Microarquitecturas orientadas a objetos para la representación matemática de problemas ingenieriles.

Ing. Zulema Beatriz Rosanigo
Investigador y Profesor Asociado Ded. exclusiva -
Facultad de Ingeniería - U.N.P.S.J.B. - Sede Trelew
e-mail: zrosanigo@acm.org

Resumen

La complejidad natural de los problemas ingenieriles puede ser reducida si se abstrae y modela el comportamiento matemático por un lado, y los conceptos físicos que lo aplican, por otro.

En este trabajo se presenta un modelo de diseño orientado a objetos para un uso flexible de matrices y funciones matemáticas en la solución de problemas ingenieriles.

La propuesta busca enfatizar la reusabilidad y posibilidad de evolución a través de aplicar sistemáticamente patrones de diseño que conducen a soluciones flexibles y modificables.

Palabras claves: patrones de diseño, microarquitecturas orientadas a objetos, comportamiento matemático

Introducción

El principal desafío en el desarrollo de software es mejorar la calidad y reducir el costo de las soluciones basadas en computadoras. Una manera de ayudar a cumplir con este objetivo es maximizar el reuso y posibilidad de evolución.

Para enfrentar y administrar la complejidad inherente a nuestro mundo real, contamos con tres mecanismos fundamentales: abstracción, generalización y composición. La *abstracción* es el acto o resultado de eliminar diferencias entre los objetos, de modo que podamos ver los aspectos comunes, como resultado, obtenemos los conceptos. La *composición* es un mecanismo que permite formar un todo a partir de las partes que lo conforman. La *generalización* nos permite distinguir que un concepto es más general que otro, pudiendo establecer jerarquías entre ellos. Lo opuesto a la generalización es la especialización y ambos mecanismos son complementarios.

La tecnología orientada a objetos es una herramienta muy poderosa para resolver problemas de gran envergadura y complejidad, que requieren alto grado de integridad en la información y facilidades para la extensión y evolución. Es la manera más efectiva de lograr reusabilidad, pues permite definir mediante abstracción y composición, los diferentes componentes de una aplicación, que son conectados para lograr el comportamiento esperado del sistema.

Con la aparición de los patrones de diseño [Gamma+95] [Pree95] aparece un nuevo nivel de reutilización no disponible hasta entonces en otras tecnologías. Un patrón es una forma o estructura identificable y reconocible en diferentes dominios; es una idea que ha sido útil en un contexto y probablemente lo sea en otros. Un patrón brinda una solución a un problema recurrente, identificando los subsistemas y componentes y los mecanismos de colaboración entre ellos. Es un modo de proveer información en forma de una declaración de problema, algunas restricciones, una presentación de una solución ampliamente aceptada al problema, y luego una discusión de las consecuencias de esa solución. Estas soluciones pueden ser catalogadas en un manual de diseño de manera semejante en que manuales de otras ingenierías describen soluciones a problemas típicos de sus respectivas áreas.

Los patrones de diseño cambian la forma en que una aplicación puede ser diseñada. Si se cuenta con un amplio conjunto de patrones bien definidos, el proceso de diseño puede transformarse en un proceso que involucra la búsqueda y selección de los patrones que resuelven cada uno de los problemas detectados y luego aplicar la solución prescripta por cada patrón seleccionado hasta que no haya más problemas para resolver [Beck94]. Este enfoque presenta una manera sistemática de reutilizar "conocimiento"de diseño.

Siguiendo esta idea, se planteó estudiar el dominio de la Ingeniería Estructural, abstraer las características y comportamiento relevantes, y diseñar aplicando los principios modernos de la Ingeniería de Software y de la Tecnología de Objetos, haciendo uso de patrones de diseño para lograr los beneficios de reusabilidad, extensibilidad y mantenibilidad.

Para reducir la complejidad inherente de los problemas ingenieriles, se abstrae y modela el comportamiento matemático por un lado, y los conceptos físicos que lo aplican, por otro.

En este artículo se presenta un modelo de diseño orientado a objetos, propuesto para uso flexible de matrices y funciones matemáticas en la solución de problemas ingenieriles. Las microarquitecturas desarrolladas han sido representadas mediante diagramas de estructura estática siguiendo los lineamientos de la notación UML (Unified Modeling Language) definida en [Rational97].

Funciones y Métodos numéricos

En general, la mayoría de los problemas de la Física y de la Ingeniería, describen el comportamiento y propiedades de sus objetos a través de funciones:

- La forma de un objeto puede describirse y calcularse conociendo su función de contorno.
- Diferentes propiedades de uno o más objetos pueden ser descriptas por la misma forma funcional.
- Un mismo objeto puede describir una propiedad o comportamiento bajo determinadas circunstancias, por ejemplo, mediante una función f_A y bajo otras circunstancias mediante una función f_B.
- Muchas propiedades físicas o químicas poseen un esquema de variación funcional. Conocida la función que describe un comportamiento resulta sencillo poder evaluarlo en diferentes casos.
- Problemas de diferente índole basan su solución en esquemas matemáticos similares. Por ejemplo, ciertas propiedades se calculan en base a otras, integrando o derivando la función que describe su comportamiento.

Para poder independizar el comportamiento de un objeto de la función matemática que lo describe, resulta necesario contar con una jerarquía de clases que definan el comportamiento matemático específico. Los objetos cuyo comportamiento o propiedades pueden ser descriptas a través de funciones, estarán compuestos con atributos que serán instancias de algún tipo de función, y delegarán en ellas, la evaluación funcional que corresponda.

Un ejemplo:

> Las cargas que actúan sobre una estructura pueden distribuirse de diferentes formas. De acuerdo al valor y forma en que se distribuyen, varía el efecto que tiene sobre la estructura. El valor de la resultante de una carga repartida depende de la forma de distribución. Puede calcularse integrando la función que representa su distribución en un intervalo.

En lugar de subclasificar a las cargas de acuerdo con la forma en que se distribuyen, y para cada una de estas subclases, redefinir las propiedades dependientes de la forma; se plantea mantener un atributo en Cargas que represente la función de distribución como una instancia de alguna subclase concreta de Función, delegando en ese componente, la evaluación del comportamiento funcional específico. De esta manera, las jerarquías Cargas y Función, pueden evolucionar independientemente y ser reutilizadas en otro contexto.

En este ejemplo, cuando a una carga repartida se le solicita su resultante, ésta retornaría una nueva carga concentrada cuyo valor lo obtiene solicitando a su distribución que se integre en el intervalo. Como la distribución es una instancia de la clase Función, sabría hacerlo.

Indudablemente, al abstraer el comportamiento matemático propio de las funciones en una jerarquía de clases independiente del contexto en que se usa, se facilita el modelado de los conceptos del dominio ingeniril.

Por ello, necesitamos definir una jerarquía de clases que representen las funciones y comportamientos funcionales.

Existen diferentes tipos y familias de funciones: polinomios, trigonométricas, logarítmica, en una o más variables, etc. También se pueden combinar a través de composición y operaciones algebraicas.

Una clase abstracta, *Función,* raíz de la jerarquía, define el comportamiento común de sus subclases y garantiza una misma interfaz. Las principales familias quedan representadas por subclasificación directa, como es el caso de las funciones polinómicas.

Para dar lugar a poder definir funciones cuya expresión analítica es conocida en tiempo de ejecución o propuesta por el usuario, se propone una subclase *Genérica* en la que para evaluar la expresión deberá recurrirse a un intérprete.

Entre las operaciones más utilizadas podemos mencionar la evaluación, la derivación y la integración. Cualquiera fuera la función, debiera brindar el servicio de evaluarse en un punto (evalAt:), evaluar su integral en un intervalo (integrarAt: unIntervalo) y de evaluar su derivada enésima en un punto (derivadaN:At:).

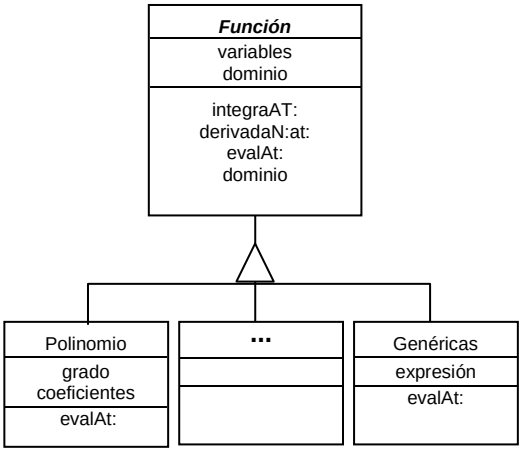
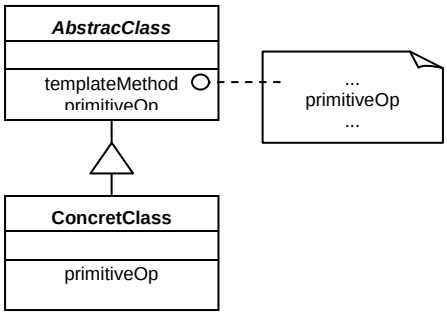


Figura 1 Función

Algunas operaciones podrán ser implementadas en la clase abstracta aplicando el patrón de diseño **Template Method**, que define el esqueleto de un algoritmo en una operación dejando que algunos pasos del algoritmo sean definidos por sus subclases. Los métodos implementados en las subclases se los conoce con el nombre de métodos Hooke.



AbstractClass*:* define las operaciones abstractas (primitiveOp) que las clases concretas redefinen para implementar el algoritmo.
Implementa el método *templateMethod* definiendo el esqueleto de un algoritmo, invcando a opera-ciones abstractas y concretas.

ConcreteClass: implementa el método *primitiveOp* para llevar a cabo los pasos específicos del algoritmo

Figura 2 Template Method

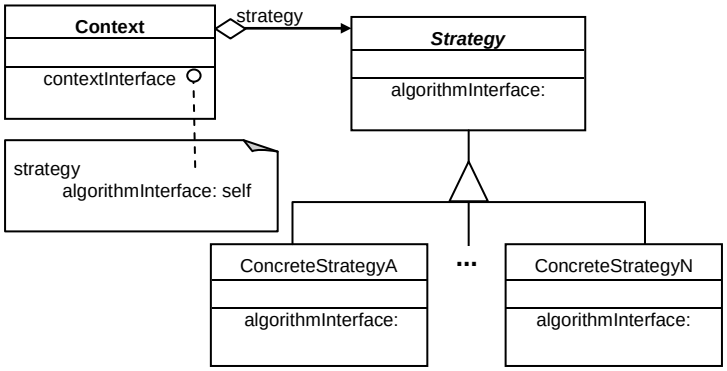
La operación básica evalAt: regresa el valor funcional para la lista de argumentos pasados. Se trata de un método *Hook* que debe ser escrito en cada subclase. Otras operaciones pueden ser definidas en la clase abstracta a partir de ésta.

Puede resultar muy útil que las familias de funciones más utilizadas queden representadas en la jerarquía y que puedan brindar sus funciones primitivas y sus funciones derivadas. En estos casos, el cálculo de la derivada enésima en un punto (derivadaN:at:) se resuelve solicitando el servicio de evaluarse en un punto (evalAt:) a su función derivada enésima. Algo similar pasaría con el cálculo de la integral en un intervalo (integrarAt: unIntervalo).

Si bien siempre es posible encontrar la expresión analítica de la función derivada, no siempre resulta práctico. Y en el caso de las integrales, no todas las funciones tienen primitivas (expresión analítica de su integral), requiriéndose de algún método numérico para su evaluación en un intervalo. Incluso en el caso que sea posible la integración analítica, la integración numérica puede ahorrar tiempo y esfuerzo si solo se desea conocer el valor numérico de la integral.

En general existe una familia de algoritmos para cada operación numérica: varios métodos de resolución numérica para integrales, derivadas, interpolación. En determinadas circunstancias, algunos resultan más apropiados que otros. El método debiera poder elegirse dinámicamente.

Cada tipo de función tendría que tener posibilidad de resolver la operación con el algoritmo que le resulte más apropiado en su contexto de aplicación. ¿Cómo podríamos resolver este problema evitando generar numerosas subclases que varían sólo en su comportamiento y configurar dinámicamente una clase única con alguno de los muchos comportamientos posibles?. La solución para este problema es aplicar el patrón de diseño **Strategy** definido en [Gamma95]. Al definir una familia de algoritmos encapsulados y hacerlos intercambiables, permite la selección del algoritmo dinámicamente.



Strategy: Declara una interfaz común para los algoritmos encapsulados. El *Context* usa esta interfaz para invocar al algoritmo definido por un *ConcreteStrategy*.

ConcreteStrategy: Implementa el algoritmo utilizando la interfaz del *Strategy*.

Context: Es configurado con un *ConcreteStrategy*. Mantiene una referencia a un objeto *Strategy*. Puede definir una interfaz que deja al *Startegy* acceder a sus datos.

Figura 3 Strategy Design Pattern

Para cada operación para la que existe una familia de algoritmos y se desea poder elegirlo dinámicamente, se aplica el patrón Strategy. En la figura siguiente, sólo se muestra para la operación de integrar.

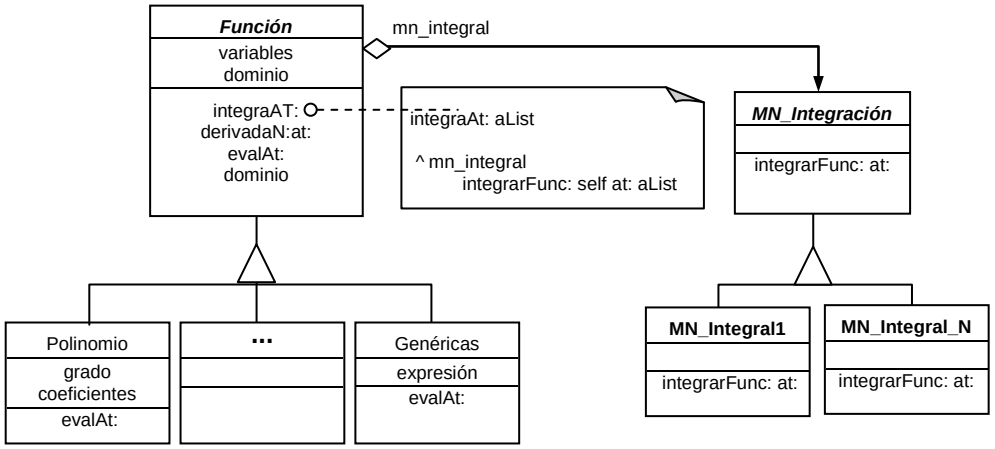


Figura 4 Función

En este modelo, la clase *MN_Integración* que representa a la clase *Strategy* del patrón, declara la interfaz común a los algoritmos soportados: *integraFunc:at:* La clase *Función* que representa a la clase *Context* del patrón, utiliza esa interfaz para invocar al algoritmo definido por un MN_Integral

concreto (*ConcreteStrategy)* y mantiene una referencia al mismo, en este caso, a través de *mn_integral*. Si en un momento resultara más conveniente resolver la integral por otro método, sólo se necesita cambiar la referencia mantenida en su atributo *mn_integral.* Cuando a una función se le solicita que se integre en un intervalo, ésta transmite la solicitud a su *mn_integral* pasándose a sí misma como parámetro:

```
integraAt: aList
"retorna el valor de su integral en el intervalo aList"

 ^ mn_integral integrarFunc: self at: aList.
```

Como consecuencia de aplicar este patrón se obtienen varios beneficios. Al crear familia de algoritmos relacionados, se puede factorizar la funcionalidad común de los algoritmos a través de la herencia. Además, al separar las funciones de los métodos numéricos, cada jerarquía puede ir evolucionando independientemente, facilitando su comprensión, mantenimiento y extensión; y el algoritmo puede ser elegido dinámicamente, simplemente cambiando el valor del atributo que lo referencia.

Funciones descriptas a través de una muestra de valores.

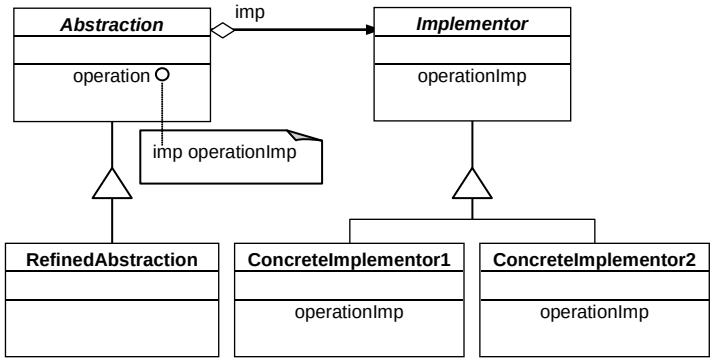
No siempre es posible representar un comportamiento funcional a través de su expresión analítica. Muchas veces sólo se cuenta con una muestra de valores funcionales en un rango del dominio y se necesita evaluar la función o su derivada en un punto que puede o no pertenecer a la muestra de puntos medidos. Los puntos pueden estar igualmente espaciados o no. Estas funciones se encuentran definidas a través de los valores que toma en un conjunto discreto de puntos del dominio, y para resolver las operaciones (evalAt:, derivadaN:At:, integrarAt:) se utilizan métodos numéricos: de interpolación, diferenciación, integración, respectivamente.

El conjunto de puntos y valores funcionales conocidos pueden tener diferentes representaciones internas. Por ejemplo, si la grilla es regular, alcanza con conocer los valores de los extremos de la muestra, la separación entre puntos y el conjunto de valores funcionales puede representarse en una lista. Esta representación no serviría para grillas no regulares, en estos casos es necesario mantener asociaciones punto - valor funcional.

Una manera de tener en cuenta las diferentes formas de representación, es a través de la herencia, donde la clase abstracta define la interfaz y cada subclase la implementa de distintas maneras. Este enfoque no es lo suficientemente flexible, ya que la herencia vincula permanentemente la abstracción con una implementación, resultando difícil extender la abstracción. Por otro lado, el problema de representación de un conjunto de mediciones, puede interesar también fuera del ámbito de las funciones.

La solución más conveniente es aplicar el patrón de diseño **Bridge** [Gamma+95] cuyo objetivo es desacoplar la abstracción de su implementación de manera que ambas clases pueden evolucionar independientemente.

El Patrón **Bridge** resuelve este problema separando la abstracción e implementación en dos jerarquías de clases distintas denominadas *Abstraction* e *Implementor*. La relación entre estas dos jerarquías es llamada *Bridge* (puente) porque vincula la abstracción y la implentación.



Abstraction define la interfaz y mantiene una referencia un objeto *Implementor*

RefinedAbstraction extiende la interfaz definida por la *Astraction*.

Implementor define la interfaz para la implementación. Esta interfaz no tiene correspondencia directa con la interfaz de la abstracción.

ConcreteImplementor implementa la interfaz del *Implementor* y define su concreta implementación.

Figura 5 Bridge Design Pattern

En nuestro caso, aplicaremos el patrón Bridge para la abstracción que representa las funciones definidas a través de un conjunto de puntos medidos, que en la jerarquía hemos llamado *Discreta.* La forma de implementar el conjunto de puntos y valores funcionales es definida en las diferentes subclases de *Representación.* Una función Discreta instanciará una Representación concreta para almacenar sus valores. Las diferentes representaciones deberán implementar los métodos definidos en la interfaz común y definir su propia implementación.

En la figura siguiente se muestra el esquema de la jerarquía de la clase *Función* utilizando el patrón **Bridge** para modelar la implementación de *Discreta* y el patrón **Strategy** para permitir elegir dinámicamente el algoritmo que implementa una operación.

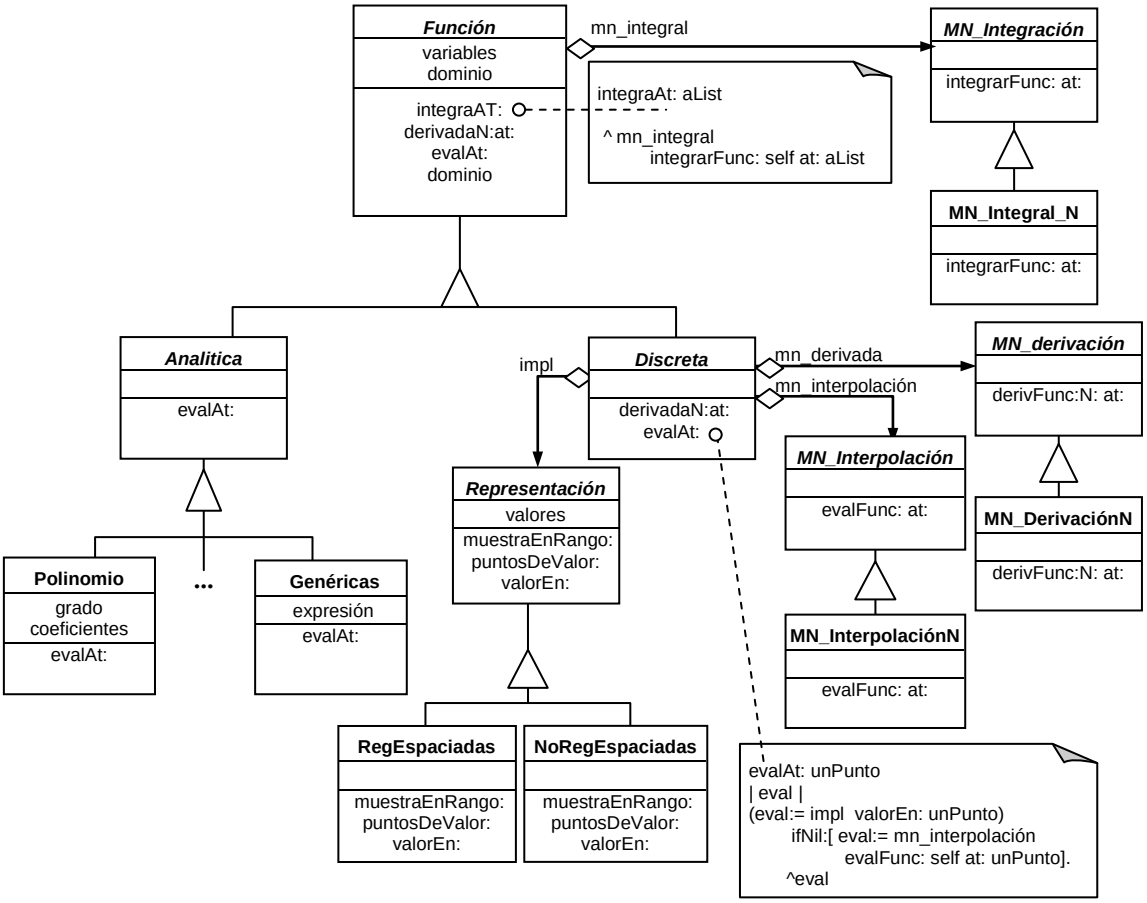


Figura 6 Jerarquía de Función

Entre las operaciones que componen la interfaz de *Representación* debieran encontrarse operaciones que permitan obtener un subconjunto de la muestra que pertenezca a un rango (muestraEnRango:), obtener el valor asociado a un punto (valorEn:), obtener el conjunto de puntos para los cuales está asociado un valor (puntosDeValor:).

Varias operaciones declaradas en *Función* serán implementadas en *Discretas* aplicando nuevamente el patrón de diseño Strategy, para permitir elegir dinámicamente el método numérico apropiado.

Matrices y Vectores

Una gran parte de problemas en Ingeniería al ser formulados mediante su modelo matemático y tratados numéricamente, requieren resolver grandes conjuntos de ecuaciones, tales como, en forma matricial:

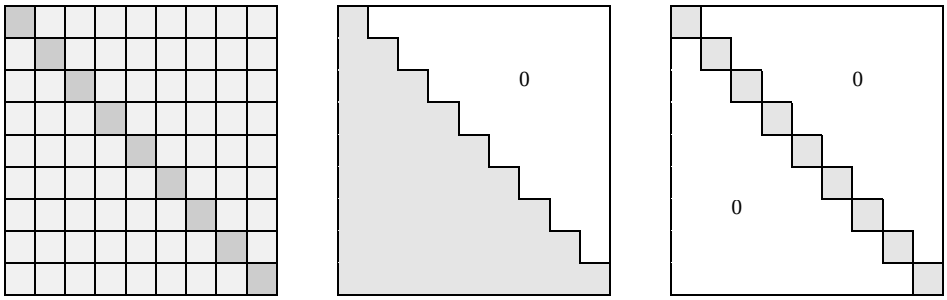
[A]{x} = {b} donde [A] representa la matriz de MxN coeficientes, {x} el vector de N incógnitas y {b} el vector de M términos independientes.

Este sistema algebraico podrá ser lineal o no, de acuerdo con el carácter físico del problema. Está formado por dos miembros en los que intervienen matrices y vectores.

Una matriz bidimensional puede pensarse como un conjunto de valores distribuidos en filas y columnas. Un vector puede ser considerado como una matriz con una sola fila, o con una sola columna. Los elementos pueden ser accedidos individualmente dando su posición.

Las principales operaciones a realizarse en estos objetos son suma, resta, transposición y multiplicación, ya que cualquier otra operación está basada en ellas. Todas ellas involucran varios accesos sobre los elementos.

Las matrices se clasifican de acuerdo a ciertas propiedades que afectan a su comportamiento en: Matriz Simétrica, Antimétrica, Matriz Diagonal, Matriz Unidad, Triangular superior, etc.



Simétrica: $m_{ij} = m_{ji}$ Triangular inferior Diagonal

Figura 7 Matrices – Clasificación teniendo en cuenta propiedades

En ingeniería, las matrices pueden ser muy pero muy grandes, por lo que es sumamente importante optimizar el espacio requerido para almacenamiento y el tiempo requerido para realizar una operación matricial.

Muy a menudo los sistemas de ecuaciones tienen matrices huecas y los elementos no nulos están situados en una banda en torno a la diagonal. De acuerdo con la forma con que se distribuyen los elementos significativos, existen diversas técnicas de almacenamiento y de operación.

En algunos casos las matrices son completas y con diferentes valores en cada posición que no permiten aplicar técnica alguna para ahorro de almacenamiento. Para otros casos, como en los que se presenta algún tipo de simetría, o son triangulares, o contienen bloques de ceros, entradas idénticas o presentan bandas no nulas, puede aprovechar esa característica para ahorrar espacio de almacenamiento.

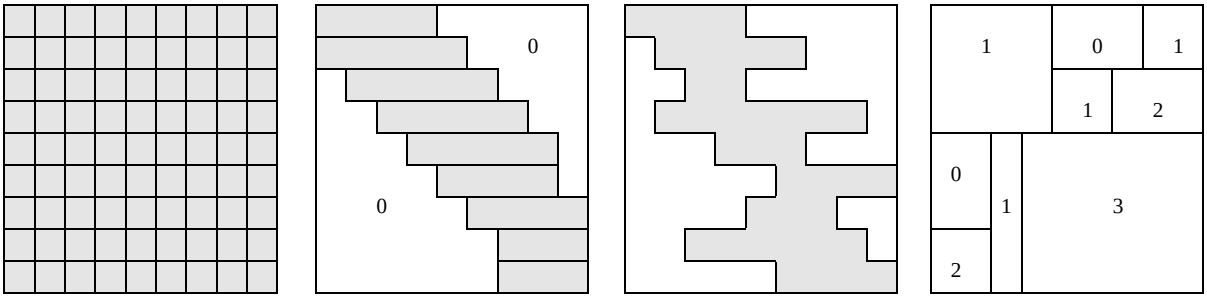


Figura 8 Matrices – Clasificación teniendo en cuenta la forma de distribución de los elementos significativos

Las diferentes formas de implementación pueden ser consideradas subclasificando otra vez la clase matriz: cada tipo de implementación da origen a una subclase, algunas de las cuales debieran replicarse en los diferentes tipos de matriz.

Este enfoque conduce a una solución poco flexible y difícil de mantener. Necesitaríamos dos subclasificaciones (una por propiedades y otra por representación) que obligarían a repetir esquemas, ya que un mismo esquema de representación puede servir para almacenar datos de distintos tipos de matrices, y un mismo tipo de matriz puede tener distintas formas de representar convenientemente sus datos. Por otro lado, el problema de representación de un conjunto de valores, puede interesar también fuera del ámbito de las matrices.

Otra vez, la solución más conveniente es aplicar el patrón de diseño **Bridge** [Gamma+95] (figura 5) desacoplando la abstracción de su implementación de manera que ambas clases pueden evolucionar independientemente. De esta manera evitamos una vinculación permanente entre la abstracción y la implementación.

En la figura siguiente, se presenta el modelado de Matriz. Una clase abstracta Matriz define el comportamiento común de sus subclases y mantiene una referencia a su representación. Las subclases tienen en cuenta las propiedades y características que distinguen algún comportamiento especial.

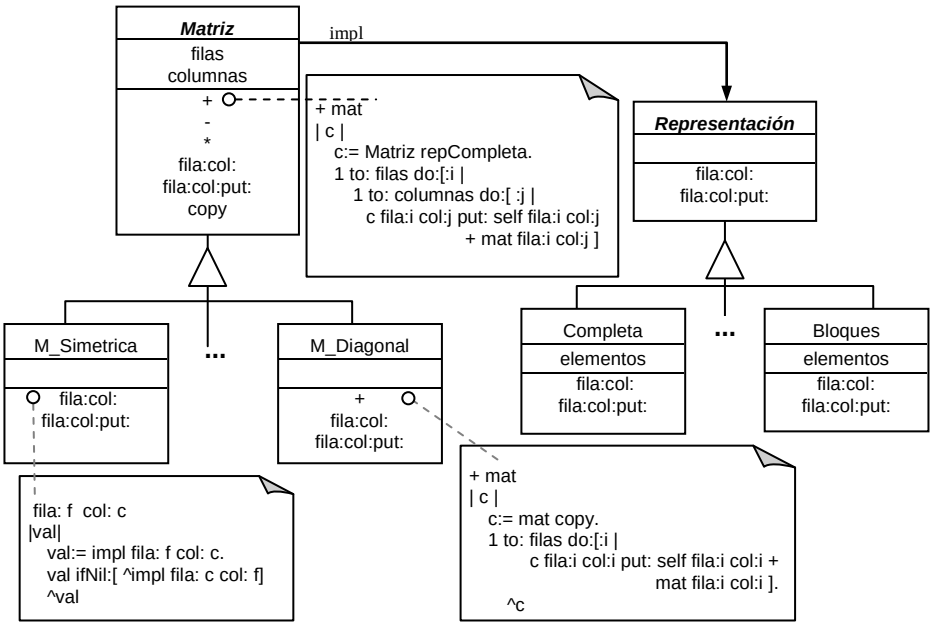


Figura 9 Clasificación de matriz

Las operaciones de acceso a un elemento, fila o columna serán implementadas en cada subclase, pues ellas conocen ciertas propiedades y si es necesario, la delegan a su implementación. Las restantes operaciones serán definidas en la superclase aplicando el patrón de diseño **Template Method** ya descripto**.** Así por ejemplo la operación suma sería definida en la superclase con la colaboración de la operación de acceso a cada elemento brindada por la implementación. Algunas subclases como Diagonal, pueden redefinirlo.

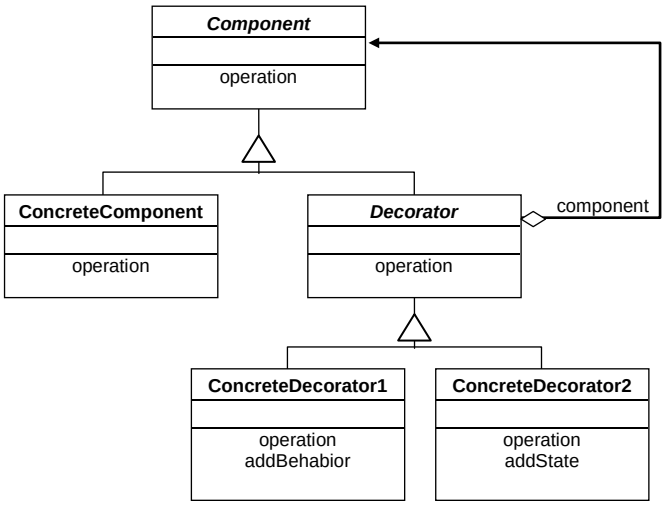
Matrices con unidades de medida

Aunque la importancia de las unidades de medida en la Ingeniería es bien conocida, no es frecuente que sean consideradas en los paquetes de software numérico, dejando la responsabilidad al usuario de utilizar cantidades en unidades consistentes.

Se podría considerarlas, permitiendo que los elementos de una matriz sean cada uno de tipo Cantidad [Fowler97]. Pero esta solución tiene dos problemas importantes: las operaciones algebraicas que requieren varios accesos a un elemento, constantemente obligarían a validar la consistencia de unidades, y como consecuencia se produce un deterioro en la perfomance. Otro problema es que no se podría utilizar funciones de bibliotecas de álgebra numérica que no trabajan con unidades.

También podríamos hacerlo, definiendo unidades para cada fila (o cada columna) o para la matriz completa, si fuera el caso. Como los elementos de una misma fila o de una misma columna de una matriz, representan el valor de una misma magnitud, todos ellos (la fila o la columna) deberán tener la misma unidad. Las verificaciones de consistencia y determinación de la unidad del resultado se harían una sola vez, antes de realizar los cálculos que involucran a la operación.

Para poder utilizar funciones de bibliotecas de álgebra numérica que no trabajan con unidades, la matriz debiera poder transformarse temporal y dinámicamente en una matriz de sólo valores. El patrón de diseño **Decorator** [Gamma+95] es especialmente adecuado para resolver este problema: permite agregar dinámicamente propiedades o comportamiento a un objeto a través de composición.



Component define la interfaz de los objetos a los cuales se le puede agregar responsabilidades dinámicamente.
ConcreteComponent define un objeto al que puede unírsele responsabilidades adicionales.
Decorator mantiene una referencia al objeto Component y define una interfaz que conforma la interfaz del componente.
ConcreteDecorator agrega responsabilidades a decorator.

Figura 10 Decorator Design Pattern

En nuestro caso, MatrizConUnits representa un ConcreteDecorator que agrega unidades a la matriz y redefine las operaciones para validar la consistencia antes de delegarla a su componente.

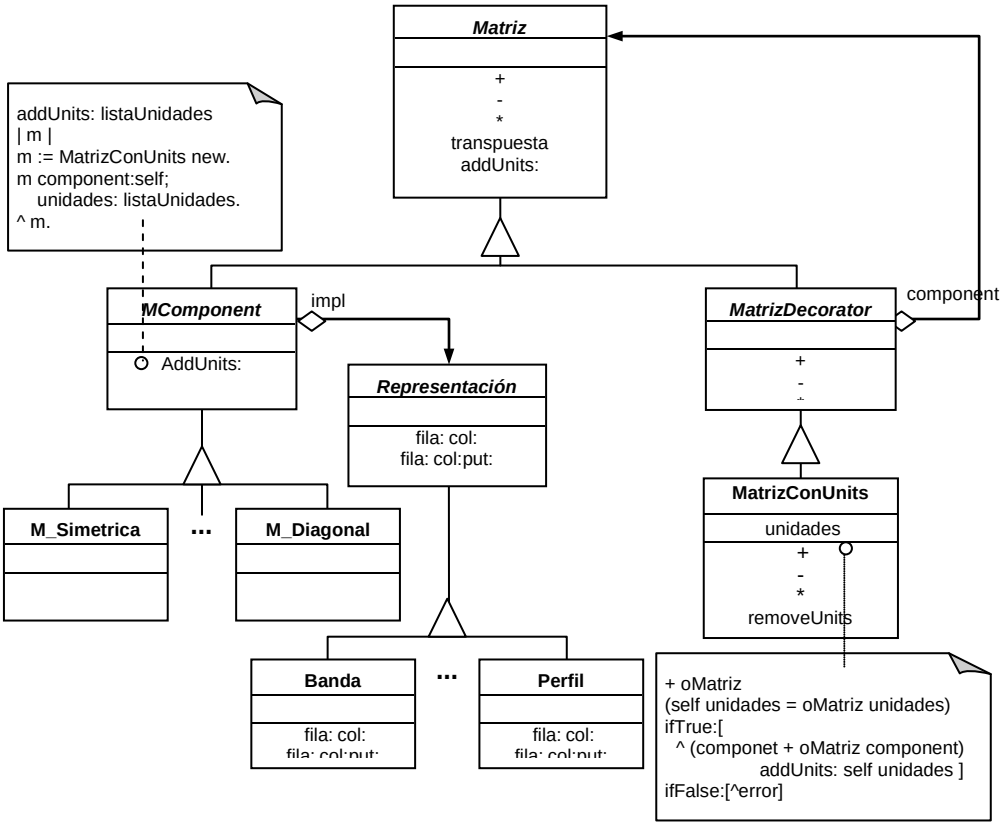


Figura 11 Matriz con unidades

Conclusiones

En este trabajo se ha presentado un modelo de diseño orientado a objetos para un uso flexible de matrices y funciones matemáticas en la solución de problemas de Ingeniería.

La complejidad natural de los problemas ingenieriles queda reducida al abstraer y modelar el comportamiento matemático por un lado, y los conceptos físicos que lo aplican, por otro. Como prueba de ello, estas abstracciones han facilitado el diseño de una familia de clases que representan los conceptos propios del dominio de la ingeniería estructural: carga, estado de carga, pieza estructural, reglamento, análisis estructural, diseño estructural [Rosanigo00]. Cada una de las clases representa el verdadero comportamiento físico, minimizando el acoplamiento y las dependencias.

Bibliografía

[Beck94] K. Beck; R. Jhonson
"Pattern Generate Architectures" presentado en ECOOP'94. 1994

[Buschmann+96] F. Bushmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal
Pattern-Oriented Software Architecture: a system of patterns. Ed. Wiley 1996.

[Coad+95] Peter Coad, David North, and Mark Mayfield
Object Models Strategies, Patterns, & Applications, Yourdon Press, 1995.

[Conde+90] C. Conde Lázaro, G. Winter Althaus
Métodos y algoritmos básicos del álgebra numérica. Ed. Reverté, 1990.

[Fowler97] Fowler, M.,
Analysis Patterns: Reusable Object Models, Addison-Wesley.-1997

[Gamma+95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.
Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley. 1995

[Johnson 96] Johnson, R., Woolf B.
"The Type Object Pattern"

[Martin+94] James Martin – James Odell
Análisis y Diseño orientado a objetos, Prentice Hall Hispanoamérica S. A. 1994

[Martin+97] James Martin – James Odell
Métodos orientados a objetos: Consideraciones prácticas. Prentice Hall Hispanoamérica S. A. 1997

[Pree95] Wolfgang Pree
Design Patterns for Object-Oriented Software Development. Ed. Addison-Wesley, 1995.

[Pressman97] Pressman, Roger S.
Ingeniería del Software - Un enfoque práctico. Cuarta edición. Ed. Mc Graw Hill. 1997.

[Rational97] "UML", www.rational.com/uml

[Rosanigo00] Rosanigo, Zulema Beatriz
"Maximizando reuso en software para Ingeniería Estructural. Modelos y Patrones"
Tesis de Magister en Ingeniería de Software bajo la dirección de Dr. Gustavo Rossi.
Facultad de Informática. UNLP. 2000.

[Rumbaugh+91] J. Rumbaugh, M. Blaha, W. Premerlani, F.Eddy, y W.Lorensen.
Object-Oriented Modeling and Design. Ed. Prentice-Hall, 1991.