referirse indistintamente a uno u otro, simplemente la Clase es el concepto y el Objeto la realidad.

Lógicamente, existen varios esquemas mentales para un mismo hecho, con lo cual la organización de clases puede ser totalmente diferente entre un programador y otro, según la percepción particular que se tenga del problema o el grado de perversidad personal. En general, la OOP no está necesariamente asociada a un lenguaje en particular, pero es evidente que hay ciertos lenguajes como el C++ que potencian algunas capacidades de este tipo de programación[6]. Entre ellas:

1. **Encapsulado**. El concepto de encapsulamiento se basa en que los Objetos se encargan de las variables a las cuales tienen acceso, porque les pertenecen o porque se lo permiten los métodos. Con ello, se potencia la idea básica de que cada parte del programa sólo sepa aquello que le interesa, y por lo tanto, no puedan existir efectos colaterales con variables de otros Objetos. En C++, además, dentro de cada Clase se puede definir otro encapsulado sobre los datos, en concreto la clasificación de *public*, *protected* y *private* limita el acceso a las variables y métodos.
2. **Herencia**. Unas Clases pueden derivar de otras y aprovechar su forma de trabajar sin tener que reescribir código. La clase hija recibe todo el conocimiento y habilidades de los padres, bueno para ser estrictos todo lo que le permiten porque cada maestrillo tiene su librillo, y los padres

[6] 'I have heard discusions of object-oriented design in C, Pascal, Modula-2 and Chill. Could there somewhere be proponents of object-oriented programming in Fortran and Cobol? I think there must be'. [S2]

puede guardarse cierta información o pasarle la herencia condicionada bajo la etiqueta *private*. La herencia, a veces, obliga a incluir funciones de tipo virtual pura en la madre, funciones que no hacen nada, sólo se declaran para que puedan desarrollarse en los hijos. De manera que una clase hija puede ser un crecimento de la madre (hace más cosas), o una particularización (hace lo mismo de forma diferente).

3. **Polimorfismo**. Los métodos de una clase pueden mantener el mismo nombre pero hacer cosas distintas según os parámetros que reciban, o las clases hijas heredadas pueden entender que ciertas cosas deben hacerse de forma diferente bajo el mismo nombre pero con diferentes objetos.

Hasta aquí se ha hecho un breve resumen de las principales características de la OOP. En la filosofía de programación con orientación al objeto la regla[7] debería ser, 'decide las clases que quieres, dótalas de unas operaciones y potencia las cosas comunes con la herencia'.

7.3 LA OOP Y LOS ELEMENTOS FINITOS

En los últimos años ha crecido el interés por la OOP y su relación con los elementos finitos. El hecho de poder asociar conceptos abstractos como: puntos de Gauss, elementos de una malla o nodos a trozos de código, ha interesado a los especialistas en programación de elementos finitos.

7.3.1 ESTADO DEL ARTE

Una de las primeras contribuciones sobre el tema se debe a Forde *et al*. (1990) [F1], en un artículo bastante claro y muy completo, define el concepto de *framework* de desarrollo para OOP en elementos finitos, y también estructura, a grandes líneas, cual debe ser la jerarquía de clases. La idea del *expandable application framework* surge de los manuales de desarrollo de aplicaciones para Macintosh, y propugna la creación librerías reutilizables de manera que el programador pueda coger aquellas partes de código que le interesen y reescribir las que necesite. De esta manera, el programa de elementos finitos se podría construir a medida de las necesidades del usuario.

[7] 'The programming paradigm is: "Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance".' [S2]

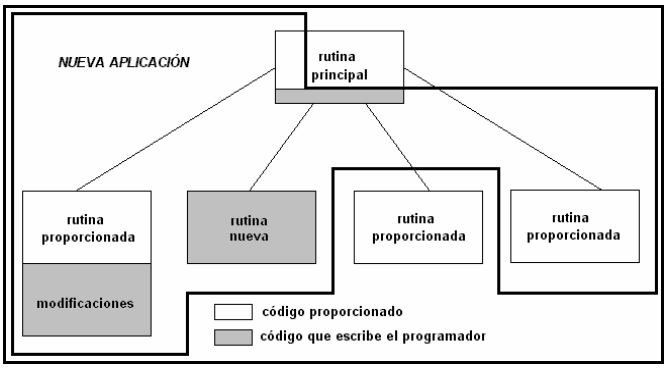


Ilustración 7.1: Esquema del framework de desarrollo

A lo largo del artículo, los autores definen de forma somera la jerarquía de clases para un programa de elementos finitos. Dicha descripción se realiza de forma inductiva: de lo particular a lo general. En concreto, los autores empiezan definiendo las Clases aisladamente, con algunas de las variables y métodos que deberían contener y finalmente, terminan relacionándolas entre ellas a través de otros Objetos. No se va a entrar al detalle de sus descripciones, y los lectores interesados podrán consultar la referencia, pero sí parece interesante resumir algunas de sus ideas que se han intentado reflejar de forma gráfica.

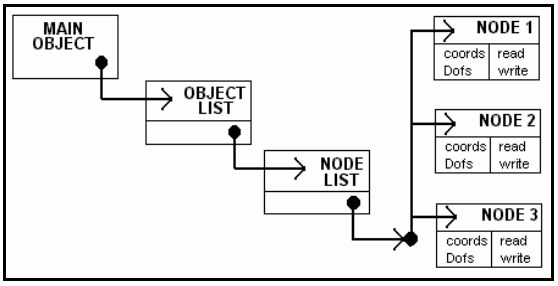


Ilustración 7.2: Organización de listas y objetos

Por ejemplo, citan la necesidad de crear listas de objetos que permitan la fácil gestión de las instanciaciones; dichas listas se controlarían mediante un objeto que podría borrarlas y crearlas, y además, las listas de objetos particulares estarían gestionados por un objeto de grupo. El esquema 7.2 aplica dichos conceptos sobre los objetos Node.

Además en el artículo se ofrece una estructura general de clases para ser implementada en un código de elementos finitos. Las descripciones son de tipo general y no definen en detalle todo el conjunto de variables y métodos que intervendrían en cada uno de los objetos, dejando en manos del lector, supuesto experto en elementos finitos, la definición última del contenido de los objetos. Reflejando de forma gráfica sus ideas se dibuja la ilustración 7.3.

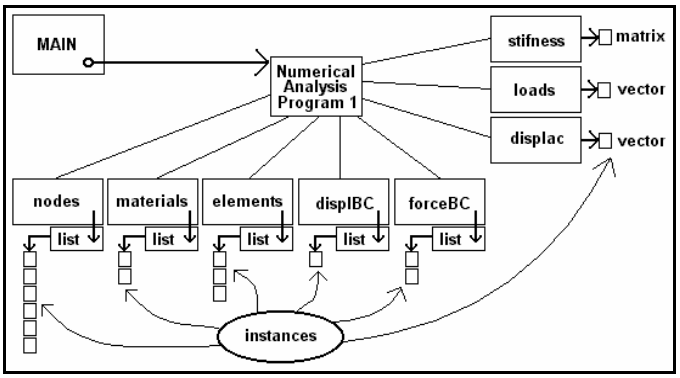


Ilustración 7.3: Jerarquía de clases y el MEF

Otra contribución se debe a Miller (1991) [M1]. En su artículo, aparecen ideas de tipo teórico sobre el interés que puede tener la OOP y el MEF, sin realizar una descripción detallada de la jerarquía de clases se limita a comentar, brevemente, la relación de los objetos con los conceptos del MEF. Así aparecen algunas ideas interesantes, por ejemplo la ineficiencia en la velocidad de ejecución de los códigos orientados a objeto o las ventajas conceptuales que presentan.

Una de las contribuciones más extensas las realizan Zimmermann *et al*. (1992) [Z1] y (1993) [D1], donde describen su visión de la organización de la jerarquía de clases y presentan dos ejemplos en lenguajes SmallTalk y C++, motivos de la tesis doctoral de uno de los autores. En las páginas se encuentra un estudio detallado sobre la eficiencia computacional del código orientado a objeto comparado con un código en FORTRAN para el cálculo con elementos finitos, los resultados muestran que el código en C++ es entre un 25% y 35% más lento que el de FORTRAN según la máquina en que se ejecute. Las razones de dicha ineficiencia son, en opinión de los autores:

- Por un lado, el encapsulamiento provoca que todo objeto usado como argumento de una función arrastre consigo todos los datos y métodos que le son propios, con lo cual se llena la pila de la máquina.
- La mayor cantidad de llamadas a funciones que requiere el manejo de objetos, también tiende a colapsar la pila.
- La herencia resta rapidez, dado que el compilador no sabe que tipo de objeto va a tener que acceder a una determinada función, y deja el enlace correspondiente para el tiempo de ejecución. Los autores estiman que esto retarda en un 5% cada llamada a una función. Ilústrese con un sencillo ejemplo, un elemento de la malla puede llamar a uno de sus métodos más o menos así: $element1 \rightarrow computeBmatrix(Gp)$,y el compilador no sabe *a priori* si el elemento será un cuadrilátero o triángulo, en cuyo caso se deja para el tiempo de ejecución la selección de la correcta matriz gradiente en el punto de Gauss.

Sin embargo, también destacan algunas ventajas como un mantenimiento más sencillo y una mayor potencia de crecimiento del código.

Otro artículo sobre el tema es de Kong *et al*. (1995)[K1], donde los autores dan su visión personal del problema sin ninguna nueva contribución relevante.

Las últimas contribuciones sobre OOP vienen referidas a un entorno general de elementos finitos, que trabaje con lenguaje simbólico y permita construir la forma débil del problema, y discretizarla con una notación parecida a las fórmulas teóricas que aparecen en los libros de texto. Las referencias son Zimmermann *et al*. (1996)[Z2] y (1996)[E1].

7.3.2 FEMLAB - THE FINITE ELEMENT LABORATORY APPLICATION

FemLab es *'an object-oriented finite element framework'* en palabras de su autor Galindo (1994)[G1]. FemLab proporciona una estructura general básica a través de una jerarquía de clases que pretende facilitar al programador la confección de un programa de elementos finitos hecho 'a medida'. El programador se encarga de seleccionar el tipo y cantidad de código que necesita a partir de su modelo matemático. Es decir, escoge las clases que le interesan y las modifica a su antojo con la seguridad que el nuevo código no va a tener efectos colaterales sobre el resto de clases por la propiedad de la encapsulación. También puede crear clases completamente nuevas o derivar algunas, utilizando los conceptos de herencia y finalmente, puede redefinir funciones con la propiedad del polimorfismo y ayudar a mantener una coherencia en la lectura del código. Por lo tanto, se puede afirmar que el código de Femlab fue escrito intentando explotar al máximo las potencialidades de OOP del C++.

La versión original de Galindo permitía el análisis lineal estático y dinámico en una, dos y tres dimensiones con un número elevado de diferentes elementos. Dentro de FemLab aparecen complejos conceptos de programación como son: los Sets o grupos de datos, los Plexes o listas enlazadas de objetos, librerías de vectores y matrices, lenguaje simbólico en las operaciones matriciales y, sobre todo, la jerarquía de clases que representa el conocimiento del funcionamiento de los elementos finitos. Todo ello permite afirmar que el trabajo desarrollado por Galindo se puede considerar de excepcional.

El entorno FemLab proporciona un código con una jerarquía de clases cuyas principales relaciones se establecen en la ilustración 7.4. Nótese que la relación de las clases con los conceptos de elementos finitos esta perfectamente relacionada.

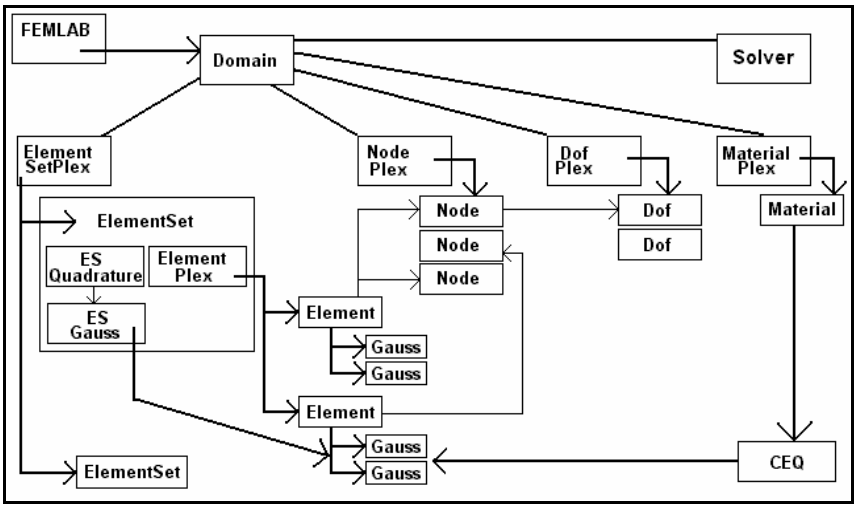


Ilustración 7.4: FemLab y el MEF

De entre los puntos fuertes de FemLab se destacan:

- La creación de una estructura lógica y consistente, con la teoría de elementos finitos, que se ha establecido al definir las relaciones entre: Sets de elementos, elementos, puntos de gauss, ecuación constitutiva y materiales , así como entre nodos y elementos.
- La definición de un Solver asociado al dominio representa que en algún momento se duplicará la información de los Dofs, pero la ganancia en facilidad de lectura de código supera ese pequeño inconveniente. Para comprobar la mayor claridad de FemLab basta con comparar la idea de Forbe [F1], comentado en el estado del arte en 7.3.1, de introducir un objeto matriz del sistema, un vector de fuerzas y otro de desplazamientos.
- La posibilidad de utilizar expresiones simbólicas en las operaciones matriciales aumenta exponencialmente la claridad del código.
- La idea de personalizar la aplicación a través de una clase Builder donde estarían definidos sólo aquellos elementos que interesan enlaza directamente con el concepto de entorno.

Sin embargo, algunos aspectos son criticables a nivel de la concepción de elementos finitos:

- La existencia de una lista de Dof *Degrees of freedom*, en la clase Domain. Domain pretende ser el dominio, por lo tanto, es lógico que contenga una lista de Nodes, de Sets de elementos y de Materiales. Pero los grados de libertad deberían ser gestionados exclusivamente por los nodos, igual que los puntos de Gauss son responsabilidad de los elementos y no del dominio. En esto Forbe presenta un esquema más lógico y atractivo.
- El problema estático y dinámico está mezclado en el interior del elemento. Esto significa que la ejecución de una u otra opción queda en manos del valor que tome una variable que se pasa

como argumento al método correspondiente. Esto resta elegancia y viola, claramente, el principio de encapsulación, ya que el análisis estático no tendría porque saber lo que hace el dinámico.

A nivel de conceptos generales de programación destacaría como negativo que:

- Un abuso en la existencia de objetos, por ejemplo FemLab es un objeto en sí mismo. Naturalemente que su intención es controlar el algoritmo general para el cálculo con elementos finitos, pero en realidad, quién acaba ejecutando los métodos es Domain. Entonces, la secuencialidad intrínseca del método de los elementos finitos se diluye en el interior de los objetos. En alguna ocasión, como este caso, se debería haber potenciado la programación secuencial al lado de la organización en objetos. Virtud que permite C++.
- La lectura y escritura de datos se dejan como responsabilidad de cada clase, esto significa que los objetos leen y escriben a medida que lo necesitan. El concepto está claramente relacionado con la filosofía de OOP pero no facilita en absoluto la manipulación del archivo de datos, en el sentido de añadir la lectura o la escritura de variables nuevas. Además, la lectura se hace a través de un intérprete de lenguaje que descodifica líneas de sintaxis, y a veces, peca de poca flexibilidad a la hora de definir tipos de variable. Por desgracia, tanto la lectura como la escritura de datos no siguen ningún estándar de preproceso o postproceso, y esto resta potencia al framework. Finalmente, comentar que, en general, las operaciones de escritura y lectura es recomendable hacerlas de golpe, en la medida de lo posible, con lo cual se debería modificar la filosofía de lectura y escritura general del entorno.
- En ocasiones se ha abusado de las definiciones restrictivas sobre los datos, de manera que algunas regiones *private* se han tenido que reprogramar para no restar potencia a la herencia.
- Por contra, a veces, se ha abusado de la compartimentación de datos y se ha violado el principio de encapsulación definiendo clases y funciones *friends* que pueden acceder libremente a datos de otras clases.
- A veces, las definiciones no siguen los estándares del lenguaje y provoca que el cambio de compilador afecte a la portabilidad del código.
- En general, el principio de entorno de trabajo no se ha conseguido porque las clases están muy relacionadas unas con otras y resulta difícil seleccionar aquellas partes de código que realmente interesan.

7.4 LAS NUEVAS IMPLEMENTACIONES EN FEMLAB

A partir del FemLab básico se han modificado en esta tesis algunos de los aspectos negativos que se destacaban en el punto anterior y, al mismo tiempo, se ha creado un conjunto de clases y estrategias de cálculo para satisfacer los objetivos planteados en el primer capítulo:

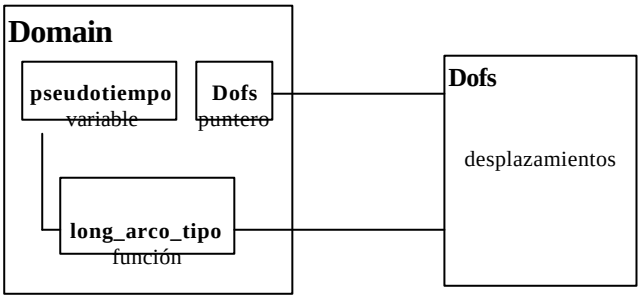
- Cálculo con el método de longitud de arco.
- Modelo de plasticidad asociada con ley de endurecimiento lineal.
- Modelo de daño.
- Análisis de sensibilidad lineal y no lineal.

A continuación, se describen algunas de las particularidades y problemas que afectan a este tipo de implementación.

7.4.1 EL CÁLCULO CON LONGITUD DE ARCO

La estrategia con longitud de arco pretende resolver el problema a fuerzas impuestas, pero con un nivel de solicitación variable, y condicionar algún desplazamiento o combinación de desplazamientos. El algoritmo aparece en el capítulo 5, algoritmo 5.3. En el caso de una organización con objetos, la longitud de arco modifica el nivel de fuerzas externas, por lo tanto, las funciones que trabajan con el control de desplazamientos deben tener acceso a la variable pseudotiempo que se encarga de incrementar las cargas. Dicha variable está contenida en uno de los objetos principales el sistema, Domain, por lo tanto en ese objeto hay que incluir todas las funciones relacionadas con la longitud de arco.

Esquema 7.1: la longitud de arco en los objetos



Para calcular la condición de arco, es necesario el acceso a los desplazamientos que se guardan en los grados de libertad, objetos Dofs. Dado que Domain los tiene en su interior, dicho acceso es trivial, sin embargo si, estuviesen dependientes única y exclusivamente de los nodos, tal y como sería deseable, también se podría acceder a la información necesaria sin mayores problemas.

7.4.2 MODELO DE PLASTICIDAD ASOCIADA

El comportamiento constitutivo viene definido por las relaciones tensión deformación, dichas relaciones fundamentan la respuesta interna de la estructura. En consecuencia, tal y como se comentó e ilustró en el capítulo 4, el cálculo de las tensiones se realiza en la evaluación de las fuerzas internas que contrarrestan la acción de las fuerzas externas solicitantes. La acción de las fuerzas internas se evalúa a través de la integración de las tensiones a lo largo del dominio, según:

$$f_i = \int_V B^t \sigma dV \qquad 7.3.1$$

Por lo tanto, el cálculo de las tensiones se materializa en los puntos de integración de los elementos que se utilizan en la integración numérica. Bajo estas consideraciones, se debe implementar el algoritmo 4.2 del capítulo 4 que controla el comportamiento constitutivo elastoplástico con una predicción-corrección.

En vista de todo ello, se han creado dos clases principales para controlar el comportamiento elastoplástico:

- Una clase para el material elastoplástico, llamada PlasticM. Dicha clase contiene información acerca de las propiedades elásticas y de la función de endurecimiento que regulará el comportamiento límite del material.

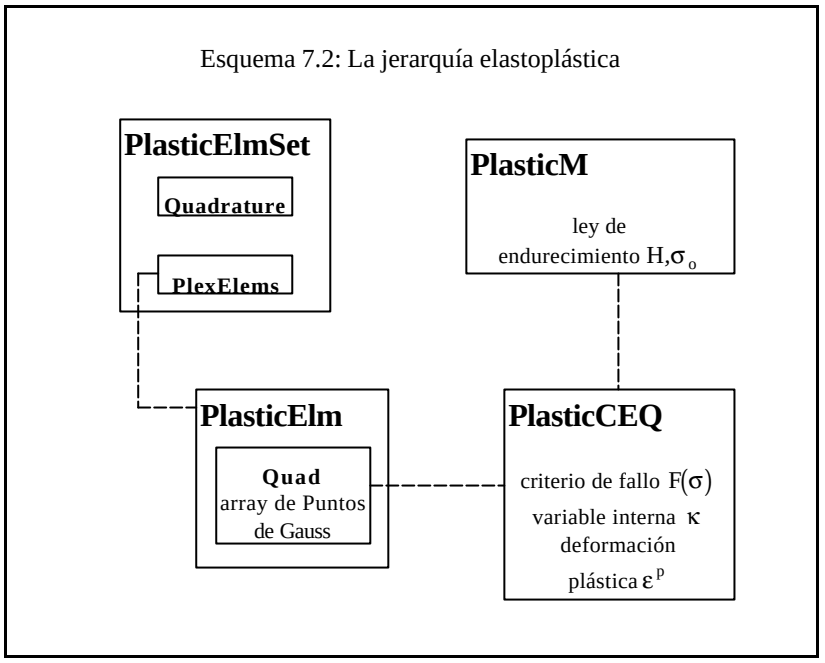
- Otra para el comportamiento constitutivo, PlasticCEQ. Esta clase contiene todas las variables internas de la formulación, el criterio de fallo, la regla de flujo, así como la matriz consistente de la formulación.

Para mantener la coherencia con el código y la organización jerárquica de clases original de FemLab, se han definido otras clases subalternas que son:

- La clase del Set de elementos elástoplásticos, PlasticElmSet, que contiene la cuadratura general y el Plex de elementos.

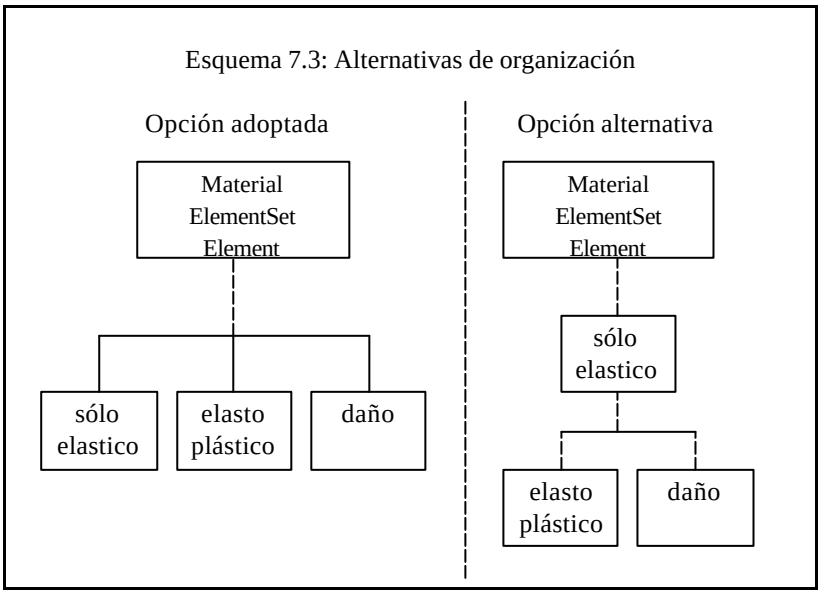
- Y la clase que representa los elementos finitos de tipo elastoplástico, PlasticElm. En su interior se guardan los procedimentos y variables que se encargan de calcular la matriz de rigidez del elemento y los punteros a los objetos de la ecuación constitutiva.

Esquema 7.2: La jerarquía elastoplástica



Todas las clases definidas anteriormente derivan de las clases generales Material, ElementSet y Element, y en consecuencia, son independientes de las clases que definen el comportamiento elástico del material. Esto puede ser motivo de controversias, dado que un posible planteamiento alternativo sería derivar las clases elastoplásticas de las clases elásticas, ElasticM, SolidElmSet y SolidElm. Según el esquema siguiente:

Esquema 7.3: Alternativas de organización

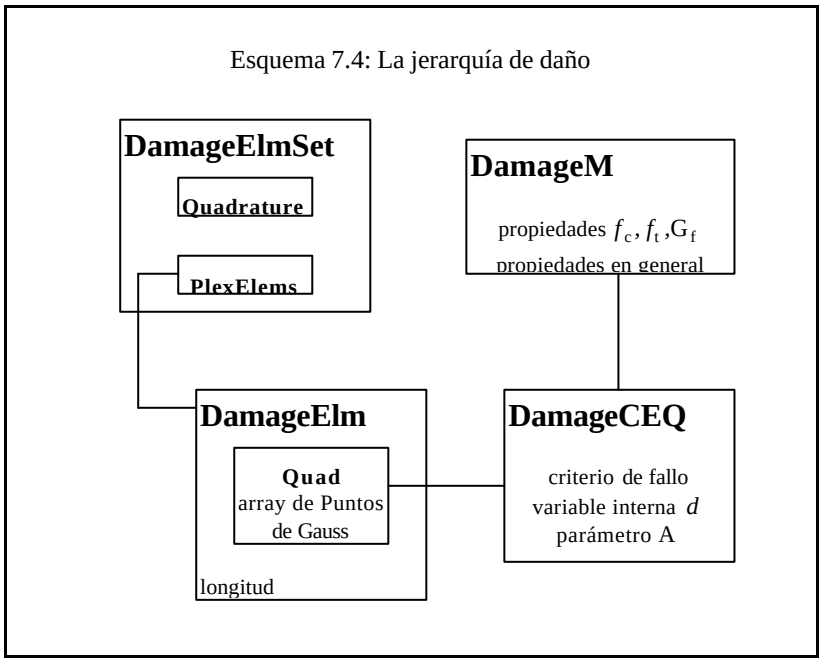


Sin embargo, dicha posibilidad se desechó porque, a nuestro juicio, la teoría elastoplástica define su propio esquema de comportamiento tanto en régimen lineal como plástico, por lo tanto es lógico que pueda existir por ella misma y no como una herencia de la elasticidad, además se tendrían que redefinir todas las funciones y complicaría la lectura del código. Aunque hay que reconocer, que también pueden formularse razonamientos que defiendan la otra posibilidad, por ejemplo reutilización de código y aprovechamiento del polimorfismo.

7.4.3 MODELO DE DAÑO

En este caso, la organización de clases es parecida a la de los elementos elastoplásticos pero con las variables y métodos del modelo de daño siguiendo el algoritmo 5.1 del capítulo 5. La estructura es parecida, generando unas clases DamageM para el material y DamageCEQ para la ecuación constitutiva; así como una DamageElm para los elementos y DamageElmSet para el Set. Como particularidad, destacar que la longitud característica se asocia al elemento y no al objeto que tiene la ecuación constitutiva, esta organización es más lógica aunque el elemento no utiliza para nada dicha variable. Gracias al puntero que relaciona el elemento con su ecuación constitutiva, se utiliza la longitud característica para calcular el parámetro A de la formulación.

Esquema 7.4: La jerarquía de daño



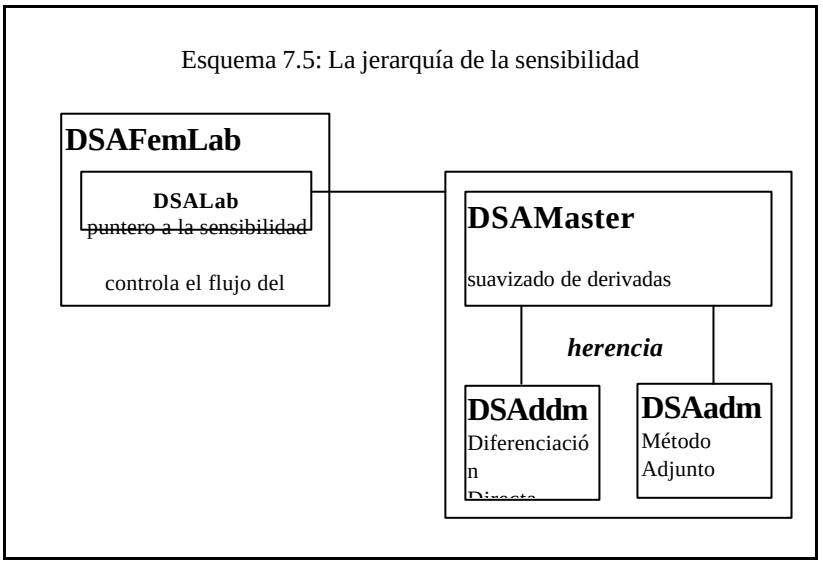
7.4.4 EL ANÁLISIS DE SENSIBILIDAD LINEAL Y NO LINEAL

El análisis de sensibilidad comprende el cálculo de las derivadas de la respuesta estructural respecto a las variables de diseño. La inclusión de dicho análisis provoca que se tenga que modificar toda la estructura interna de FemLab a distintos niveles.

- Creación de clases bases para organizar el cálculo de las derivadas a nivel algorítmico. Creación de clases nuevas DSAMaster, DSAddm, DSAadm y derivadas de las del análisis normal DSAFemLab.
- Derivación de las clases elementales para poder calcular el sistema de ecuaciones de sensibilidad en régimen lineal y no lineal: SolidDSAElmSet, SolidDSAElm, PlasticDSAElmSet, PlasticDSAElm, DamageDSAElmSet y DamageDSAElm. En caso de modificación de cargas SolidDSALoadSet y SolidDSALoad.
- Finalmente modificar clases existentes para poder almacenar la información que añade la sensibilidad: Dofs, CEQs y cuadraturas.

A modo de ejemplo se muestra, en el esquema 7.6, la jerarquía que se ha definido para los elementos lineales, entendiendo que dicho esquema es generalizable al resto de elementos. Nótese que las partes oscuras son las modificadas o añadidas, y las flechas representan la herencia entre clases. Con esta estructura se sigue el principio de ir construyendo compartimentos estancos y relacionados de forma lógica entre ellos.

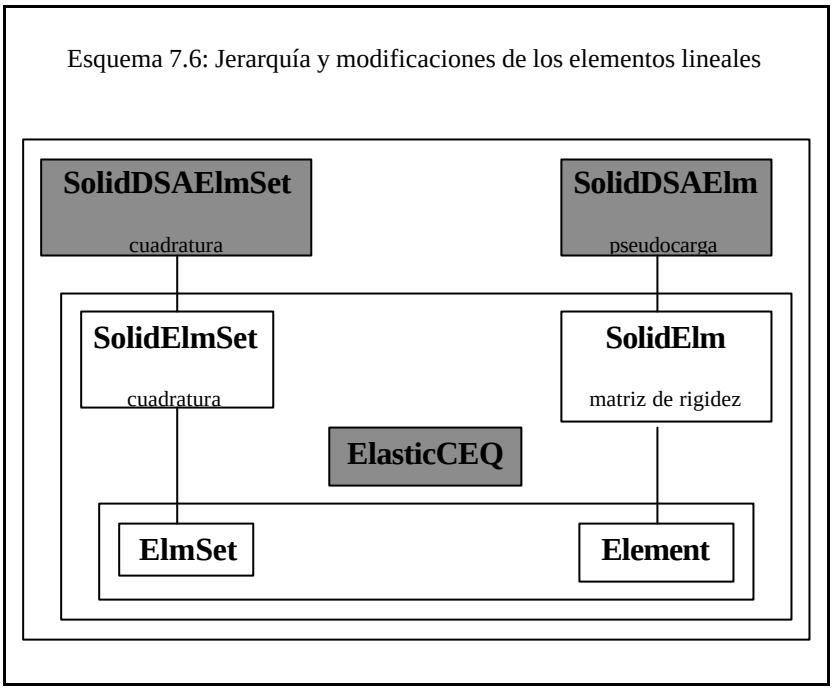
Esquema 7.5: La jerarquía de la sensibilidad



En detrimento de la organización del esquema 7.6 que aparece en la página siguiente, se deben comentar los inconvenientes de la herencia. Es deseable que en las funciones principales del código se realicen llamadas a los objetos padres, para no tener que retocar el código de carácter secuencial que ejecuta el cálculo por elementos finitos. Esto representa que existe la necesidad de

definir funciones virtuales puras en las clases madres para que el compilador[8] anuncie que se debe hacer una llamada a la función correcta de la hija en tiempo de ejecución.

Esquema 7.6: Jerarquía y modificaciones de los elementos lineales

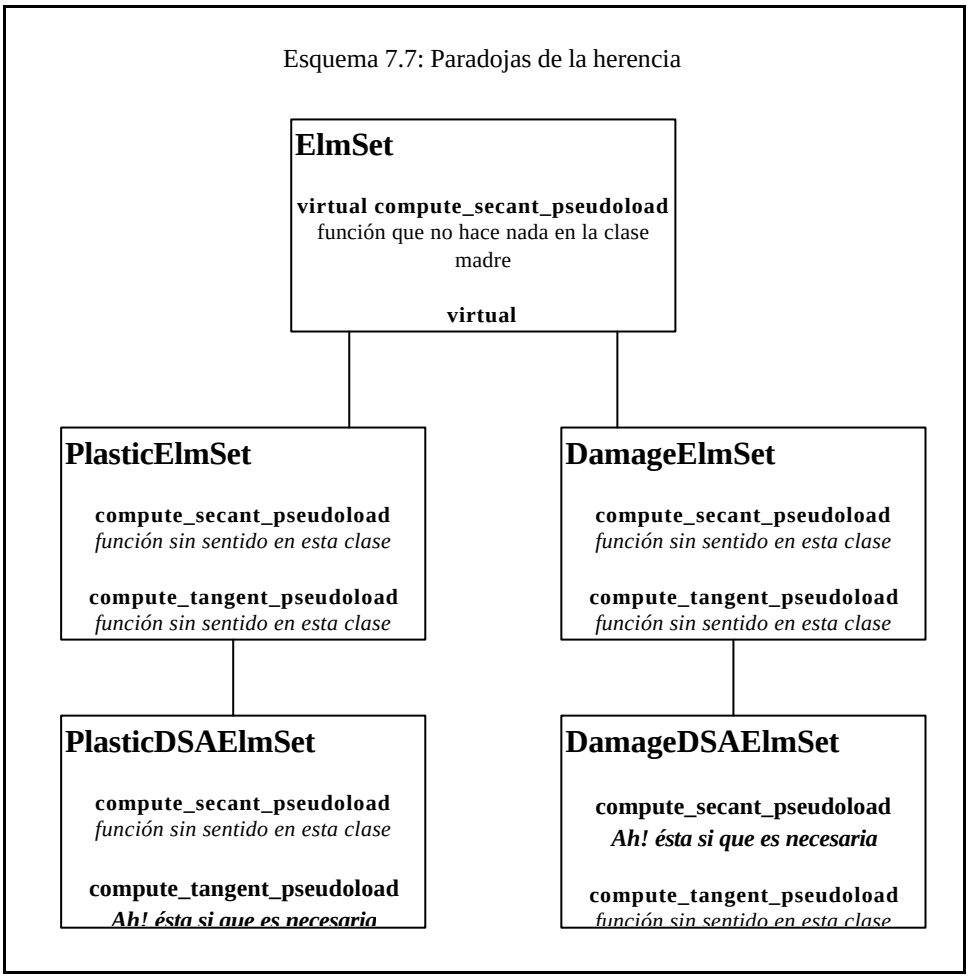


De forma inevitable, este hecho comentado, implica que se va a producir un incremento de definiciones de funciones que no hacen nada. Hay que reconocer que hasta cierto punto es lógico, según las propiedades de la herencia, pero desafortunadamente provoca una mayor complejidad en la lectura del código.

Por ejemplo, en la implementación del cálculo de la pseudocarga para el análisis de sensibilidad se produce la siguiente paradoja, las clases derivadas que no trabajan con los conceptos de sensibilidad tienen funciones que no hacen nada, entonces, ¿por qué están allí?... Pues simplemente por exigencias del compilador. Este problema queda ilustrado en el esquema 7.7. La solución pasa por definir todo un conjunto de clases paralelas que trabajen con la sensibilidad, pero esto representaría no aprovechar en absoluto la herencia y duplicar el código excesivamente.

[8] Se ha utilizado el compilador g++ de *gnu project.*

Esquema 7.7: Paradojas de la herencia



Finalmente, comentar que en la clase de los Dofs el cálculo de sensibilidad provoca que aumente el número de datos que se guardan, dado que se debe calcular un vector de las derivadas del grado de libertad con respecto a las variables de diseño que se definan. La naturaleza vectorial, provoca que las dimensiones en memoria puedan desbordarse.

Por lo tanto, es necesario un almacenamiento dinámico de dichas magnitudes. De esta manera, si el análisis es sin sensibilidad, sólo se debe reservar un espacio para el apuntador del vector y de esta manera no se producen pérdida de memoria.

BIBLIOGRAFÍA ESPECÍFICA DEL CAPÍTULO

[D1] Devis, R. *Programación Orientada a Objetos en C++*. Ed. Paraninfo. 1993.

[D2] Durkin, J. *Expert Systems. Design and Development*. Ed. Prentice-Hall. 1994.

[D3] Dubois-Pèlerin, Y. y Zimmermann, T. "Object-Oriented finite element programming: III An efficient implementation in C++". *Computer Methods in Applied Mechanics and Engineering*. 108, 165-183, 1993.

[E1] Eyheramendy, D. y Zimmermann, T. "Object-Oriented finite element. II A symbolic environment fo automatic programming". *Computer Methods in Applied Mechanics and Engineering*. 132, 277-304, 1996.

[F1] Forde, B. W. R., Foschi, R. O. y Stiemer, S. F. "Object-Oriented Finite Element Analysis". *Computers and Structures*, 34, 355-374, 1990.

[G1] Galindo, M. *"FemLab versión 1.0."* Technical Report núm IT-114. Published by CIMNE. Barcelona, 1994.

[H1] Hernández E., y Hernández, J. *Programación en C++*. Ed. Paraninfo. 1993.

[K1] Kong, X. A. y Chen, D. P. "An Object-Oriented Design of FEM Programs". *Computers and Structures*. 57, 157-166, 1995.

[M1] Miller, G. R. "An Object-Oriented Approach to Structural Analysis and Design" *Computers and Structures*, 40, 75-82, 1991.

[S1] Stroustrup, B. *El lenguaje de programación C++*. Ed. Addison-Wesley Iberoamericana, 1993.

[S2] Stroustrup, B. "What is Object-Oriented Programming?", *IEEE Software*, pp 10 -20. May 1988.

[W1] Winder, R. *Developing C++ software*. Ed. John Wiley & sons. 1993

[Z1] Zimmermann, T., Dubois-Pèlerin, Y. y Bomme, P. "Object-Oriented finite element programming: I Governing principles". *Computer Methods in Applied Mechanics and Engineering*. 98, 291-303, 1992.

[Z2] Zimmermann, T. y Eyheramendy, D. "Object-Oriented finite element. I Principles of symbolic derivations and automatic programming". *Computer Methods in Applied Mechanics and Engineering*. 132, 259-276, 1996.