Capítulo 2

Introducción a la Programación Orientada a Objetos

El desarrollo de aplicaciones basado en lenguajes propios de la Programación Orientada a Objetos (OOP) supone una serie de ventajas respecto de otros lenguajes de programación que no lo son. En la actualidad, la gran mayoría de los programadores profesionales, realiza su actividad con algún tipo de lenguaje orientado a objetos. Este tipo de lenguajes no es exclusivo de la plataforma .NET, ya que existen otras, de igual potencia y versatilidad, que reúnen los mismos requisitos (por ejemplo, la plataforma Java).

La fase previa a la programación consiste en una metodología para efectuar el análisis y la definición funcional de la aplicación, que constituye de esta forma la base de la codificación.

El objetivo del presente capítulo es múltiple; por un lado introducir al programador en alguno de los conceptos fundamentales de la OOP, acercarle a ellos definiendo ejemplos desde el punto de vista mecánico; y por otro, definir una metodología para el desarrollo de aplicaciones sobre la que se basa la aplicación asociada a este Proyecto.

2.1 Definición de OOP

La OOP (*Object Oriented Programming*) es una evolución de los lenguajes procedurales que trata de modelar mediante código los objetos reales con los que se pretende trabajar. El objetivo fundamental es simplificar el proceso de desarrollo de las aplicaciones, y de alguna manera independizar éste de los propios objetos.

En la OOP se definen algunas propiedades que son difícilmente asimilables de forma directa (como puede ocurrir con la última parte del párrafo anterior), pero una vez explicados y entendidos mediante ejemplos, son de aplicación a todos los lenguajes OOP.

Por tradición se han dividido los lenguajes OOP en dos grupos: aquéllos que son totalmente OOP (como puede ser C# o Java) y otros que incorporan algunas características de la orientación a objetos, pero no sacrifican flexibilidad y rendimiento (el caso de C++). Utilizar como herramienta la OOP conlleva asumir la metodología OOP; afecta a todo el proceso, desde el análisis hasta la codificación final.

Existen diversas metodologías de análisis y diseño de aplicaciones por OOP; algunas de ellas están incluidas en los principales entornos de programación (como ocurre con Visual Studio.NET y la metodología UML –*Unified Modeling Language*). Por otro lado, los patrones de diseño permite así mismo desarrollar aplicaciones basadas en estas

metodologías; de esta forma, patrones como MVC (el Modelo–Vista–Controlador) pueden llevarse a cabo de forma casi instintiva.

2.2 Conceptos de OOP

A continuación se definen los conceptos fundamentales para comprender inicialmente el funcionamiento de la OOP. Para ello se hará uso de un ejemplo de tipo mecánico como es el cálculo de las tensiones en un contacto esférico. A continuación se define una clase de objetos: la clase *Tensiones*.

En primer lugar hay que analizar la situación; se deben identificar los objetos reales (ya sean físicos o no) con los que se tiene que trabajar y que por tanto deben ser modelados e implementados. Este estatus se le atribuye a las componentes del tensor de tensiones que se modelan a través de la clase *Tensiones*.

Estos objetos pueden presentar una serie de atributos y sobre éstos podrán realizarse determinadas acciones; estos elementos, a la hora de la codificación reciben el nombre de propiedades y métodos respectivamente. En el caso de la clase *Tensiones*, las variables que forma parte de la clase (variables miembro) serán las componentes del tensor (*sx, sy, sz, txy, tyz* y *txz*), y adicionalmente, deben disponerse una serie de propiedades adicionales como son: módulo de Young, coeficiente de Poisson, coeficiente de rozamiento, radio de contacto, cargas, etc.; formando todas ellas el conjunto de las propiedades de la clase. Por otro lado, debe implementarse "funciones" que para unos determinados valores de las propiedades adicionales anteriores, generen un valor para cada una de las componentes del tensor de tensiones. Estas "funciones" reciben el nombre de métodos, y pueden denominarse "CalculaTensiones" por ejemplo. De forma adicional a ellos, puede implementarse algunos métodos adicionales para dar valor al resto de propiedades; por ejemplo, el método "CalculaRadio" puede asignar un valor a la propiedad "radioContacto" partiendo de otras propiedades como la geometría, las cargas y el material.

En el la figura 2.1 se muestra un pequeño esquema con algunas de las propiedades del párrafo anterior. Observe que no se encuentran ordenadas, esto sirve para hacer ver que las propiedades existen pero no tienen definida las relaciones entre ellas; de relacionar las distintas propiedades se encargan los métodos. En la figura 2.2, se muestran las mismas propiedades pero organizadas a través de los métodos correspondientes, como lo comentados anteriormente.

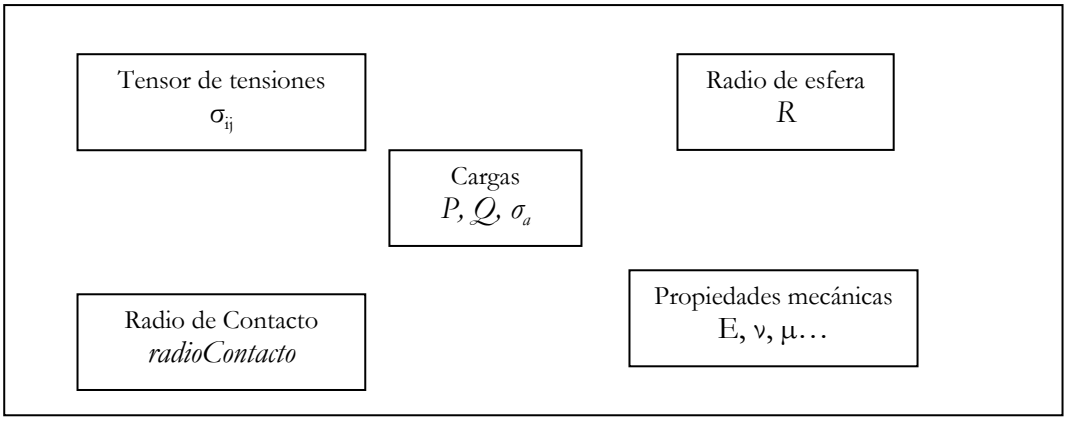

Figura 2.1: Propiedades de la clase *Tensiones*.

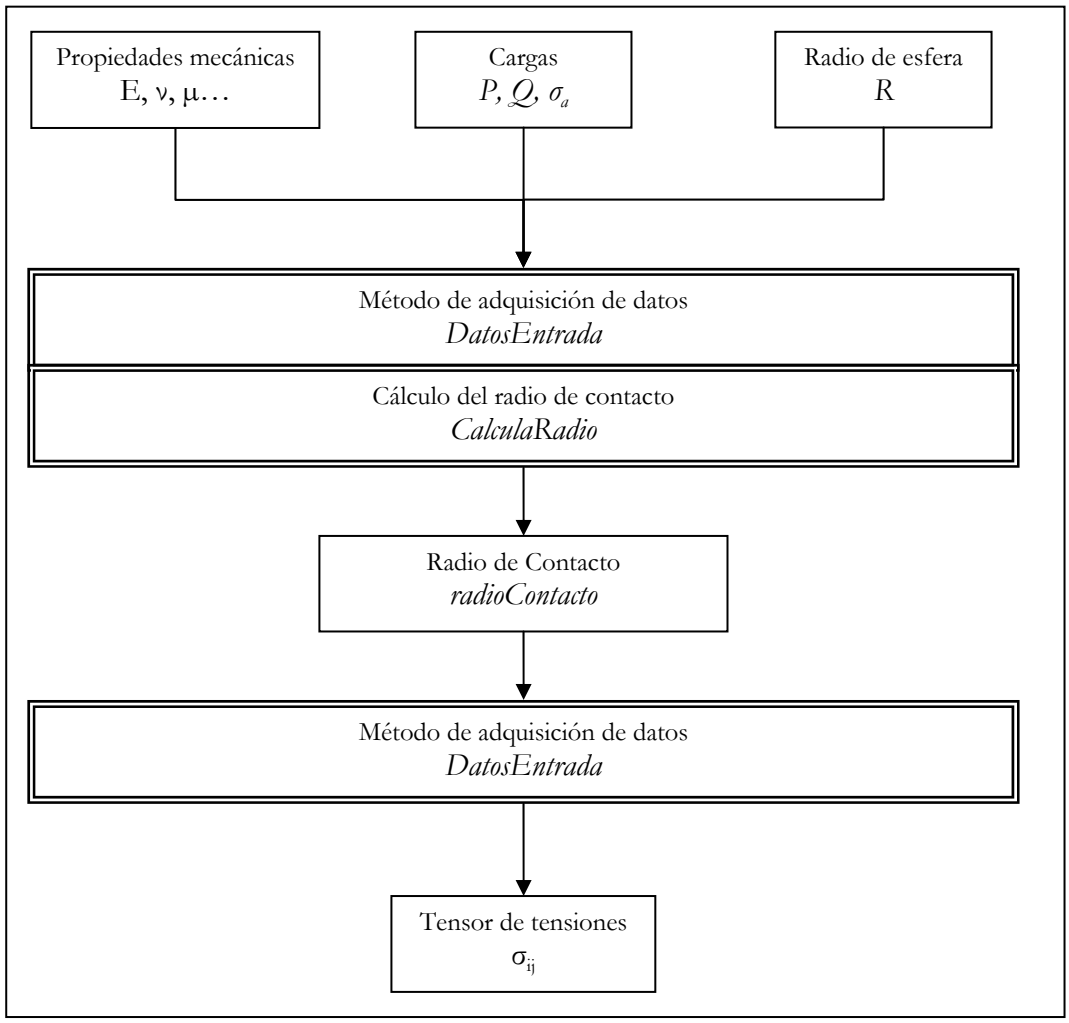

Figura 2.2: Propiedades y métodos de la clase *Tensiones*.

2.2.1 Diferencias de una clase y un conjunto de datos

La diferencia fundamental entre una clase de objeto y otro tipo de conjunto de datos (por ejemplo las estructuras), es que la primera no contiene únicamente miembros de datos (como le sucede a las estructuras), sino que dispone también de código necesario para efectuar las acciones sobre el objeto y que, por lo general, harán uso de estos datos: los métodos.

Otra diferencia importante es que las estructuras, enumeraciones y otros tipos que puede definir el programador, son declaraciones compuestas de tipos fundamentales u otro tipo definidos anteriormente; mientras que las clases deben contar, además, con una implementación; es decir, con el código correspondiente a los métodos que forman parte de ella.

2.2.2 Definición de Objeto

Un objeto puede definirse como una entidad en cuyo "interior" existen datos y código relacionados entre sí. Una propiedad bastante interesante es que el acceso a estos datos está restringido, de forma que el usuario del objeto no suele tener acceso a los datos salvo a través del código del objeto, de forma indirecta. Los procedimientos y funciones que ejecutan este código es lo que se conoce como métodos.

De esta forma, un objeto puede ser utilizado desde dos puntos de vista, desde su programación y desde su utilización. La programación requiere la implementación del código propio de la clase; mientras que el usuario del objeto hace uso de sus propiedades mediante los métodos.

El hecho de que los datos no sean totalmente accesibles para los usuarios recibe el nombre de "ocultación"; en los lenguajes OOP más usuales existen determinadas características para determinar el acceso, como por ejemplo: *private*, *protected* y *public*. Dependiendo de cada situación, el acceso a una propiedad puede ser calificado con una de las anteriores opciones y no otra; a éstas se les conoce con el nombre de "niveles de ocultación".

2.3 Características diferenciales de la OOP

Se definen a continuación una serie de propiedades de la OOP comunes en cualquier caso a todos los lenguajes. Algunas de ellas no son nuevas, y estaban ya integradas en anteriores lenguajes de programación, sin embargo, lo novedoso es la conjunción de todas ellas.

2.3.1 La abstracción

Este caso, por ejemplo, existe desde casi el comienzo de la informática. El objetivo es conseguir el diseño de una solución para una cierta necesidad abstrayéndose de los detalles, haciendo uso de objetos cuya implementación, en principio, no es importante.

Un ejemplo claro es la construcción de una vivienda de uso familiar; en principio no se estudiarían los detalles acerca de la distribución de esfuerzos o la distribución de las instalaciones, sino que se centrará en la distribución de las habitaciones y su diseño, abstrayéndose de los elementos que puedan existir en su interior.

Por tanto, el ejercicio de abstracción permite dividir un problema en objetos sin entrar en detalles sobre su funcionamiento o implementación. Una vez dispuesto el diseño del modelo a seguir, sobreviene la fase de análisis del diseño individual, desarrollando cada clase por separado.

Un ejemplo que se ajusta bastante al propósito de este Proyecto, lo constituye el desarrollo de un modelo para la predicción de vida a fatiga por fretting. La figura 2.3 muestra un esquema de dicho modelo; observe que existen diferentes objetos que se encargan de cada una de las partes que entran en juego: las tensiones, la fase de iniciación y la de propagación. El control (Modelo de predicción) se encarga de iniciar el proceso, mandado una petición al objeto "iniciación"; éste calcula las tensiones a través del objeto correspondiente y determina la primera fase del crecimiento (la iniciación de grietas), tras

ello da paso al objeto de "propagación", que hace uso del objeto "tensiones" para proporcionar una predicción de vida; ambos resultados de predicción son devuelto al control para que realice la tarea de mostrar los resultados.

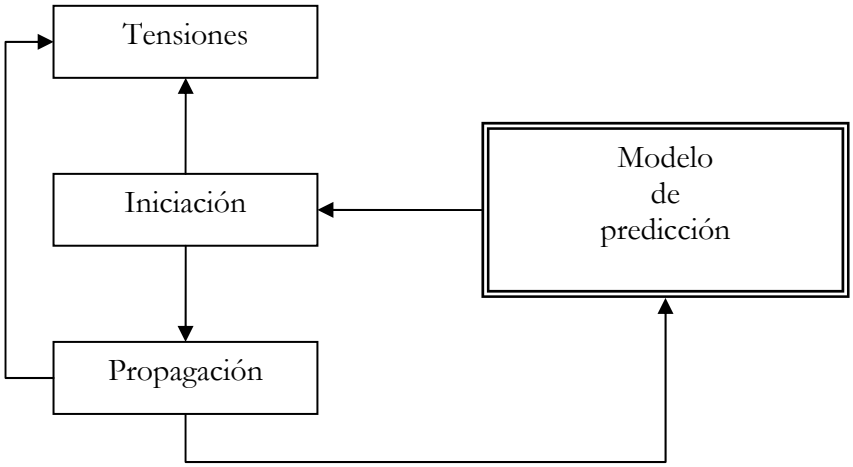

Figura 2.3: Modelo para la predicción de vida en fatiga por fretting.

2.3.2 La encapsulación

Es consecuencia de la propia definición de objeto, en la que datos y código comparten una misma entidad, a modo de "cápsula". La encapsulación hace que no sea posible la incorrecta manipulación del objeto (siempre que así se pretenda, esté correctamente diseñado y se use la ocultación de datos de manera apropiada).

La reunión de datos, códigos y esta propiedad da lugar a la definición de una clase.

2.3.3 La herencia

La herencia es, sin duda, la característica más preciada y requerida de un lenguaje OOP; se trata de que, basándose en la misma idea de los objetos, éstos pueden estar formados a su vez por otros más. Permite ahorrar gran cantidad de tiempo y trabajo, a cambio de dedicar más tiempo en el diseño y la implementación de los componentes, pero sin duda que merece la pena.

Un ejemplo que se refleja bastante en la aplicación del Proyecto (así como toda la programación basada en Formularios Windows) lo constituye el uso de las ventanas "tipo Windows". Todas ellas se heredan de una misma clase la clase *Form*.

NOTA:

Para justificar el uso de la herencia, piense la utilidad que representa a tenor del ejemplo anterior. La dedicación de programadores profesionales para realizar la clase Form, se puede aprovechar en cada una de las ventanas que forma parte de cada aplicación en entorno Windows; lo cual, sin lugar a dudas, justifica la importancia de la herencia.

2.3.4 El polimorfismo

Está muy ligada a la característica anterior, y representa la cualidad por la cual un objeto puede adoptar diversas formas. En realidad, lo que se hace es que cuando se llame a un método de un determinado objeto ("hijo") del cual se tiene una referencia, el lenguaje se encarga (en tiempo de ejecución) de determinar el enlace necesario para que se llame al método en el objeto adecuado ("padre").