Una clase responde a la idea de concepto, entendido como una noción compartida que se aplica a determinados objetos. Sin embargo, no puede haber objetos que no respondan a un concepto. Para ver si un objeto está comprendido en un concepto se le aplican ciertas pruebas de reconocimiento. Por ejemplo, para saber si un determinado objeto es un pino, podríamos ver si es un árbol, si tiene frutos en forma de piña, si tiene las hojas cilíndricas y alargadas, etc.: si no pasa alguna de las pruebas concluiremos que no es un pino.

En el ambiente de objetos se suele pensar en los objetos como entidades activas, que realizan tareas. A los datos de la programación tradicional se les aplican procesos, se les hace tal cosa, se les hace tal otra... Cuando se trabaja con objetos son éstos los que actúan, estimulados por mensajes y respondiendo con el comportamiento esperado. Se piensa en los objetos como actores en el escenario, a los cuales el programador (autor) les define su comportamiento (papel) y el usuario (director) dirige su actuación.

Por todo esto, un buen diseño orientado a objetos debe partir de identificar las clases con las que hay que trabajar tratando de observar lo que se quiere modelizar. Y a partir de allí, definir comportamientos y atributos de los objetos de dichas clases.

Ya volveremos sobre estos temas.

PARADIGMAS Y TÉCNICAS PREVIAS DE DESARROLLO DE SOFTWARE

Hay una serie de técnicas, anteriores en el tiempo a la orientación a objetos, que han tenido una fuerte influencia sobre la misma. Entre ellas, la programación estructurada, la programación modular, la abstracción y el ocultamiento de información. Dado que, si bien son técnicas desarrolladas fuera del paradigma de objetos, permanecen en él, las vamos a repasar a continuación.

Programación estructurada

La programación estructurada es un concepto que surge como respuesta a la crisis del software de los años 60, cuando el desarrollo de programas estaba ocupando una porción cada vez mayor de los costos de computación y, a su vez, el mantenimiento de los mismos era cada vez más inmanejable.

La programación estructurada se convirtió, hace unos 30 años, en un *must*, y todo producto que se lanzaba al mercado se proclamaba *estructurado*.

El primer inconveniente que se atacó fue el uso del ya olvidado *Goto*[7], sobre todo luego de un artículo de Dijkstra titulado "Goto considered harmful" (El *Goto* considerado dañino). A menudo se ha resumido la programación estructurada como el rechazo del *Goto* y nada más.

7 Más correctamente, ruptura de secuencia o interrupción incondicional de la secuencia.

Esta búsqueda de la supresión del uso del Goto había provocado la enunciación del teorema de Bohm-Jacopini [Bohm], ya en 1966, bajo la caracterización de *una entrada, una salida*: "Cualquier segmento de programa con una entrada y una salida que tenga todas las proposiciones en algún camino de la entrada a la salida se puede especificar usando sólo secuencia, selección e iteración".

Sin embargo, la programación estructurada involucraba otras prácticas, de las cuales las más importantes parecen ser:

- Utilizar un estilo disciplinado de programación.
- Ser una técnica que lleve a programas fácilmente modificables.
- Sencillez, claridad y elegancia.

Estos objetivos se logran sólo parcialmente con la eliminación del Goto y su reemplazo por las tres estructuras fundamentales de secuencia, selección e iteración.

Otros aspectos, menos formalizados, apuntaban a usar lenguajes que poseyeran unas pocas sentencias de uso habitual, con construcciones conceptualmente simples y ampliamente aplicables en la práctica, sin reglas demasiado permisivas, el uso de nombres claros, el aislamiento de las dependencias de la máquina en unas cuantas rutinas separadas, etc..

Fue la primera vez que el rendimiento y las características técnicas dejaron de ser las únicas consideraciones de calidad de los sistemas de software.

Los lenguajes de programación como Fortran y Algol se hicieron más estructurados, y luego surgieron nuevos lenguajes, como C y Pascal.

Programación modular

La llamada programación modular buscó cubrir los baches que dejaba la simple aplicación de la programación estructurada, sobre todo en el desarrollo de sistemas de tamaño medio a grande. Por lo demás, buscaba objetivos similares.

La idea consiste en dividir un programa en un conjunto de módulos o subprogramas autónomos que son programados, verificados y modificados individualmente. Dichos módulos pueden ser desarrollados por distintos programadores y testeados y mantenidos en forma independiente.

Así, los módulos son las partes funcionales del sistema que se plantea como solución.

Cuando se elige desarrollar el sistema haciendo un refinamiento de un problema desde lo más complejo a lo más elemental, el mismo se escribe como una secuencia de acciones no primitivas que luego son desarrolladas como subprogramas. Esto se conoce como programación "top-down" o descendente, el primer estilo encarado en el marco de la programación modular.

En otros casos los subprogramas sirven para escribir una serie de acciones no primitivas, de uso relativamente frecuente, que luego se usan en varios programas. Así se llega a construir bibliotecas de subprogramas. Estas bibliotecas (en inglés "libraries"[8]) pueden ser tan amplias que los programadores escriban programas en un lenguaje casi propio, que permite aun el manejo de ciertos tipos y estructuras de datos que el lenguaje original no manejaba. A menudo esto es llamado programación "bottom-up" o ascendente, y pronto se convirtió, no en un sucedáneo de la programación descendente, sino en su mejor complemento.

Se enfatiza asimismo el uso de declaraciones locales, que confina identificadores a espacios de nombres más reducidos, y se desalienta el de datos globales. En esta misma línea, se recomienda que las interacciones entre módulos se limiten al pasaje de parámetros.

Otras recomendaciones apuntan a reducir el acoplamiento entre módulos y aumentar la cohesión. Esto es, garantizar que cada módulo tenga la menor cantidad de interacciones posibles con los demás módulos (conexiones angostas), a la vez que cada uno debe hacer una sola cosa simple, trabajando sobre la misma estructura de datos.

Las modificaciones en lenguajes de programación que apuntaban a la programación estructurada también implementaron la programación modular. Así, Fortran 77, Algol 80, Pascal y C, manejaban mejor la modularidad que sus predecesores.

Abstracción

Pero la programación estructurada y modular, así como estaba encarada, ya no resolvía todos los problemas a mediados de la década de los 80. Por eso, muchos empezaron a cuestionar por qué la producción de software no seguía los pasos de otras industrias, que construyen sus productos en base a componentes desarrollados por un tercero.

Así, por ejemplo, una fábrica de automóviles no pretende producir los equipos de audio o de aire acondicionado, las baterías, las lamparitas o las cubiertas que incluye en sus modelos, y a veces ni siquiera los motores. ¿Por qué hacen esto? Precisamente porque comprárselos a quien produce estos componentes en forma masiva suele ser más económico, además de que permite que cada uno se dedique a lo que mejor sabe hacer.

Lo mismo puede aplicarse al desarrollo de software, utilizando lo desarrollado por otros en nuestras aplicaciones o incluso lo desarrollado *en casa* para otros sistemas. Esto se llama **abstracción**.

De esta forma se puede ir construyendo un lenguaje de nivel superior. Por ejemplo, supongamos que agregamos a Pascal una función de exponenciación, que el lenguaje no tiene. Si incluimos esta función en una unidad de biblioteca, podremos utilizarla luego en todas nuestras aplicaciones. Por añadidura, si el módulo a desarrollar fuera más complejo, podríamos pensar en comprarlo a alguien que lo haya desarrollado.

8 Esto ha dado lugar a que en general se las conozca como "librerías", lo que considero un barbarismo inaceptable.

Por lo tanto, no olvidemos que la forma más fácil de escribir un programa es usando simplemente bibliotecas escritas por otras personas, ya probadas, generalizadas y optimizadas. En general, a los principiantes les resulta trabajoso reunir información e integrar productos de diversos fabricantes, pero los beneficios suelen ser mayores que los costos.

En realidad, hay dos formas de abstracción: la **abstracción de procesos**, que se corresponde con lo explicado en los párrafos anteriores, y la **abstracción de datos**, que veremos enseguida. La diferencia fundamental entre ambas es que la primera se centra en subprogramas, trabajando sobre un conjunto de datos accesibles a todos los módulos, mientras la segunda se centra en datos, que contienen asimismo los subprogramas que permiten manipularlos.

La abstracción de datos se utiliza para lo que se denomina implementación de **tipos definidos por el programador**[9]. Diremos que trabajamos con un tipo definido por el programador cuando manejamos un tipo de dato que no está predefinido en el lenguaje utilizado, simulándolo de manera totalmente transparente, mediante un adecuado uso de una biblioteca de subprogramas.

Puesto que los tipos de datos no son conjuntos sino álgebras, es decir, conjuntos más las operaciones definidas sobre los elementos de los mismos, se hace necesario especificar ambas cosas. Por ello, la implementación de un tipo definido por el programador implica siempre las siguientes tres tareas:

- Determinación de las operaciones necesarias sobre el tipo.
- Elección de una estructura de datos sobre la cual se materializará la implementación.
- Implementación de la estructura de datos y las operaciones.

La ventaja principal de los tipos definidos por el programador es que se puede definir un tipo que se ajuste a unas necesidades particulares en vez de tener que usar los definidos en el lenguaje que está usando.

Para que su uso sea sencillo, los lenguajes deben tratar a los tipos definidos por el programador como a los tipos predefinidos.

Los lenguajes de programación estructurados no tenían en general un buen soporte para la abstracción. Sólo Ada pareció reconocer esta necesidad y en algunas implementaciones de Pascal se introdujo el concepto de unidad de biblioteca. C implementó una funcionalidad incompleta.

A pesar de los años que tiene el concepto, la abstracción se ha utilizado en sus primeros tiempos casi siempre dentro de una misma empresa o equipo de desarrollo, sin que se diera ese intercambio de componentes de software que se viene anunciando desde los 80. Sin

9 Un comentario importante respecto de los nombres. En general este concepto se conoce como "tipo abstracto de datos" (TDA o TAD), pero el término ha sido muy cuestionado por el uso de la palabra "abstracto". En efecto, "abstracto" tiene una connotación diferente en muchas metodologías, y en particular en la orientación a objetos. Por lo tanto, y en aras de mantener la claridad al máximo, en lo sucesivo lo llamaremos tipo de datos definido por el programador.

embargo, desde hace una década aproximadamente, gracias al auge de Internet, ha surgido este mercado que está comenzando a revolucionar el desarrollo de software. Hoy los desarrolladores de software construimos las aplicaciones *enchufando* componentes y probándolos en conjunto.

Ocultamiento de Implementación

El problema del uso de la abstracción, sobre todo si hablamos de tipos de datos definidos por el programador, consiste en que debería haber forma de impedir que el usuario del mismo (un programador de aplicaciones que denominaremos cliente o *consumidor*) haga un uso indebido de los datos por utilizar aspectos de la implementación que no debieran ser de su incumbencia.

Volviendo a la analogía de la industria automotriz, quien provee a una fábrica de automóviles de equipos de audio, no suele especificar cómo están construidos los mismos. Solamente basta con que asegure una serie de conectores o interfaces que permitan su instalación y una serie de características y funcionalidades a prestar. Así, el fabricante del equipo de audio puede establecer cambios en los circuitos internos sin necesidad de que el fabricante de autos deba modificar nada si se respetan las interfaces y las prestaciones establecidas. Esto lo logra porque el equipo de audio es, desde el punto de vista de la fábrica de automóviles, una **caja negra**, que se testea cerrada, teniendo en cuenta solamente que cumpla las prestaciones establecidas por un contrato y que tenga las interfaces necesarias para poder conectarlo.

¿Cómo podemos trasladar esta idea a la industria del software? Con el principio del **ocultamiento de implementación**, también llamado ocultamiento de información u ocultamiento de datos. Este concepto permite el desarrollo de software de modo tal que el cliente no pueda usar la estructura de datos y los algoritmos subyacentes. En otras palabras, se pueden usar las interfaces de los tipos de datos definidos por el programador, pero no la implementación. De este modo, la interfaz se convierte en una especie de contrato entre las partes, que debe ser respetado.

El ocultamiento de implementación precisa un par de normas respecto de las especificaciones:

- Al cliente, se le debe proveer sólo la información que necesita para usar la función correctamente.
- Al que implemente la función (proveedor), se le debe dar sólo la información necesaria para comprender el uso que se hará del mismo.

Por lo tanto, una especificación de una operación debe señalar cuáles son las funciones que realiza, describiendo el dominio de cada una, los valores iniciales, los parámetros que se deben utilizar y el efecto de la misma. Dicho de otra manera, se deben definir las **precondiciones** y **postcondiciones** de la función. Respecto del tipo de datos, importa definir los **invariantes** del mismo, es decir, aquellas condiciones que se deben cumplir siempre para los datos del tipo.

El ocultamiento implica separar el *qué* del *cómo*. Es decir, el cliente debe conocer qué hace el módulo, pero no necesariamente *cómo* lo hace.

Todo esto puede parecer extraño: ¿por qué ocultarle al usuario de la biblioteca lo que no necesita? Las ventajas del ocultamiento son básicamente dos:

- Permite cambios de implementación.
- Impide violaciones de restricciones entre datos internos.

La primera ventaja se refiere a que, por no poder usar el cliente la estructura de los datos y algoritmos, no va a utilizar características exclusivas de éstos, de modo que en futuras modificaciones se podrá alterar dicha estructura sin que sean necesarios cambios en los programas clientes.

Por ejemplo, si se desarrolla un tipo de datos definido por el programador que maneja pilas[10] de enteros trabajando sobre arreglos, y se le da al cliente acceso a la implementación, éste puede utilizar facilidades propias de la forma de implementación. Si luego, por consideraciones diversas, se establece que conviene implementar las pilas sobre listas encadenadas, no se podrá hacer sin obligar al cliente a modificar sus programas, lo cual alteraría el *contrato* entre implementador y cliente.

Esto se hace mucho a lo largo de la vida de un sistema: en las primeras versiones se usa una implementación simple con tal de salir lo antes posible al mercado y luego se va mejorando en versiones futuras. Notemos, en consecuencia, que de no ocultar la implementación, se estaría impidiendo su evolución.

La segunda ventaja que se mencionó se refiere a las restricciones que pueda haber entre datos internos, que hayan sido establecidas por el implementador como resguardo para el buen funcionamiento, y que puedan ser arruinadas por el acceso indebido del cliente.

Por ejemplo, si el implementador establece datos internos de un tipo vinculados entre sí, dejar librado el acceso a la implementación al cliente puede malograr el trabajo hecho. Un ejemplo de esto puede ser un grafo[11] implementado de forma tal que por cada arco que se agrega en un sentido se agrega automáticamente su simétrico, y cada vez que se elimina un arco se eliminan ambos. Un cliente que no entiende del todo la implementación puede intentar agregar o suprimir un arco sin preocuparse por la simetría.

En definitiva, se oculta la implementación para que ningún cliente sea dependiente de una forma de implementación en particular. O dicho de otro modo, para que el cliente pueda estar seguro de que está usando el tipo de datos definido por el programador correctamente, y el implementador pueda introducir cambios y mejoras en el futuro. Gracias al

10 Una pila es una estructura de datos en la cual los elementos se insertan y se extraen por un mismo extremo, denominado tope.

11 Un grafo es una estructura de datos en forma de red, como una red caminera o ferroviaria o una red de computadoras.