Maximizando reuso en software para Ingeniería Estructural

Modelos y Patrones

Ing. Zulema Beatriz ROSANIGO

Director: Dr. Gustavo ROSSI

Tesis presentada a la Facultad de Informática de la Universidad Nacional de La Plata como parte de los requisitos para la obtención del título Magister en Ingeniería de Software.

La Plata, Febrero de 2000

Facultad de Informática
UNIVERSIDAD NACIONAL DE LA PLATA
ARGENTINA

A Silvina, Hernán, Pablo Andrés y Pedro que con toda paciencia y cariño acompañaron el desarrollo de esta tesis.

Agradecimientos

Quiero agradecer especialmente a mi director de tesis, Gustavo Rossi, por su prontitud de respuesta a mis mails, y por su ayuda y aliento a lo largo de todo este estudio.

Mi reconocimiento y gratitud para todos aquellos que me apoyaron y alentaron durante el desarrollo de esta tesis: a Alicia Paur, que además compartió discusiones y leyó con detenimiento buscando los errores que el corrector ortográfico no detectaba; a Silvia Gordillo por su constante apoyo y aliento, quien, a pesar de la distancia, cada vez que mi ánimo declinaba, con una frase vía e-mail lo levantaba; a todos mis amigos y compañeros de trabajo, quienes me escucharon y proporcionaron ánimo oportunamente.

Finalmente, mi amor y agradecimiento para mi querida familia, por tolerar mis largas tardes de trabajo, comprender mis momentos de euforia y de angustia, y estar a mi lado brindándome todo su cariño y fuerza.

RESUMEN

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.

En Ingeniería Estructural, la mayor parte de los programas existentes están escritos en lenguaje procedural, como Fortran, con miles de líneas de código y complejas estructuras de datos. Modificar o extender un componente requiere un alto grado de conocimiento del mismo y también del programa entero. El costo de mantenimiento, reuso y extensión de estos sistemas resulta muy grande y sin garantías.

Para contar con herramientas flexibles, extendibles y fáciles de modificar y mantener, es necesario diseñar aplicando sistemáticamente los principios de la ingeniería de software moderna. La tecnología orientada a objetos ha demostrado ser 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.

Siguiendo con esta idea, en este trabajo se presenta un conjunto de modelos orientados a objetos, aplicables al dominio de la Ingeniería Estructural y basado en los conceptos modernos de la Ingeniería de Software.

Los conceptos del dominio son identificados y modelados enfatizando la reusabilidad a través de aplicar sistemáticamente patrones de diseño que conducen a soluciones flexibles, extendibles y modificables, generando microarquitecturas orientadas a objetos que representan los componentes fundamentales de este dominio.

Finalmente, estas microarquitecturas son integradas en la definición de un framework de aplicaciones que abordan la problemática de la Ingeniería Estructural, minimizando la dependencia entre componentes y estableciendo mecanismos claros de comunicación y articulación.

ABSTRACT

The main challenge in the software development is to improve the quality and reduce the cost. A helpful way to fulfill this goal is maximizing the reuse and evolution possibility.

The vast majority of existing structural engineering packages are written in procedural language, usually Fortran, with thousands of code lines and complex structures of data. Modifications or extension of a component requires a high degree of knowledge of the same one, and also, of the whole program. The maintenance, reuse and extension of these systems are very expensive and without guarantees.

To have flexible and expandable software, easy to modify and maintain, it is necessary to design applying the principles of the modern software engineering, systematically. Object Oriented Technology has demonstrated to be a very powerful tool to solve problems of great span and complexity that require high degree of integrity in the information, and facilities to extension and evolution.

Continuing with this idea, this work presents a group of objected oriented models for structural engineering domain, which are based on the modern concepts of the Software Engineering.

The domain concepts are identified and modeled emphasizing the reusability through applying systematically design patterns that lead to flexible solutions, easy to modify and extend, generating objected oriented micro-architectures that represent the fundamental components of this domain.

Finally, these micro-architectures are integrated in the definition of an applications framework for Structural Engineering, minimizing the dependence among components and establishing clear mechanisms of communication and articulation.

INDICE

Capítulo 1 Introducción 1

1.1 Objetivo 2

1.2 Principales Contribuciones 2

1.3 Organización del texto 3

Capítulo 2 Tendencias de la Ingeniería de software 5

2.1 Introducción 5

2.2 Requisitos de Calidad 6

2.3 El modelo orientado a objetos 7

2.4 Beneficios de la tecnología OO 8

2.5 Medida de la complejidad 10

Capítulo 3 Conceptos sobre reutilización 11

3.1 Introducción 11

3.2 Tipos de reutilización 12

3.2.1 Reutilización de código 12

3.2.2 Reutilización de Componentes 12

3.2.3 Modelos y patrones 12

3.2.4 Frameworks 13

3.3 La práctica de la reutilización 13

3.4 Análisis de Dominio 14

Capítulo 4 Patrones 16

4.1 Qué son? 16

4.2 Cómo surgen? 16

4.3 Por qué son útiles? 17

4.4 Cualidades de un buen patrón 17

4.5 Beneficios que reportan 17

4.6 Elementos esenciales en la descripción de un patrón 18

4.7 Categorías 18

4.7.1 Patrones de análisis 18

4.7.2 Patrones arquitectónicos 19

4.7.3 Patrones de diseño 20

4.7.4 Expresiones idiomáticas 21

4.8 Catálogos 21

4.9 Patrones de diseño vs Frameworks 22

4.10 Patrones vs Algoritmos 23

Capítulo 5 Ingeniería Estructural *24*
5.1 El proceso de diseño ingenieril 24
5.2 Análisis y Diseño estructural 26
5.3 Forma estructural 26
5.3.1 Estructuras de barra 26
5.3.2 Estructuras superficiales 27
5.3.3 Tipos de Vínculos y apoyos 28
5.4 Cargas – Esfuerzos – Tensiones – Deformaciones 28
5.4.1 Tipos de cargas 29
5.4.2 Tipos de esfuerzos, tensiones y deformaciones 30
5.5 Características mecánicas y geométricas. 31
5.6 Métodos de Análisis estructural 31
5.7 Tipos de problemas 32

Capítulo 6 Microarquitecturas OO para el dominio de la I.E. *34*
6.1 Cantidades y mediciones 34
6.1.1 Magnitudes 34
6.1.2 Unidades de medida 37
6.1.3 Mediciones 40
6.1.4 Sistemas de Referencia 42
6.2 Matrices y Vectores 44
6.2.1 Matrices con unidades de medida 47
6.2.2 Sistema de Ecuaciones 49
6.3 Funciones y Métodos numéricos 49
6.4 Cargas – Estados de Cargas – Hipótesis de carga 54
6.4.1 Carga 54
6.4.2 Estado de carga 56
6.4.3 Hipótesis de carga 58
6.5 Normas y Reglamentos 59
6.6 Sistemas Discretos. Estructuras de barras 60
6.6.1 Nodos y grados de Libertad 63
6.6.2 Elemento 64
6.6.3 Pieza estructural 68
6.7 Métodos de análisis y diseño estructural 69
6.7.1 Análisis Estructural 69
6.7.2 Diseño Estructural 71

Capítulo 7 Un modelo arquitectural OO para aplicaciones en el dominio I.E. 73
7.1 El modelo arquitectónico 73
7.2 El modelo 75
7.2.1 Editor gráfico 76
7.2.2 Problema Estructural 78
7.2.3 Análisis 79
7.2.4 Diseño 80
7.3 La Vista 82
7.4 El Controlador 83
7.5 Mecanismo de propagación de cambios 84
7.6 Hacia un framework OO 85

Capítulo 8 Trabajos relacionados 87
8.1 Graham Archer 87
8.2 Jun Lu, Donald White y Wai-Fah Chen 88
8.3 Butenweg C., Ebenau C., Gajenwski R., Trierauf G. 89
8.4 J. Drolet 91
8.5 Discusión 92
8.6 Aportes de este modelo 92

Capítulo 9 Conclusiones y trabajos futuros 94
9.1 Contribuciones 94
9.2 Futuros tra bajos 95

Bibliografía 96

Anexo Notación y diagramas 99
A.1. Diagrama de clases 99
A.2. Diagrama de objetos 99
A.3. Clase 100
A.4. Objeto 101
A.5. Asociación 101
A.6. Tipos de asociaciones 102
A.7. Paquete 102
A.8. Nota 103

TABLA DE FIGURAS

Figura 4.1 Accountability ... 19
Figura 4.2 Estructura de MVC (Model-View-Controller) ... 19
Figura 4.3 Estructura de un Agente PAC ... 20
Figura 4.4 Estructura del Broker ... 20
Figura 4.5 Composite ... 21
Figura 4.6 Strategy ... 21
Figura 4.7 Clasificación de patrones de diseño[Gamma95] ... 22
Figura 5.1 Formas estructurales sometidas a tracción o flexión puras ... 26
Figura 5.2 Estructura típica de vigas y marcos ... 27
Figura 5.3 Estructuras superficiales ... 27
Figura 5.4 Tipo de Cargas de acuerdo con la forma ... 29
Figura 5.5 Tipo de Cargas de acuerdo con el tiempo de aplicación ... 29
Figura 5.6 Tipo de Cargas de acuerdo con el origen ... 30
Figura 5.7 Tipo de esfuerzos ... 30
Figura 5.8 Secuencia operacional ... 33
Figura 6.1 Cantidades escalares ... 35
Figura 6.2 Cantidades Vectoriales ... 35
Figura 6.3 Cantidades generalizadas ... 36
Figura 6.4 Unidades y Conversiones ... 38
Figura 6.5 Type Object Pattern ... 39
Figura 6.6 Composite Design Pattern ... 40
Figura 6.7 Measurement ... 41
Figura 6.8 Mediciones ... 41
Figura 6.9 Desplazamiento y ángulo entre dos sistemas de Referencia ... 42
Figura 6.10 Componentes de un vector según el sistema de coordenadas ... 43
Figura 6.11 Sistema de Referencia ... 44
Figura 6.12 Matrices – Formas de distribución de los elementos significativos ... 45
Figura 6.13 Bridge Design Pattern ... 46
Figura 6.14 Clasificación de matriz ... 46
Figura 6.15 Template Method ... 47
Figura 6.16 Decorator Design Pattern ... 48
Figura 6.17 Matriz con unidades ... 48
Figura 6.18 Sistema de Ecuaciones ... 49
Figura 6.19 Strategy Design Pattern ... 51
Figura 6.20 Función ... 51
Figura 6.21 Jerarquía de Función ... 53
Figura 6.22 Carga Estática ... 55

Figura 6.23 Carga 56
Figura 6.24 Estado de carga 57
Figura 6.25 Iterator Design Pattern 57
Figura 6.26 Hipótesis de carga 58
Figura 6.27 Range Pattern 59
Figura 6.28 Reglamento 60
Figura 6.29 Sistemas discretos 60
Figura 6.30 Barra sometida a fuerzas axiles 61
Figura 6.31 Analogía entre Red Eléctrica y Red de Tubería 62
Figura 6.32 Nodo y Grado de Libertad 63
Figura 6.33 Clasificación de Elemento 65
Figura 6.34 Elementos uni, bi y tridimensionales 66
Figura 6.35 Jerarquía de Elemento 67
Figura 6.36 Pieza Estructural 68
Figura 6.37 Análisis Estructural 70
Figura 6.38 Diseño Estructural 71
Figura 7.1 Estructura del MVC 74
Figura 7.2 Componentes del MVC 74
Figura 7.3 Componentes del Modelo 76
Figura 7.4 Objetos Gráficos 77
Figura 7.5 Adapter Design Pattern 77
Figura 7.6 Problema Estructural 78
Figura 7.7 Módulo de Análisis 79
Figura 7.8 Módulo Diseño 81
Figura 7.9 View Handler 82
Figura 7.10 Command Processor 83
Figura 7.11 Publisher-Subscriber 84
Figura 7.12 Esquema del MVC con el publisher-subscriber 84
Figura 8.1 Diagrama ER de los objetos de alto nivel en [Archer96] 87
Figura 8.2 Jerarquía de Program_Control en BUB++ 89
Figura 8.3 Clase Domain en BUB++ 90
Figura 8.4 Estructura del programa BUB++ 91
Figura A.1. Diagrama de clases 100
Figura A.2. Clase 100
Figura A.3. Objeto 101
Figura A.4. Cardinalidad 101
Figura A.5. Tipos de asociación 102
Figura A.6. Paquete 102
Figura A.7. Nota 103

Capítulo 1

Introducción

Los cambios sociológicos, los adelantos en la industria y el comercio, la economía y los avances en la tecnología de la construcción imponen una creciente responsabilidad a los diseñadores y constructores de edificios. Estos profesionales requieren cada vez más conocimientos y aptitudes para enfrentarse a las exigencias que se le presentan.

La economía exige el uso de los materiales con mayor racionalidad y los reglamentos fijan normas más estrictas sobre la seguridad de las estructuras. El ingeniero estructural necesita contar con herramientas hechas a medida, que contemple modalidades reglamentarias, tecnológicas, burocráticas y de trabajo en obra, y que además, sean fácilmente adaptables a nuevos requerimientos y a las exigencias que caracterizan la época actual.

Los programas existentes de cálculo de estructuras son, en su mayoría, paquetes monolíticos. Consisten en cientos de líneas de código procedural, comúnmente escritos en FORTRAN, con complejas estructuras de datos, las cuales son accedidas a través del programa. Resulta difícil modificar y extender el código existente para adaptarlo a nuevos usos, modelos y soluciones. La integridad de las estructuras de datos no está asegurada, y varias rutinas del programa a menudo las acceden directamente. Modificar o extender un componente requiere un alto conocimiento del mismo y también del programa entero, de modo que resulta difícil asegurar que una modificación o adaptación local no esté afectando a otras partes del programa. El costo de mantenimiento, reuso y extensión de estos sistemas resulta muy grande y sin garantías.

Mejorar la calidad y reducir el costo de las soluciones basadas en computadoras es el principal desafío en el desarrollo de software. Una manera de ayudar a cumplir con este objetivo es maximizar el reuso y posibilidad de evolución. El reuso produce reducción de tiempo y costo e incrementos de calidad, en la medida que el desarrollador pueda encontrar, utilizar y adaptar al nuevo contexto, aquellas soluciones que ya han sido probadas y usadas exitosamente. Se puede hacer reuso de un dominio del problema, de una clase, de una componente, de un diseño, de mecanismos, de una idea o de un patrón.

Un patrón es una idea que ha sido útil en un contexto y probablemente lo sea en otros. 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.

Los patrones ayudan a aliviar la complejidad del software en varias fases en el ciclo de vida. Capturan ideas y soluciones obtenidas por experiencia y las hacen accesible para los desarrolladores, analistas e ingenieros menos expertos o principiantes.

Las etapas de análisis y diseño son fundamentales en el desarrollo de software. En la fase de análisis se estudia y especifica el dominio del problema dentro del contexto de objetivos y metas preestablecidas, buscando comprender los detalles y complejidades concernientes al problema para ir hacia un modelo mental de lo que se quiere alcanzar. A menudo encontramos que muchos aspectos de un problema en un dominio ya han aparecido en otros proyectos diferentes. Abstraer el modelo conceptual y aplicarlos al problema en cuestión es el desafío. De esto tratan los patrones.

Para contar con herramientas flexibles, manejables, extendibles, fáciles de modificar y mantener, es necesario diseñar aplicando sistemáticamente los principios de la ingeniería de software moderna. La tecnología orientada a objetos ha demostrado ser 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.

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

1.1 Objetivo

El propósito de esta tesis es analizar el dominio de la Ingeniería Estructural, de manera de definir microarquitecturas y modelos de diseño basados en objetos, para facilitar el desarrollo de aplicaciones ingenieriles que sean manejables, extendibles, fáciles de modificar, mantener y reusar.

1.2 Principales Contribuciones

Las principales contribuciones de este trabajo son:

- Caracterización de los objetos fundamentales del dominio de la IE
- Aplicación de los principios de la Ingeniería de Software y de la Tecnología de Objetos, en su concepción actual, al campo de la IE.

- Definición de un conjunto de microarquitecturas que modelan los principales objetos del dominio en las que se han aplicado sistemáticamente patrones de diseño que conducen a soluciones flexibles y modificables.

- Integración de todos esos objetos en la definición de un framework de aplicaciones para el dominio estructural.

- Definición de una arquitectura OO para aplicaciones que abordan la problemática de la Ingeniería Estructural, y descripción de sus componentes principales y los mecanismos de comunicación.

1.3 Organización del texto

El resto del texto se organiza de la siguiente manera:

En el capítulo 2 se describen los principales factores que determinan la calidad del software y se introducen los principios y conceptos básicos en que se basa el modelo orientado a objetos y los beneficios que esta tecnología aporta para producir software de alta calidad y confiabilidad.

El capítulo 3 discute conceptos sobre la reutilización: beneficios que reporta, tipos de reutilización, características del proceso, importancia del análisis de dominio.

El capítulo 4 describe los patrones de software en sus diferentes categorías y los beneficios que reportan para la construcción de software flexible y reusable. Algunos ejemplos de patrones de análisis de [Fowler 97], de diseño de [Gamma+95] y patrones arquitectónicos de [Buschamann+96] son utilizados para ilustrar las diferentes categorías.

El capítulo 5 introduce los conceptos del dominio de la Ingeniería Estructural, describiendo las características más relevantes y aquellas que tendrán mayor impacto en la toma de decisiones de diseño.

El capítulo 6 analiza y modela conceptos del dominio. Los tópicos discutidos en los capítulos 2 a 5 intervienen en forma conjunta para definir esquemas de modelado de los conceptos del dominio, enfatizando la reusabilidad a través de aplicar sistemáticamente patrones de diseño que conducen a soluciones flexibles y modificables. Como resultado se obtienen microarquitecturas orientadas a objetos que representan los componentes fundamentales del dominio de la IE. Los conceptos están agrupados en categorías que van desde las más genéricas a las más específicas del dominio: cantidades y mediciones; vectores y matrices; funciones y métodos numéricos; cargas y estados de cargas; normas y reglamentos; sistemas discretos; análisis y diseño estructural.

El capítulo 7 define una arquitectura para aplicaciones del dominio estructural basada en las nuevas tendencias de diseño conducido por patrones y en la que se integra a las microarquitecturas modeladas en el capítulo anterior, minimizando la dependencia entre componentes.

En el capítulo 8 se describen algunos trabajos relacionados y se discuten diferencias entre ellos y la aproximación descripta en esta tesis.

El capítulo 9 presenta las conclusiones y futuros trabajos.

Finalmente en el anexo se describe el método de modelado y la notación gráfica utilizada para la representación de las abstracciones de software presentadas en esta tesis.

Capítulo 2

Tendencias de la Ingeniería de software

2.1 Introducción

Una de las preocupaciones actuales más urgentes de la industria del software es crear sistemas confiables y de mayor calidad con menor inversión de tiempo y costo, que resuelvan problemas cada vez más complejos. Es preciso utilizar técnicas avanzadas de la ingeniería de software que ayuden a aliviar el esfuerzo en las diferentes etapas del ciclo de vida.

Tal como lo manifiestan J. Martin y J. Odell [Martin 94], en el software se necesita un avance en:

- Complejidad
- Capacidad de diseño
- Flexibilidad
- Rapidez de desarrollo
- Facilidad de modificación
- Confiabilidad

La Tecnología Orientada a Objetos ha demostrado ser una excelente herramienta para resolver problemas de gran envergadura y complejidad, permitiendo obtener sistemas interoperables, modulares, evolutivos y con alto índice de reusabilidad. La reutilización conduce a un desarrollo más rápido y programas de mejor calidad.

Las técnicas orientadas a objetos combinadas con otras herramientas como las CASE (ingeniería de software asistida por computadora), programación visual, generadores de código, metodologías basadas en depósitos, bases de datos, bibliotecas de clases que maximicen la reutilización, tecnología cliente servidor, etc.; pueden proporcionar la magnitud del cambio necesario para lograr ese salto anteriormente mencionado.

En este capítulo se describen los principales factores que determinan la calidad del software y se introducen los principios y conceptos básicos en que se basa el modelo orientado a objetos y los beneficios que esta tecnología aporta para producir software de alta calidad y confiabilidad.

2.2 Requisitos de Calidad

La calidad del software la definimos como la concordancia con los requisitos funcionales y cualitativos explícitamente establecidos y con los estándares de desarrollo explícitamente documentados. Es una compleja mezcla de factores que varían a través de diferentes aplicaciones, algunos de los cuales pueden medirse directamente (cantidad de errores por unidad) y otros indirectamente (facilidad de uso o de mantenimiento).

McCall y colegas [Pressman 97] proponen una categorización de factores según tres aspectos importantes del producto de software: características operativas, capacidad de cambio y adaptabilidad a nuevos entornos, los cuales son mostrados en el siguiente cuadro.

Factor		Descripción	Pregunta
Características Operativas	Correctitud	Grado en que el programa satisface sus especificaciones	Hace lo que quiero?
	Fiabilidad	Grado en que se puede esperar que un programa lleve a cabo sus funciones	Lo hace de forma fiable todo el tiempo?
	Eficiencia	Cantidad de recursos requeridos para llevar a cabo sus funciones	Se ejecuta lo mejor que puede?
	Integridad	Grado en que puede controlarse el acceso al software o a los datos por personal no autorizado	Es seguro?
	Facilidad de uso	Esfuerzo requerido para aprender un programa, trabajar con él, preparar la entrada e interpretar la salida	Puedo usarlo sin dificultades?
Capacidad de cambio	Facilidad de mantenimiento	Esfuerzo requerido para arreglar un error	Puedo corregirlo?
	Flexibilidad	Esfuerzo requerido para modificarlo	Puedo cambiarlo?
	Facilidad de prueba	Esfuerzo requerido para probar un programa de forma que asegure que realiza su función	Puedo probarlo?
Adaptabilidad	Portabilidad	Esfuerzo requerido para transferir el programa de un sistema a otro	Podré usarlo en otra máquina?
	Reusabilidad	Grado en que un programa o partes de él se pueden usar en otras aplicaciones	Podré reusar alguna parte?
	Facilidad de interoperación	Esfuerzo requerido para acoplar un sistema a otro	Podré hacerlo interactuar con otro sistema?

2.3 El modelo orientado a objetos

El modelo orientado a objetos (OO) ve a la realidad como un conjunto de objetos cooperantes y eventos que activan operaciones, las cuales modifican el estado de los objetos.

Durante muchos años el término OO se utilizó para distinguir un enfoque de desarrollo de software que usaba un lenguaje de programación OO. Hoy en día el paradigma OO encierra una visión completa de la ingeniería de software.

Un **objeto** es una entidad que representa un concepto real o abstracto, que tiene un conjunto de responsabilidades y que encapsula un estado interno. Un objeto es instancia de una *clase* particular.

Una **clase** es una implementación de un tipo de objeto. Especifica una estructura de datos y los métodos operativos permisibles que se aplican a cada uno de los objetos. El concepto de **herencia** permite organizar las clases en jerarquías de sub y superclases, por especialización y generalización respectivamente. Algunas clases (llamadas *clases abstractas*) no están previstas para tener instancias sino para capturar común comportamiento y favorecer el *reuso* a través de herencia. Los objetos heredan todo su comportamiento y estructura interna de la clase a la que pertenece y de todas las superclases de la jerarquía.

En este modelo, todas las acciones son causadas por la llegada de **mensajes** a un objeto, y estos responden ejecutando un **método**. La forma en que objetos diferentes responden a un mismo mensaje puede variar (esto se conoce como *polimorfismo*). Los atributos de un objeto sólo se pueden consultar y modificar mediante mensajes específicos. Los atributos mantienen el estado de un objeto y están contenidos en sus variables de instancia, las cuales pueden referenciar a otros objetos. El comportamiento está definido por los métodos los cuales aplican transformaciones a las variables de instancia.

Los objetos pueden ser complejos desde el punto de vista interno pero basta con conocer el comportamiento y su forma de actuar, para poder ser utilizados. Un objeto puede estar compuesto por otros objetos, y formar de esta manera, objetos complejos con un mínimo esfuerzo.

La visión OO demanda un enfoque evolutivo de la Ingeniería de Software, el proceso se mueve a través de una espiral evolutiva que comienza con la comunicación con el usuario, en el cual se define el dominio del problema y se identifican las clases básicas del problema con sus atributos y comportamiento relevante. El software OO evoluciona iterativamente y debe dirigirse teniendo en cuenta que el producto final se desarrollará a partir de una serie de incrementos.

La ingeniería de software OO hace hincapié en la reutilización. Por lo tanto las clases se buscan en una biblioteca de clases existentes antes de construirse, y si no se encuentra, el desarrollador aplica para crearla, análisis orientado a objetos (AOO), diseño orientado a objetos (DOO), programación orientada a objetos (POO) y pruebas orientadas a objetos (PrOO). La nueva clase se coloca en la biblioteca de manera que pueda ser reutilizada en el futuro. Aún, cuando una clase se necesita crear, maximizar el reuso sigue siendo un principio a respetar, el cual se logra aprovechando la herencia directa o indirectamente, para lo cual son muy importantes las actividades de abstracción, generalización y composición.

El análisis OO puede subdividirse en análisis de la estructura de objetos en el cual se definen las categorías de los objetos que percibimos y la forma en que los asociamos; y en análisis del comportamiento de objetos en el que se identifica los estados del objeto, los tipos de eventos, las reglas de activación, las condiciones de control y las funciones.

Existen varios métodos de análisis y de diseño orientados a objetos, cada uno de los cuales introduce un proceso, un conjunto de modelos y una notación que posibilita al ingeniero de software crear cada modelo de una manera consistente. El análisis y discusión de estos métodos y sus diferencias está fuera del alcance de esta tesis.

2.4 Beneficios de la tecnología OO

Tres conceptos importantes diferencian el enfoque OO de la ingeniería de software convencional. El encapsulamiento empaqueta datos y operaciones que los manejan. La herencia permite que los atributos y operaciones de una clase sean heredados por todas las subclases y objetos que se instancien de ella. El polimorfismo permite que una operación pueda implementarse de maneras diferentes y conserve el mismo nombre.

Estas características propias de la tecnología OO, sumadas a una disciplina de desarrollo que las potencie y maximice el reuso, permiten desarrollar aplicaciones complejas y de alta calidad con menor inversión de tiempo y esfuerzo.

Los principales beneficios que la tecnología OO aporta son:

- **Reducción de la brecha semántica entre el mundo real y el modelo**: El mundo está formado por objetos. Desde una etapa temprana categorizamos los objetos y descubrimos su comportamiento. Los usuarios finales y los especialistas del dominio piensan de manera natural en término de objetos, eventos y mecanismos de activación. El análisis se traduce de manera directa en el diseño y la implementación.

- **Diseño más rápido:** Muchas aplicaciones se crean a partir de componentes ya existentes, probados y verificados, lo cual acelera el proceso de diseño y reduce la posibilidad de falla. Los componentes se pueden adaptar para un diseño particular.

- **Mantenimiento más sencillo:** Las modificaciones no tienen impacto global. El encapsulado separa muy bien el qué del cómo. Cada clase efectúa funciones independientemente de las demás. Las clases son como cajas negras y sólo dentro de la misma clase se puede ver su interior.

- **Independencia de la plataforma:** Las clases están diseñadas para ser independientes del ambiente de plataformas, hardware y software al utilizar solicitudes y respuestas con formato estándar.

- **Mayor modularidad y extensibilidad:** Los programas se forman a partir de piezas pequeñas, cada una de las cuales se puede crear fácilmente. El encapsulamiento y la comunicación vía mensajes contribuye a generar pequeños módulos. Se construyen clases a partir de otras clases, permitiendo construir componentes complejos de software que a su vez intervienen en otros bloques más complejos. La herencia y el polimorfismo facilitan la extensión de las clases.

- **Integridad:** Las estructuras de datos sólo se pueden utilizar con métodos específicos. El objeto esconde sus datos de los demás objetos y permite el acceso mediante sus propios métodos. El encapsulado evita la corrupción de los datos de un objeto y los protege del uso arbitrario y no pretendido.

- **Reusabilidad:** La generación de componentes reusables (clases) es una consecuencia natural del paradigma. Las clases están diseñadas para que se reutilicen en muchos sistemas. La reusabilidad proporciona beneficios inmediatos y sumamente valiosos en la calidad del producto y en la productividad de los procesos.

- **Mayor calidad:** Los diseños suelen tener mayor calidad puesto que se integran a partir de componentes probados y verificados, y porque la mayoría de los factores que determinan la calidad de un producto de software (fiabilidad, integridad, reusabilidad, flexibilidad, facilidad de mantenimiento) son naturalmente encontrados en los productos desarrollados con tecnologías OO.

2.5 Medida de la complejidad

La complejidad de un programa es una medida de su comprensibilidad: cuanto más complejo es, más difícil será de comprender.

La complejidad es función de la cantidad de posibles rutas de ejecución de un programa y de la dificultad de rastreo de esas rutas. Evitar la complejidad excesiva mejora su confiabilidad y reduce el esfuerzo requerido para su desarrollo y mantenimiento.

La métrica de complejidad más ampliamente usada para el software es la complejidad ciclomática McCabe [Pressman97], la cual proporciona una medida cuantitativa para probar la dificultad y una indicación de la fiabilidad última. Existe una fuerte correlación entre la métrica de McCabe y el número de errores que existen en el código fuente, así como el tiempo requerido para encontrarlos y corregirlos.

El diseño y la programación orientadas a objetos dan como resultado menores métricas McCabe, debido principalmente a que cada método es relativamente sencillo y autocontenido.

Para enfrentar y administrar la complejidad inherente a nuestro mundo de objetos, 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.

Estos mecanismos son parte del patrimonio humano y están presentes en las actividades mentales que nos dan la capacidad de percibir y controlar la complejidad del mundo.

La tecnología OO incita a aprovechar plenamente estos mecanismos, para lograr mayor flexibilidad y reusabilidad de sus componentes, contribuyendo también a una reducción de la métrica McCabe.

Capítulo 3

Conceptos sobre reutilización

3.1 Introducción

Las organizaciones necesitan aumentar la productividad y reducir los costos. Esto es válido para cualquier tipo de producción y en particular para la construcción de software. Una forma de lograrlo es mediante la reutilización sistemática en todas y cada una de las etapas del desarrollo y ciclo de vida: incluye las bibliotecas (de programas, clases, componentes), los marcos estructurales, el análisis de dominio, la administración de proyectos y las arquitecturas distribuidas. La práctica de la reutilización es muy común en otras ramas de la ingeniería.

La *reutilización* de una idea, proceso o artefacto se da cuando es utilizado en un contexto diferente para el que originalmente fue diseñado. Si el contexto y las condiciones se mantienen, simplemente se lo estaría *utilizando* nuevamente. Para incrementar las posibilidades de reuso, hay que hacer que la idea, proceso o artefacto sea más general. ¿Es posible?. Sí, de hecho, existen formas bien conocidas de incrementar la generalidad hasta un nivel muy elevado. Al incrementar la generalidad puede disminuir la eficiencia y aumentar la complejidad, lo que constituye una desventaja que habrá que comparar con factores como la posible mejora de fiabilidad que se produciría como consecuencia de utilizar un componente conocido y verificado y con los ahorros de tiempo y costo resultantes de la reutilización [Pressman 97].

Los programadores han reutilizado ideas, algoritmos, abstracciones, argumentos y procesos desde los primeros días de la computación pero en una forma *ad hoc*. En la actualidad, es preciso un enfoque más organizado de la reutilización para poder construir sistemas complejos y de alta calidad en períodos de tiempo muy breves.

Para una reutilización a gran escala, es necesario contar con bibliotecas de componentes para reutilización compartida, herramientas automatizadas para explorarlas y mecanismos que permitan buscar el requisito indicado en la especificación actual entre los descriptos para los componentes reutilizables ya existentes. Con la reutilización sistemática se logra mejoras en la calidad, en la productividad y en los costos en general.

3.2 Tipos de reutilización

Entre los artefactos reutilizables, además del código fuente, podemos mencionar: diseños, planes, datos de comprobación, documentación, interfaces.

La reutilización de código es la que produce menores ganancias de productividad y fiabilidad, se alcanzan mayores beneficios al reutilizar diseños, modelos, componentes, frameworks.

3.2.1 Reutilización de código

Es la forma más utilizada y menos metódica, conocida también como el método de copiar y pegar. Segmentos de código desarrollados en una aplicación son incorporados en otra después de ser adaptados.

3.2.2 Reutilización de Componentes

La reutilización de componentes requiere la inclusión de un producto de desarrollo dentro de la aplicación. Los componentes reutilizables son elementos independientes de una especificación o de una implementación.

Un componente de software puede ser una estructura de datos, un módulo, una abstracción, una unidad funcional. En gran medida el futuro del software depende de los componentes reutilizables. Los componentes reutilizables encapsulan tanto datos como procesos que se aplican a los datos. Pueden ser componentes de *caja negra* o de *caja blanca*. Los primeros se utilizan sin conocer ni modificar su implementación; los segundos pueden ser examinados y modificados.

Para que un componente pueda ser conocido y localizado para su reutilización, es necesario por un lado, brindar una descripción, y por otro, contar con algún tipo de depósito o biblioteca con mecanismos para la clasificación, gestión y recuperación. La descripción debiera abarcar el *concepto*, es decir comunicar la intención del componente, el *contenido* (en el caso de ser de caja blanca) para describir la forma en que se construye el concepto, y el *contexto* para situar el componente en el seno de su dominio de aplicabilidad.

3.2.3 Modelos y patrones

Un modelo representa un conjunto coherente de conceptos de un problema, es un conjunto de tipos de objetos, tipos de relación y de otros constructores que transmiten una calidad particular al dominio [Martin 97].

Un patrón es una forma o estructura identificable y reconocible en diferentes dominios. Un patrón brinda una solución a un problema recurrente, identificando los subsistemas y componentes y los mecanismos de colaboración entre ellos. En el capítulo siguiente se desarrolla en forma más completa los patrones de software.

La reutilización de modelos y patrones es aplicable en las diferentes etapas del ciclo de vida del software facilitando la construcción de componentes complejos.

3.2.4 Frameworks

En el nivel más alto de granularidad aparece la reutilización de un diseño completo o framework.

Un framework da una solución para la descripción y construcción de aplicaciones. Un framework de especificación describe algún tema o problema, como por ejemplo, CORBA que define un enfoque de sistemas distribuidos. Un framework de aplicación es un conjunto de componentes de implementación que da una base sobre la cual construir una aplicación. El desarrollador de la aplicación lo perfecciona mediante especialización y lo extiende agregando nuevos componentes.

Un framework es una arquitectura reusable que provee la estructura y comportamiento genérico para una familia de abstracciones de software, junto con un contexto que especifica su colaboración y su uso dentro de un dominio dado. Es una clase de "máquina virtual".

Un framework OO es un diseño arquitectónico de un tipo particular de aplicación o dominio [Johnson 92]. Consiste de un conjunto de clases abstractas y concretas, y define la interacción entre los componentes mediante el planteo de restricciones, herencia, polimorfismo y reglas informales de composición.

3.3 La práctica de la reutilización

Los beneficios de la reutilización pueden resumirse en mejoras en la calidad del producto, en la productividad del desarrollador y en los costos en general. Sin embargo, siguen existiendo dificultades para la práctica sistemática de la reutilización. Los beneficios suelen verse opacados por problemas organizativos, por una falta de verdadera comprensión de la naturaleza de la reutilización del software, y por una estrategia débil para impulsar e implementar la tecnología de reutilización.

Los desarrolladores se oponen al empleo de componentes por diversas razones como:

- Dificultad de localizar componentes apropiados
- El costo de elaborar componentes reutilizables
- La falta de confianza en otros desarrolladores

Para poder practicar la reutilización, una organización deberá contar con una infraestructura básica: asignación de personal, definición de papeles y de responsabilidades, informes y mecanismos para compartir los componentes y adquisición de herramientas y bibliotecas que contribuyan de la forma más positiva a la reutilización. También es necesario que tenga establecido un plan interno de reutilización de software, con control de calidad y costos y que siga y mida tanto la reutilización como el impacto de la misma.

La disciplina de reutilización abarca tanto al productor como al consumidor de un componente.

Las técnicas de análisis y diseño de componentes reutilizables se basan en los mismos principios y conceptos que forman parte de las buenas prácticas de ingeniería de software: abstracción, ocultamiento, independencia funcional, refinamiento. El productor además de aplicar los métodos de diseño para crearlo, deberá tener especial cuidado en la selección de la interfaz del componente, en la definición de parámetros de implementación, en el grado de generalidad a alcanzar. Los componentes diseñados en un entorno que establece estructuras de datos estándar, protocolos de interfaz y arquitecturas de programas para todos los dominios de aplicación, tienen más posibilidades de ser reusado.

El productor, una vez que ha creado y probado el componente, deberá hacerlo conocer para que pueda ser usado, esto es, catalogarlo, proporcionar documentación, colocarlo en un depósito de componentes reutilizables.

Las actividades del consumidor pueden resumirse en encontrar, comprender, modificar e integrar el componente. Idealmente, se analiza el modelo de análisis para determinar aquellos elementos que indican artefactos reutilizables ya existentes. Se utilizan herramientas automatizadas para explorar el depósito en un intento de encontrar el requisito indicado en la especificación actual entre los descriptos para los componentes reutilizables ya existentes. Si existe un componente que se ajusta a las necesidades de la aplicación en cuestión, se lo utiliza directamente integrándolo en la aplicación. A veces es necesario realizar modificaciones tanto en la nueva aplicación como en el componente. Las modificaciones locales del componente debieran realizarse por especialización o extensión para abarcar alguna característica adicional.

3.4 Análisis de Dominio

La comprensión del dominio es un prerrequisito para la reutilización. El análisis de dominio es el estudio de la naturaleza fundamental de un dominio. Se sirve de las mismas técnicas de abstracción, composición, generalización y especialización que se emplea en el análisis OO, pero no trata aplicaciones en particular. Como resultado, los modelos de dominio son independientes de preocupaciones de

implementación, describen con claridad el problema tanto para desarrolladores como para el experto del dominio.

En el análisis de dominio se seleccionan y abstraen las funciones y objetos específicos, se identifican las características comunes y las relaciones fundamentales, se categorizan los elementos extraídos, se buscan las trama repetidas en los problemas que se presentan en ese dominio.

Un modelo de dominio sirve como base para diseñar y construir objetos del dominio. Cuando un negocio, sistema o producto de dominio es definido como estratégico a largo plazo, puede desarrollarse un esfuerzo continuado para crear una biblioteca reusable robusta. El objetivo es ser capaz de crear software dentro del dominio con un alto porcentaje de elementos reusables [Pressman 97].

Capítulo 4

Patrones

4.1 Qué son?

Un patrón es una idea que ha sido útil en un contexto y probablemente lo sea en otros. 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.

Un patrón involucra una descripción general de una solución recurrente a un problema recurrente con diversos objetivos y restricciones. Pero un patrón identifica algo más que simplemente una solución, también explica por qué la solución se necesita.

Un patrón orientado a objetos es una abstracción formada por algunas clases, que resulta ser útil una y otra vez en desarrollos de Objetos.

4.2 Cómo surgen?

El movimiento de patrones de diseño comenzó con el trabajo del arquitecto Christopher Alexander en el año 1970. En 1977 Alexander publicó el libro *A Pattern Language*, seguido en 1979 por *The Timeless Way of Building*. Ambos libros presentaron un modo de representar conocimiento de arquitectura como una serie de problemas, junto con las restricciones, una solución al problema y un patrón o ejemplo que puede ser seguido para resolver el problema.

Estas ideas fueron tomadas y aplicadas al software y tuvieron una amplia aceptación con la aparición del libro “Design Patterns: Elements of Reusable Object-Oriented Software” de Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides (referidos como the Gang of Four o simplemente GoF) [Gamma95]. A partir de aquí los patrones de diseño se hicieron populares y son utilizados en diferentes dominios.

Los patrones no son privativos del diseño de software. Existen en diferentes disciplinas. Y dentro del área del software, aparecen en diferentes enfoques: patrones de análisis, patrones de arquitectura, patrones de expresiones idiomáticas, patrones de diseño. Tampoco están confinados a la comunidad de orientación a objetos, existen para el modelado de datos, para bases de datos relacionales y otros.

4.3 Por qué son útiles?

Ayudan a aliviar la complejidad del software en varias fases en el ciclo de vida. En las fases de análisis y diseño, pueden servir de guía en la selección de arquitecturas de software que han probado ser exitosas. En las fases de implementación y mantenimiento, ayudan a documentar las propiedades estratégicas de sistemas de software en un nivel de abstracción más alto que el código fuente. Capturan ideas y soluciones obtenidas por experiencia y las hace accesible para los desarrolladores, analistas e ingenieros menos expertos o principiantes.

4.4 Cualidades de un buen patrón

Encapsulamiento y Abstracción: Cada patrón encapsula un problema bien definido y su solución en un dominio particular. También abstrae conocimiento y experiencia del dominio.

Equilibrio Tiene que comprender algún tipo de balance entre sus metas y sus restricciones para minimizar el conflicto dentro del espacio de la solución.

Apertura y Variabilidad: Cada patrón puede trabajar junto con otros para resolver un problema mayor y en una variedad de implementaciones.

Capacidad de generación y de composición Un patrón, una vez aplicado, genera un contexto resultante que puede coincidir con el contexto inicial de otros patrones. Por aplicación de estos patrones posteriores se progresa hacia el objetivo final para generar un "todo" o solución global completa. Aplicando un patrón se provee un contexto para la aplicación del siguiente patrón. En un nivel particular de abstracción y granularidad puede estar compuesto con otros patrones.

4.5 Beneficios que reportan

- Capturan conocimiento y experiencia y los hace accesible para los desarrolladores, analistas e ingenieros menos expertos o principiantes. Los diseñadores expertos usualmente toman buenas decisiones y encuentran buenas soluciones, estas soluciones son documentadas en forma de patrones, explicitando el problema y las razones de cada decisión. Cuando el mismo problema aparece en otra aplicación, podemos comprender claramente las razones para adoptar una solución en particular.

- Con sus nombres se crea un vocabulario que ayuda a los desarrolladores a comunicarse mejor. Proveen un vocabulario común, poderoso y conciso, permitiendo mayor grado de abstracción.

- Los sistemas documentados con los patrones que utiliza, son más fáciles de comprender. Los patrones son un medio ideal para documentar qué problema están resolviendo, la forma en que lo hacen y los motivos de esa solución.

- Facilitan la reestructuración de un sistema, haya sido o no, diseñado con patrones. Posibilitan un vasto reuso de arquitecturas de software

4.6 Elementos esenciales en la descripción de un patrón

Aunque existen múltiples formatos de presentación de un patrón, como mínimo debiera contener ciertos componentes esenciales:

- **Nombre:** Tiene que tener un nombre significativo. Sería muy incontrolable tener que describir el patrón cada vez lo utilizamos en una discusión.

- **Intención:** Descripción sucinta de cuál es el problema que resuelve.

- **Descripción del problema:** Describe cuándo aplicarlo, explica el problema y su contexto y la lista de precondiciones que deben encontrarse, si las hubiera.

- **Solución:** Describe los elementos que lo componen: clases, objetos, relaciones, responsabilidades y colaboraciones.

- **Consecuencias:** Describe los costos y beneficios de aplicarlo. Incluye el impacto sobre la flexibilidad, extensibilidad y portabilidad del sistema.

4.7 Categorías

Los patrones cubren diferentes rangos de escala de abstracción. Algunos ayudan a estructurar un sistema en subsistemas, otros a refinar estos subsistemas y sus relaciones, otros solucionan una implementación particular.

4.7.1 Patrones de análisis

Reflejan estructuras conceptuales de procesos de negocios más que implementaciones de software [Fowler97].

Ejemplos:

Party: Dado que las personas, los puestos y las organizaciones tienen responsabilidades similares, una solución de modelado es crear un tipo Party como supertipo de ellas. Esto facilita el modelado de estructuras de organizaciones y personas más complejas. Un tipo Party puede estar representando a una persona, una organización, una empresa o en ciertos casos, un puesto.

Accountability: El concepto de Accountability se presenta cuando una persona u organización es responsable sobre otra. Para representar las relaciones entre dos partes se crea el tipo *Accountability* y el *Accountability Type* para expresar la clase de relación.

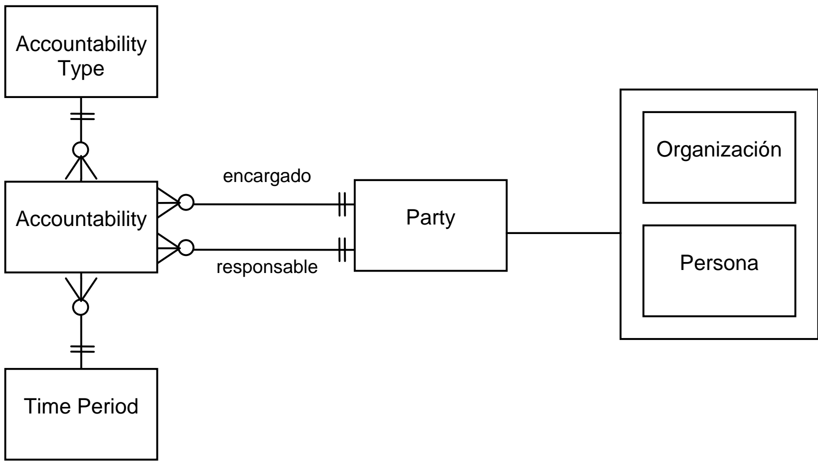


Figura 4.1 Accountability

4.7.2 Patrones arquitectónicos

Describen los principios fundamentales de la arquitectura de un sistema de software. Identifica los subsistemas, define sus responsabilidades y establece las reglas y guías para organizar las relaciones entre ellos.

Ejemplos: [Buschmann96]

MVC (Model-View-Controller): divide una aplicación interactiva en tres componentes: el **Modelo** que contiene la funcionalidad y los datos, la **Vista** que despliega la información al usuario y el **Controlador** que maneja la entrada y coordina la actividad de la vista.

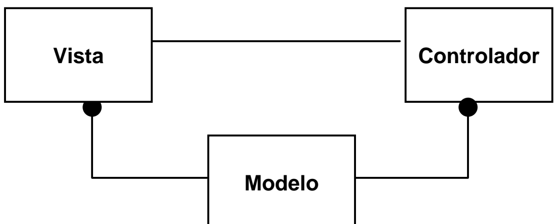


Figura 4.2 Estructura de MVC (Model-View-Controller)

PAC (Presentation-Abstraction-Control): define una estructura para software interactivo como una jerarquía de agentes cooperantes. Cada agente es responsable de un aspecto específico de funcionalidad de la aplicación y consiste de tres componentes: la **Presentación**, la **Abstracción**, y el **Control**. Esta subdivisión separa la interacción hombre-máquina de la funcionalidad y de la comunicación con otros agentes.

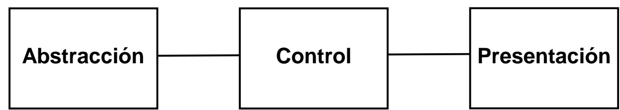


Figura 4.3 Estructura de un Agente PAC

Broker: puede ser usado en sistemas distribuidos: con clientes que interactúan por invocaciones a un servidor remoto con bajo nivel de acoplamiento. Un **broker** o corredor es responsable para coordinar la comunicación entre **cliente** y **server**.

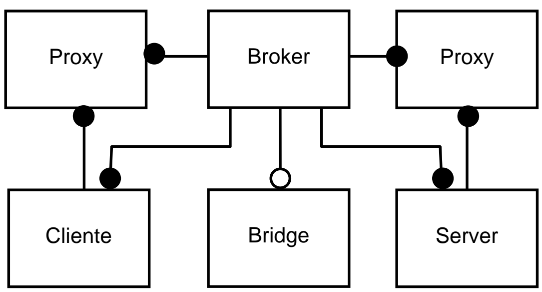


Figura 4.4 Estructura del Broker

4.7.3 Patrones de diseño

Los patrones de diseño proveen un esquema para refinar los subsistemas y las relaciones entre los componentes de un sistema. Describen una estructura recurrente de comunicación entre componentes para resolver un problema general de diseño dentro de un contexto particular. Se encuentran en un nivel intermedio de granularidad entre los patrones arquitectónicos y las expresiones idiomáticas. Tienden a operar independientemente de un paradigma particular de programación o lenguaje de programación.

Ejemplos: [Gamma95]

Singleton: Asegura que una clase tenga una única instancia y provee un punto global de acceso a ella, este ejemplar único debe ser fácilmente accesible por muchos otros objetos.

Composite: Compone objetos en una estructura de árbol para representar la jerarquía de las partes y el todo. Permite tratar en forma uniforme a los objetos simples (*Leaf*) y compuestos (*Composite*), siendo la superclase de ambos *Component*. Los clientes utilizan la interface del *Component* para interactuar con los objetos de la estructura del patrón composite sin necesidad de realizar una diferencia entre objetos simples y compuestos. Si el receptor es una hoja el requerimiento es manejado directamente, si es un objeto compuesto, el composite delega el requerimiento a sus componentes hijos.

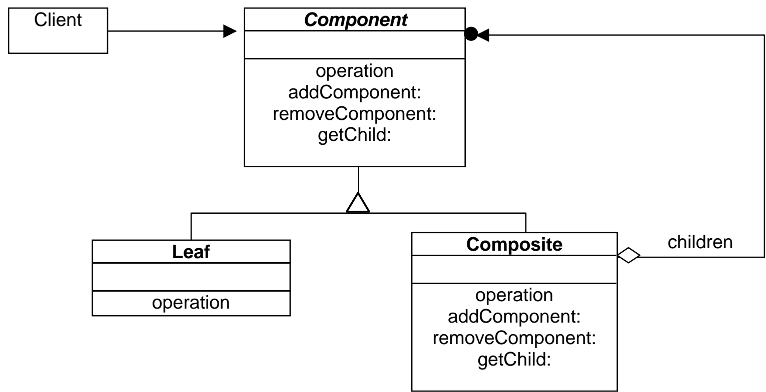


Figura 4.5 Composite

Strategy: Define una familia de algoritmos, encapsula a cada uno y los hace intercambiables. Permite que el algoritmo varíe independientemente de los clientes que lo usan. Una clase *Strategy* define el algoritmo que encapsula y una clase *Context* delega el requerimiento desde su cliente a su *Strategy*.

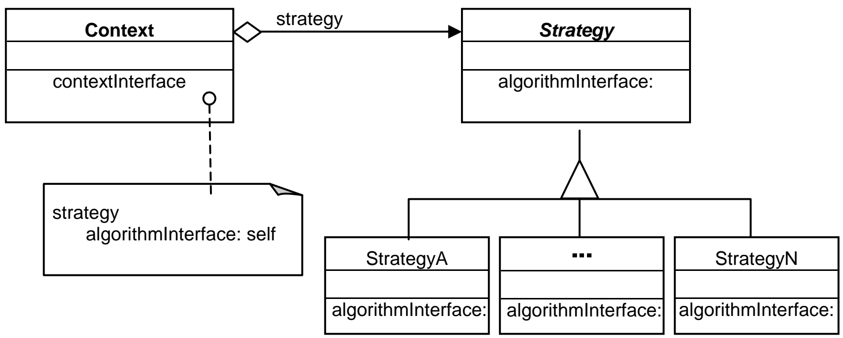


Figura 4.6 Strategy

4.7.4 Expresiones idiomáticas

Los modismos o expresiones idiomáticas son el nivel más bajo de abstracción en un sistema de patrones. La mayoría de los modismos son específicos de un lenguaje. Describen como implementar una parte particular de diseño al utilizar un lenguaje de programación determinado.

4.8 Catálogos

Un catálogo de patrones es una colección de patrones relacionados que cubren dominios y disciplinas particulares con variación de concurrencia, distribución, diseño organizacional, reuso de software, sistemas de tiempo real, negocio y comercio electrónico, y diseño de interfaces humana. Estos catálogos presentan una colección de soluciones relativamente independientes a problemas de diseño comunes, agrupándolos en categorías.

Los patrones se agrupan de acuerdo a dos criterios fundamentales: 1) categoría del problema o funcionalidad que los patrones soportan y 2) categoría de los patrones (arquitectónicos, diseño, idiomas)

Propósito / *Alcance*	CREACIONAL	ESTRUCTURAL	COMPORTAMIENTO
CLASE	• Factory Method	• Adapter (clase) • Bridge (clase)	• Template Method
OBJETO	• Abstract Factory • Prototype • Singleton	• Adapter (objeto) • Bridge (objeto) • Facade • Flyweight • Proxy	• Chain of Responsability • Command • Iterator (objeto) • Mediator • Memento • Observer • State • Strategy
COMPUESTOS	• Builder	• Composite • Decorator	• Interpreter • Iterator (componente) • Visitor

Figura 4.7 Clasificación de patrones de diseño[Gamma95]

Los catálogos de patrones nos ayudan para ubicar los patrones que pueden utilizarse en nuestros proyectos. Para ello, uno busca en la intención y motivación que inspiró a cada patrón para encontrar uno o más que sean relevantes para nuestro problema, y de esa preselección, estudiarlos plenamente, y seleccionar el más adecuado.

4.9 Patrones de diseño vs Frameworks

Tanto los patrones como los frameworks son sumamente útiles en la resolución de problemas de diseño y tienen objetivos semejantes, pero existen diferencias importantes entre ellos:

- Los frameworks siempre modelan un dominio particular, mientras los patrones de diseño generalmente pueden utilizarse en distintos dominios;
- Los patrones de diseño pueden emplearse tanto en el diseño como en la documentación de un framework. Estos patrones pueden ser vistos como descripciones abstractas de frameworks que facilitan el reuso de arquitectura de software. Igualmente, los frameworks pueden ser vistos como realizaciones concretas de patrones que facilitan el reuso de diseño y código.

- Los patrones actúan como bloques o microarquitecturas para la construcción de diseños más complejos. Los frameworks son macroarquitecturas que generalmente contienen varios patrones.

- Un framework es software ejecutable, mientras que un patrón de diseño representa conocimiento y experiencia acerca de software, sólo un ejemplo de patrón puede codificarse. En lo que a esto respecta, los frameworks son de una naturaleza física, mientras que los patrones son de una naturaleza lógica: Los frameworks son la comprensión física de una o más soluciones de patrones de software.

- Los patrones están descriptos en un lenguaje independiente, mientras que los frameworks están instrumentados generalmente en un lenguaje particular.

4.10 Patrones vs Algoritmos

Tanto los algoritmos como los patrones dan solución a un problema en un contexto y se los identifica con un nombre, pero no son lo mismo y ambos son necesarios.

Los algoritmos y estructuras de datos resuelven generalmente problemas computacionales de granularidad más fina, son más determinísticos y con menor variación entre sus estrategias y sus tácticas de instrumentación que los patrones.

Los algoritmos y estructuras de datos pueden ser empleados en la instrumentación de uno o más patrones.

Los ingenieros de software requieren encontrar tanto arquitecturas como soluciones apropiadas a problemas computacionales. Así siempre habrá una necesidad de patrones como de algoritmos y estructuras de datos y de su uso conjunto.

Capítulo 5

Ingeniería Estructural

La *Ingeniería Estructural (IE)* se centra en lo relacionado con la concepción, diseño, y construcción de los sistemas estructurales que se necesitan para apoyar las actividades humanas. Aunque la Ingeniería Estructural está más directamente asociada con la Ingeniería Civil, interviene en cualquier disciplina ingenieril que requiere un sistema estructural o una componente estructural, como un paso integral hacia el cumplimiento de sus objetivos. Proyectos específicos que involucran Ingeniería Estructural incluyen puentes, edificios, presas, medios de transporte, almacenamiento de líquidos o gases y medios de transmisión, generación y transmisión de unidades de potencia, plantas de tratamiento de agua potable y de alcantarillado, fabricas industriales y plantas, estructuras vehiculares, y componentes de máquinas. Cada uno de estos proyectos requiere sistemas o componentes estructurales que deben concebirse para satisfacer las necesidades para las que han sido construidos, diseñados para soportar en forma segura y útil las cargas que chocaran contra ellas, y construidos para proporcionar un producto final consistente con la concepción y diseño.

El cálculo de estructuras tiene por objeto el estudio de la estabilidad y resistencia de las construcciones, de manera que bajo las acciones que aquellas deben soportar, tanto las fuerzas internas llamadas tensiones, como las deformaciones que se presentan, queden dentro de ciertos límites establecidos.

5.1 El proceso de diseño ingenieril

El *proceso de diseño ingenieril* abarca las siguiente etapas:

Etapa conceptual. Cualquier proyecto ingenieril debe estar dirigido hacia la satisfacción de un conjunto único de objetivos. Durante la etapa de concepción o planificación, se identifican las necesidades y los objetivos se articulan cuidadosamente para satisfacer a éstas. La etapa conceptual debería sacar a luz un plan que maximice la satisfacción de los objetivos establecidos, mientras que minimice cualquier característica objetable del proyecto.

Etapa preliminar de diseño. El plan que emerge a partir de la etapa conceptual incluye frecuentemente varias alternativas que van a investigarse a través de la preparación de diseños preliminares individuales.

Cada diseño preliminar involucra una consideración completa de la forma estructural que se va a usar, la manera cómo están conectados los componentes, las cargas y acciones que tendrá que soportar la estructura, incluyendo las condicio-

nes que ocurrirán durante su fabricación. Para cada caso es necesario un análisis estructural, es decir, deben determinarse las fuerzas y las deformaciones en toda la estructura. Frecuentemente, los diseños preliminares se basan en teorías aproximadas de análisis estructural para minimizar el tiempo y el esfuerzo invertidos en la fase preliminar.

Esta fase debe producir suficientes detalles de modo que puedan hacerse decisiones inteligentes en la selección final de una de las alternativas que se propusieron en la etapa conceptual.

Etapa de selección. Una vez que están terminados los diseños preliminares correspondientes a las alternativas consideradas, debe hacerse una selección. En este punto se convocan las partes involucradas en la etapa conceptual de modo que puedan participar en el proceso de selección. La principal consideración se centra en cómo cada alternativa satisface los objetivos establecidos originalmente. Otra vez, se da una debida consideración a cualquier característica objetable de las alternativas.

En esta etapa al ingeniero estructural le conciernen las economías relativas de las alternativas, el impacto que pudiera tener cualquier característica única de cada alternativa sobre el comportamiento estructural o lo práctico de la construcción, y cualquier otra área relacionada con la decisión.

El resultado de esta etapa es usualmente una decisión para proseguir con una de las alternativas para las cuales se ha preparado un diseño preliminar. En algunos casos se encargan nuevos diseños preliminares, lo que implica un regreso a la etapa preliminar de diseño. Sin embargo, antes de proceder a la siguiente etapa, es necesario hacer una selección entre las alternativas.

Etapa final de diseño. Los resultados de la etapa preliminar de diseño constituyen un punto de partida para la etapa final de diseño; sin embargo, el ingeniero estructural debe proceder con mayor cuidado a partir de este punto. Aquí las cargas se determinan con mayor exactitud de la que fue necesaria durante el diseño preliminar, y todas las posibles circunstancias y combinaciones de carga deben considerarse. El análisis estructural que se requiere para esta etapa debe llevarse a cabo con gran precisión, y deben moderarse las condiciones de las suposiciones restrictivas de la etapa preliminar de diseño. Se proporciona cada miembro y se detallan las conexiones para asegurar que la estructura se comportará de acuerdo con las hipótesis hechas en el análisis estructural.

Etapa de construcción. La meta de esta etapa es dar vida a lo que se describió en la etapa final de diseño. Los documentos completos de la etapa final de diseño sirven como base de concurso para los contratistas de construcción en prospecto.

5.2 Análisis y Diseño estructural

El análisis y diseño estructural requiere la aplicación del criterio del ingeniero para producir un sistema estructural que satisfaga de manera adecuada las necesidades del cliente. A continuación, este sistema se incorpora a un modelo matemático para obtener las fuerzas en los miembros. Como el modelo matemático no representa con exactitud la estructura real, basándose en las propiedades de los materiales, la función estructural, las consideraciones ambientales y estéticas, se efectúan modificaciones en el modelo, y se repiten los procesos de resolución hasta obtener una solución que produce un equilibrio satisfactorio entre la selección del material, la economía, las necesidades del cliente y diversas consideraciones arquitectónicas.

5.3 Forma estructural

La forma que va a tener una estructura depende de muchas consideraciones, tales como: requisitos funcionales, requisitos estéticos, limitaciones económicas, disponibilidad de materiales, condiciones de la cimentación.

La mayoría de las estructuras está compuesta de una combinación de formas estructurales, cada subestructura sirve de una manera única y el conjunto satisface los objetivos funcionales requeridos [West 84].

5.3.1 Estructuras de barra

Estructuras a tracción y compresión pura: Muchas veces se busca que los miembros de una estructura estén sometidos a tensión o compresión pura para un uso sumamente eficiente del material.

Las formas más comunes para soportar compresión pura son el *arco* y la *columna*. Esta última aparece en casi todas las formas estructurales. Una forma estructural común en la que se combinan elementos sometidos a tracción y compresión pura son los *reticulados* o *armadura articulada*. Las estructuras verdaderamente articuladas son raras pero bajo ciertas restricciones en el tamaño de los miembros y conexiones y posiciones de las cargas, pueden ser consideradas como tales.

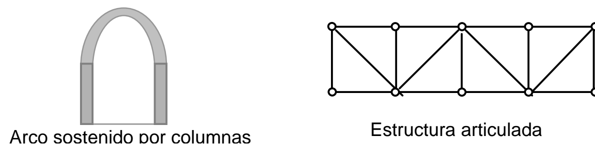
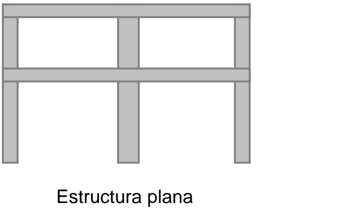


Figura 5.1 Formas estructurales sometidas a tracción o flexión puras

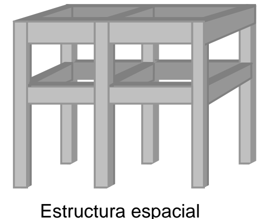
Una de las formas estructurales más simples en que los elementos están sometidos a tensión pura es la *estructura sostenida por cables,* como el Puente colgante Golden Gate en San Francisco, California, o el sistema de techo colgante para Madison Square Garden en Nueva York.

Estructuras a flexión: Lograr que la estructura esté sometida sólo a tensiones de compresión y tracción pura, no siempre es posible, y entonces aparecen los esfuerzos de flexión. La flexión induce compresión en un lado del elemento y tracción en el otro.

El miembro estructural más simple sometido a flexión es la *viga*. Los *marcos a flexión* están formados de una combinación de vigas y columnas, algunos de los cuales están sometidas a flexión pura y otros combinan flexión con tracción o compresión. Se diferencia de los reticulados por tener conexiones resistentes a la flexión que aseguran continuidad en los extremos del miembro. Las estructuras de este tipo son las más comunes y aparecen en edificios, en marcos de puentes, etc.



Estructura plana



Estructura espacial

Figura 5.2 Estructura típica de vigas y marcos

5.3.2 Estructuras superficiales

Las estructuras superficiales obtienen su configuración espacial a través de superficies tridimensionales continuas, y las cargas están resistidas mediante las superficies de las mismas mientras que las anteriormente descriptas forman una estructura esquelética en el espacio que sirve para soportarse a sí misma y todo lo que está unido a ella.

A esta categoría pertenecen las *placas plegadas, membranas inflables, cascarones.*

Placa plegada

Cascarón semicircular

Figura 5.3 Estructuras superficiales

5.3.3 Tipos de Vínculos y apoyos

Existen diferentes maneras de vincular dos elementos estructurales entre sí y de apoyar la estructura, de acuerdo con el grado de libertad de movimiento que se restringe, y en función de ello, varía el comportamiento de la estructura.

Ejemplos: articulación, apoyo móvil, apoyo fijo, empotramiento, apoyo elástico, etc.

5.4 Cargas – Esfuerzos – Tensiones – Deformaciones

El establecimiento de las cargas que actúan en una estructura es uno de los pasos más difícil y más importante del proceso total de diseño.

Las **cargas** son fuerzas y momentos externos que actúan sobre una estructura, incluyendo los desplazamientos impuestos en los vínculos. Los **esfuerzos** son las fuerzas internas que resisten las cargas. **Tensión** es la relación entre el esfuerzo y el área sobre la que está aplicada. En respuesta a las fuerzas que actúan sobre la estructura, ésta experimenta **deformaciones,** los que se manifiestan como un conjunto de **desplazamientos**. Cada desplazamiento es una traslación o una rotación de algún punto particular de la estructura.

El diseño final de una estructura debe ser consistente con la combinación de cargas más crítica que la estructura va a soportar. Las combinaciones de carga que deben considerarse son especificadas mediante los reglamentos que rigen la construcción, y en algunos casos el proyectista ejerce su propio criterio. En estas combinaciones se tiene en cuenta la probabilidad de concurrencia de determinados fenómenos, generando diversas hipótesis de combinación de cargas. Para cada hipótesis se calculan los esfuerzos y deformaciones en cada miembro de la estructura y se dimensionan para soportar la peor de las combinaciones.

5.4.1 Tipos de cargas

Las cargas externas aplicadas a una estructura se clasifican de diferentes maneras. A continuación se explican aquellas clasificaciones que interesan por su comportamiento especial en el cálculo de estructuras.

a) por su forma:

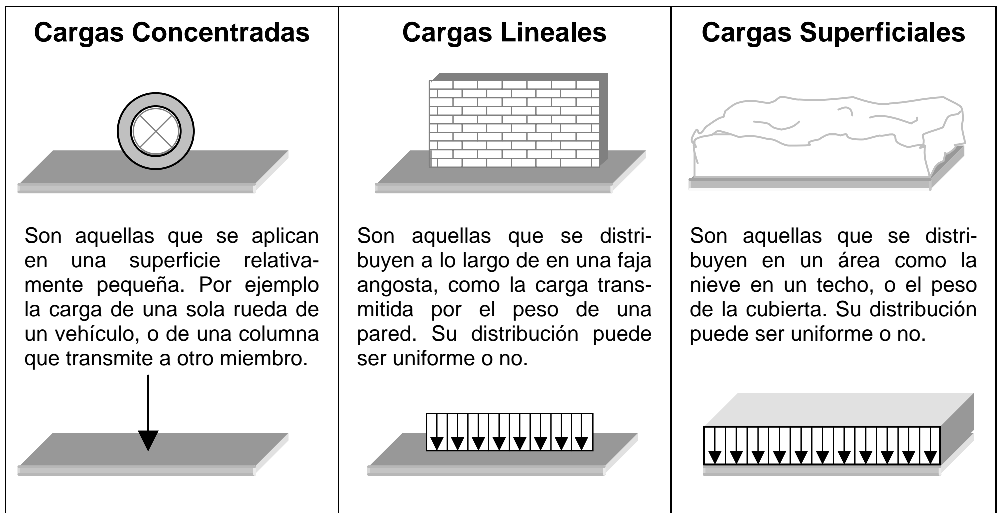


Figura 5.4 Tipo de Cargas de acuerdo con la forma

b) por el tiempo de aplicación:

Cargas Estáticas	**Cargas dinámicas**:
Son fuerzas aplicadas lentamente y que luego permanecen casi constantes. Un ejemplo es el peso de un entrepiso.	Varían con el tiempo. En ellas se incluyen las de índole repetitiva y las de impacto. **Cargas repetitivas:** son fuerzas aplicadas cierto número de veces, de modo que ocasionan una variación en la magnitud de las fuerzas internas. Un buen ejemplo es un puente grúa. **Cargas de impacto:** son fuerzas que obligan a la estructura o a sus componentes a absorber cierta cantidad de energía en un corto tiempo. Un ejemplo es la caída de un gran peso sobre la losa de un entrepiso; otro son las ondas de choque de una explosión al golpear los muros y techos de un edificio.

Figura 5.5 Tipo de Cargas de acuerdo con el tiempo de aplicación

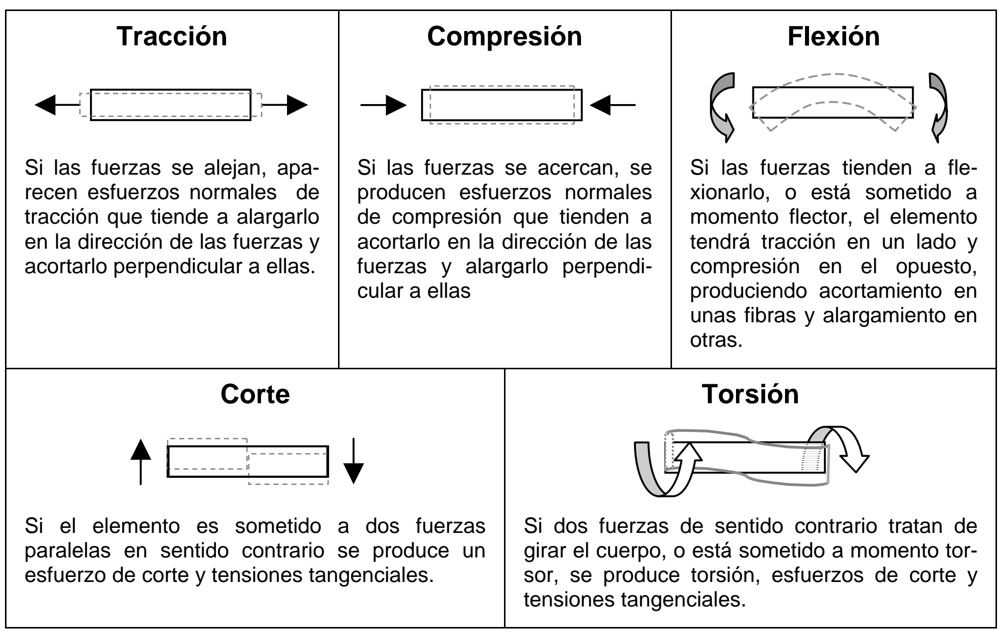
c) por su origen:

Cargas Muertas	Cargas Vivas	Cargas por viento (eólicas)	Cargas por nieve	Cargas sísmicas
Son las fuerzas debidas a los materiales, equi-pos, estructuras y otros elementos pesados que se apoyan sobre el edificio,incluyendo su propio peso, y que están de modo permanente en un sitio.	Son las fuerzas debidas a los ocupantes, materiales, equipos y estructuras apoyados sobre el edificio y que se mueven o pueden ser movidas o reubicadas durante el transcurso de la vida útil del inmueble.	Son las fuerzas máximas que el viento puede aplicar a un edificio dentro de un intervalo promedio de recurrencia.	Son las fuerzas aplicadas a un inmueble por la máxima acumulación de nieve ocurrida en un intervalo promedio de recurrencia.	Son las fuerzas que generan los máximos esfuerzos o deformaciones en un edificio durante un terremoto.

Figura 5.6 Tipo de Cargas de acuerdo con el origen

5.4.2 Tipos de esfuerzos, tensiones y deformaciones

Según el modo en que se apliquen las cargas tienden a deformar la estructura y sus componentes: las fuerzas de tracción los estiran, las de compresión los aplastan, las de torsión los retuercen y las de cortante hacen que algunas partes de la estructura se deslicen respecto de otras. Los esfuerzos normales producen tensiones normales de tracción o de compresión. Los esfuerzos de corte y el momento torsor producen tensiones tangenciales. El momento flector produce tensiones normales de tracción y de compresión.



Tracción	Compresión	Flexión
Si las fuerzas se alejan, aparecen esfuerzos normales de tracción que tiende a alargarlo en la dirección de las fuerzas y acortarlo perpendicular a ellas.	Si las fuerzas se acercan, se producen esfuerzos normales de compresión que tienden a acortarlo en la dirección de las fuerzas y alargarlo perpendicular a ellas	Si las fuerzas tienden a flexionarlo, o está sometido a momento flector, el elemento tendrá tracción en un lado y compresión en el opuesto, produciendo acortamiento en unas fibras y alargamiento en otras.

Corte	Torsión
Si el elemento es sometido a dos fuerzas paralelas en sentido contrario se produce un esfuerzo de corte y tensiones tangenciales.	Si dos fuerzas de sentido contrario tratan de girar el cuerpo, o está sometido a momento torsor, se produce torsión, esfuerzos de corte y tensiones tangenciales.

Figura 5.7 Tipo de esfuerzos

5.5 Características mecánicas y geométricas.

El comportamiento de una estructura depende en gran medida de la forma y material con que está hecho cada uno de sus miembros. Existen materiales más aptos que otros para soportar determinados esfuerzos. La elección del material para la estructura obedece a distintos intereses, pero cualquiera sea el material elegido, debe conocerse determinadas mediciones llamadas **características mecánicas del material**, las cuales se obtienen mediante ensayos normalizados. Las principales son: Módulo de Elasticidad Longitudinal (**E**), Módulo de Elasticidad Transversal (**G**), peso específico (γ), Coeficiente de dilatación térmica (α), Resistencia a ruptura (σ_r).

De la forma y dimensiones de la pieza estructural interesan determinados valores conocidos como **características geométricas**, tales como: Inercia de torsión (**It**), Inercia alrededor del eje local y (**Iy**), Inercia alrededor del eje local z (**Iz**), Sección transversal (**Ax**). Todas estas pueden ser calculadas a partir de las dimensiones y forma del elemento.

Los materiales más utilizados en estructuras de obras civiles son el hormigón (armado, pretensado), el acero (conformado, laminado, armado) y la madera. Cada uno de estos materiales están clasificados en categorías o tipos de acuerdo con su resistencia a rotura, y las características mecánicas pueden ser consultadas en tablas.

Coeficiente de rigidez: Es un valor que depende de características geométricas y mecánicas y que representa la fuerza que se asocia con un desplazamiento unitario. Para una barra de sección **A**, longitud **l** y módulo de elasticidad longitudinal del material **E**, sometida a esfuerzo de tracción, el coeficiente de rigidez es **E A / l**

Coeficiente de flexibilidad: Es un valor que depende de características geométricas y mecánicas y que representa el desplazamiento que se asocia con una fuerza unitaria. Para una barra de sección **A**, longitud **l** y módulo de elasticidad longitudinal del material **E**, el coeficiente de flexibilidad es **l /E A.**

5.6 Métodos de Análisis estructural

El análisis estructural se desarrolló históricamente a lo largo de dos caminos diferentes [Tuma74]:

- La mecánica vectorial, basada en los principios del equilibrio estático y de la geometría de deformación.
- La mecánica del trabajo virtual, basada en el principio de la conservación de la energía.

Los métodos de cálculo de estructuras de barras más potentes actuales utilizan técnicas de análisis matricial, planteando el “equilibrio” de las diferentes barras que la componen. Pero una gran parte de las estructuras en ingeniería son de naturaleza continua, y por lo tanto su comportamiento no puede expresarse en forma precisa en función de un número pequeño de variables discretas [Oñate95]. Aunque las estructuras continuas son inherentemente tridimensionales, en algunos casos es posible describirlas adecuadamente por modelos matemáticos uni o bidimensionales.

El **método de los elementos finitos** es el procedimiento más potente para el análisis de estructuras de carácter uni, bi o tridimensional sometidas a las acciones exteriores más diversas, existiendo una gran analogía entre este método y el del análisis matricial [Oñate95].

El método de los elementos finitos consiste en subdividir previamente a la estructura en un número de subregiones llamadas “elementos finitos” donde la forma geométrica y la función estructural son simples. Dentro de cada uno de estos elementos se hace una aproximación sobre uno o más campos (campos de desplazamiento, de tensión, de deformación) reduciendo a un número pequeño de nodos, descriptos por funciones simples. La malla de elementos finitos puede estar constituida por elementos de diferente geometría. La estructura entera es idealizada por varios de estos elementos cuya unión posterior restituirá el campo total. La solución completa se obtiene combinando esas distribuciones de tensiones o de desplazamientos de manera de satisfacer equilibrio de fuerzas y compatibilidad de desplazamientos en las uniones de esos elementos.

Las operaciones algebraicas son representadas en forma matricial y dentro de los métodos matriciales se puede tomar indistintamente los desplazamientos o las fuerzas como incógnitas: A partir de la expresión del Principio de los Trabajos Virtuales (PTV) se obtienen las matrices de rigidez $\mathbf{K}^{(e)}$ y el vector de cargas $\mathbf{f}^{(e)}$ (o de desplazamientos $\mathbf{a}^{(e)}$) para cada elemento; luego se procede al ensamblaje de las matrices para obtener la matriz de rigidez global $\mathbf{K}$ de toda la malla de elementos finitos y el vector de cargas $\mathbf{f}$ (o de desplazamientos $\mathbf{a}$) y finalmente se resuelve el sistema de ecuaciones $\mathbf{K}\,\mathbf{a} = \mathbf{f}$ para calcular las variables incógnitas $\mathbf{a}$ (o $\mathbf{f}$) utilizando algún método de solución de ecuaciones algebraicas simultáneas lineales.

5.7 Tipos de problemas

Los problemas que se presentan en la Ingeniería Estructural son básicamente de dos clases: *análisis estructural* y *diseño estructural*, ambos están relacionados entre sí y con una secuencia como la ilustrada a continuación.

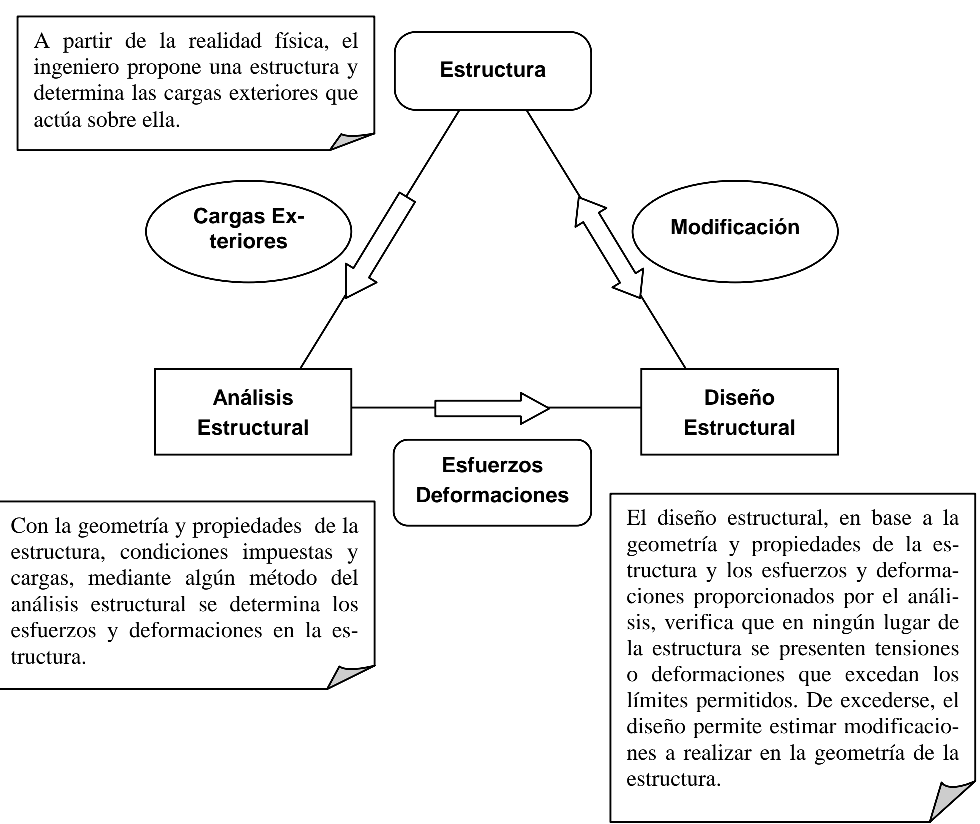


Figura 5.8 Secuencia operacional

Los tipos de problemas pueden subclasificarse según su comportamiento elástico o plástico, estático o dinámico, si los efectos térmicos son tenidos en cuenta. Cada subtipo de problema tiene métodos de análisis y diseño asociados.

Capítulo 6

Microarquitecturas OO para el dominio de la I.E.

En esta sección se analizan y modelan conceptos del dominio enfatizando la reusabilidad a través de aplicar sistemáticamente patrones de diseño que conducen a soluciones flexibles y modificables. Como resultado se obtienen *microarquitecturas* orientadas a objetos que representan los componentes fundamentales del dominio de la I.E.

Los conceptos están agrupados en categorías que van desde las más genéricas a las más específicas del dominio: cantidades y mediciones; vectores y matrices; funciones y métodos numéricos; cargas y estados de cargas; normas y reglamentos; sistemas discretos; análisis y diseño estructural.

6.1 Cantidades y mediciones

6.1.1 Magnitudes

Magnitud es todo ente abstracto que se puede medir. Para poder medir, se requiere de alguna unidad de medida.

Hay dos tipos de magnitudes que aparecen en el dominio de la ingeniería. Se trata de las magnitudes escalares y las magnitudes vectoriales.

Una **magnitud escalar** requiere para su medición de un número y una unidad asociada con él. Ejemplos de magnitudes escalares son: la longitud, el tiempo, el volumen. Ejemplos de cantidades de estas magnitudes pueden ser: 3 metros, 5 segundos, 10 m^3. Una manera de modelar atributos que se representan por cantidades escalares es utilizar el patrón ***Quantity*** [Fowler97], que es un tipo que combina un número con la unidad que está asociada con él.

Quantity: Representa las cantidades como un valor y su unidad. Esto permite poder hacer conversiones posteriores, y da mayor conocimiento acerca de qué representa, por ejemplo, el valor 50 en la representación del peso de una persona. Este "pequeño" patrón permite que un sistema de cálculo de ingeniería no tenga necesidad de estar atado a un sistema de medición particular, el cual en general cambia de un país latino a un país anglosajón, por ejemplo. Este patrón junto con el **Conversion Ratio,** que veremos más adelante, resuelven el problema.

Dado que este patrón modela cantidades escalares, se lo renombra como **CantidadEscalar**, dejando el tipo Cantidad reservado para generalizar ambas cantidades**.**

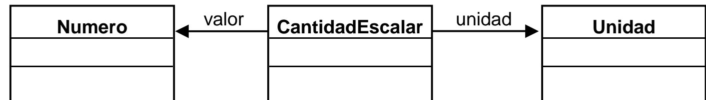


Figura 6.1 Cantidades escalares

Las **magnitudes vectoriales** se representan por un vector y requieren, para estar completamente definidas, que se dé información de su módulo o intensidad, dirección y sentido, o bien de sus componentes en el sistema de coordenadas elegido: esférico, cilíndrico o cartesiano. Ejemplos de magnitudes vectoriales son las fuerzas, los desplazamientos, la velocidad y la aceleración. Para indicar el desplazamiento que sufre un cuerpo se requiere indicar tres cantidades. Estos tres valores son las magnitudes escalares de las componentes del vector en alguno de los sistemas de coordenadas, por ejemplo: (3m; 4m; 1m) representa las componentes de un desplazamiento en un sistema de coordenadas cartesianas; o (36,9°; 5m; 1m) representa las coordenadas del mismo desplazamiento en un sistema de coordenadas cilíndrico.

La cantidad de coordenadas depende de la dimensión de espacio que estamos trabajando, en el plano (espacio bidimensional) son dos y en el espacio tridimensional es tres. Existen fórmulas que permiten convertir coordenadas desde un sistema a otro.

Para modelar atributos que se representan mediante cantidades vectoriales se define el tipo **CantidadVectorial**.

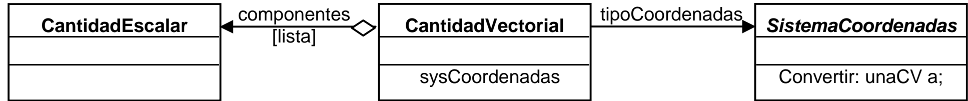


Figura 6.2 Cantidades Vectoriales

El orden con que son expresadas las componentes es importante ya que representa una coordenada específica. Así, para el sistema cartesiano la primer componente es el valor de la coordenada *x*, la segunda es *y* y la tercera es *z*; mientras que para el cilíndrico el orden es: (ϕ; r; z). Por eso el mapeo entre **CantidadVectorial** y **CantidadEscalar** se expresa como lista y la lista es única para una magnitud en un sistema de coordenadas.

Generalizando estos dos tipos de cantidades, definimos a ***Cantidad*** como un supertipo de ambos, y responsable de establecer la interface común con operaciones que permitan la suma, resta, comparación, etc. Para poder realizar operaciones aritméticas y de comparación entre cantidades, *Cantidad* requiere de la colaboración de *Unidad* en cuanto debe convertir la unidad del operando en la unidad del receptor (ver punto 6.2.2) y de SistemaReferencia cuando se trata de cantidades vectoriales.

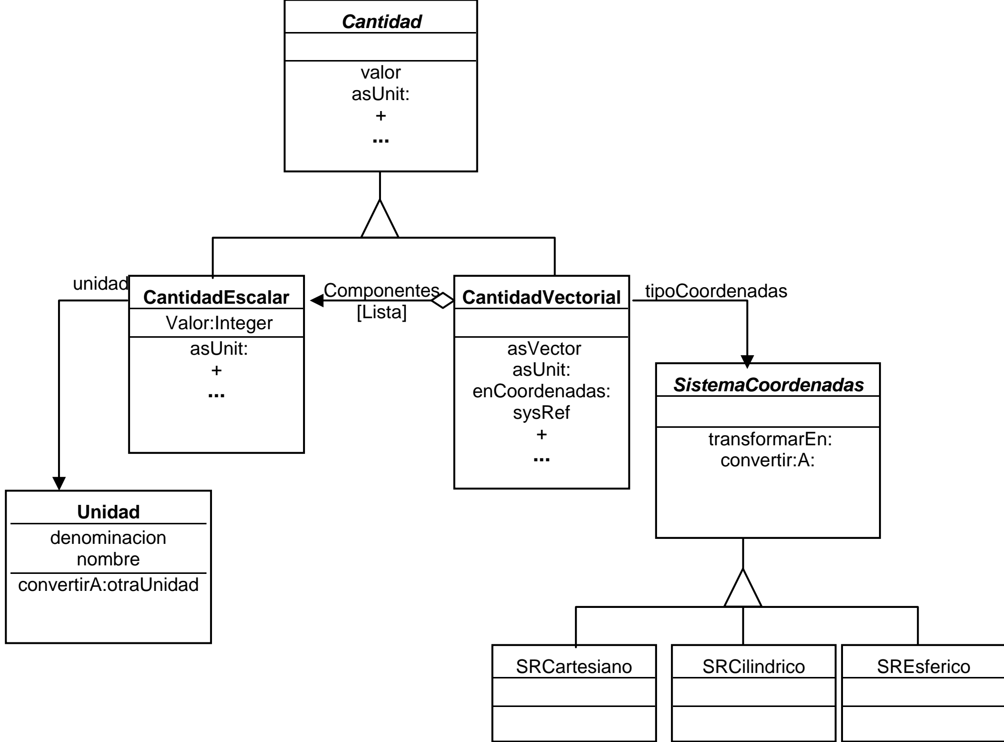


Figura 6.3 Cantidades generalizadas

Cuando el mensaje + otraCantidad es recibido por una cantidad escalar, si la unidad de otraCantidad es diferente de la del receptor, primero se la convierte a la unidad del receptor, para lo cual requiere de la colaboración de *Unidad* y finalmente se retorna una cantidad cuyo valor es la suma de los valores de ambas cantidades y la unidad es la del receptor.

Si el mensaje es recibido por una cantidad vectorial, el operando debe ser vectorial y se debe verificar primero que estén referidos a un mismo tipo de sistemas de coordenadas, de no estarlo, el operando debe ser transformado al tipo de sistema del receptor para lo cual colabora la clase *SistemaCoordenadas*. Luego, se retorna una cantidad vectorial donde cada una de sus componentes resulta de la suma de las correspondientes componentes del receptor y el operando, siendo la clase CantidadEscalar (e indirectamente Unidad) quien colabora en esta parte.

6.1.2 Unidades de medida

La **unidad de medida** es una cantidad de la misma especie de la que se quiere medir. Permite medir un tipo de fenómeno o magnitud.

En la Física hay gran cantidad de magnitudes que son objeto de estudio. Si para cada una de ellas hubiera que definir una unidad independiente de las demás se crearían grandes problemas tanto como para recordarlas como para operar con ellas, por lo que se procura relacionarlas entre sí para que a partir de un mínimo número de ellas se puedan obtener todas las demás. A las unidades que forman este conjunto mínimo del que se puede partir para adoptar las demás unidades se las llama *unidades fundamentales* y a las otras *unidades derivadas*.

Las **unidades fundamentales** corresponden a las magnitudes: longitud, masa, tiempo, intensidad de corriente eléctrica, temperatura, cantidad de sustancia e intensidad iluminación. Para cada una de ellas hay varios sistemas de medición, con múltiplos y submúltiplos de unidad, y existen fórmulas que permiten conversiones. Ejemplo: para la medición de longitud: metro, kilómetro, pulgadas, etc. Y la fórmula que permite convertir metros a pulgadas es 1 m = 254 pulgadas.

Las **unidades derivadas o compuestas** se obtienen por combinación de las fundamentales, por ejemplo, en la medición de velocidad interviene una unidad de longitud y una unidad de tiempo elevada a la potencia dos negativa. Ej: [m s^{-2}] , [km h^{-2}].

Se dice que una unidad es homogénea si sólo interviene un único tipo de unidad elevado a cualquier potencia distinta de cero.

Para convertir una unidad en otra, debe referirse a un mismo tipo de fenómeno o magnitud y en el caso de ser una unidad compuesta, el factor de conversión puede calcularse a partir de los factores de conversión de cada unidad fundamental que aparece.

La mayoría de las conversiones de unidades pueden realizarse a través de un factor de conversión, pero no todas. Una conversión de grados Celsius a Farenheit requiere de un producto y una suma.

El modelado de unidad está basado en los patrones ***Unit y ConvertionRatio*** [Fowler97], readaptándolos para facilitar la conversión de unidades.

En general, podemos decir que una unidad está formada por una o más unidades homogéneas y que una unidad homogénea es una unidad de referencia elevada a una potencia. Bajo esta consideración, una unidad fundamental es una unidad formada por una sola unidad básica elevada a la potencia uno (1).

Entre las operaciones que debe poder realizar las unidades para que las operaciones con cantidades den resultados consistentes, se encuentran: **=** (compara si dos unidades son las mismas), ***** y **/** (multiplica o divide dos unidades dando como resultado otra unidad), igualTipo: (devuelve verdadero si los tipos son equivalentes) y convertirA: (permite pasar de una unidad a otra de tipo equivalente).

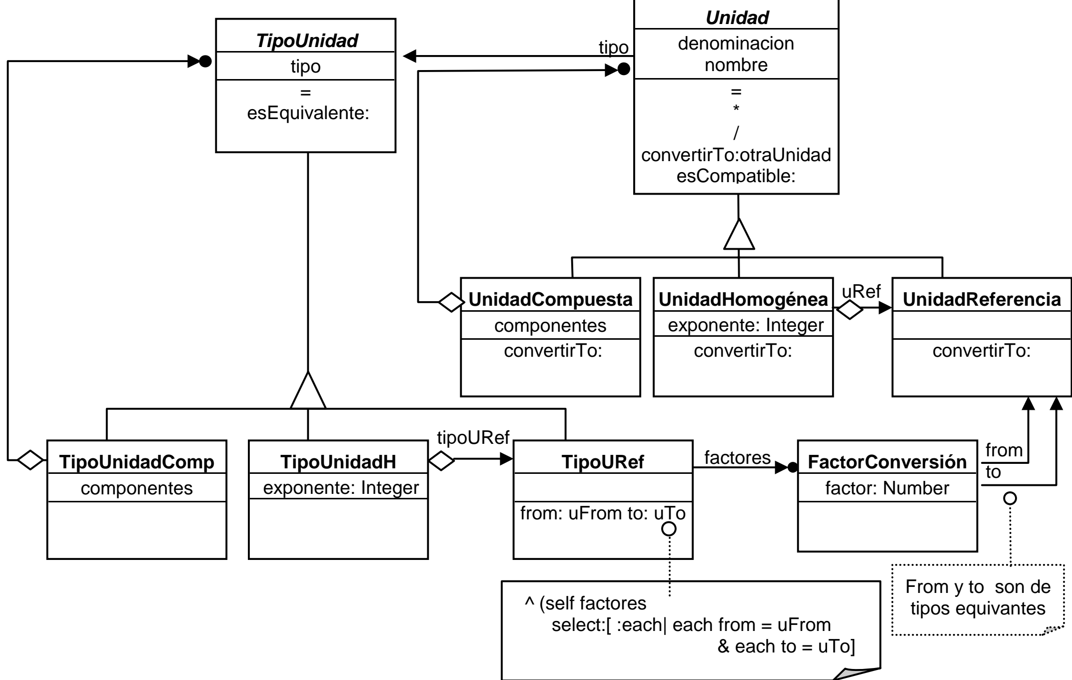


Figura 6.4 Unidades y Conversiones

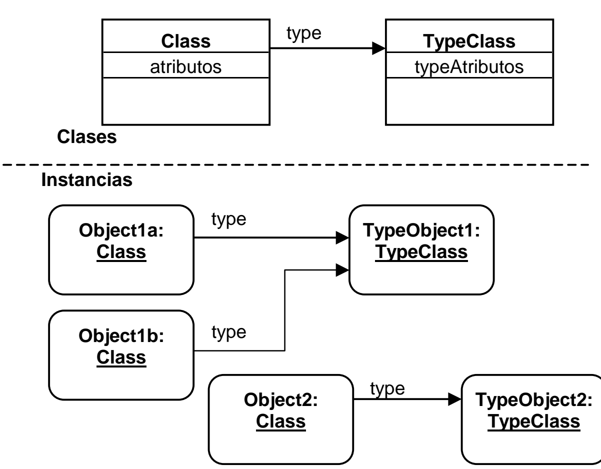
Los factores de conversión son una triada de un par de unidades de referencia de un mismo tipo y un número que representa el factor de conversión desde la primer unidad a la segunda.

Ejemplos:

La unidad **m^2** (metros cuadrados) es una unidad homogénea de tipo **superficie** constituida por la unidad referencia **m** (metro) con potencia **2.**
La unidad **dyna** está compuesta por las unidades **g** (gramo) con potencia **1, cm** (centímetro) con potencia **1** y **s** (segundo) con potencia **- 2**.
Las unidades de referencia **m** y **cm** corresponden al tipo Unidades de Longitud; **g** a unidades de masa y **s** a unidades de tiempo.

¿Cómo representamos en el modelo que una unidad de fuerza está compuesta por una unidad de masa a la potencia 1, una unidad de longitud y una unidad de tiempo a la potencia menos dos [M L T^{-2}]?.
Creando una instancia TipoUnidadCompuesta con nombre “Fuerza” y tres componentes: una de tipo Masa otra de tipo Longitud y otra de tipo Tiempo con potencia -2.

Consideraciones de diseño: Para plantear la relación entre la unidad y su tipo aplicamos el ***Type Object Pattern*** [Johnson 96] que permite que varias instancias de una clase, en este caso *Unidad*, sean agrupadas de acuerdo con comunes atributos y/o comportamiento, evitando una explosión de numerosas subclases y permitiendo agrupar recursivamente de modo que un grupo es en sí mismo un ítem en otro grupo. Consta de dos clases: una que representa los objetos y otra que representa sus tipos. Cada objeto tiene referencia a su correspondiente tipo y delega alguna de sus responsabilidades a él.



TypeClass: es la clase de *TypeObject*. Tiene una separada instancia de cada tipo de *Object*.

TypeObject: es una instancia de *TypeClass*. Representa a un tipo de *Object*. Establece todas las propiedades de un *Object* que son comunes para todos los *Objects* del mismo tipo.

Class: es la clase de *Object*. Representa una separada instancia de *TypeClass*.

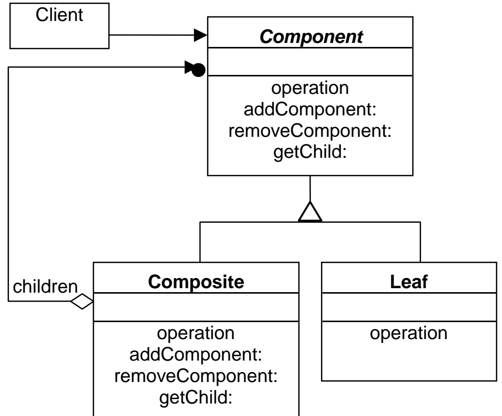
Object: es una instancia de *Class*. Representa un ítem único que tiene un único contexto. Establece todas las propiedades del ítem que lo diferencia entre los ítems del mismo tipo. Delega propiedades definida por su tipo en su asociado *TypeObject*

Figura 6.5 Type Object Pattern

La clase *TypeClass* define la estructura y comportamiento común que tendrán sus instancias, las que oficiarán como supertipos de las instancias de *Class*. Cada instancia de *TypeClass* representa un *tipo* de objeto y establece las propiedades y comportamiento para todos los objetos del mismo tipo. Ambas clases pueden ser subclasificadas independientemente.

En nuestro caso el *TipoUnidad* corresponde al *TypeClass* y *Unidad* al *Class*.

Para representar la estructura recursiva de las unidades compuestas y de los correspondientes tipos, aplicamos el pattern ***Composite*** [Gamma+95] que permite componer objetos en estructuras de árbol para representar jerarquías de partes y tratar en forma uniforme a los objetos simples (*Leaf*) y compuestos (*Composite*), siendo la superclase de ambos *Component*. Los clientes utilizan la interface del *Component* para interactuar con los objetos de la estructura del patrón composite sin necesidad de realizar una diferencia entre objetos simples y compuestos. Si el receptor es una hoja el requerimiento es manejado directamente, si es un objeto compuesto, el composite delega el requerimiento a sus componentes hijos.



Component: Define la interfaz para los objetos en la composición. Implementa el comportamiento por defecto para la interfaz común de todas las clases. Define la interfaz para acceder y manejar sus componentes hijos.

Leaf: Representa objetos simples, que no tienen hijos. Define el comportamiento para objetos primitivos en la composición.

Composite: Define el comportamiento para los componentes que tiene hijos. Almacena los componentes hijos. Implementa operaciones definidas en la interfaz del *Component*.

Client: Manipula objetos en la composición a través del *Component*.

Figura 6.6 Composite Design Pattern

Cuando una instancia de Unidad recibe el mensaje convertirA: otraUnidad, si es una instancia de la clase UnidadHomogenea, solicita el factor a su unidadReferencia (quien a su vez se lo solicita a su tipo), y lo eleva a la potencia expresada por su exponente. Si es una instancia de UnidadCompuesta, delega parte de la tarea en sus componentes: la conversión se efectúa multiplicando las conversiones parciales que cada una de sus componentes le entrega.

Con esta forma de componer las unidades alcanza con mantener unos pocos factores de conversión y unidades de referencia. Por ejemplo, todas las conversiones de unidades de superficie, volumen y longitud, se pueden realizar a partir de conocer las conversiones para longitud solamente.

Se deja al tipo como responsable de mantener y brindar los factores de conversión. Este esquema permitiría también realizar conversiones de una unidad A en otra C a través de convertir la unidad A en una intermediaria unidad B y esta última en C si no hubiera un factor o método directo entre A y C.

6.1.3 Mediciones

Modelar cantidades como atributos puede ser útil, pero cuando se requieren varias mediciones de diferentes fenómenos resulta mejor modelarlas con el patrón **Measurement** [Fowler97] que relaciona un tipo de fenómeno con un objeto sobre el que se aplica la medición (Party) y una cantidad (Quantity).

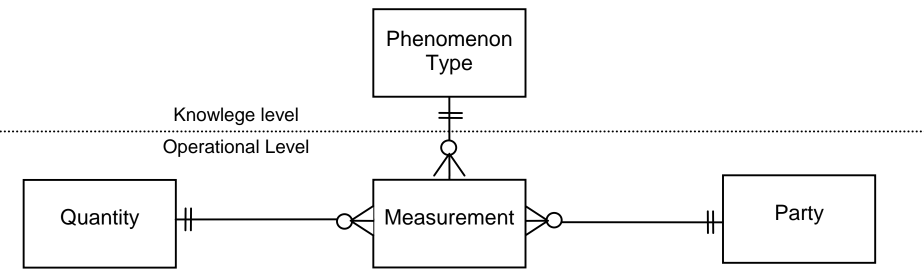


Figura 6.7 Measurement

Dado que las mediciones pueden hacerse sobre magnitudes escalares como vectoriales, se presenta una modificación al modelo de Fowler para contemplar estos casos.

En las mediciones vectoriales es importante saber el lugar o punto del objeto de medición en el que se realiza dicha medición. La medición de la presión (fenómeno vectorial) en la pared de un tanque de agua (objeto de medición) no es la misma en un punto medio que en un punto extremo, ni siquiera es la misma en ambos extremos. El modelo debe tener en cuenta esta información. Una forma de resolverlo es que aparezca en la relación el punto explícitamente. Esta solución afectaría sólo a las mediciones vectoriales.

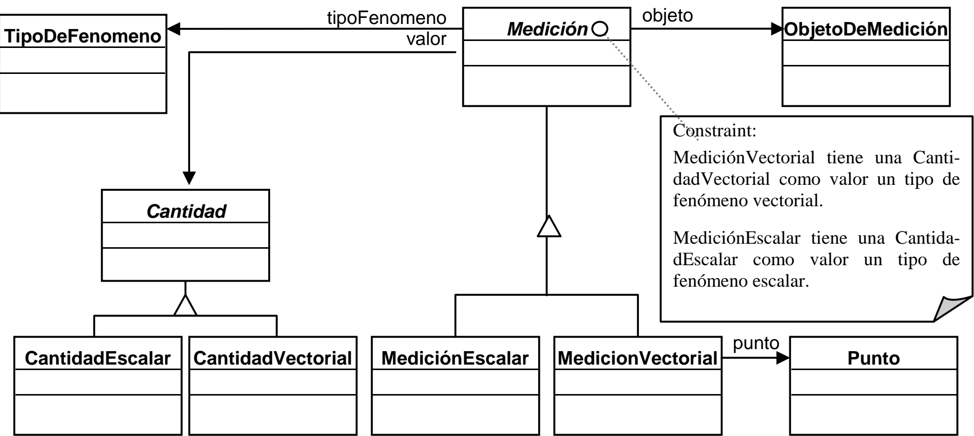


Figura 6.8 Mediciones

El TipoDeFenómeno, planteado por Fowler en el nivel de conocimiento, es modelado con el patrón **Type Object**, de manera que las características comunes a todas las mediciones de un mismo tipo de fenómeno queden encapsuladas en la clase TipoDeFenómeno, y ambas clases puedan evolucionar independientemente.

6.1.4 Sistemas de Referencia

En Ingeniería se presenta a menudo la necesidad de ubicar un objeto en el espacio. Para ubicar un objeto en el espacio es necesario dar la posición (magnitud vectorial) de cada uno de sus puntos y para ello requerimos de un sistema de referencia. Si el sistema de referencia es único para todos los objetos de mi modelo, el vector posición se modela como cualquier otra magnitud vectorial, ya sea como atributo o como medición, indicando sus componentes en el tipo de sistema de coordenadas elegido (figura 6.2).

Muchas veces es provechoso emplear dos o más sistemas de referencia para describir el comportamiento de un objeto en el que interviene directa o indirectamente su posición. Por ejemplo: Nos interesa describir el movimiento de un cuerpo. Supongamos que el cuerpo se encuentra apoyado en el asiento de un tren en movimiento. Para un observador ubicado dentro del tren el cuerpo no se mueve, para otro observador ubicado en el andén el cuerpo se mueve ya que la posición del mismo respecto de la suya va cambiando. La medición de la posición del cuerpo en los instantes t_0 y t_1, es la misma respecto de un sistema de referencia ubicado en el tren y varía respecto de un sistema de referencia ubicado en el andén. Si se conoce la posición del cuerpo respecto de un sistema de referencia (R_1) y la posición de este sistema respecto de otro (R_2), puede calcularse cual es la posición de ese cuerpo respecto de este otro sistema R_2. Y esto resulta muchas veces más fácil de hacer que referir todo a un único sistema.

Un sistema de referencia tiene un tipo de sistema de coordenadas (cartesianas, cilíndricas, esféricas) y puede ser global o local. Un sistema de referencia local (figura 6.9) tiene un desplazamiento (magnitud vectorial) respecto de otro y un ángulo de rotación (magnitud escalar). Existen fórmulas para cambiar de tipo de sistema (cartesianas a cilíndricas, por ejemplo) y para cambiar de un sistema de referencia a otro.

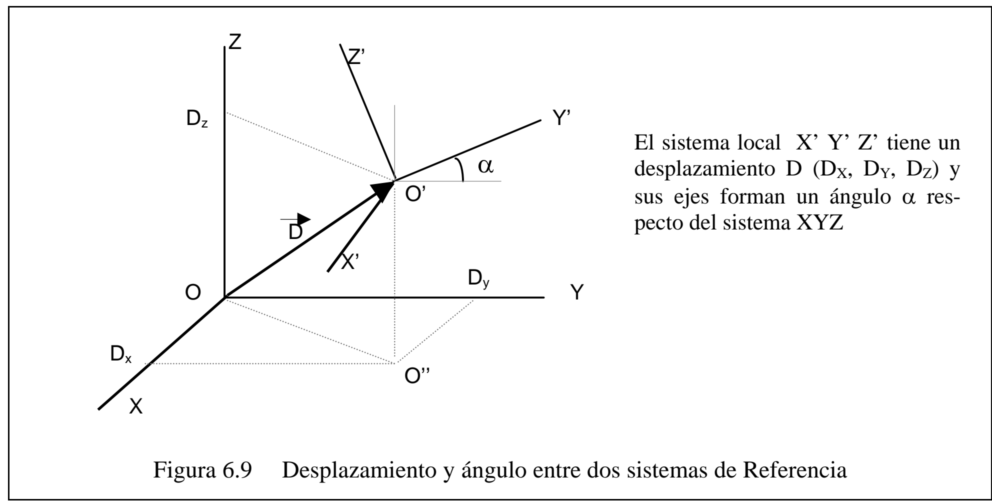


Figura 6.9 Desplazamiento y ángulo entre dos sistemas de Referencia

Las componentes de una cantidad vectorial de cualquier fenómeno (no sólo los relacionados con la posición) pueden variar su valor de acuerdo con el sistema de referencia adoptado, y este cambio no sólo se debe a que varíe el tipo de sistema de coordenadas como fue planteado en el modelo de la figura 6.2., sino al hecho de trabajar con más de un sistema de referencia y que uno pueda estar girado respecto del otro. En la figura 6.10 se muestran las componentes del vector V en los sistemas planos de coordenadas cartesianas XY y X'Y' en los que se ve claramente la diferencia de valores.

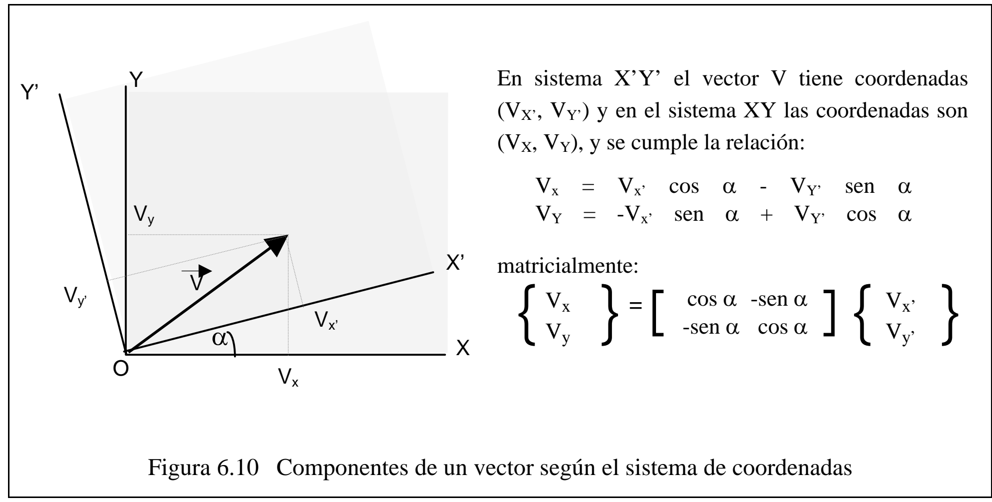


Figura 6.10 Componentes de un vector según el sistema de coordenadas

En la figura 6.11 se muestra el modelado de Sistema de Referencia y Cantidad. El tipo de coordenadas queda reflejado como una subclase, en lugar de una clase independiente. La relación entre la cantidad vectorial y el tipo de sistema de coordenadas que fuera planteado en la figura 6.2 es reemplazada por la del sistema de referencia.

El conocimiento que posee una CantidadVectorial de su Sistema de Referencia, le va a permitir poder hacer correctamente las operaciones algebraicas vectoriales (suma, resta, producto vectorial, producto cartesiano), tan comunes y necesarias en la Ingeniería, la Matemática y la Física.

Una solución similar es presentada en [Balaguer+97] en el que define el patrón **Reference System** como una solución para resolver problemas en sistemas de información geográfica (GIS).

En el dominio de la Ingeniería Estructural, las principales operaciones que requiere de un sistema de referencia, son las transformaciones de local a global y viceversa, y obtener la matriz de rotación respectiva.

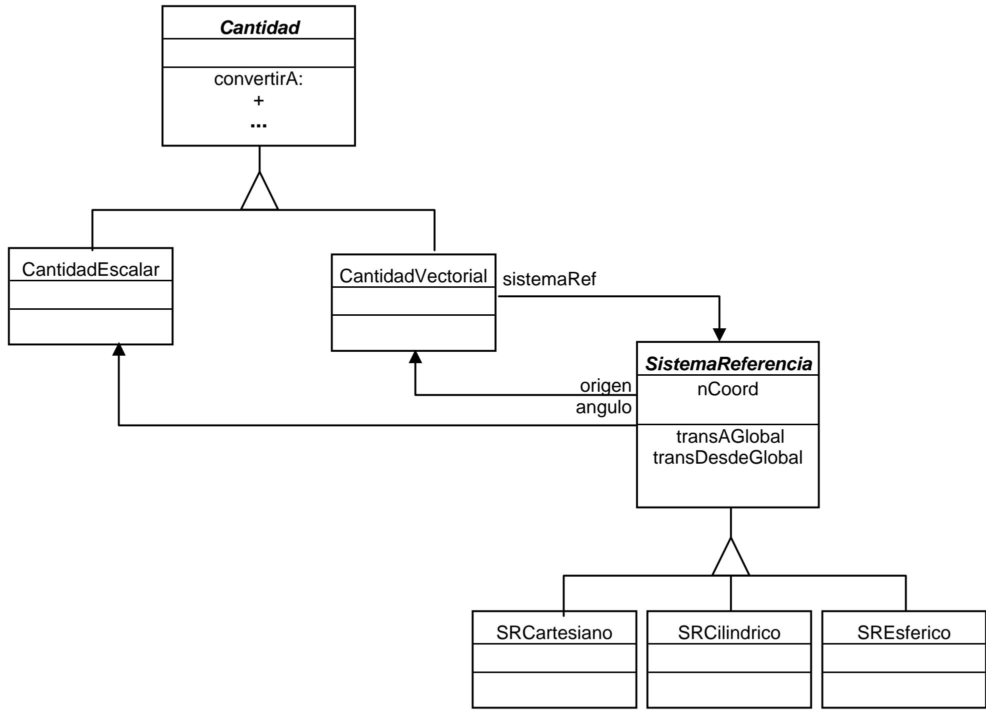


Figura 6.11 Sistema de Referencia

6.2 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 lineales, tales como, en forma matricial, $[A]\{x\}=\{b\}$. Este sistema algebraico podrá ser lineal o no lineal de acuerdo con el carácter físico del problema.

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.

Puesto que las matrices pueden ser muy pero muy grandes, 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.

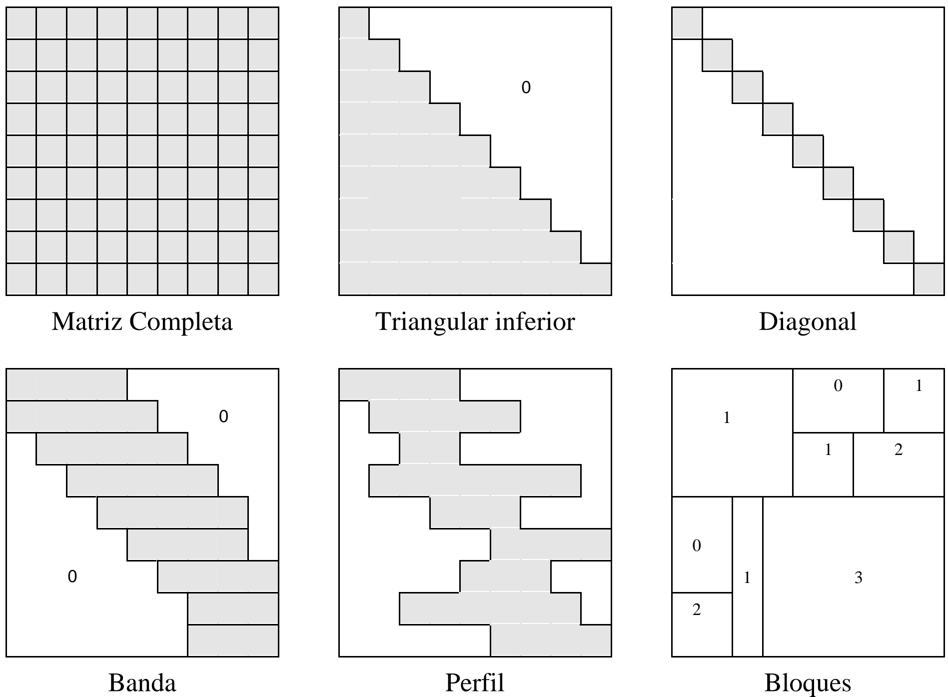


Figura 6.12 Matrices – Formas de distribución de los elementos significativos

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.

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 no es lo suficientemente flexible, ya que la herencia vincula permanentemente la abstracción con una implementación, lo que hace difícil extender la abstracción. Por otro lado, el problema de representación de un conjunto de valores, puede interesar también fuera del ámbito de las matrices.

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.

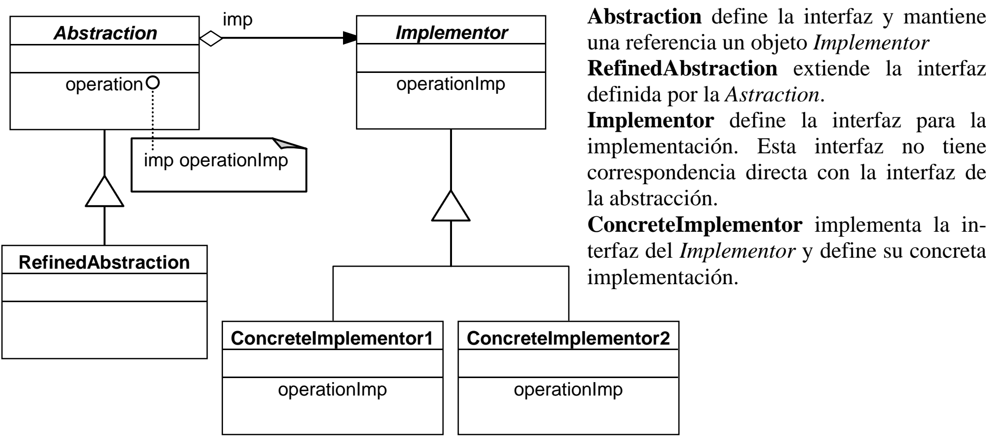


Figura 6.13 Bridge Design Pattern

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.

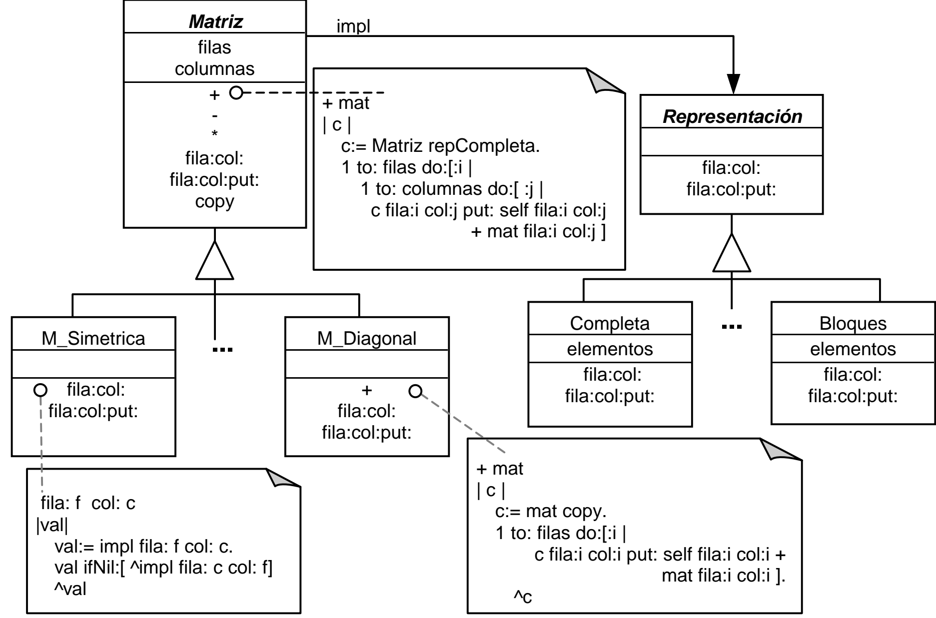
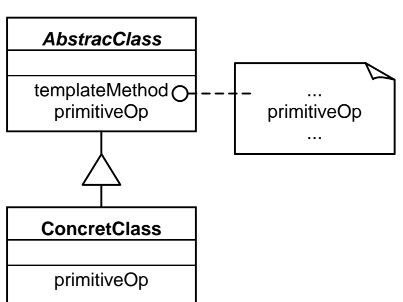


Figura 6.14 Clasificación de matriz

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** que define el esquele-

to 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 operaciones abstractas y concretas.

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

Figura 6.15 Template Method

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.

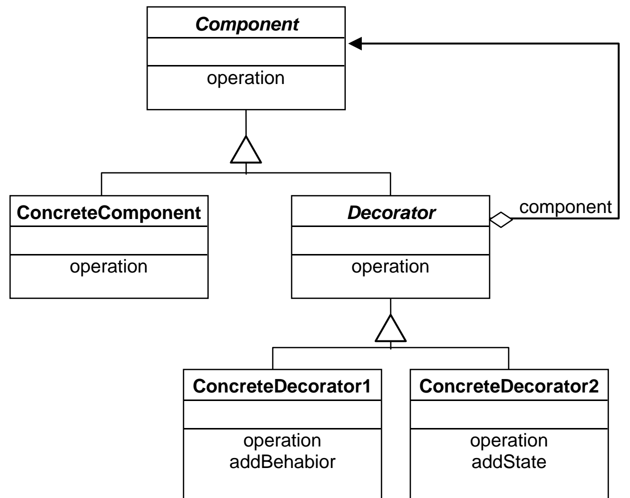
6.2.1 Matrices con unidades de medida

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.

Una forma de considerarlas, podría ser permitiendo que los elementos de una matriz sean cada uno de tipo Cantidad. 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.

Otra forma es 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 6.16 Decorator Design Pattern

El Decorator para la matriz agrega unidades y redefine las operaciones agregando validación de consistencia antes de delegarla a su componente.

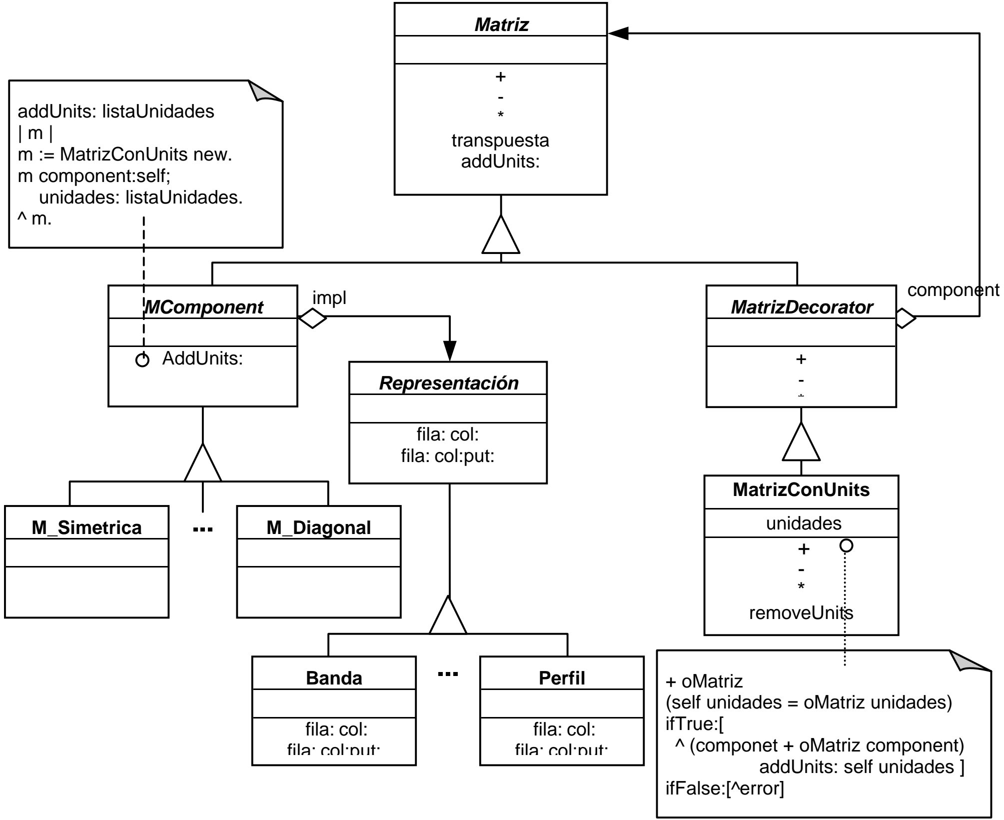


Figura 6.17 Matriz con unidades

6.2.2 Sistema de Ecuaciones

Un sistema de ecuaciones como **[A]{x}={b}**, consta de dos miembros. Uno de ellos representa el vector de términos independientes {b} y el otro por un producto entre la matriz de coeficientes [A] y el vector incógnita que se pretende resolver. Los datos son [A] y {b} y el resultado que debe devolver cuando se resuelve es {x}.

Existen diferentes tipos de ecuaciones: lineales, no lineales, diferenciales, etc. Para cada tipo de ecuaciones, existen familias de algoritmos que permiten resolver el sistema. Algunos métodos comparten una parte del algoritmo y difieren en otra. Se puede pensar en una jerarquía de clases que factorice el comportamiento común hacia clases abstractas.

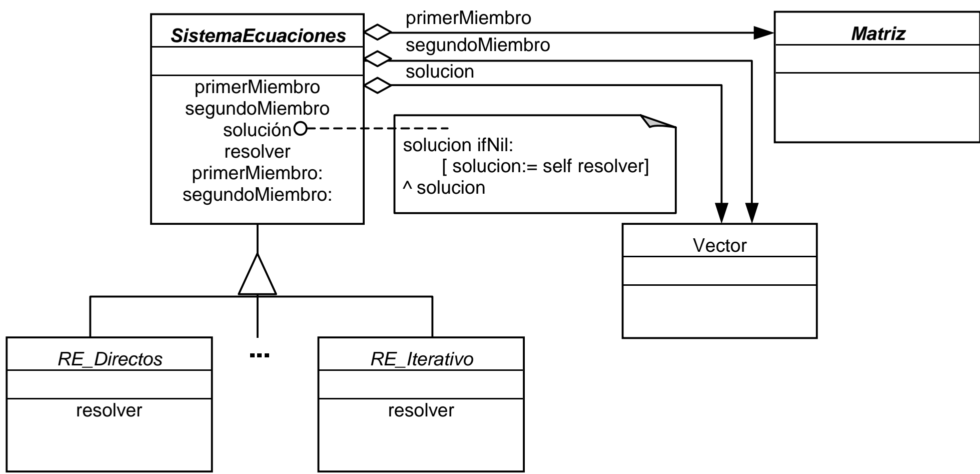


Figura 6.18 Sistema de Ecuaciones

La representación de un sistema de ecuaciones como un objeto que encapsula ambos miembros, la solución y el método, permite reflejar conceptualmente mejor el planteo del problema ingenieril y además, el sistema de ecuaciones se puede ir preparando parcialmente, entre diferentes objetos clientes.

6.3 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, muchas propiedades físicas o químicas poseen un esquema de variación funcional, problemas de diferente índole basan su solución en esquemas matemáticos similares.

Una misma pieza estructural bajo determinadas circunstancias describe su comportamiento tensorial, por ejemplo, mediante una función f_A y bajo otras

circunstancias mediante una función f_B. Diferentes comportamientos de una o más piezas pueden ser descriptos por la misma forma funcional. Conocida la función que describe un comportamiento resulta sencillo poder evaluarlo en diferentes casos.

Para poder independizar la evaluación del comportamiento de un objeto de la función matemática que lo describe, resulta necesario contar con una clase base *Función* que defina el comportamiento común de sus subclases y garantice una misma interfaz.

Existen diferentes tipos de funciones: de acuerdo con el número de variables, de acuerdo con su forma, etc. Los tipos de funciones más usadas podrían estar representados 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.

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 podrán ser implementadas en la clase abstracta aplicando el patrón de diseño **Template Method** ya descripto, y delegando la evaluación en la subclase.

Entre las operaciones aplicadas a funciones más utilizadas en la Ingeniería se encuentran la integración y derivación. Los métodos de integración numérica pueden utilizarse para integrar funciones dadas. 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. Por otro lado, no todas las funciones tienen expresión analítica de su integral.

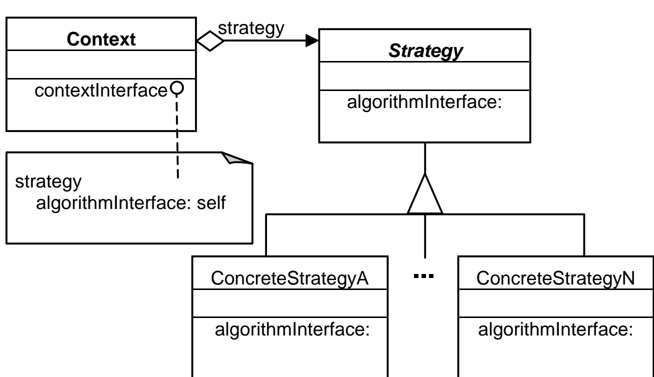
En cambio, matemáticamente siempre es posible encontrar la expresión analítica de la función derivada. La derivada de orden n se obtiene por aplicaciones sucesivas de la regla de derivación. La derivada en un punto se puede hallar evaluando la función derivada en ese punto. Y también puede hallarse por métodos numéricos.

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.

Cualquiera fuera la función, debiera brindar el servicio de evaluar su integral en un intervalo (integrarAt: unIntervalo) y de evaluar su derivada enésima en un punto (derivadaN:At:).

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

La solución para este problema es aplicar el pattern de diseño **Strategy** definido en [Gamma95]. Al definir una familia de algoritmos encapsulados y hacerlos intercambiables, permite la selección del algoritmo dinámicamente. Además, las funciones y los métodos numéricos pueden ir evolucionando independientemente.



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

Figura 6.19 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.

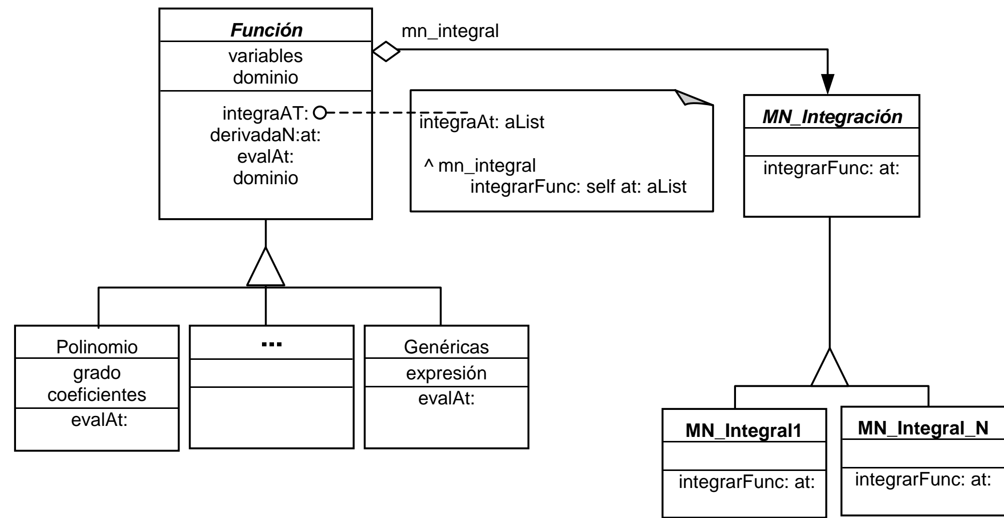


Figura 6.20 Función

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.*

En general, sólo una instancia de la clase del MN_Integral elegido es definida. Para asegurar una única instancia, nos valemos del patrón de diseño Singleton.

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 en forma discreta y para resolver las operaciones (evalAt:, derivadaN:At:, integrarAt:) se utilizan métodos numéricos: de interpolación, diferenciación, integración.

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 un vector o matriz. Esta representación no serviría para grillas no regulares, en estos casos es necesario mantener asociaciones punto - valor funcional. Si el problema de la representación se tratara juntamente con los propios de Función, las diferentes formas de representación darían origen a distintas subclases. Este enfoque no es lo suficientemente flexible, ya que la herencia vincula permanentemente la abstracción con una implementación, lo que hace 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.

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 definido 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.

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.

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.

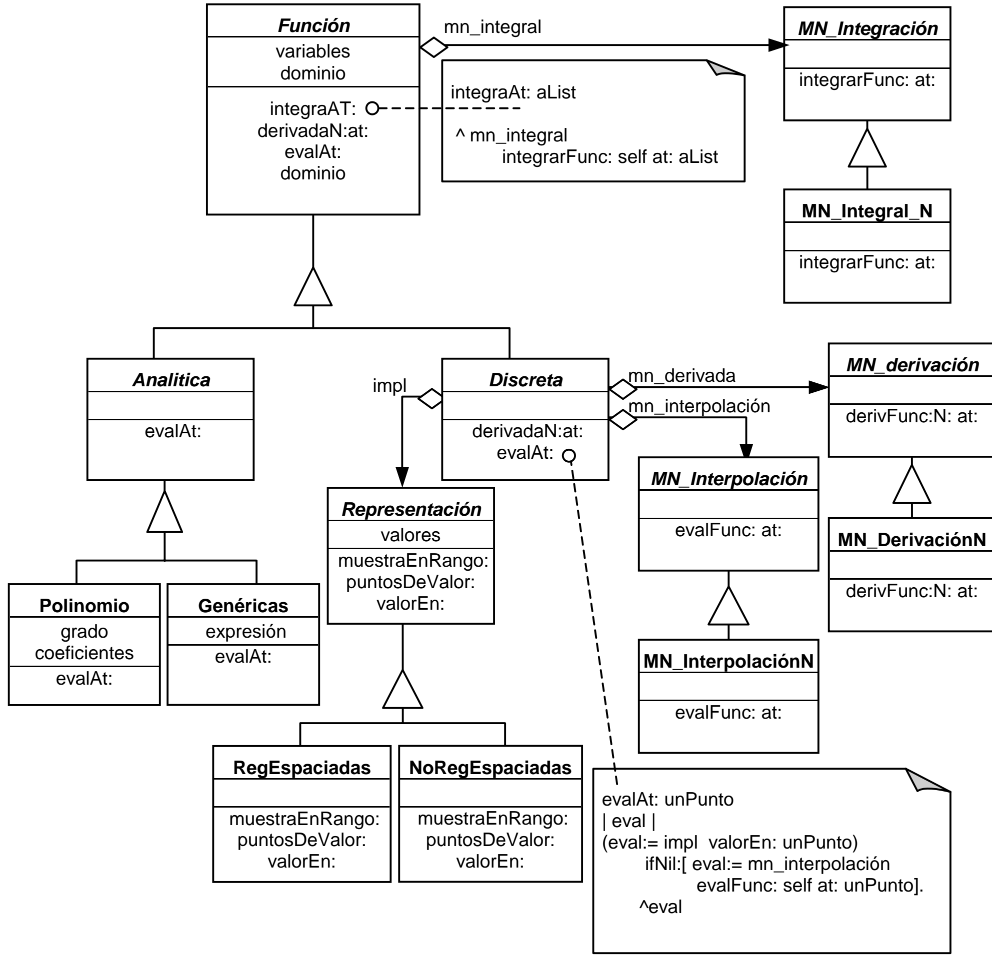


Figura 6.21 Jerarquía de Función

6.4 Cargas – Estados de Cargas – Hipótesis de carga

En general, las fuerzas externas no se conocen completamente y algunas son estimadas de acuerdo con la probabilidad de aparición. Además aparecen fuerzas diferentes en distintos momentos de la vida útil de la estructura, para los cuales debe verificarse el diseño estructural. Para considerar esta situación, el ingeniero establece diferentes hipótesis de combinación de cargas, y para cada una de ellas verifica el estado de tensiones y de deformaciones.

Una hipótesis combina uno o más estados de cargas multiplicados cada uno por un coeficiente que tiene en cuenta la incidencia de ese estado dentro de esa hipótesis.

Podemos decir que una hipótesis puede ser expresada mediante la fórmula:

$$\sum_{i=1}^{n} a_i \, Ec_i$$

donde a_i es el coeficiente de aporte y Ec_i es uno de los estados de carga.

Un Estado de carga es un conjunto de cargas aplicadas en nodos o en elementos que actúan simultáneamente.

6.4.1 Carga

Como se mencionara en la sección 5.4, las **cargas** son fuerzas y momentos externos que actúan sobre una estructura, incluyendo los desplazamientos impuestos en los vínculos mientras que los **esfuerzos** son las fuerzas y momentos internos que resisten las cargas. Las cargas son datos para el análisis estructural y los esfuerzos son calculados por el mismo. El modelo que a continuación se describe, sirve tanto para representar cargas externas como esfuerzos. También es aplicable a las tensiones y deformaciones.

Para representar una carga necesitamos una clase en la cual especificar la información propia y el comportamiento. La clase abstracta *Carga* describe el comportamiento genérico de sus subclases. De acuerdo con lo que se explicó en el apartado 5.4.1, las cargas pueden clasificarse según diferentes criterios en puntuales y repartidas, dinámicas o estáticas, y teniendo en cuenta el origen; pudiendo darse cualquier combinación: fuerza concentrada dinámica, fuerza repartida (en cualquiera de sus variantes) estática, etc.

La mayoría de los comportamientos son dependientes de la forma (concentrada o repartidas). La diferencia entre los distintos tipos de cargas repartidas está dada por su función de reparto, y los valores característicos como resultante y momento pueden calcularse con operaciones sobre la función de reparto.

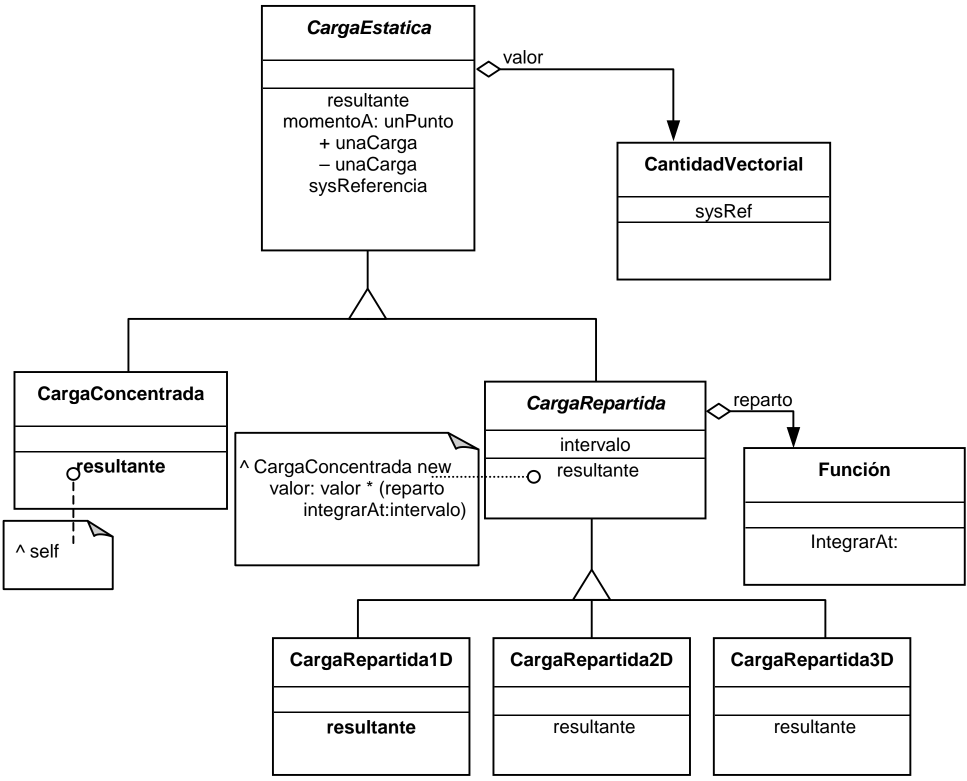


Figura 6.22 Carga Estática

Las cargas dinámicas pueden ser tratadas como las estáticas pero considerando un comportamiento adicional. La clasificación por el origen importa en el momento de calificar un estado de carga pero no afecta al comportamiento de una carga en particular.

Para considerar el comportamiento adicional de las cargas dinámicas, podríamos valernos de la especialización, extendiendo cada una de las clases de la jerarquía. Pero hay una solución mejor utilizando el pattern ***Decorator*** [Gamma+95] que permite agregar dinámicamente propiedades o comportamiento a un objeto a través de composición, y provee una alternativa flexible para extender funcionalidad, frente a la especialización, que en este caso, no resulta práctica debido al número de extensiones independientes. La clase CargaDinámica representa al *Decorator* y mantiene una referencia a un objeto Carga (*Component*) y define la interfaz del objeto decorado.

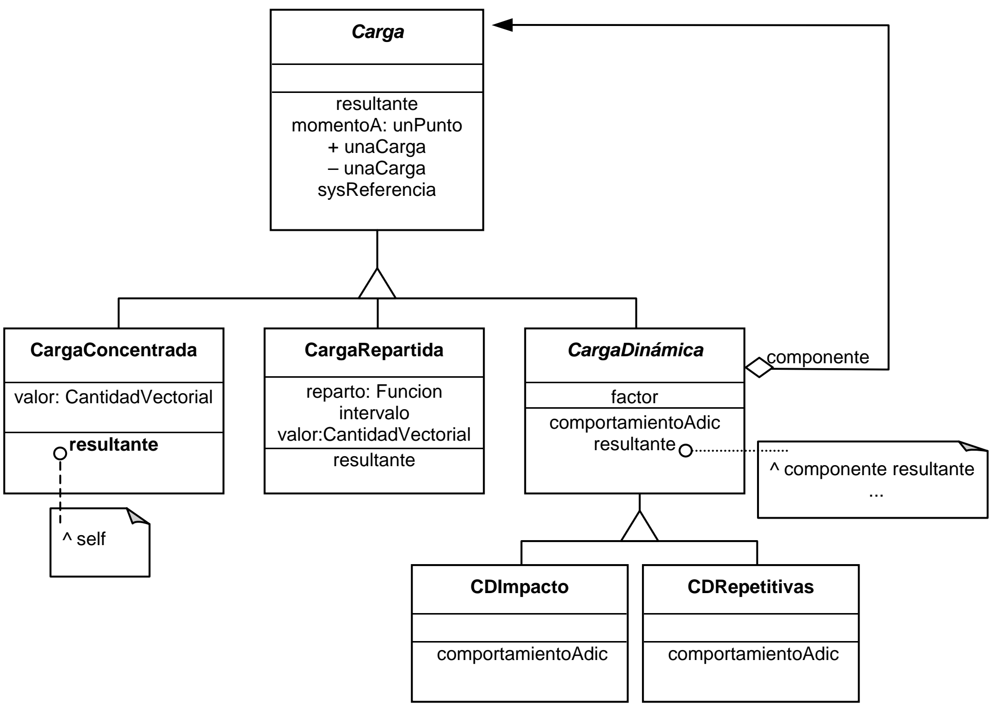


Figura 6.23 Carga

6.4.2 Estado de carga

Un estado de carga es un conjunto de cargas aplicadas en la estructura (nodos o elementos) que actúan simultáneamente y responden a un mismo origen. Para modelarlo, necesitamos de una clase que defina el comportamiento de la asociación carga-objeto (CargaAplicada) y de otra clase (EstadoDeCarga) encargada de mantener esas asociaciones.

CargaAplicada es modelada con el patrón Medición descripto en 6.2.3, la carga representa el valor medido, el elemento o nodo representa el objeto de medición y el tipo de fenómeno distingue si se trata de una carga externa, un esfuerzo, una tensión o deformación. La clase CargaAplicada se subclasifica según se aplique a elementos o a nodos y mantiene el comportamiento.

Las cargas aplicadas sobre elementos pueden convertirse en cargas equivalentes aplicadas en los nodos del elemento. En una etapa del análisis estructural importa conocer y distinguir las cargas aplicadas en nodos, (incluidas las equivalentes) y los desplazamientos impuestos. En otra etapa, importa conocer o determinar ciertos comportamientos debido a las cargas reales que actúan sobre los elementos. La clase CargaAplicada es responsable de proveer las operaciones para determinar las cargas equivalentes y el comportamiento debido a cargas reales de sus instancias.

Independientemente de la implementación elegida, la clase EstadoDeCarga debe proveer mecanismos para recorrer y acceder a todos los nodos y elementos cargados.

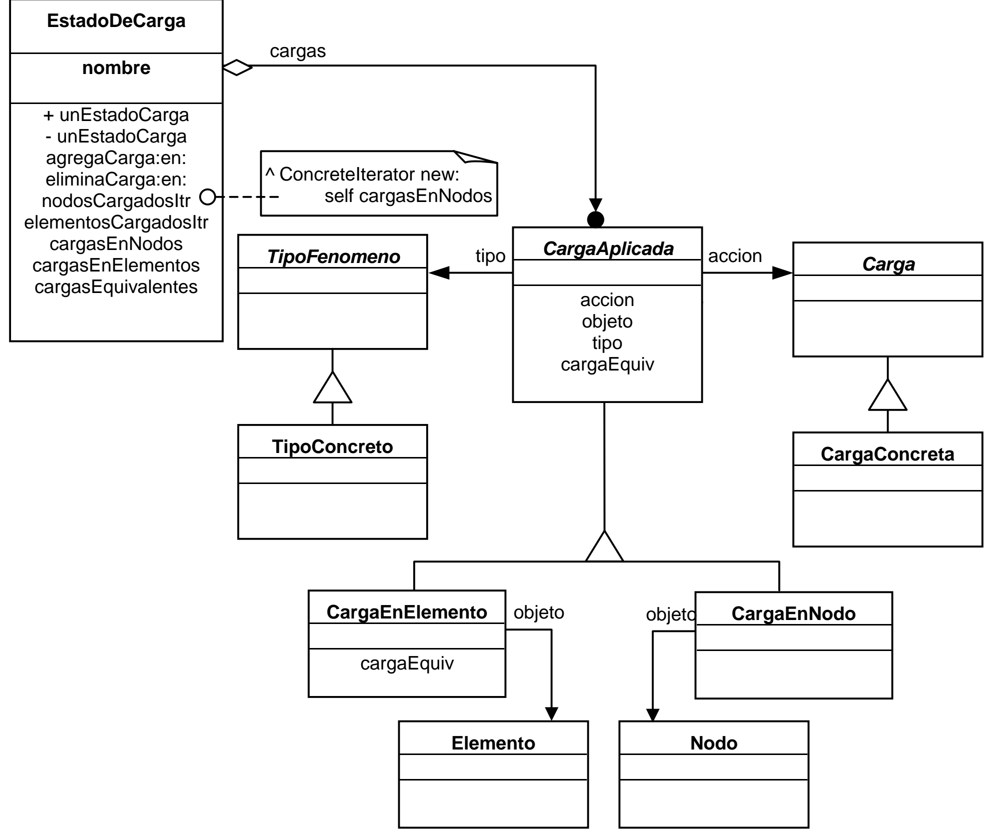
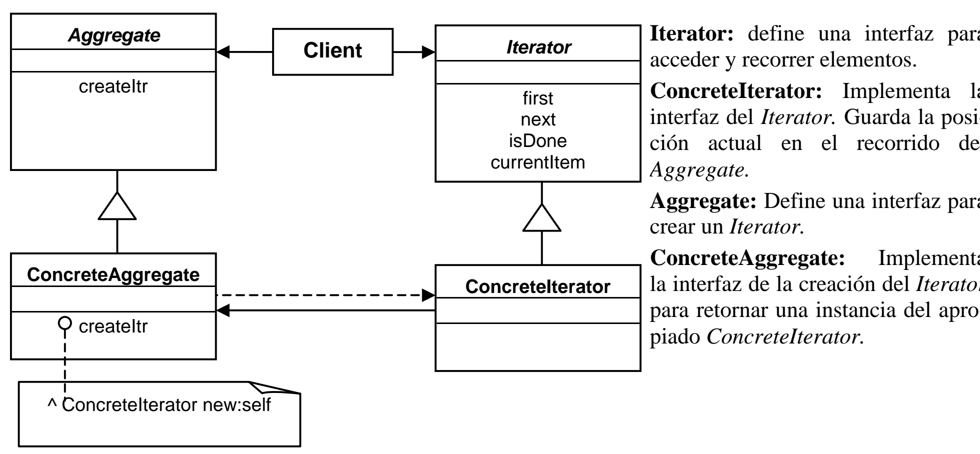


Figura 6.24 Estado de carga

Para permitir que otros objetos puedan acceder y recorrer las componentes de cada contenedor, sin exponer la representación interna, conviene utilizar el pattern ***Iterator*** [Gamma95], el cual provee una interfaz para controlar el recorrido (first, next, isDone) y el acceso (currentItem).



Iterator: define una interfaz para acceder y recorrer elementos.

ConcreteIterator: Implementa la interfaz del *Iterator.* Guarda la posición actual en el recorrido del *Aggregate.*

Aggregate: Define una interfaz para crear un *Iterator.*

ConcreteAggregate: Implementa la interfaz de la creación del *Iterator* para retornar una instancia del apropiado *ConcreteIterator.*

Figura 6.25 Iterator Design Pattern

Para ello necesitamos de una clase abstracta Iterator que defina la interfaz y clases concretas que implementen las operaciones según el tipo de iterador que se pretende: Array, Lista, Cola. Por otro lado, la clase que requiere de un iterador, necesita definir un método para crearlo.

EstadoDeCarga además de proveer operaciones para mantener las asociaciones como agregaCarga:en:, eliminaCraga:en:, + (unEstadoCarga), provee las operaciones elementosCargadosItr y nodosCargadosItr que retornan una instancia del correspondiente Iterator.

Las clases que necesiten iterar sobre los ítems de un estado de carga, lo harán con la interfaz polimórfica de Iterator.

6.4.3 Hipótesis de carga

La clase HipótesisDeCarga mantiene las asociaciones de los estados de carga y los factores con que participan en una hipótesis. Al igual que en EstadoDeCarga, se aplica el pattern ***Iterator*** para proveer un mecanismo de acceso y recorrido sin exponer su representación interna. La operación estadoDeCargaItr provee el iterador correspondiente.

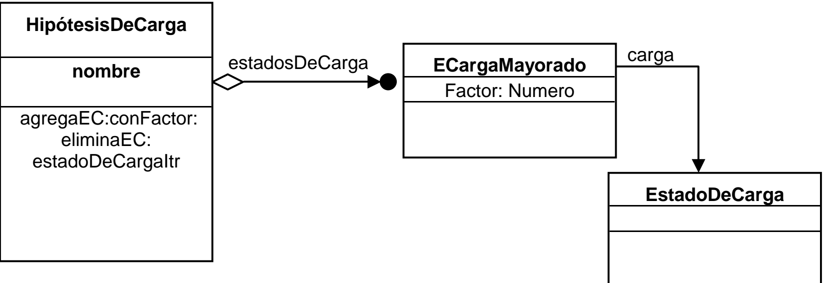


Figura 6.26 Hipótesis de carga

6.5 Normas y Reglamentos

El ingeniero como cualquier otro profesional, pocas veces es completamente libre de escoger o aceptar cualquier valor para las propiedades del componente de un sistema. En general existen varias restricciones: de tipo legal, económico, físico, temporal o estético que determinan valores de las propiedades de los componentes o fijan límites en cual deben permanecer.

En Ingeniería Estructural, muchas propiedades son obtenidas mediante cálculo a partir de otras iniciales, y deben verificar que cumplan con determinadas restricciones. Por ejemplo, en el análisis estructural se determinan los esfuerzos, tensiones y deformaciones a partir de los datos del problema y en el diseño se verifican que se encuentren dentro de límites permitidos, los cuales están fijados por normas dentro del reglamento vigente para ese tipo de problema. Un valor de tensión de tracción de $50Kg/cm^2$ es aceptable en el reglamento cuando el material es acero y no lo es para el de hormigón.

Un reglamento contiene un conjunto de normas que tienen validez en un determinado contexto. Una norma es un valor o límite de valores que gobierna una propiedad del sistema. Una norma mínima fija sólo el límite inferior y una norma máxima, el límite superior.

Se hace necesario modelar las normas y reglamentos como objetos capaces de interactuar con otros objetos que requieren verificar sus propiedades. Para ello se definen dos clases que colaboran entre sí: **Reglamento** y **Norma**.

Reglamento se encarga de la administración de las normas: agrega, modifica o elimina normas, conoce qué fenómenos o propiedades están reglados y brinda una interfaz para que los clientes del reglamento puedan consultar si un cierto valor de un fenómeno es aceptable por ese reglamento o cuál es el tope inferior o superior fijado para un fenómeno.

Norma vincula un valor o rango de valores con un tipo de fenómeno. Es capaz de responder si un valor cumple o no con ella. Puede resultar conveniente subclasificar en NormaMínima, NormaMaxima y NormaConRango. Las dos primeras vinculan el tipo de fenómeno con una cantidad, y la última con un rango.

El rango puede ser modelado basándonos en el patrón de análisis **Range** [Fowler97], que vincula dos magnitudes.

Figura 6.27 Range Pattern

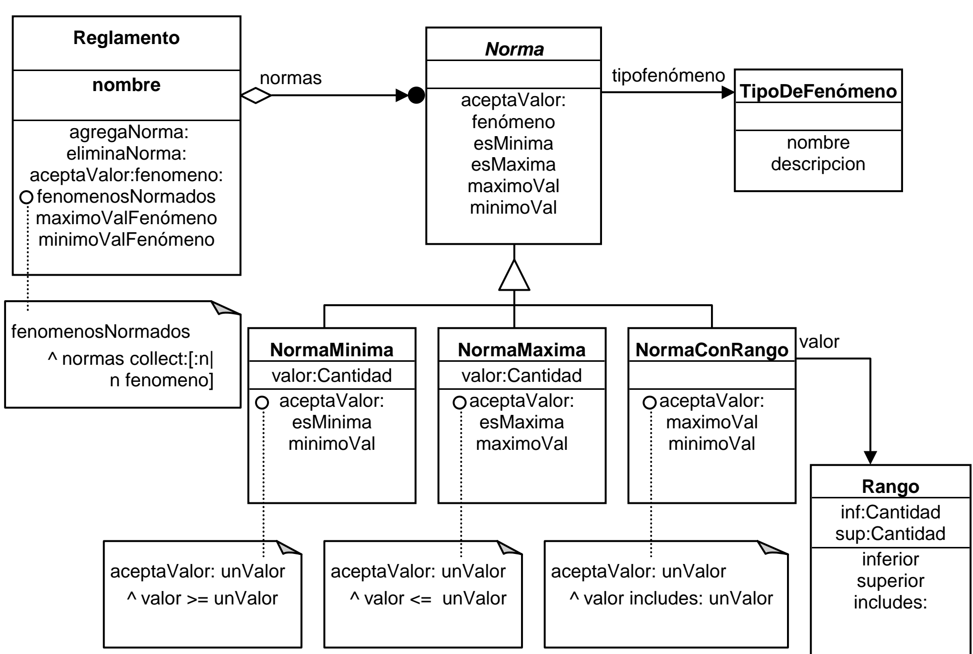


Figura 6.28 Reglamento

6.6 Sistemas Discretos. Estructuras de barras

En numerosas ocasiones de la vida práctica el ingeniero se enfrenta con el problema de analizar un sistema tipo malla compuesto de una serie de "elementos" diferentes, físicamente diferenciables, conectados por sus extremidades o "nudos" y sometidos a un conjunto de "acciones", en el sentido más amplio de la palabra, normalmente externas al sistema. Ejemplos de dichos sistemas, que denominaremos "discretos", abundan en ingeniería. Relacionados con las estructuras, por ejemplo, podemos considerar sistemas discretos todas las estructuras de barras, tales como pórticos, simples y compuestos, celosías, entramados de edificación.

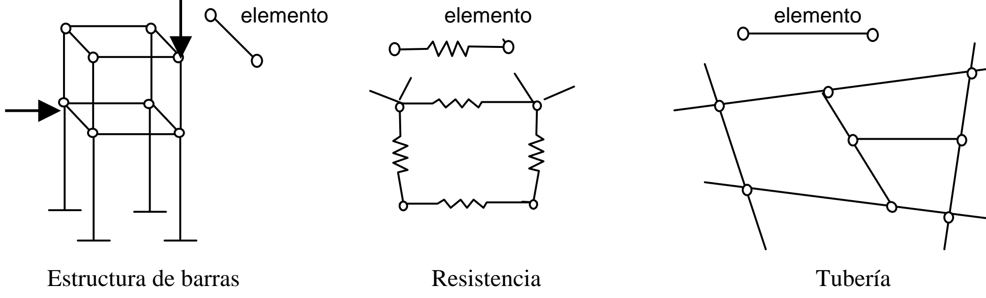


Figura 6.29 Sistemas discretos

En otras áreas de la ingeniería tenemos ejemplos de este tipo de sistemas en las redes hidráulicas y eléctricas, en los métodos de optimización de la producción (PERT etc.), y en los sistemas de organización del transporte. En la Figura 6.29 se han representado algunos de dichos sistemas discretos.

La mayoría de los sistemas discretos pueden analizarse utilizando técnicas de cálculo matricial muy similares, y que a su vez guardan una estrecha relación con el método de elementos finitos.

Las ecuaciones matriciales de una estructura de barras se obtienen a partir del estudio del "equilibrio" de las diferentes barras que la componen. En estas ecuaciones intervienen características geométricas de la barra, características mecánicas del material que la compone, las fuerzas actuantes y los desplazamientos. Así por ejemplo, el estudio de equilibrio de una barra *e* de longitud $\ell_{(e)}$ sometida únicamente a fuerzas axiles como la de la Figura 6.30,

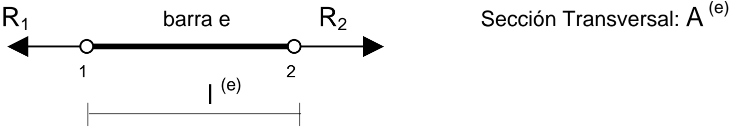


Figura 6.30 Barra sometida a fuerzas axiles

conduce a:

$$R_2^{(e)} = -R_1^{(e)} = (EA/\ell)^{(e)}\,(u_2^{(e)} - u_1^{(e)}) = k^{(e)}\,(u_2^{(e)} - u_1^{(e)}) \qquad (1)$$

donde R_1 y R_2 son las fuerzas en los nodos de la barra, E es el módulo de elasticidad del material, A y ℓ son la sección y longitud de la barra, y u_1 y u_2 son los desplazamientos de los nodos 1 y 2, respectivamente. El índice *e* indica que los valores se refieren a una barra en particular.

También puede ser expresada en forma matricial:

$$q^{(e)} = \begin{Bmatrix} R_1^{(e)} \\ R_2^{(e)} \end{Bmatrix} = k^{(e)} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{Bmatrix} u_1^{(e)} \\ u_2^{(e)} \end{Bmatrix} = K^{(e)} a^{(e)} \qquad (2)$$

$K^{(e)}$ se denomina matriz de rigidez y es función de la geometría de la misma y de sus propiedades mecánicas; y $a^{(e)}$ y $q^{(e)}$ son los vectores desplazamientos y de fuerzas en los nudos de la barra.

La expresión de equilibrio de una estructura compuesta de barras como la considerada se obtiene a partir de la regla que expresa que la suma de las fuerzas en un nudo, debida a las diferentes barras que concurren en el mismo, es igual a la fuerza exterior que actúa en dicho nudo:

$$\sum_{e=1}^{n_e} R_i^{(e)} = R_j^{exterior} \qquad (3)$$

donde la sumatoria se extiende a todas las barras n_e que concurren en el nodo de numeración global j.

Las ecuaciones (1), (2) y (3) son muy similares para la mayoría de los sistemas discretos. Así por ejemplo, un elemento aislado de una red eléctrica (resistencia), proporciona, de acuerdo con la Ley de Ohm, una relación entre los voltajes y las intensidades que entran por cada nodo muy similar a la (1). Algo similar ocurre con la ecuación de equilibrio de caudales q y altura piezométricas h de los nudos de una tubería. En la figura siguiente se muestra la analogía entre elementos aislados de una red eléctrica y de un tramo de tubería.

Resistencia eléctrica	Tramo de tubería
$I_1^{(e)}$ r (e) s $I_2^{(e)}$ 1 2 $V_1^{(e)}$ $R^{(e)}$ $V_2^{(e)}$	(e) $q_1^{(e)}$ r s $q_2^{(e)}$ 1 2 $h_1^{(e)}$ $k^{(e)}$ $h_2^{(e)}$
Ecuación (1) $I_1^{(e)} = - I_2^{(e)} = k^{(e)} (V1^{(e)} - V2^{(e)})$ $k^{(e)} = 1 / R^{(e)}$ I: intensidad de corriente V: Voltaje R: resistencia	$q_1^{(e)} = - q_2^{(e)} = k^{(e)} (h_1^{(e)} - h_2^{(e)})$ q: caudal h: altura piezométrica k: coeficiente que depende de la rugosidad de la tubería y las alturas piezométricas
Ecuación (2) $\frac{1}{R^{(e)}} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{Bmatrix} V_1^{(e)} \\ V_2^{(e)} \end{Bmatrix} = \begin{Bmatrix} I_1^{(e)} \\ I_2^{(e)} \end{Bmatrix}$	$k^{(e)} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{Bmatrix} h_1^{(e)} \\ h_2^{(e)} \end{Bmatrix} = \begin{Bmatrix} q_1^{(e)} \\ q_2^{(e)} \end{Bmatrix}$
Ecuación (3) $\sum_{e=1}^{n_e} I_i^{(e)} = I_j^{exterior}$ esta ecuación representa a la Ley de Kirchhoff que establece que la suma de intensidades de corriente que concurren a un nodo es igual a cero.	$\sum_{e=1}^{n_e} q_i^{(e)} = q_j^{exterior}$ La suma de caudales que concurren en un nudo es igual al caudal aportado desde el exterior al nudo.

Figura 6.31 Analogía entre Red Eléctrica y Red de Tubería

En este tipo de problemas se parte definiendo una malla de elementos discretos (barras) conectados entre sí por nodos. Cada elemento tiene propiedades geométricas y mecánicas conocidas. En un nodo pueden concurrir más de una barra y puede haber fuerzas externas aplicadas. En el proceso de análisis estructural se determinan los esfuerzos (fuerzas internas), las tensiones y las deformaciones (desplazamientos y rotaciones), aplicando un método adecuado para el tipo de problema. En el proceso de diseño estructural, se verifica que el estado de tensiones y de deformaciones esté dentro de rangos permitidos.

6.6.1 Nodos y grados de Libertad

El nodo es una importante abstracción tanto para los modelos matemáticos basados en métodos matriciales como para la formulación por elementos finitos. Puede representar tanto a un encuentro entre piezas estructurales, como a un apoyo o a un punto de la discretización.

Un nodo tiene una posición, la cual es representada por una cantidad vectorial y tiene asociado un sistema de coordenadas, y un conjunto de grados de libertad, representados también por cantidades, escalares o vectoriales según el tipo.

Un grado de libertad es caracterizado por su naturaleza física: ¿es un desplazamiento, una rotación, una temperatura?, por su nivel de diferenciación: ¿es un desplazamiento generalizado, una velocidad generalizada o una aceleración generalizada? y por su estado: ¿fijo o libre? ¿localizado en el contorno o no?, ¿moderado o no?

La naturaleza física puede ser de tipo vectorial o escalar, dando origen a dos subclases de grado de libertad: GLVectorial y GLEscalar.

El número de grados de libertad para un problema dado puede ser muy grande, y la cantidad de grados de libertad por nodo depende del problema. Un nodo es responsable de administrar sus grados e libertad y su posición, de responder a requerimientos sobre sus grados de libertad, sistema de referencia, etc. Un grado de libertad es responsable de mantener su estado, nivel, valor y responder a preguntas tales cómo esLibre?, es un desplazamiento fijo? etc.

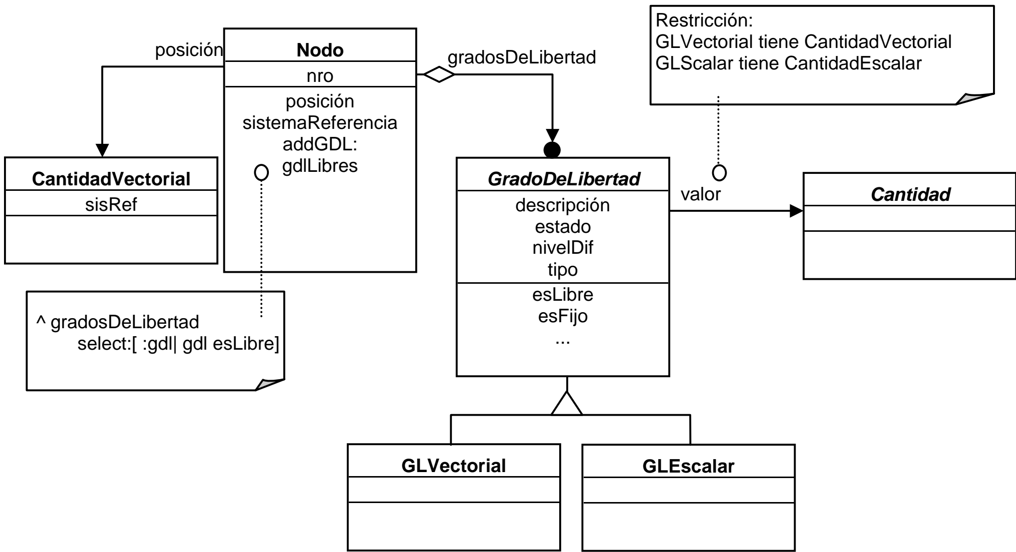


Figura 6.32 Nodo y Grado de Libertad

6.6.2 Elemento

El elemento es una de las más importantes abstracciones tanto para la formulación teórica como para el desarrollo por métodos discretos. Un elemento puede representar una pieza estructural o una parte discreta de la misma. Es un modelo propuesto por el ingeniero que simplifica la compleja realidad física para dar soluciones aceptables desde el punto de vista de la seguridad y economía.

El elemento debe ser responsable de determinar sus características geométricas y mecánicas, establecer un sistema de coordenadas locales que utilizará para comunicarse con otras componentes del sistema, proveer la información que requiere el Análisis tales como: matriz de rigidez, matriz de flexibilidad, de masa, de amortiguación; actualizar su estado tensorial y de deformación, etc.

Como las características geométricas dependen de la forma de su sección y las características mecánicas dependen del material es conveniente delegar en las clases FormaGeométrica y Material la responsabilidad de proveer las características respectivas. De esta manera todos los elementos que tengan igual forma de sección, pueden compartir la misma instancia de FormaGeométrica, y lo mismo sucede con los elementos de igual material constitutivo. Además, esta solución permite cambiar el material o la geometría en forma dinámica sin perder las restantes características.

Existe una amplia clasificación y subclasificación de tipos de elementos que representan las diferentes simplificaciones estructurales que puede plantear el ingeniero para modelar una realidad física compleja: elementos de viga, columna, placas, elasticidad plana, etc. Cada una de ellas tiene una forma propia de calcular la información requerida por el análisis.

Una clase base *Elemento* podría encapsular el comportamiento común del amplio espectro de subclases definiendo una interfaz que cada una de las subclases deberá implementar. Algunas operaciones comunes pueden ser resueltas en la clase base. En la figura 6.33 se muestra el modelo resultante.

El número grande de subtipos se da especialmente para los elementos del modelo matemático basado en “elementos finitos”, que es uno de los métodos más utilizados en el análisis estructural. Las razones de la explosión de tipos se debe fundamentalmente a que para cada simplificación estructural las aproximaciones de los campos de desplazamiento, tensiones, etc. pueden representarse por diferentes funciones, con mayor o menor grado de exactitud y complejidad.

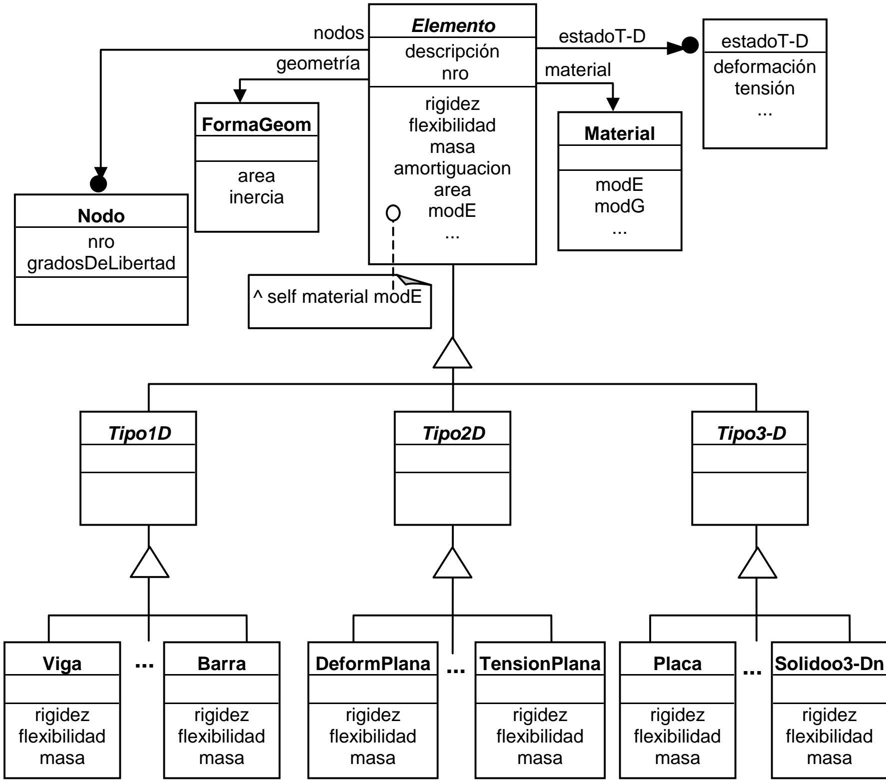


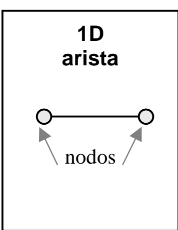
Figura 6.33 Clasificación de Elemento

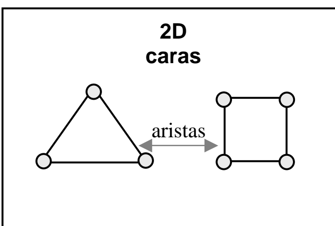
Como la determinación de la rigidez, flexibilidad varía según sea el tipo de problema y simplificación estructural adoptada, número de nodos del elemento, forma en que se distribuyen los campos de desplazamientos, tensiones y deformaciones; cada una de estas formas y combinaciones da origen a un tipo distinto de elemento. Estos tipos aparecen agrupados en "familias". Así en la bibliografía especializada se encuentran por ejemplo, entre los tipos bidimensionales subtipos rectangulares y triangulares y cada uno de ellos a su vez con varias subclasificaciones como: "rectangular Langragiano de cuatro nodos", "rectangular Langragiano cúbico de dieciséis nodos", "rectangular Serendípito cuadrático...", que obedecen a la función de forma que se elige para la distribución de los desplazamientos nodales.

Esto conduce a una gran variedad de tipos y subtipos y la solución planteada en la figura anterior, produce una jerarquía rígida con muchas subclases que puede resultar difícil de mantener. Si se desea cambiar el tipo de elemento en forma dinámica, por otro para el mismo tipo de problema pero con función de distribución diferente debiera crearse un nuevo elemento en la clase del otro tipo, copiar la información existente y luego eliminarlo.

Una solución mejor se obtiene si se desacoplan las dependencias que dan origen a tantas subclases: la forma del elemento, la tipología estructural y las funciones de forma.

Teniendo en cuenta su forma, los elementos pueden ser uni, bi o tri dimensionales. Los elementos unidimensionales consisten en una arista con dos nodos como contorno, los bi dimensionales en una cara con aristas como contorno y los tri dimensionales, en un cuerpo con caras como contorno, esto es, los elementos unidimensionales aparecen en la formación del contorno de elementos bidimensional y estos en la de los tridimensionales.





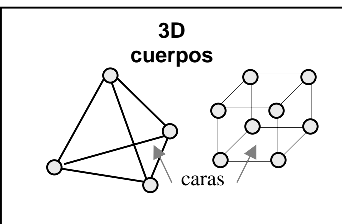


Figura 6.34 Elementos uni, bi y tridimensionales

La forma del elemento puede ser lineal o curvo para los unidimensionales, triangular o rectangular para los bidimensionales y tetraédrico o hexaédrico para los tridimensionales. Para cada una de estas formas la cantidad de nodos perteneciente a la frontera es fija y conocida.

Las funciones de forma pueden ser lineales, cuadráticas, cúbicas, etc. y son instancias de la clase Función, cuyo comportamiento ya fue establecido. Según la función de forma elegida, aparece la necesidad de nodos adicionales interiores: por ejemplo, una función de forma cuadrática introduce un nodo interior en la arista, una cúbica, dos nodos interiores. La cantidad de nodos interiores es conocida para cada función de forma.

La tipología estructural se subclasifica según el modelo de comportamiento simplificado adoptado: deformación plana, tensión plana, flexión, flexión con corte, etc. y para cada caso define los grados de libertad a considerar y la expresión general de rigidez, flexibilidad, tensión, etc.

Para modelarlo aplicamos el pattern ***Type Object*** [Johnson96] que desacopla instancias de sus clases evitando una explosión de subclases y permite crear nuevas “clases” dinámicamente en tiempo de ejecución. Como se explicara anteriormente, consta de dos clases: una que representa los objetos (Class) y otra que representa sus tipos (TypeClass). Cada objeto (Object) tiene un puntero o referencia a su correspondiente tipo (TypeObject) y delega alguna de sus responsabilidades a él.

La clase *TipologíaEstructural* de la figura 6.35 está representando el *TypeClass* del pattern y *Elemento* al *Class*. Ambas son subclasificadas independientemente.

TipologíaEstructural es responsable de proveer la expresión general de rigidez, flexibilidad, tensión, etc. y los grados de libertad por nodo. Una instancia de una clase concreta de tipología actuaría como superclase de elemento. Los clientes de objetos de la clase Elemento no necesitan estar enterados de la separación entre el objeto y su tipo: efectúan los requerimientos al elemento y éste decide cuáles delega a su tipología.

Elemento es subclasificada en los tipos *1D*, *2D* y *3D* y sus subclases de forma (rectangular, triangular, etc.). Cada elemento conoce las funciones de forma con que representa la distribución de sus desplazamientos, deformaciones y tensiones y conoce también su tipología estructural. En función de las coordenadas de sus nodos de contorno, puede calcular las coordenadas de los nodos interiores.

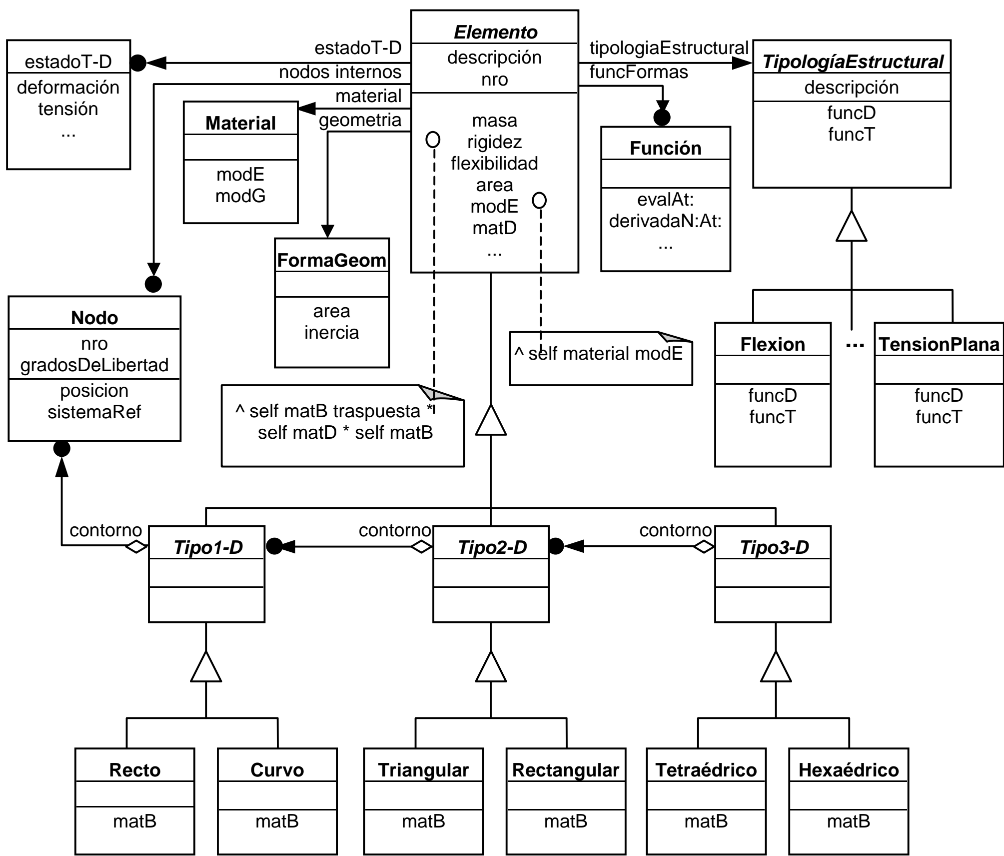


Figura 6.35 Jerarquía de Elemento

Para proveer la información requerida por el Análisis, como la matriz de rigidez, flexibilidad, masa y amortiguación, Elemento necesita reunir y combinar datos que le aportan su material, su forma geométrica, su tipología estructural y sus funciones de forma. Parte del comportamiento puede ser factorizado para que sea resuelto en las clases más altas. Por ejemplo, la matriz de rigidez puede calcularse en función de la matriz constitutiva D y la matriz de deformación elemental B como la integral sobre el elemento de $B^T \cdot D \cdot B$. Este producto puede ser planteado en la clase más abstracta, aplicando el patrón de diseño **Template Method** ya descripto,

y delegando parte de la evaluación en la subclase. Como D depende de la tipología estructural y de las características geométricas y mecánicas también pueden ser resueltas en la clase más abstracta. En cambio como B depende de las funciones de forma en que se distribuyen los desplazamientos y del tipo de elemento debe ser resuelta en cada clase concreta. Para la evaluación de la matriz B se necesita conocer las derivadas de las funciones de forma y por ser éstas instancias de la clase Función, saben hacerlo.

Cuando se crea un elemento debe indicarse su tipología estructural. Varios elementos pueden compartir la misma instancia de TipologíaEstructural, de material, de funciones de forma. Por ejemplo: entre un elemento de tipo "Rectangular Langragiano cúbico de 16 nodos" y otro "Rectangular Serendípeto cuadrático de 8 nodos" para una tipología estructural de flexión plana la diferencia reside en la función de forma y la cantidad de nodos internos. Ambos son subclases de Rectangular y pueden compartir el material, la geométrica, y la tipología estructural.

6.6.3 Pieza estructural

Una pieza estructural puede ser representada por un elemento o un conjunto de elementos que interactúan. Para modelar la estructura, nos basaremos en el pattern ***Composite*** [Gamma+95] que permite componer objetos en estructuras de árbol para representar jerarquías de partes y tratar en forma uniforme a los objetos simples (*Leaf*) y compuestos (*Composite*), siendo la superclase de ambos *Component*. La clase Elemento anteriormente descripta representa el objeto simple (*Leaf*) y la clase ElementoCompuesto representa el objeto compuesto (*Composite*) siendo la superclase de ambos PiezaEstrutural (*Component*).

Esta forma de tratarlos permite componer la estructura a través de sus piezas estructurales y éstas a través de su discretización. La discretización es necesaria para facilitar el análisis estructural y las piezas son necesarias para el diseño y verificación de los estados de tensión y de deformación.

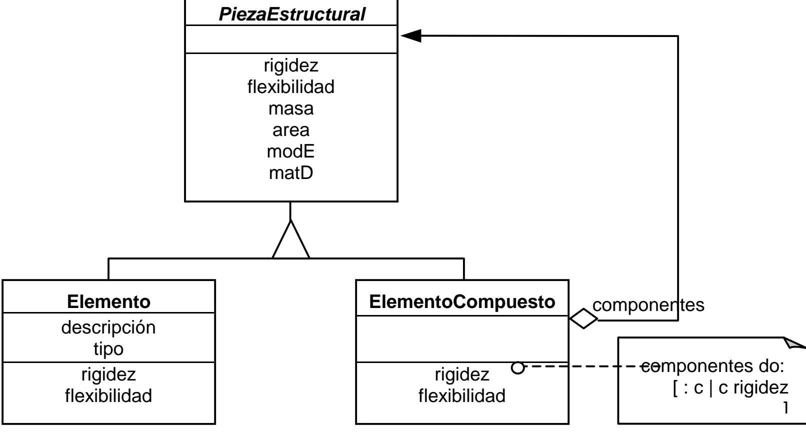


Figura 6.36 Pieza Estructural

6.7 Métodos de análisis y diseño estructural

A diferencia del paradigma estructurado en que los métodos de análisis y de diseño son resueltos por funciones o subrutinas, generalmente complejas, en el paradigma orientado a objeto, puede ser representado por una clase, con su propia estructura interna y métodos.

6.7.1 Análisis Estructural

En los problemas de Ingeniería no siempre es posible obtener soluciones matemáticas rigurosas. Cuando los problemas implican propiedades de materiales, distribución de cargas y condiciones de contorno complejas, es necesario introducir simplificaciones o idealizaciones para reducir el problema a una solución matemática que sea capaz de dar resultados aceptables desde el punto de vista de la seguridad y economía. El nexo entre el problema físico y la posible solución matemática se obtiene con el *modelo matemático*, que es una manera de designar simbólicamente al sistema idealizado de sustitución que incluye todas las simplificaciones impuestas al problema físico. Un mismo problema físico puede ser representado aceptablemente por diferentes modelos matemáticos que lo simplifican.

La mayoría de las estructuras de ingeniería civil normalmente se estudian estáticamente. Sin embargo, hay varias situaciones donde el comportamiento y los efectos dinámicos necesitan ser considerados, como por ejemplo: Viento en edificios altos, terremotos, estudios de impacto, comprobación no-destructiva.

Estas situaciones son estudiadas bajo los lineamientos del análisis estático para la primera y del análisis dinámico para la segunda. En cada uno, existen diferentes métodos de análisis que abarcan el estudio de las distintas tipologías estructurales.

Sin embargo, los diferentes tipos de análisis tienen un esquema semejante de resolución: consisten en resolver un conjunto de sistema de ecuaciones que plantea el equilibrio de fuerzas, la compatibilidad de los desplazamientos y las condiciones de contorno.

Por ejemplo, en un análisis estático de estructuras por el método de la rigidez, la primera ecuación a resolver tiene la forma **{f} = [K] {u}** donde {f} es el vector de fuerzas nodales de la estructura, [K] es la matriz de rigidez total y {u} el vector desplazamiento que representa las incógnitas del sistema, mientras que en el análisis dinámico toma la forma general: **{f} = [K] {u} + [C] {u'} + [M] {ü}** donde {f} es el vector de fuerzas nodales de la estructura y es dependiente del tiempo, [K] es la matriz de rigidez total, {u} el vector desplazamiento, [C] es la matriz de amortiguación, {u'} es la velocidad (derivada primera de {u} respecto del tiempo), [M] es la matriz de masa y {ü} es la aceleración (derivada segunda de {u}).

Resuelto el sistema se puede calcular otras magnitudes de interés como las deformaciones, tensiones, esfuerzos y reacciones, algunas de las cuales pueden ser resueltas de la misma forma en todos los métodos de análisis.

La forma que toman las matrices y vectores de cada ecuación en las diferentes tipologías varía, pero puede obtenerse con un procedimiento común: ensamblando el aporte respectivo de cada elemento.

El esquema de resolución puede ser diseñado 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.

De esta manera el esqueleto de la resolución de varios algoritmos puede estar en las clases más altas de la jerarquía quedando algunas operaciones a definirse en las subclases concretas.

La responsabilidad principal del objeto análisis estructural es determinar los esfuerzos y deformaciones en la estructura. Para ello, necesita preparar los datos que intervienen en las ecuaciones de equilibrio y compatibilidad, plantear las ecuaciones y resolverlas. Cada tipo de análisis conoce qué ecuaciones debe plantear. Para la solución matemática, requiere la colaboración de las clases: matrices, vectores, sistema de ecuaciones, funciones, métodos numéricos. Para preparar las matrices que intervienen en los sistemas de ecuaciones, necesita que cada elemento de la estructura aporte su contribución.

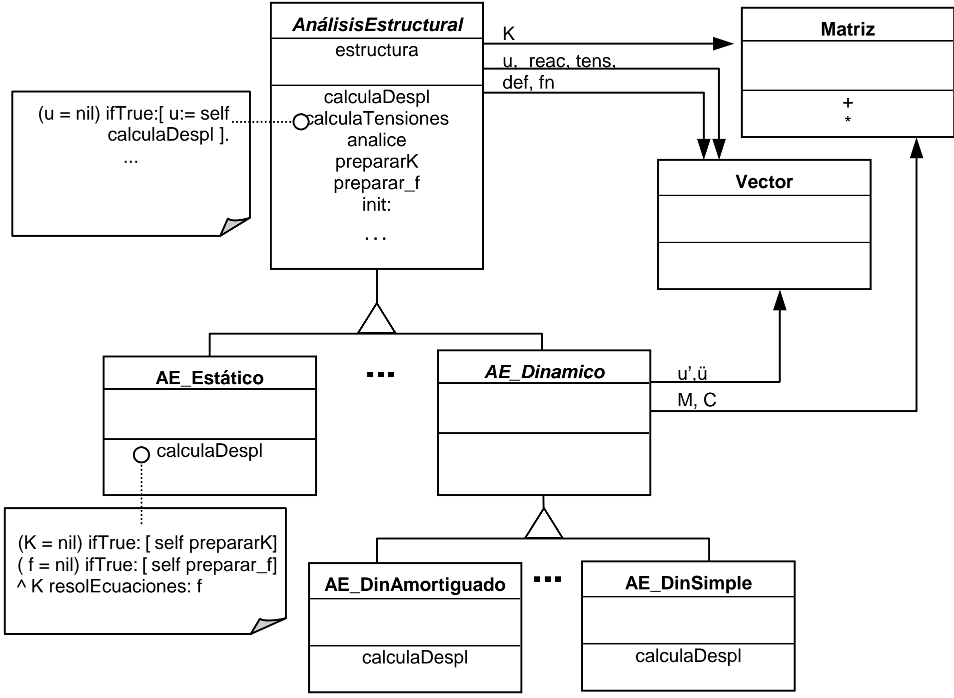


Figura 6.37 Análisis Estructural

El aporte de un elemento depende de la geometría y material del mismo y de la tipología del problema (tensión plana, flexión de vigas, placas delgadas). Si el elemento es capaz de dar correctamente su aporte, la tarea del Análisis estructural se ve notablemente simplificada.

Una vez planteadas y resueltas las ecuaciones, debe encargarse de que cada elemento actualice su estado de tensiones y deformaciones.

6.7.2 Diseño Estructural

Al igual que en el análisis los diferentes tipos de diseño tienen un esquema semejante de resolución que puede ser factorizado en una clase abstracta, dejando que las concretas, definan o redefinan algunas operaciones.

La responsabilidad fundamental es verificar que las tensiones y deformaciones de la pieza estructural se encuentren dentro de los límites fijados por las normas vigentes para el material y determinar las propiedades geométricas (sección, inercia, etc.) mínimas que permitan cumplir con las normas.

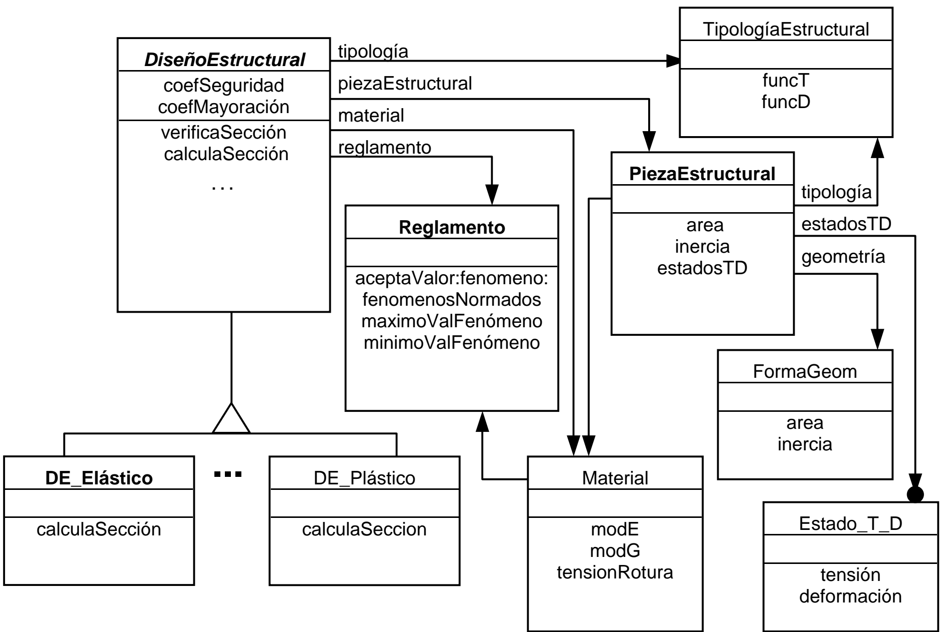


Figura 6.38 Diseño Estructural

Para ello requiere la colaboración de Reglamento, Material, TipologíaEstructural y PiezaEstructural. Se inicializa con la pieza estructural, el material y tipología de la misma y el reglamento asociado a ese material. La pieza estructural colabora con el diseñoEstructural brindado información sobre sus características geométricas y sus estados de tensión-deformación, los cuales fueron actualizados por el análisis

estructural. El material colabora brindando información de sus características mecánicas. La TipologíaEstructural brinda las expresiones genéricas de tensión y de deformación a partir de las cuales puede redimensionarse la pieza estructural si no cumple con alguna norma del reglamento. El reglamento aporta las normas que deben cumplirse y los coeficientes de seguridad y mayoración requeridos.

Capítulo 7

Un modelo arquitectural OO para aplicaciones en el dominio I.E.

7.1 El modelo arquitectónico

El objetivo primario del diseño arquitectónico es presentar una estructura del sistema modular, flexible y extensible y representar las relaciones de control entre las partes. Para alcanzar esta meta las dependencias entre objetos debe ser minimizada y claramente definida.

Las aplicaciones del dominio estructural requieren interactividad con el usuario. Aún aquellas que operan secuencialmente con una entrada, proceso y salida, obligan al usuario, ejecutarlas varias veces, con sucesivas modificaciones en la entrada de datos, antes de alcanzar resultados satisfactorios.

La entrada de datos es la tarea más tediosa, pero puede facilitarse enormemente utilizando interfaces gráficas (GUI). Un editor gráfico es la mejor solución para aplicaciones en este dominio. El editor deberá contar con funciones que permitan dibujar la estructura, definir los estados de carga, material, etc. El editor no debiera ser el único mecanismo de ingreso de datos, ya que a veces se cuenta con información en archivos.

La salida también requiere representación visual, y el usuario puede necesitar verla en diferentes formas.

Un patrón arquitectónico que provee una organización estructural muy apropiada para sistemas interactivos es el **Model-View-Controller** (MVC) [Buschmann+96] que divide una aplicación interactiva en tres componentes: el **modelo** que contiene el corazón de la funcionalidad y los datos; la **vista** que despliega información al usuario; y el **controlador** que maneja la entrada del usuario. El modelo es independiente del comportamiento de la entrada y de las representaciones específicas de salida. Cada vista tiene asociada un controlador que recibe la entrada, generalmente como eventos producidos por movimiento del mouse, activación de botones o entrada desde teclado. Los eventos son convertidos en requerimientos de servicios que se envían o al modelo o a la vista.

La interfaz de usuario queda representada con la vista y el controlador, permitiendo encapsular la dependencia de plataforma de modo que los cambios no afecten al corazón de la aplicación. El MVC también permite manejar diferentes vistas del mismo modelo, abriendo y cerrando vistas dinámicamente, y mantenerlas sincronizadas. Un modelo tiene uno o más pares de asociaciones vista-controlador.

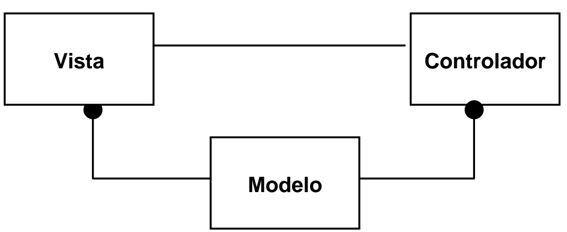


Figura 7.1 Estructura del MVC

Esta solución permite cambiar un subsistema sin causar mayores efectos en los restantes. Por ejemplo, se podría pasar de una interfaz no gráfica a otra gráfica sin modificar el modelo. También se puede adicionar soporte para nuevos dispositivos de entrada sin afectar la vista y el modelo.

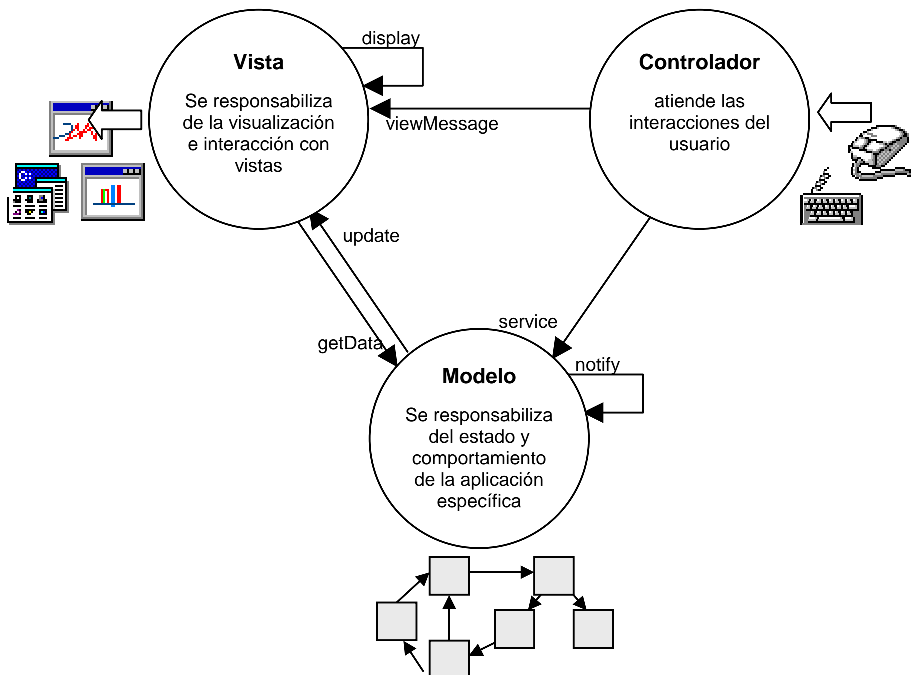


Figura 7.2 Componentes del MVC

Cuando se produce una entrada de usuario, el controlador maneja el evento, lo interpreta y solicita el servicio asociado al modelo. El modelo realiza el requerimiento, si esto provoca un cambio en sus datos, debe notificar a las vistas y controladores registrados con el mecanismo de propagación de cambios llamando a sus procedimientos de actualización. Cada vista solicita los datos que han cambiado y los despliega en la pantalla. Cada controlador recupera datos del modelo para activar o desactivar ciertas funciones, habilitar un menú, por ejemplo.

7.2 El modelo

La esencia de las interfaces gráficas (GUI) son los objetos gráficos. Estos objetos tienen la habilidad de saber dibujarse y de comunicarse con el usuario para permitirle cambiar sus características. Varios de ellos representan objetos del dominio estructural, tales como cargas, nodos, elementos. Esta representación se puede lograr agregando funcionalidad para su representación a los objetos involucrados o diseñar los objetos gráficos con una total independencia de los objetos representados. En la primera solución cada objeto es responsable de sus cálculos y de su representación gráfica. En la segunda, se requieren mensajes adicionales que permitan la comunicación entre los módulos, desde los objetos gráficos hacia los representados en orden de recuperar o actualizar datos, pero se mantiene una completa independencia que facilita el reuso, evolución y mantenimiento de las clases.

Los tipos de problemas que se presentan en Ingeniería Estructural tienen una secuencia operacional que ha sido mostrada en la figura 5.8, conformada por tres componentes: la definición del problema estructural compuesto por la estructura y cargas exteriores, el cálculo de esfuerzos y deformaciones realizado por el análisis estructural, la verificación de la estructura y propuesta de modificación de características geométricas realizada por el diseño estructural.

En correspondencia con lo antedicho, el modelo queda compuesto por cuatro componentes separadas que interactúan y colaboran entre sí: el **Editor Gráfico** que es responsable de la edición y modificación del dibujo que representa la estructura y las cargas, el **Problema Estructural** que consiste en la definición de la estructura a analizar, es decir su geometría, propiedades mecánicas de los materiales, condiciones de contorno y estados de carga; el **Análisis** que es la responsable de resolver el problema (determinar los esfuerzos, tensiones y deformaciones) de acuerdo con el modelo matemático adoptado y el método elegido; y el **Diseño**, que es responsable de las verificaciones de acuerdo con los reglamentos, el material y los estados de tensión y deformación alcanzados.

Tanto el EditorGráfico como el ProblemaEstructural tienen asociados pares de vista–controlador, mediante los cuales el usuario podrá ir “creando” la estructura y definiendo las características propias de su problemática.

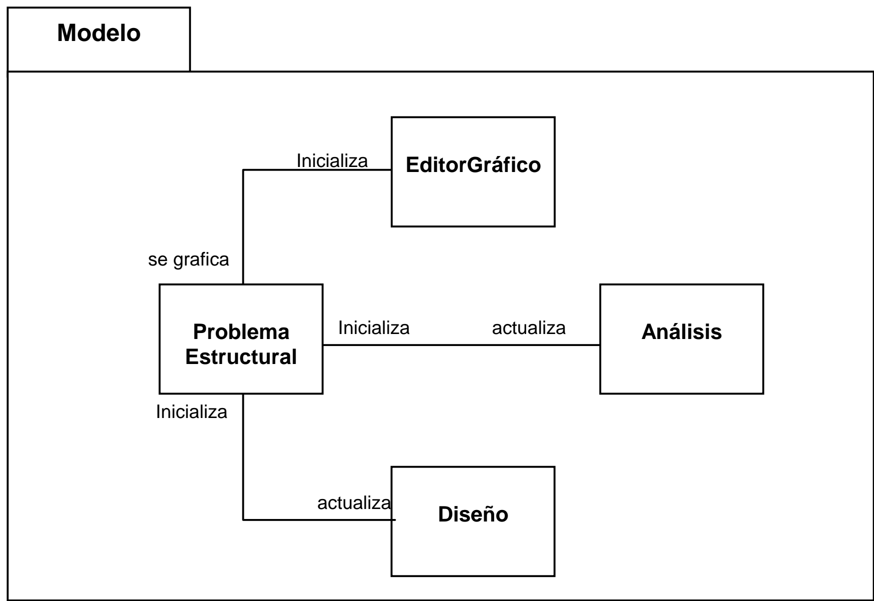


Figura 7.3 Componentes del Modelo

7.2.1 Editor gráfico

El editor gráfico, utilizado especialmente para facilitar el ingreso y visualización de datos, podría desarrollarse especializando algún framework existente. Se trata de un editor gráfico que permita dibujar la estructura y las cargas a través de líneas, nodos y aristas, definir caras, superficies y volúmenes y distintos tipos de apoyos y vínculos. También requiere tener funcionalidad para animación de modo que pueda poder visualizar las deformaciones de la estructura.

El EditorGráfico tiene muchas responsabilidades, entre ellas, responder a las opciones del menú que se muestra cuando el usuario clickea sobre una selección, agregar, modificar y quitar figuras, animar dibujos, realizar zoom. Si una figura cambia su aspecto entonces el Dibujo debe notificar a las Vistas dependientes. Indudablemente, son tareas comunes a todos los editores gráficos, por lo que nos concentraremos en los aspectos que lo especializan para el ambiente de Ingeniería Estructural.

Las figuras que maneja el editor representan objetos del problema estructural, tales como cargas, nodos, elementos. Los objetos gráficos resultantes pueden ser objetos simples o complejos y es necesario poder manejarlos uniformemente. Los simples o primitivos definen las formas básicas a partir de las cuales se crean las demás. Los objetos complejos se forman agrupando tanto objetos simples como compuestos.

La mejor manera de organizar estas clases es aplicando el patrón **Composite** [Gamma+95] ya descripto.

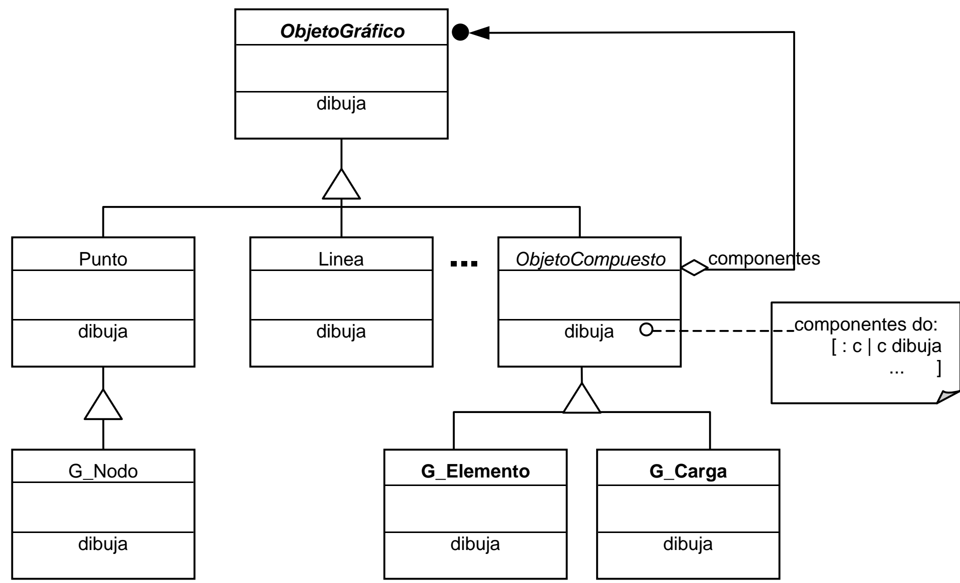


Figura 7.4 Objetos Gráficos

Si se trata de un framework existente, ya tendrá una jerarquía de objetos gráficos que habrá que especializar si fuera necesario. Otros objetos del problema estructural, como el estado de tensiones o de deformaciones, también necesitarán visualizarse. Determinados atributos o comportamiento de uno se convierten en atributos o comportamiento del otro. G_Nodo, G_Elemento y G_carga de la figura deben interactuar con las respectivas clases representadas: Nodo, Elemento y Carga.

Esto requiere un gran conocimiento mutuo entre las interfaces del editor y del problema estructural. Para mantener independencia entre ellos, de modo de facilitar reuso de ambas clases con otros fines o de poder usar un editor existente, se aplica el patrón de diseño **Adapter** [Gamma95]. El patrón Adapter convierte la interfaz de una clase en una interfaz apta para el cliente. Permite que las clases con interfaces incompatibles trabajen juntas.

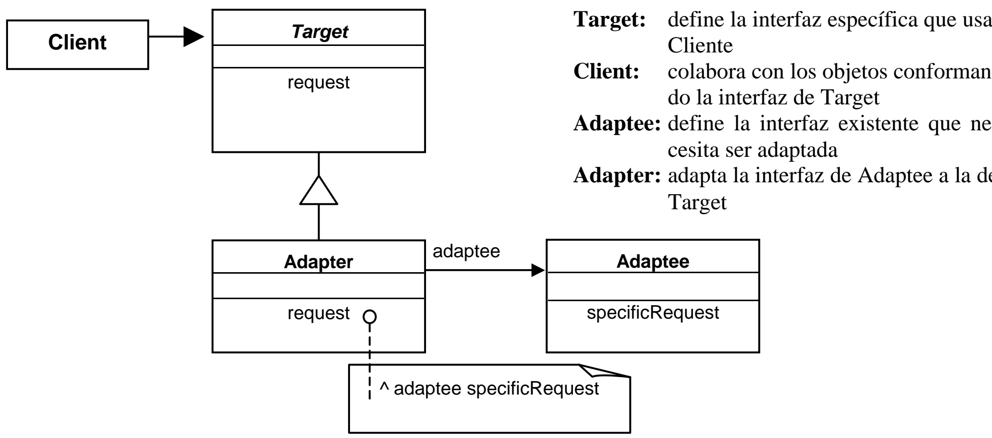


Figura 7.5 Adapter Design Pattern

7.2.2 Problema Estructural

El objeto Problema Estructural representa la estructura a analizar, es decir su geometría, propiedades mecánicas de los materiales, condiciones de contorno y estados de carga: fuerzas exteriores, tensiones, deformaciones.

Está compuesto por la estructura que es una colección de piezas estructurales, los reglamentos y las hipótesis que es una colección de hipótesis de cargas. Como se describiera en 6.7, las piezas estructurales están compuestas por elementos quienes tienen conocimiento del material, características geométricas, tipología estructural y se encuentran vinculados por nodos quienes conocen sus propios grados de libertad. Las hipótesis de carga descriptas en la sección 6.5 asocian factores de combinación con estados de cargas y éstos mantienen asociaciones entre cargas y elementos o nodos. Como consecuencia, el problema estructural tiene conocimiento directo o indirecto de las cargas actuantes, los nodos, elementos, material, tipología estructural.

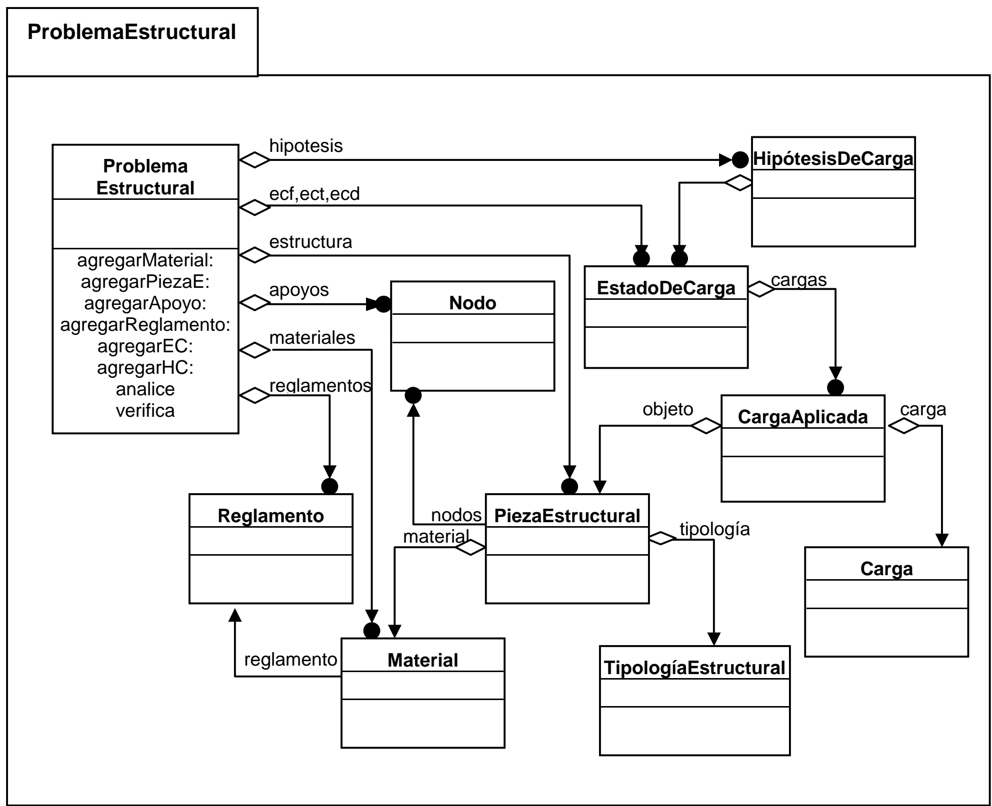


Figura 7.6 Problema Estructural

Además provee funcionalidad para interactuar con la interfaz de usuario: registra las vistas y controladores dependientes, notifica sus cambios y provee métodos de acceso a sus datos para las vistas puedan obtener la información a ser mostrada, brinda procedimientos para realizar acciones específicas de la aplicación, los cuales son invocados por los Controllers en nombre del usuario.

Se inicializa vía la interfaz de usuario, o vía el editor gráfico. Los objetos generados a partir del dibujo del usuario como nodos, elementos y cargas, actualizan sus valores vía el editor, mientras que reglamentos, materiales, tipología estructural pueden hacerlo en forma directa mediante un comando enviado desde la interfaz. Algunos de estos objetos podrían tener una inicialización por defecto.

7.2.3 Análisis

Este módulo representa el proceso de análisis que se aplica. Su responsabilidad es obtener las propiedades del problema estructural, incluyendo las condiciones iniciales y cargas, preparar la matriz de rigidez y el vector de fuerzas, efectuar el análisis y actualizar al objeto problema estructural con los resultados obtenidos.

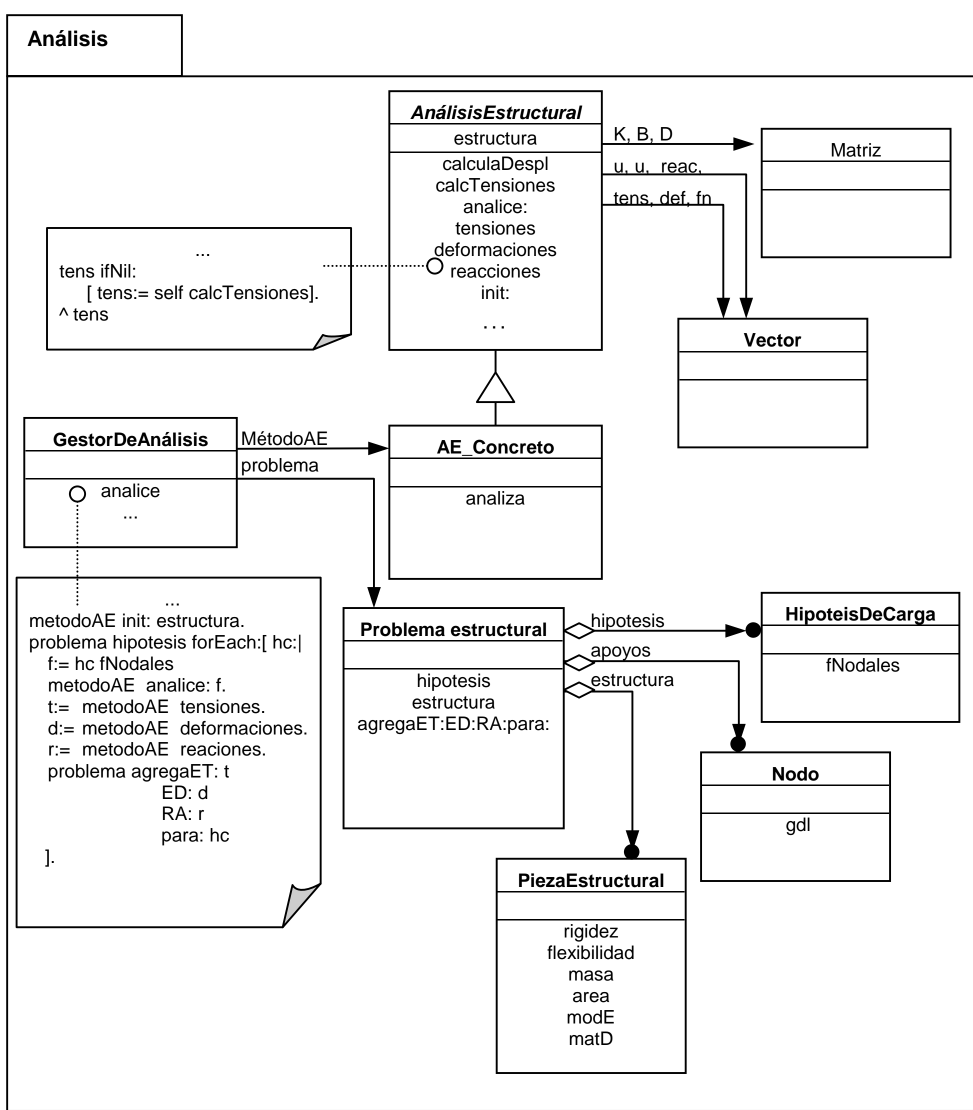


Figura 7.7 Módulo de Análisis

Está compuesto por el GestorDeAnálisis, quien es inicializado con la instancia del ProblemaEstructural y el método de análisis elegido por el usuario.

Requiere colaboración de objetos numéricos como matrices, vectores, tensores y delega el análisis propiamente dicho en el objeto AnálisisEstructural. Los grados de libertad de la estructura se transforman en incógnitas del análisis. Las cantidades son convertidas a unidades consistentes y transformadas en valores adimensionales, para aliviar las comprobaciones y no perjudicar la perfomance. Los resultados del análisis son valores adimensionales que deberán transformarse en cantidades representativas de desplazamientos o de tensiones de los nodos y elementos correspondientes.

El GestorDeAnálisis se encarga de preparar la estructura a analizar y para cada hipótesis de carga del problema, determina el vector de fuerzas nodales y solicita a su métodoAE el cálculo de tensiones, deformaciones y reacciones de la estructura, y actualiza el estado de tensión-deformación correspondiente a esa hipótesis.

7.2.4 Diseño

Este módulo representa el proceso de diseño que se aplica. Su responsabilidad es obtener las propiedades del problema estructural, geometría, material, y estado de tensiones y deformaciones que brindó el análisis para efectuar el diseño correspondiente.

Está compuesto por el GestorDeDiseño, quien es inicializado con la instancia del ProblemaEstructural y el método de diseño elegido por el usuario.

Cuando se le envía el mensaje *verifica*, el GestorDeDiseño se encarga de verificar cada piezaEstructural de la estructura del problema, delegando en el método de diseño elegido (métodoDE) la verificación del estado de tensiones y deformaciones más desfavorables. Si no cumple la verificación, solicita su dimensionamiento y las nuevas dimensiones son registradas para ser mostradas al usuario, quien decidirá o no, tomarlas en cuenta. La piezaEstructural mientras tanto mantiene sus dimensiones originales.

Cuando se abre la aplicación, una instancia de ProblemaEstructural, Gestor de Análisis y GestorDeDiseño son creadas.

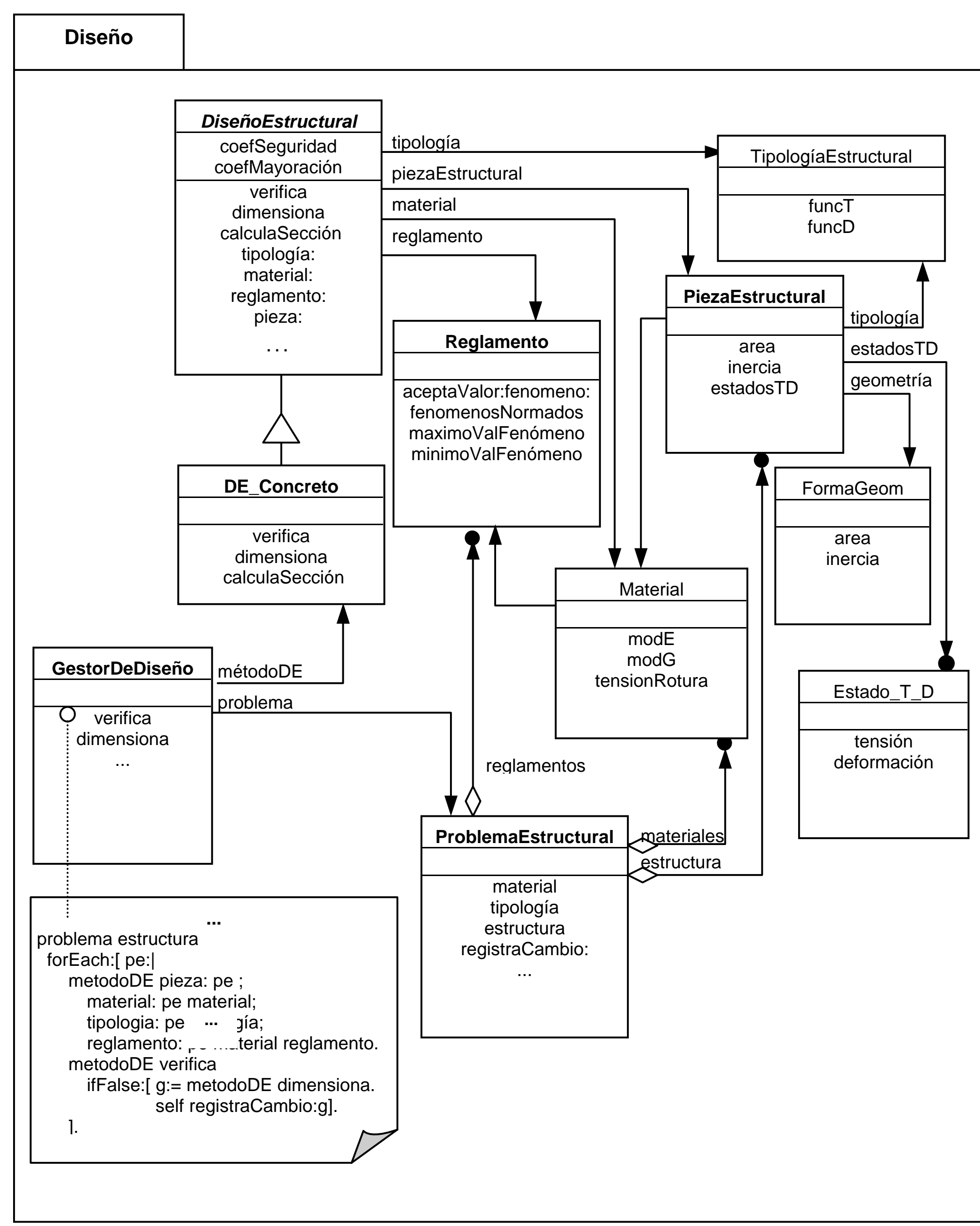


Figura 7.8 Módulo Diseño

7.3 La Vista

La vista muestra texto y gráficos que representan estados del modelo. Diferentes tipos de vistas son usadas para mostrar los datos y resultados del problema, gráficos 3D, gráficos 2D, tablas de valores, formularios, cuadros de selección.

Cada vista define el procedimiento para actualizar la información el cual es activado mediante el mecanismo de propagación de cambios. También define el procedimiento para dibujarse en el display.

Existen varias bibliotecas de clases reusables de vistas y controladores para los elementos más frecuentes en interfaces de usuario: menúes, botones, listas, cuadros de texto. Se pueden construir nuevas vistas en base a las existentes, combinándolas jerárquicamente aplicando el patrón Composite ya descripto.

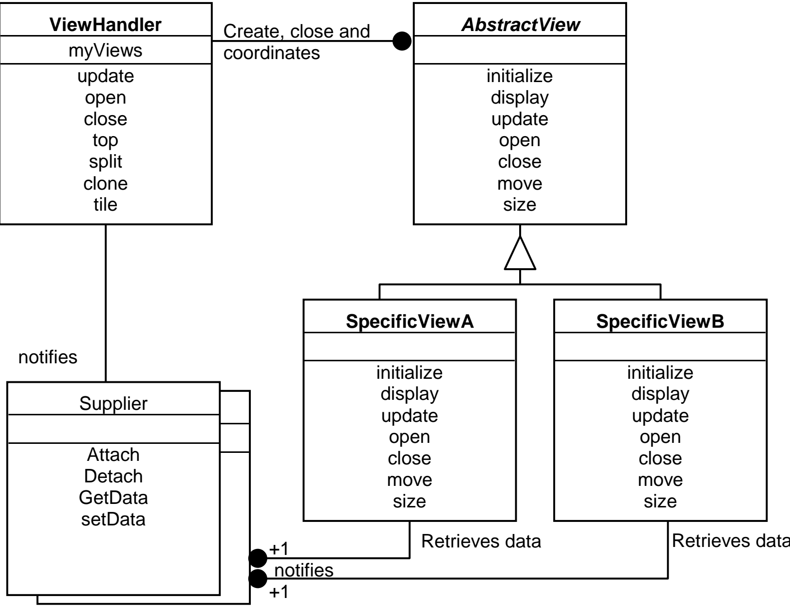


Figura 7.9 View Handler

Se pueden abrir y cerrar vistas dinámicamente. Puede tenerse una vista principal que cuando es cerrada, implique cerrar todas las demás y finalizar la aplicación, previa consulta al usuario.

El patrón **View Handler** ayuda a manejar todas las vistas de la aplicación, permite que los clientes abran, manipulen y dispongan de las vistas y también coordina sus dependencias y organiza sus actualizaciones. La componente principal del patrón es el ViewHandler que es responsable de abrir nuevas vistas, manipularlas y disponerlas. El AbstractView define el protocolo de sus subclases. SpecificView implementa las funciones, recupera los datos de los Suppliers, los prepara para desplegarlos y los presenta al usuario. Los componentes Suppliers provee una interfaz que permite recuperar y cambiar datos, y notifica de sus cambios a las vistas específicas o al View Handler. En el MVC, los Suppliers son parte del Modelo.

7.4 El Controlador

El controlador responde a la entrada del usuario, acciones del mouse, plaqueta digitalizadora, teclado. Recibe e interpreta los eventos usando un procedimiento dedicado: handleEvent.

Cada vista tiene su controlador, lo cual permite direccionar apropiadamente la solicitud de servicio asociada al evento detectado por el controlador. Los controladores de vistas que sólo muestran información, asocian los eventos de usuario con un handleEvent vacío.

Para mantener independencia entre el controlador y el modelo, de modo de proveer controladores reusables y evitar tener que modificar el manejador de eventos cuando se producen cambios en la funcionalidad del modelo, se aplica el patrón de diseño **Command Processor** [Buschmann96]. Este patrón separa un requerimiento de un servicio de su ejecución y está construido sobre el patrón de diseño Command [Gamma95]. Ambos patrones siguen la idea de encapsular los requerimientos como objetos.

Una clase abstracta AbstractCommand define el protocolo para ejecutar una operación, *do*. La clase concreta Command define el vínculo entre el objeto receptor (el Modelo, en el MVC) y una acción e implementa el *do* para invocar las operaciones correspondientes (*action*) sobre el receptor. La clase Command Processor mantiene los objetos comandos, inicia su ejecución y provee servicios adicionales relacionados con la ejecución del comando. Receiver conoce cómo ejecutar la operación asociada para llevar a cabo un requerimiento.

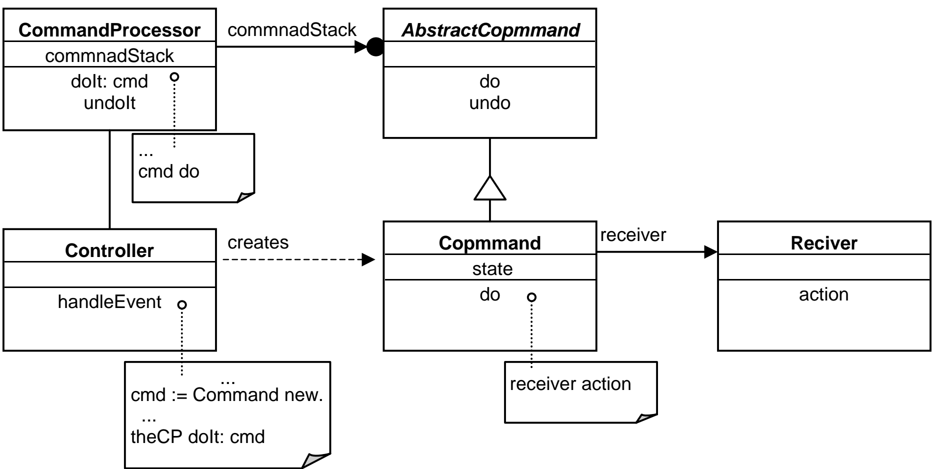


Figura 7.10 Command Processor

El controlador acepta el requerimiento del usuario, transforma el requerimiento en un comando (crea un objeto comando y establece su receptor) y se lo transfiere al commandProcessor quien hace que se ejecute y mantiene la información necesaria para los servicios adicionales.

7.5 Mecanismo de propagación de cambios

La comunicación general entre los componentes del MVC puede ser especificada con ayuda del patrón de diseño: **Publisher-Subscriber** para el mecanismo de propagación de cambios

El patrón **Publisher-Subscriber** [Buschmann+96] conocido también como Observer [Gamma95], define una dependencia uno a muchos entre objetos, de esta manera cuando un objeto cambia de estado, todos sus dependientes son notificados y actualizados. Un componente lleva el rol de *publisher* (*subject* en [Gamma95]). Todos los componentes dependientes de los cambios en el *publisher* son los *subcribers* (*observers* en Gamma+95) los cuales mantienen una referencia al *publisher.* El *publisher* conoce todos sus *subcribers*.

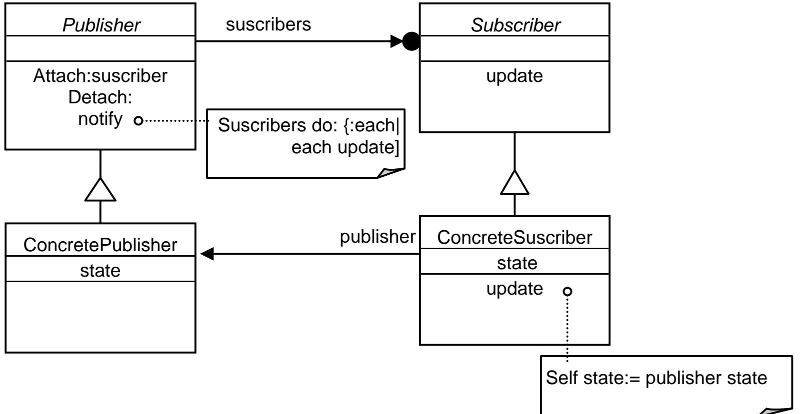


Figura 7.11 Publisher-Subscriber

En el MVC, el modelo es quien cumple el rol de publisher y las vistas y los controladores actúan como subscribers.

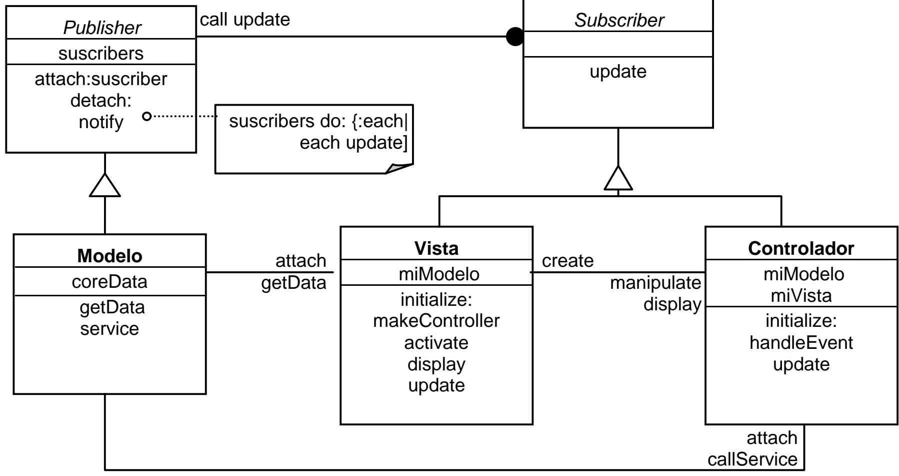


Figura 7.12 Esquema del MVC con el publisher-subscriber

7.6 Hacia un framework OO

Un framework de aplicación es un conjunto de componentes de implementación que da una base sobre la cual construir una aplicación. El desarrollador de la aplicación perfecciona el marco mediante especialización y lo extiende agregando nuevos componentes. Es una aplicación reusable, semi completa que puede ser especializada para producir una aplicación adaptada a una necesidad particular.

Consiste de un conjunto de clases abstractas y concretas, y define la interacción entre los componentes mediante el planteo de restricciones, herencia, polimorfismo y reglas informales de composición.

En el framework se especifican los aspectos del dominio de la aplicación que se mantienen estables en todas las aplicaciones de ese dominio y se provee mecanismos para poder expresar las variaciones que sean necesarias.

Las clases abstractas conforman los componentes principales de la arquitectura, representan el esqueleto que va a conservar toda aplicación del dominio que el framework modeliza y son los lugares de articulación: "hot spots" [Pree95], aquellos que comprometen los aspectos del dominio que pueden variar de una aplicación a otra, donde el framework debe proveer flexibilidad.

Un framework orientado a objetos para aplicaciones del dominio estructural, tendrá una arquitectura como la descripta anteriormente. Cada componente del Modelo: Editor Gráfico, Problema Estructural, Análisis y Diseño y el módulo matemático de soporte, constituye un subframework del soporte completo de nuestro modelo y sus mecanismos de interacción quedaron definidos en el modelo arquitectónico propuesto.

Los lugares de articulación ("hot spots") debieran quedar perfectamente definidos y el resto de la arquitectura debería ser una "caja negra" que un usuario no necesite conocer para poder usar el framework [Pree 95].

Los hot-spots provistos por el framework deben ser los suficientes como para proveer la mayor flexibilidad y simpleza posible en la arquitectura para que pueda ser ampliamente utilizada en el dominio sin que resulte terriblemente complicado utilizarlo. Encontrar este equilibrio no resulta fácil y requiere sucesivos refinamientos que aseguren y testeen que los hot-spots encontrados son los correctos.

Para poder cubrir cualquier tipo de aplicación en el dominio de la Ingeniería Estructural, el usuario del framework podrá agregar y especializar clases en los siguientes lugares previstos de articulación:

- **Métodos numéricos:** Pueden agregarse nuevos métodos de integración, diferenciación, interpolación, resolución de ecuaciones. La constante investigación en este tema provoca que aparezcan algoritmos que resultan ser más eficientes que otros en determinadas condiciones.

- **Representación de matrices y funciones:** Las representaciones más comunes estarán desarrolladas, quedando muchas otras pendientes.

- **Tipología estructural:** Los tipos de problemas y su correspondiente simplificación estructural son muy variados como para que todos puedan estar considerados. Este es otra “ventana” abierta que puede ser especializada y le da posibilidades al investigador estructuralista de “crear” nuevas tipología para probar comportamientos reales.

- **Tipos de elementos:** Los elementos más comunes estarán representados en clases concretas. Al igual que con la tipología, el ingeniero puede experimentar y probar con nuevos tipos de elementos, especializando las clases existentes.

- **Material:** Si bien los materiales aptos para estructuras más comunes pueden estar completamente desarrollados, queda abierta la posibilidad de incorporar nuevas variantes.

- **Métodos de análisis:** Los diferentes problemas estructurales son resueltos de diversas formas y hay una variedad importante de métodos de análisis. Un framework para este dominio, debe dejar prevista una ventana abierta para ello.

- **Métodos de diseño:** Habiendo dejado abierta la posibilidad de incorporar nuevas tipologías estructurales y materiales, se hace más imperioso aún, prever crecimiento en métodos de diseño.

Capítulo 8

Trabajos relacionados

La aplicación de diseño orientado a objetos para Ingeniería Estructural es relativamente nueva. En los últimos años han aparecido variados productos interactivos y con interfaz gráfica, que permiten realizar el análisis y diseño para determinados tipos de estructuras y material. Pero se trata, en su mayoría, de productos finales, sin posibilidad de adaptación y con manuales de uso como única documentación accesible.

Los trabajos más afines con lo desarrollado en esta tesis, y que muestran algo de su diseño, están referidos al análisis estático o dinámico por el método de los elementos finitos. A continuación se describen algunos de estos trabajos y se discuten las diferencias entre ellos y la aproximación descripta en esta tesis.

8.1 Graham Archer

Graham Archer [Archer96] en su tesis doctoral describe una arquitectura de software orientada a objeto para análisis dinámico por elementos finitos. Los componentes principales son: **Model** que está compuesto por un conjunto de elementos, nodos, cargas, restricciones y condiciones de borde; **Analysis** que realiza un análisis sobre un modelo, **Map** quien establece la comunicación entre Model y Analysis para evitar interdependencias entre estos últimos, **Constrain Handler** que maneja las restricciones del modelo de acuerdo con el modo requerido por el análisis y **Reorder Handler** reordena la numeración de las ecuaciones de acuerdo al criterio elegido por el análisis típicamente para reducir requerimientos de almacenamiento.

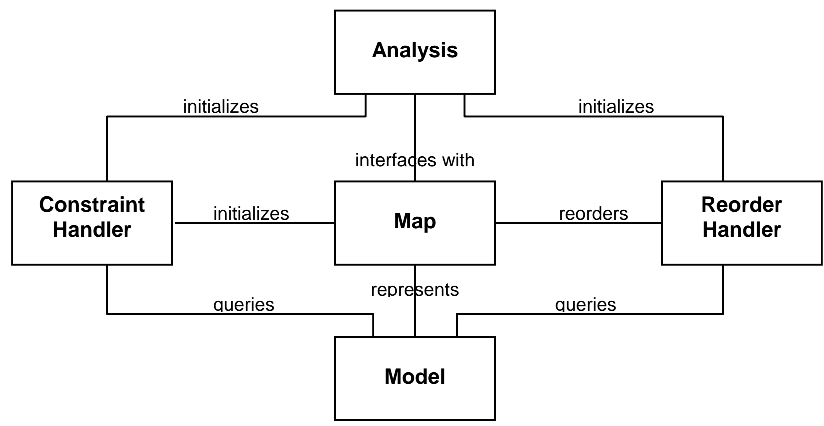


Figura 8.1 Diagrama ER de los objetos de alto nivel en [Archer96]

Los elementos están subclasificados de acuerdo con los diferentes tipos de elementos finitos, generando un gran número de subclases. Pueden ser cargados por los objetos *Element Load*, que son específicos para cada tipo de elemento. También puede tener un estado inicial dado por el objeto *Initial Element State*. Element Load y Initial Element State replican la clasificación de Element, es decir, para cada tipo de elemento existe el correspondiente tipo de element Load y de Initial Element State.

Cada elemento tiene un modelo constitutivo calibrado por un material, del que obtiene las relaciones acción-deformación. El modelo constitutivo es subclasificado en correspondencia con cada tipo de relación. El elemento y el modelo constitutivo utilizan objetos acción-deformación para trasmitir los valores de uno al otro. La jerarquía *Action/Deformation* espeja la jerarquía de *Constitutive Model* para proveer al modelo constitutivo con una medida de acción y deformación.

Los elementos están conectados por grupos de grados de libertad de los cuales el nodo es una subclase. Cada grado de libertad tiene un sistema de coordenadas asociado.

El sistema es capaz de modelar y simular el comportamiento estructural incluyendo efectos dinámicos y estáticos no lineales. Una innovación del diseño del sistema es la creación de un objeto (*Map*) el cual separa tareas del análisis de los detalles del modelo: separa la información basada en los grados de libertad del modelo de la gobernada por ecuaciones en el análisis. El análisis pone el énfasis sobre los algoritmos numéricos. El tipo de matriz usada para las matrices de propiedades del sistema en el objeto análisis puede ser modificado simplemente cambiando el tipo de *Matrix Handler* utilizado.

8.2 Jun Lu, Donald White y Wai-Fah Chen

Los autores [Lu+93] presentan una arquitectura flexible para un programa de análisis de elementos finitos. El corazón del sistema es un objeto del Ensamblador que realiza la transformación de propiedades del elemento de coordenadas locales a coordenadas globales y las ensambla en el conjunto. Los elementos proporcionan al Ensamblador sus sistemas de coordenadas locales y los nodos proporcionan el sistema de coordenadas estructurales.

Una amplia librería numérica fue desarrollada por los autores que incluye matrices Banda, triangular, esparcida, simétrico y perfil. También presentan una clase EquationSolver, cuyo propósito es aislar el propósito de los algoritmos generales de solución de la formulación de los elementos finitos.

Las principales clases presentadas son: La jerarquía de elementos, **Element**, que está clasificada teniendo en cuenta las diferentes formulaciones de elementos finitos sobre la base del método de los desplazamientos. Los elementos pueden proporcionar sus matrices de rigidez, masa y amortiguación. **Nodo** que posee una posición, un sistema de coordenadas y un conjunto de grados de libertad. **Material** que encapsula las relaciones constitutivas y proporciona la ley básica de tensión-deformación a los elementos. **Assembler** que ensambla los aportes de los elementos para producir la matriz de rigidez global. **Solver** que resuelve el sistema de ecuaciones resultantes.

No se describe ni se proporciona ningún esquema de estructuras ni de solución.

8.3 Butenweg C., Ebenau C., Gajenwski R., Trierauf G.

Los autores presentan un enfoque orientado a objeto para la construcción de software que puede aplicarse a programas centrado en ecuaciones diferenciales resueltas por el método de los elementos finitos y describen su programa BUB++ que está escrito en C++ [Butenweg+96].

Todos las clases del sistema necesarias para representar los elementos finitos están asociadas con la clase **Domain** que es la clase principal del programa y contiene funciones para la entrada, salida y cálculo. El flujo del programa es controlado por la clase **Program_control**.

La clase **Program_control** consiste de subclases conectadas por herencia, que representan los diferentes métodos de resolución: **Solve_linear**, **Solve_nonlinear**, etc. La conexión con la clase central **Domain** es realizada por un puntero. Dependiendo del tipo de cálculos que se definen en el archivo de la entrada del usuario se crea una instancia de una subclase de Program_control. La función virtual *run()* controla la ejecución del programa e invoca a la función específica en cada subclase.

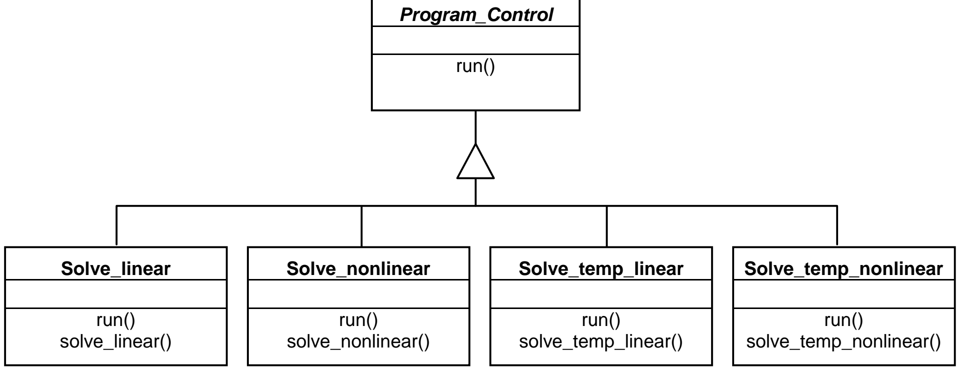


Figura 8.2 Jerarquía de Program_Control en BUB++

La clase de **Domain** se construye con el concepto de herencia múltiple. Una clase base virtual **fe_data** y tres subclases **fe_input, fe_solver, fe_output** (Fig. 8.3) que contienen las funciones responsables para las diferentes fases de ejecución del programa, actúan de superclases de Domain. Esta clase reúne todos los datos para el programa y contiene información general sobre el modelo de elemento finito, como la matriz de rigidez global o atributos definidos. Mantiene listas de nodos, de elementos, de constantes reales, estados de carga, de asociaciones de elementos y nodos cargados.

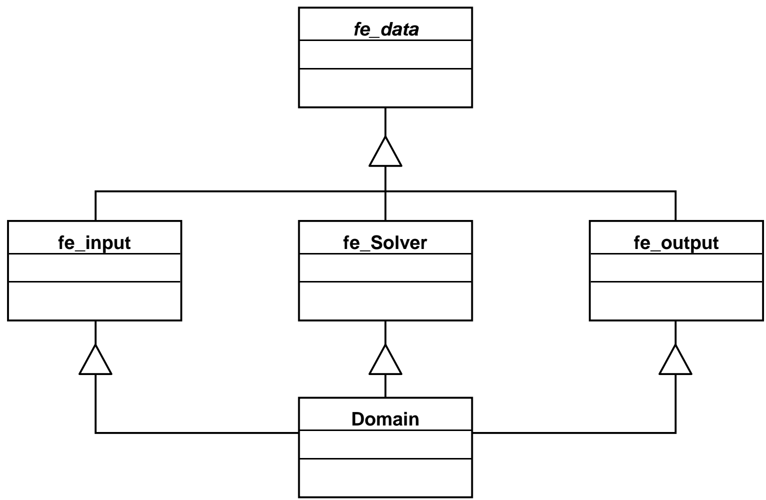


Figura 8.3 Clase Domain en BUB++

Los elementos están conectados por nodos y pueden tener cargas aplicadas. **Element** es una clase base abstracta y consiste de diferentes tipos de elementos conectados a través de herencia. El primer nivel de herencia contiene los tipos básicos de elementos, por ej. barras, cáscaras, platos. Cada uno de estas clases se extiende en niveles subsecuentes de herencia para aprovechar los datos y métodos comunes.

La clase **Node** está basada en múltiple herencia de las clases **Location** (define la posición por medio de tres coordenadas cartesianas globales) y **Degree_of_ freedom** (contiene la información sobre desplazamientos y rotaciones posibles y fijos). Algunos de sus atributos son el número del nodo, la lista de elementos unidos y, si es necesario, la matriz de rotación para el sistema de coordenada de ese nodo. Adicionalmente guarda una lista de nodos adyacentes conectados por elementos. Cada grado de libertad es un solo atributo. Ellos tienen números globales en la estructura entera y se guardan en un vector de enteros.

Load es la clase base que define las características comunes a los diferentes tipos de carga, los cuales se conectan por herencia. El primer nivel, distingue si son aplicadas a elementos o a nodos: **loads_on_elements** y **loads_on_nodes** y en un segundo nivel si son de superficie, repartidas, simples fuerzas, desplazamientos,

variaciones de temperatura. **Step_load** representa un estado de carga, y mantiene las asociaciones entre las cargas y elementos o nodos.

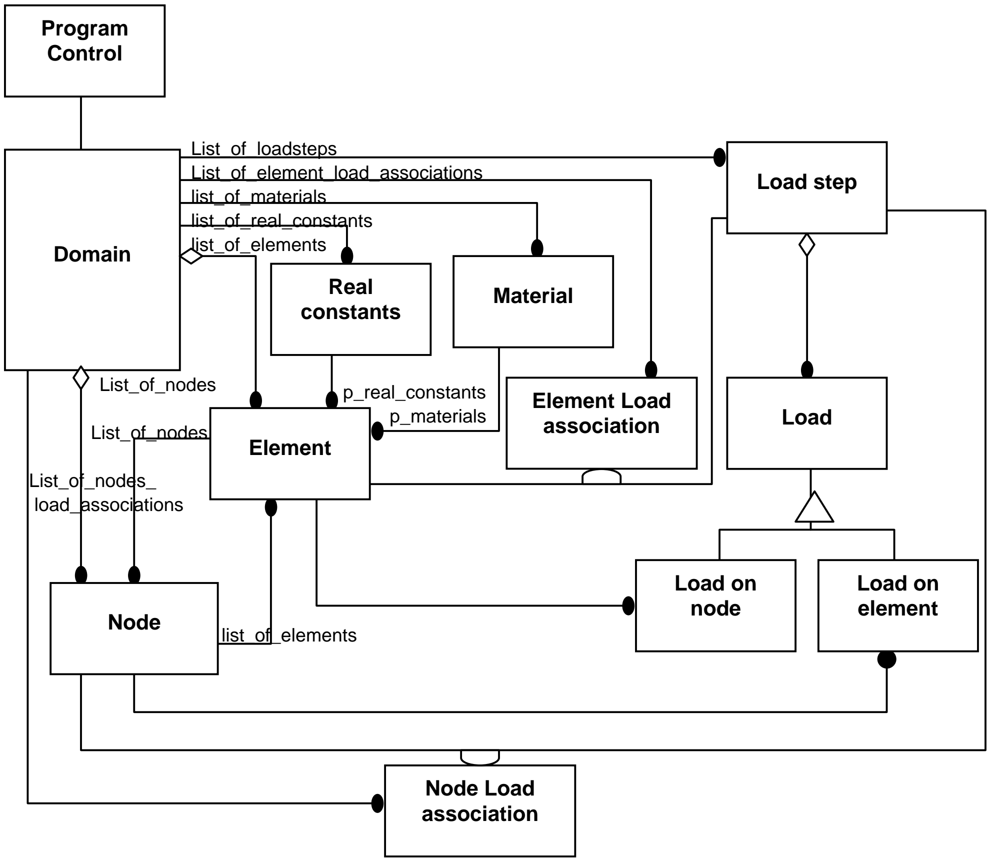


Figura 8.4 Estructura del programa BUB++

8.4 J. Drolet

El autor presenta en su artículo [Drolet96] la descripción de una herramienta para simplificar las simulaciones mediante elementos finitos. La representación matemática del problema es elegida como modelo de metáfora de software. Además de tipos de datos como matrices, vectores y tensores, introduce los tipos parámetros, incógnitas y condiciones de borde. Los vectores y matrices son derivados de una clase más general llamada Tensor que ha sido creada para forzar el chequeo del tipo de datos en operaciones que involucran vectores y matrices.

Los elementos y sus partes constituyentes: caras, aristas y nodos tienen significado geométrico y representación visual, pero no presenta más detalle que lo mencionado.

8.5 Discusión

Las arquitecturas anteriormente revisadas demuestran una amplia gama de vistas sobre el diseño de un programa orientado a objetos de elementos finitos. Un tema recurrente es el deseo de separar los distintos componentes del sistema entre sí.

Archer con su clase Map y Lu con su Ensamblador intentan desacoplar dependencias entre el modelo y el análisis. Tanto Lu como Drolet definen un conjunto de clases responsables de reflejar comportamientos matemáticos especiales que alivian el comportamiento de los objetos físicos pero no hacen referencia directa a estos últimos. Archer y Butenweg incluyen descripción de cargas y estados de cargas y las subclasifican según se apliquen a nodos o a elementos.

La arquitectura presentada por Butenweg se relaciona más estrechamente con los programas de elementos finitos tradicionales. Si bien utiliza los conceptos básicos del modelo orientado a objetos, estos no están plenamente aprovechados, como puede observarse en el caso de la jerarquía de Program_control de la figura 8.2, en que cada subclase nombra de diferente manera a la función que representa la resolución específica: *solve_linear(), solve_nonlinear()*, no haciendo uso adecuado del polimorfismo.

8.6 Aportes de este modelo

Trabajo con unidades: cualquiera sea el tipo de magnitud, es posible asociarla a una unidad de medida. El patrón presentado por Fowler es completado para magnitudes vectoriales. Se presenta una manera original de asociar unidades a matrices y vectores mediante la aplicación del Decorator Design Pattern.

Sistema de referencia: Para un conocimiento pleno del significado de una magnitud vectorial se requiere un sistema de referencia y tipo de coordenadas. Algunos autores lo han tenido en cuenta como atributo del nodo. F. Balaguer, S. Gordillo, F. das Neves [Balaguer+97] definen el patrón **Reference System** como una solución para resolver problemas en sistemas de información geográfica (GIS).

Abstracción del comportamiento matemático: Caracterización de una familia de clases para tratamiento de problemas algebraicos y funcionales: Matriz, Sistema de ecuaciones, Función y Métodos numéricos. Las operaciones para las que existen familias de algoritmos, se han modelado aplicando el patrón de diseño Strateggy. Las diferentes representaciones se modelaron con el patrón de diseño Bridge.

Abstracción de los conceptos del dominio estructural: Caracterización de una familia de clases que representan los conceptos propios del dominio estructural: Carga, estado de carga, pieza estructural, reglamento, análisis estructural, diseño estructural. Cada una de las clases intenta representar el verdadero comportamiento físico, minimizando el acoplamiento y las dependencias.

Arquitectura modular: Definición de una arquitectura modular y flexible, y que además tiene en cuenta la característica interactiva de las aplicaciones que abordan la problemática de la Ingeniería Estructural.

Capítulo 9

Conclusiones y trabajos futuros

Los frameworks de aplicaciones orientadas a objetos y los patrones de diseño ayudan a reducir el costo y mejorar la calidad del software al basarse en diseños probados e implementaciones para producir componentes reusables que pueden ser adaptadas para cubrir los requerimientos de una nueva aplicación. Contar con un framework de aplicaciones OO para el dominio estructural es una idea atractiva y prometedora.

En esta tesis ha sido presentado un original conjunto de modelos orientados a objetos aplicables al dominio de la I.E. Los conceptos del dominio han sido identificados y modelados enfatizando la reusabilidad a través de aplicar sistemáticamente patrones de diseño que conducen a soluciones flexibles, extendibles y modificables. Como resultado se obtuvieron un conjunto de microarquitecturas OO que representan los elementos claves en este dominio.

También ha sido definido un modelo de arquitectura para aplicaciones que abordan la problemática de la I.E., basado en las nuevas tendencias de diseño conducido por patrones, que integra y conecta esas microarquitecturas formando artefactos más complejos. La dependencia entre los componentes ha sido minimizada, de modo que puedan ser fácilmente reemplazados y/o reutilizados.

Finalmente, ha quedado descripto un framework OO de aplicaciones para este dominio en el que se han previsto posibles lugares de especialización y/o extensión (hot spots).

9.1 Contribuciones

El principal aporte de este trabajo consiste justamente en:

- Caracterización de los objetos fundamentales del dominio de la I.E.
- Aplicación de los principios de la Ingeniería de Software y de la Tecnología de Objetos, en su concepción actual, al campo de la I.E.
- Definición de un conjunto de microarquitecturas que modelan los principales objetos del dominio en las que se han aplicado sistemáticamente patrones de diseño que conducen a soluciones flexibles y modificables.
- Integración y articulación de esas microarquitecturas para conformar artefactos más complejos que compondrán el sistema total.

- Definición de una arquitectura OO para aplicaciones que abordan la problemática de la I.E., y descripción de sus componentes principales y los mecanismos de comunicación.

- Definición de un framework OO para Ingeniería Estructural estableciendo sus “hot spots” para extensión y especialización.

9.2 Futuros trabajos

Los frameworks OO constituyen el estado del arte para el reuso de abstracciones de diseño a gran escala, en un dominio de aplicación particular. Diseñar un framework no es una tarea fácil. Se requieren muchas iteraciones para obtener jerarquías sólidas. En algunas áreas, como en interfaces gráficas, ya han alcanzado la madurez suficiente. Como dije anteriormente, contar con un framework de aplicaciones OO para el dominio estructural, es una idea atractiva y prometedora. Este trabajo es sólo un comienzo. La inmediata tarea a seguir es la implementación y validación del modelo. Algunas otras futuras líneas de trabajo se enumeran a continuación:

- **Integración con otros minicomponentes**: En algunas áreas, existen muy buenos componentes probados y testeados, como en el área de los objetos numéricos. Facilitar la integración contribuye a lograr mejoras en calidad, productividad y costos.

- **Independencia de la plataforma:** Si bien en teoría podría garantizarse la compatibilidad en diferentes plataformas, en la práctica, la multiplicación de arquitecturas de hardware y sistemas operativos, y la coexistencia de diferentes estándares, demandan investigación y trabajo en esta línea para lograrlo.

- **Entrada y Salida:** Incorporar soporte completo para captura y despliegue de información para diferentes dispositivos y formatos.

- **Extensiones:** En el capítulo 7 se mencionaron lugares previstos para extensión del framework, los cuales abarcan especializaciones de métodos numéricos, representación de matrices y funciones, incorporación de nuevas tipologías estructurales, tipos de elementos, material y métodos de análisis y diseño.

Bibliografía

[Applenton 97] Brad Appleton
"Patterns and Software: Essential Concepts and Terminology" [HTML]
http://www.enteract.com/~bradapp/docs/patterns-intro.html, 1997

[Archer96] Graham Archer
"Objected-Oriented Finite Element Analisys". Tésis doctoral. [HTML]
http://ce.ecn.purdue.edu/~archer/PhD/PhD.html, 1996.

[Balaguer+97] F. Balaguer, S. Gordillo, F. das Neves
"Pattern for GIS Applications Design". PLOP 1997.

[Beck+94] Beck,K. ; Johnson, R
"Patterns Generate Architectures" presentado en ECOOP'94

[Budinsky+96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu
"Automatic code generation from design patterns" - Vol. 35, No. 2, 1996 - Object technology

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

[Butenweg+96] Butenweg C., Ebenau C., Gajenwski R., Trierauf G.
"Objected Oriented finite element computations: Requirements, model and implementation". [HTML]. http://www.omk.il.pw.edu.pl/~rgajewski/Publ.html.

[Campo+97] Marcelo R. Campo, Roberto Tom Price
Apuntes del curso: "Desarrollo de Frameworks Orientados a Objetos" - CACIC 97

[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.

[Coplien 94] James O. Coplien,
"Software Design Patterns: Common Questions and Answer", [HTML],
http://st.www.cs.uiuc.edu /users/patterns/patterns.html, 1994

[Drolet96] J. Drolet
"Toward a cross-platform finite element aplication framework: a tool to simplify finite element simulations". CSCE Conference, 1996.

[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

[Garrido+97] A, Garrido y G. Rossi.
"Using Patterns to Define a Framework's Architecture". First Conference on Using Patterns, 1997.

[Johnson 92] Johnson, R.
"Documenting frameworks using patterns", presentado en OOSPLA'92

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

[Kardestuncer75] Hayrettin Kardestuncer
Introducción al análisis estructural con matrices. Ed. McGraw Hill - 1975

[Lu+93] Lu J.; White D. W.; Chen W. F.
"Applying Objected Oriented Design to Finite Element Programing"
ACM-SAC, 1993.

[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. Ed. Prentice Hall Hispanoamérica S. A. 1997

[Mattsson+99] Michael Mattsson, Jan Bosch and Mohamed Fayad
"Framework integration: Problems, causes, solutions". Comunications of ACM, Vol. 42, Nro.10 Oct-99

[Meijers96] Marco Meijers.
"Tool support for object-oriented design patterns". Master's thesis, Utrecht University, 1996.

[Merritt90] Frederick Merritt
Enciclopedia de la Construcción - Grupo Editorial Océano - 1990

[Merritt92] Frederick Merritt
Manual del Ingeniero Civil – Ed. Mc. Graw Hill – 1992

[Oñate95] Eugenio Oñate
Cálculo de Estructuras por el método de Elementos Finitos - CIMNE - 1995

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

[Pree97] Wolfgang Pree
"Essential Framework Design Patterns", Object Magazine 7, pp. 34-37.

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

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

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

[Salviano 97] Clenio F. Salviano,
"Aplicação de Design Patterns para Reuso de Software Orientado a Objetos", Anais do II ERI, Piracicaba, SP, 1997

[Schmidt 95] Douglas Schmidt,
"Experience Using Design Patterns to Develop Reuseable Object-Oriented Communication Software", Communications of ACM, Vol. 38, No. 10, October 1995.

[Schmidt+96] Douglas C. Schmidt, Ralph E. Johnson, Mohamed Fayad
"Software Patterns" - Communications of the ACM, Special Issue on Patterns and Pattern Languages, Vol. 39, No. 10, Oct 1996.

[Schmidt 98] Douglas C. Schmidt
"Using Design Patterns and Frameworks to Develop Object-Oriented Communication Systems" – www.cs.wustl.edu/~schmidt/ Washington University.

[Tuma74] j. Tuma y R. Munshi,
Análisis Estructural Avanzado. Ed. McGraw Hill, 1974.

[Vlissides 97] John Vlissides,
"Patterns: The Top Ten Misconceptions", Object Magazine, 1997.

[West84] Harry H. West
Análisis de estructuras - C.E.C.S.A 1994

Anexo

Notación y diagramas

Un modelo es una descripción completa de un sistema desde una perspectiva particular. Representa un aspecto de la realidad y se construye de modo que nos ayude a comprenderla. Los modelos se representan mediante diagramas y estos diagramas pueden reflejar aspectos estáticos o aspectos dinámicos.

Las microarquitecturas desarrolladas en este trabajo han sido representadas mediante diagramas de estructura estática siguiendo los lineamientos de la notación UML (Unified Modeling Language) definida en [Rational97].

Un diagrama de estructura estática muestra el conjunto de clases y objetos importantes que son parte de un sistema, junto con las relaciones existentes entre estas clases y objetos. Muestra, de una manera estática, la estructura de información del sistema y la visibilidad que tiene cada una de las clases, dada por sus relaciones con las demás en el modelo.

En la representación de aquellas abstracciones que corresponden a patrones reconocidos de análisis, de diseño, o arquitecturales se ha respetado la notación original tal como están representadas en la referencia bibliográfica indicada: [Fowler97], [Gamma+94] y [Buschmann+96]. Estos dos últimos utilizan la notación OMT (Object Modeling Technique).

A continuación se describen los elementos que intervienen en los diagramas y se da la notación correspondiente, distinguiéndose, cuando resulta necesario, la forma que toman en Fowler, OMT y UML.

A.1. Diagrama de clases

En estos diagramas, los elementos fundamentales son las clases y las asociaciones. Son construidos y refinados a través del desarrollo.

A.2. Diagrama de objetos

Son similares a los de clases, pero los elementos que intervienen son objetos (instancias de clases). Son construidos durante el análisis y diseño con el propósito de ilustrar datos y estructuras de objetos.

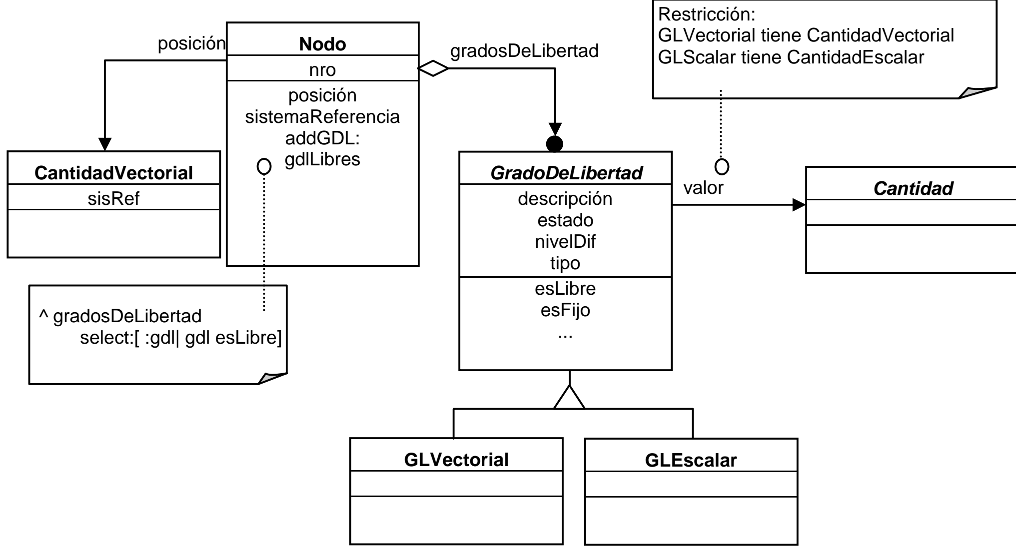


Figura A.1. Diagrama de clases

A.3. Clase

Las clases son los elementos fundamentales del diagrama. Una clase describe un conjunto de objetos con características y comportamiento idéntico.

Se representa por un rectángulo. Dependiendo el nivel de detalle que se desee mostrar, puede tener tres divisiones internas:

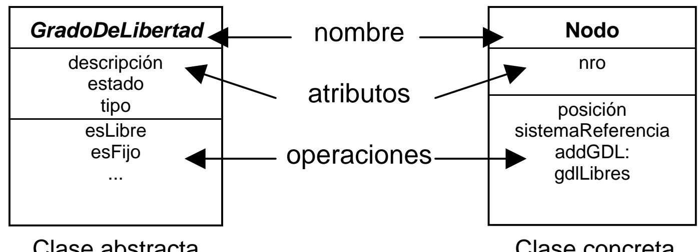


Figura A.2. Clase

El nombre no puede faltar y sirve para identificarla. Las clases abstractas son reconocidas en el diagrama porque su nombre figura en itálica.

Los atributos identifican las características propias de cada clase. Generalmente son de tipos simples, ya que los atributos de tipos compuestos se representan mediante asociaciones de composición con otras clases.

El conjunto de operaciones describe el comportamiento de los objetos de una clase. Se utilizó la sintaxis de Smalltalk. En Smalltalk, la indicación de que una operación lleva parámetros lo denotan la presencia de dos puntos (:) en el nombre de la operación. Por ejemplo addGDL: indica que requiere un parámetro.

A.4. Objeto

Representa una instancia. La representación gráfica difiere entre las notaciones OMT y UML y se ilustra en la figura siguiente.

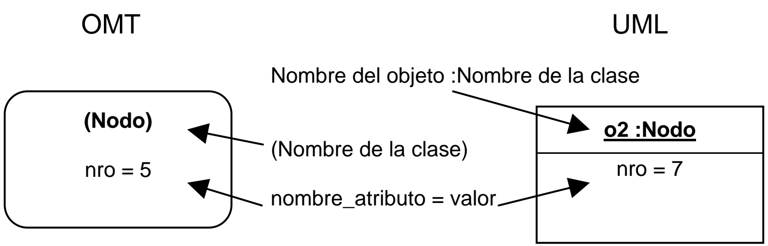


Figura A.3. Objeto

A.5. Asociación

Una asociación en general es una línea que une dos o más símbolos. Cada asociación puede presentar algunos elementos adicionales que dan detalle a la relación, como son el rol, la multiplicidad y el cualificador.

Rol: Identificado como un nombre al final de la línea, describe la semántica de la relación en el sentido indicado. Por ejemplo, la asociación entre Nodo y Cantidad-Vectorial de la figura A.1, recibe el nombre posición.

Multiplicidad: Describe la cardinalidad de la relación, es decir, la cantidad de objetos de una clase que se deben asociar con objetos de la otra clase. En la representación de la cardinalidad se utilizó la notación OMT (Object Modeling Technique), y las razones fueron exclusivamente debidas a efectos visuales: mantener la simbología propuesta por UML en forma proporcional al tamaño de gráficos utilizados, restaba claridad al dibujo.

Significado	Fowler	OMT	UML
a lo sumo uno			0..1
sólo uno			1
cero o más			*
1 o más			1..*

Figura A.4. Cardinalidad

Cualificador: es un atributo especial que reduce la multiplicidad efectiva de una asociación; se aplica en asociaciones uno-a-muchos o muchos-a-muchos.

A.6. Tipos de asociaciones

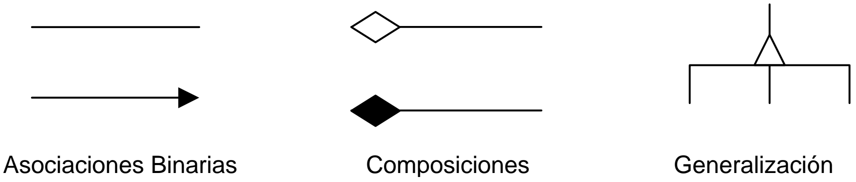
Asociación binaria: Se identifica como una línea sólida que une dos clases. Representa una relación de algún tipo entre las dos clases, no muy fuerte.

Composición: Es una relación "parte-todo" o "una-parte-de". Se denota dibujando un rombo del lado de la clase que contiene a la otra en la relación. Indica una pertenencia fuerte: se puede decir que el objeto contenido es parte constitutiva y vital del que lo contiene.

Un rombo relleno indica una asociación fuerte con dependencia existencial: el elemento dependiente desaparece al destruirse el que lo contiene y, si es de cardinalidad uno, es creado al mismo tiempo.

Un rombo hueco indica una relación de composición menos fuerte, una agregación.

Generalización: La relación de generalización denota una relación de herencia entre clases. Se representa dibujando un triángulo sin rellenar que conecta una superclase con sus subclases. La superclase se conecta mediante la línea de la parte superior del triángulo y las subclases mediante líneas a una barra horizontal asociada a la base del triángulo. La subclase hereda todos los atributos y mensajes descritos en la superclase. En el ejemplo de la figura A.1, *GradoDeLibertad* (superclase) es una generalización de GLEscalar y de GLVectorial (subclases).



Asociaciones Binarias | Composiciones | Generalización

Figura A.5. Tipos de asociación

A.7. Paquete

Un paquete es una forma de agrupar clases (u otros elementos en otro tipo de diagramas) en modelos grandes. Pueden tener asociaciones de dependencia o de generalización entre ellos. Se representa con un rectángulo con una solapa. En este ejemplo, no se está dando visibilidad a su contenido, pero podría mostrar las relaciones internas entre las clases que lo componen.

Figura A.6. Paquete

A.8. Nota

Es un comentario dentro de un diagrama. Puede estar relacionado con uno o más elementos en el diagrama mediante líneas punteadas. Pueden representar aclaraciones al diagrama o restricciones sobre los elementos relacionados. Se representa mediante un rectángulo con su borde derecho doblado.

En el ejemplo de la figura A.1 se encuentran dos notas: una que representa una restricción del atributo valor y otra que representa un ejemplo aclaratorio de la codificación de la operación gdlLibres.

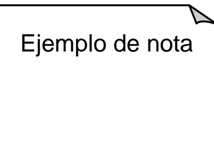


Figura A.7. Nota