Capítulo 1.

INTRODUCCIÓN: LA ORIENTACIÓN A OBJETOS

CONCEPTOS PREVIOS

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 20 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*[6], 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.

Esta búsqueda de la supresión del uso del *Goto* había provocado la enunciación del teorema de Bohm-Jacopini [13], 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, dentro de los propósitos de la programación estructurada había otros, de los cuales los más importantes parecen ser:

- Utilizar un estilo disciplinado de programación.
- Ser una técnica que lleve a programas altamente 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

[6] Más correctamente, ruptura de secuencia o interrupción incondicional de la secuencia.

de nombres claros, no utilizar un identificador para propósitos múltiples, abundancia de comentarios aclaratorios, uso de sangría para aumentar la legibilidad y denotar la estructura del programa, 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.

El concepto de programación estructurada lo he analizado a fondo en mi obra "Curso de Programación" [5], en el tomo I.

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, aunque buscaba objetivos similares.

La idea consistió en dividir un programa en un conjunto de módulos o subprogramas autónomos que fueran individualmente programables, verificables y modificables. En muchos casos estos módulos fueron desarrollados por distintos programadores y eran testeados y mantenidos en forma independiente.

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, constituyendo lo que se conoce con el nombre de biblioteca de subprogramas. Estas bibliotecas (en inglés "libraries"[7]) 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.

Es también de gran utilidad la división de un programa en módulos en la etapa de prueba, pues cada subprograma puede ser probado en forma separada antes de inte-

[7] Esto ha dado lugar a que en general se las conozca como "librerías".

grarlos al sistema. Esto permite que en el momento de la integración del sistema se pueda tener la seguridad de que cada uno de los módulos no posea errores internos, restando solamente la prueba en conjunto para ver la interacción entre los mismos.

El adecuado uso de módulos permite también un mejor mantenimiento, ya que si se necesita modificar alguna tarea de las que efectúa el sistema, esto se podrá hacer modificando el módulo que realiza dicha tarea, sin necesidad de tocar todo el programa.

Se enfatiza asimismo el uso de declaraciones locales, que confina identificadores a espacios de nombres más reducidos, y se desalienta el de identificadores globales, sobre todo cuando se trata de variables. 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.

El concepto de programación modular también fue introducido en mi obra antes mencionada, en el tomo I. También en el tomo II se analiza el tema.

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 industria del software no hacía lo que 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 estas *autopartes* 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 nuestras aplicaciones. Asimismo, si el módulo a desarrollar fuera más complejo, podríamos pensar en comprarlo a alguien que lo haya desarrollado.

La abstracción se utiliza también para lo que se denomina implementación de **tipos de datos abstractos** (TDA). Diremos que trabajamos con un TDA 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. Esto permite manejar tipos de datos que el lenguaje no posee, ya testeados en otros contextos.

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 TDA implica siempre las siguientes 3 tareas:

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

Los lenguajes de programación modulares 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 comercio de componentes de software que se viene anunciando desde los 80. Sin embargo, desde hace un lustro aproximadamente, gracias a la tecnología de objetos y el auge de Internet, ha surgido este mercado que está comenzando a revolucionar el desarrollo de software. Tal vez en unos años más los desarrolladores de software terminemos comprando la mayor parte de lo que necesitemos, *enchufando* componentes y probándolos en conjunto.

La abstracción de datos está analizada en el tomo II de mi obra ya citada.

Ocultamiento de implementación

El problema del uso de la abstracción, sobre todo si hablamos de TDA, consistía en que no había forma de impedir que el usuario del mismo (un programador de aplicaciones que denominaremos *cliente*) hiciera un uso indebido de los datos, porque tendía a utilizar conceptos propios 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. De esta manera, 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 puede este concepto ser llevado 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 TDA, 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 la especificación[8]:

- Al cliente, se le debe proveer sólo toda la información que necesita para usar el módulo correctamente.
- Al que implemente el módulo, se le debe proveer toda la información necesaria para comprender el uso que se hará del mismo, pero ninguna información más.

Por lo tanto, una especificación debe señalar cuáles son las funciones que el módulo realiza, describiendo el dominio de cada una, los valores iniciales, los parámetros que se deben utilizar y el efecto de dichas funciones.

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:

- Permitir cambios de implementación.
- Impedir 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 propiedades exclusivas de éstos, de modo que en futuras modificaciones se podrá alterar dicha estructura sin que sean necesarios cambios en los programas clientes.

[8] Esto lo tomé de Boria [11].

Por ejemplo, si se desarrolla un TDA que maneja pilas 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 podrá hacerse sin obligar al cliente a modificar sus programas, lo cual alteraría el *contrato* entre implementador y cliente.

Nótese, 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 tres datos de un TDA vinculados entre sí, dejar librado el acceso a la implementación al cliente puede arruinar el trabajo hecho. Un ejemplo de esto puede ser un grafo no dirigido 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.

De todas maneras, es importante destacar que el ocultamiento de implementación debe ser cuidadoso en respetar el contrato entre cliente e implementador en todo momento. Vale decir, las versiones futuras de un mismo TDA o módulo deben tener, por lo menos, la misma funcionalidad y la misma interfaz que la primera versión, y no introducir efectos colaterales. Por lo tanto, podré agregar nuevas operaciones, pero debo garantizar que las antiguas operaciones permanecen y mantienen la misma interfaz.

Asimismo, la implementación debe respetar en todo momento las restricciones entre datos internos que sean necesarias.

Además, para que sea útil al cliente, toda la funcionalidad esperable de un TDA debería estar accesible en forma de subprogramas. Decimos en este caso que la implementación es *completa*. Por eso suele ocurrir que se le oculta al usuario la estructura interna de un TDA y las implementaciones de los módulos, pero se le deja ver el encabezado de estos últimos.

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 TDA correctamente, y el implementador pueda introducir cambios y mejoras en el futuro.

El concepto de ocultamiento de implementación sólo se dio en algunos lenguajes, como Ada y Modula-2, antes del surgimiento de la programación orientada a objetos.

Hacia un nuevo paradigma

La gran pregunta que cabe hacerse es, después de todo lo visto, ¿por qué seguimos buscando nuevas técnicas de desarrollo?

El problema principal es que la crisis del software sigue, y probablemente no terminará nunca. Una de las causas más fuertes es la huida hacia adelante[9] que se verifica por el aumento de complejidad de los sistemas. Así, cuando una técnica estaba resultando útil, el enorme crecimiento de la complejidad la hace obsoleta.

Esto es bastante lógico si lo pensamos humana y tecnológicamente. Cuando las computadoras eran muy caras y el costo de desarrollar software muy alto, los programas resolvían tareas elementales. A medida que el costo del hardware bajó, surgieron nuevos lenguajes de programación y hubo técnicas mejores de desarrollo de software, se pudieron crear sistemas cada vez más complejos, por lo que esos hardware, lenguajes y técnicas resultaron insuficientes. Con el tiempo el hardware siguió abaratándose, surgieron ambientes de desarrollo amigables y se crearon metodologías de desarrollo que permitieran manejar la complejidad que se estaba precisando, pero los requerimientos crecieron en complejidad. Es obvio que una crisis así no va a finalizar nunca (aunque personalmente creo que se debería hacer todo lo posible por acotarla).

Desde el punto de vista de la complejidad y del riesgo de error, podemos hablar de categorías de aplicaciones. En primer lugar, estarían los programas aficionados, que son los que se hacen por individuos dentro de grupos que comparten intereses comunes, como los estudiantes: entre éstos, el costo del error es bajo y el desarrollo es económico. Luego vendrían los programas renovables o de consumo, como los típicos productos de procesamiento de textos y planillas de cálculo, de un diseño con poca interacción con el usuario real, muy económicos por su gran difusión, y de escasos riesgos en caso de errores. Le siguen los llamados sistemas esenciales, que serían aquellos que con sus fallas pueden poner en peligro el funcionamiento de una empresa. Y finalmente, los sistemas vitales, de los que depende la vida de seres humanos o los que pueden poner en peligro la vida humana. Evidentemente, cuanto mayor es el riesgo de error, mayor es la complejidad, pues las salvaguardias para evitar errores introducen gran cantidad de código y relaciones entre módulos.

El aumento de la complejidad no es un asunto trivial. Si pensamos, haciendo una grosera simplificación, que cada nueva funcionalidad agregada a un producto de soft-

[9] La expresión la tomé de Muller [10].

ware agrega un nuevo módulo que implemente esa funcionalidad, podemos medir el aumento de complejidad asimilándolo al aumento del número de módulos. Pero, como sabemos, la cantidad de interacciones posibles entre módulos crece mucho más que linealmente con éstos, y es del orden del cuadrado del número de módulos[10].

Encima, lo usual es enfrentar la complejidad, no intentando escribir la menor cantidad de código posible, sino que se suele traducir en un aumento, a veces innecesario, del tamaño del código.

Otras razones para el uso de nuevas técnicas de desarrollo tienen que ver con nuevos usos de la informática o nuevas formas de uso. Por ejemplo, el auge de las interfaces gráficas de usuario, sobre todo luego de 1995, ha impulsado nuevas formas de desarrollo, entre ellas la programación visual y la programación por eventos. La World Wide Web también ha impulsado la aparición de aplicaciones especiales para ese medio. Estas técnicas, a su vez, precisan de nuevas metodologías de desarrollo y programación.

Calidad del software

El objetivo primordial de la orientación a objetos es la **calidad**. Todo lo demás le está subordinado.

Para entender la importancia de la calidad, mencionemos el problema del año 2000. Debido a este grueso error de diseño, que muchos informáticos tratan inútilmente de justificar, se gastaron miles de millones de dólares en todo el mundo en mantenimiento correctivo. ¿Hemos pensado en los usos alternativos que pudo haber tenido esa cantidad de dinero?

La calidad del software está influenciada tanto por *factores internos*, que son aquéllos que sólo afectan a los informáticos, y *externos*, que perciben los usuarios finales. Estos últimos son los únicos realmente importantes, y los primeros deben ser condición para que se den estos.

A continuación presento una lista de factores, de más a menos importantes:

- Confiabilidad. Éste es el factor de calidad por excelencia. Se basa a su vez en dos necesidades: corrección (que el sistema cumpla con las especificaciones) y robustez (capacidad de reaccionar bien ante situaciones excepcionales).
- Extensibilidad: es la facilidad de adaptarse a cambios de especificación.
- Reutilización: es la capacidad de las partes de un sistema para servir en la construcción de aplicaciones distintas.

[10] Hoy en día existen métricas para evaluar la complejidad, pero en este contexto preferimos este análisis simplificado.

- Compatibilidad: se refiere a la interoperabilidad con otros productos de software, y se logra a través del uso de estándares, como las normas de comunicación, formatos de archivos, etc..
- Facilidad de uso o usabilidad: significa que el sistema pueda ser utilizado por personas de diferentes formaciones y capacidades.
- Portabilidad: es la facilidad de correr el sistema en otras plataformas distintas de aquellas para las cuales fue diseñado.
- Eficiencia. Consiste en usar la menor cantidad posible de recursos de hardware, tiempo de procesador y ancho de banda. Este requisito de calidad no se debe exagerar, ya que, como dice un refrán conocido, "no importa cuán rápido es hasta que no esté correcto".
- Funcionalidad. El sistema no debe pecar ni por exceso ni por defecto respecto de los requerimientos. En general, conviene comenzar el desarrollo por los aspectos más importantes, y nunca sacrificar la calidad por agregar funcionalidad.
- Oportunidad: que el sistema esté en el mercado en el momento en que los usuarios lo precisan, o apenas un poco antes.
- Costo.

ORIENTACIÓN A OBJETOS

La tecnología de orientación a objetos es un nuevo paradigma de desarrollo de sistemas. Con esta tecnología se busca esa meta siempre inalcanzable de obtener sistemas económicos de desarrollar y mantener[11].

En realidad, el paradigma de objetos no es nuevo en absoluto, aunque su auge periodístico haya empezado hace unos 10 años. Nació de la mano de los problemas de simulación, y de lenguajes como Flavors y Smalltalk, hace ya más de 20 años. Tampoco es cierto que sea una idea totalmente diferente de la "programación tradicional", al punto que quien quiera aprender a programar con objetos deba sepultar sus conocimientos previos en programación, como pretenden algunos autores apasionados por el paradigma.

Lo que se ha gestado en estos últimos años es una fiebre de consumo (periodístico por lo menos) de todo lo que de alguna manera tenga algo que ver con los *objetos*. Y, paralelamente, han aparecido en el mercado un sinnúmero de ambientes de trabajo, ambientes de desarrollo e implementaciones de lenguajes que se procla-

[11] Cuando digo "inalcanzable" me refiero a que cada unos 10 años surgen nuevas ideas que mejoran las técnicas que se estaban utilizando y que supuestamente eran el último escalón en este sentido.

man "orientados a objetos". Y eso es lo que la mayoría ha oído: mucho ruido, personas que hablan de algo que no entienden del todo y mucha confusión.

La metodología de orientación a objetos se impulsó con una serie de objetivos, que analizaremos en los ítems subsiguientes:

- Reducción de la brecha entre el mundo de los problemas y el mundo de los modelos.
- Aumento del nivel de complejidad de los sistemas.
- Fomentar la reutilización y extensión del código.
- Uso de prototipos.
- Programación en ambientes de interfaz de usuario gráfica.
- Programación por eventos.

Reducción de la brecha entre el mundo de los problemas y el mundo de los modelos

La orientación a objetos intenta crear una correspondencia unívoca entre elementos del espacio del problema y objetos en el espacio de soluciones.

Desde la perspectiva de la orientación a objetos, los objetos de un programa interactúan para resolver un problema, respondiendo a estímulos —mensajes o eventos— del medio externo (generalmente otros objetos).

Para la mayoría de las personas, la forma de pensar orientada a objetos es más natural que las técnicas tradicionales. Al fin y al cabo, el mundo está formado por objetos. Comenzamos a aprender sobre ellos en la infancia y descubrimos que se comportan de determinadas maneras: desde una etapa muy temprana categorizamos los objetos y descubrimos su comportamiento. Las personas piensan de manera natural en términos de objetos, eventos y mecanismos de activación. Y los programas no son más que modelizaciones de la realidad.

Miremos un poco alrededor de nosotros: las sillas en que estamos sentados, la mesa que tenemos delante, el colectivo que nos transporta, el supermercado donde hemos ido a comprar comida, las cajas, el pollo y los tomates que compramos, los cajeros... Estamos rodeados por objetos.

Ésta es una de las principales razones por las cuales los ambientes gráficos se suelen proclamar como orientados a objetos.

Si tuviéramos que hacer un sistema que simulara las colas en las paradas de colectivos o los stocks del supermercado, nadie dudaría en utilizar POO. Es por ello que el primer lenguaje que tuvo cierta orientación a objetos fue Simula 67, un lenguaje orientado a simulación, ¡en la década del sesenta!

Un objeto es cualquier cosa, real o abstracta, acerca de la cual almacenamos datos y los métodos que controlan dichos datos. Son ejemplos de objetos: una empre-

sa, un recibo, un plano de la República Oriental del Uruguay, una disquetera, un tranvía, el recorrido de un tranvía, un proceso para emitir un recibo, un icono en una pantalla, toda la pantalla, este texto, un carácter del texto, etc.

En el ambiente de la POO 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.

También se puede pensar en un objeto como un elemento a priori pasivo, que es despertado por la llegada de un mensaje proveniente de otro objeto, y que responde activamente a este mensaje.

Una clase responde a la idea de concepto, entendido como una noción compartida que se aplica a determinados objetos en forma consciente. Un concepto puede no tener instancias, como el concepto de *elefante ovíparo*. Sin embargo, no puede haber objetos que no respondan a un concepto. Así surge la noción de clase.

Desde esta perspectiva, los objetos de un programa interactúan para resolver un problema, respondiendo a estímulos —mensajes o eventos— del medio externo (generalmente otros objetos).

Aumento del nivel de complejidad de los sistemas

Como dijimos más arriba, ésta es una de las principales causas que provocan la continuidad de la crisis del software. Desde el principio de esta crisis se intentó atacar el problema de la complejidad, en un principio mediante la descomposición descendente (top-down) de un sistema, y luego incorporando la construcción ascendente (bottom-up) de máquinas virtuales cuyas abstracciones fueran acercando el modelo al problema en forma iterativa.

La orientación a objetos, al reducir la brecha entre los modelos y la realidad, ya colabora en este aspecto. Adicionalmente, un mejor manejo de los grandes bloques de código, la agrupación de TDA, bibliotecas y demás, facilita la administración de proyectos grandes. Tampoco es despreciable la facilidad para el trabajo de grupos grandes de desarrolladores.

Pero lo que cambia radicalmente la tecnología de orientación a objetos es nuestra forma de pensar sobre los sistemas. La complejidad de los objetos que podamos utilizar seguirá en aumento, ya que nuevos objetos se pueden construir a partir de otros. Éstos a su vez están constituidos por otros objetos, y así sucesivamente. Podemos tener una biblioteca como un depósito de clases, en parte comprada y en parte construida en casa. Es muy probable que estas clases sean más poderosas a

medida que crece su complejidad. De esta manera, se puede llegar a un sistema de un millón de líneas de código adquiriendo novecientas mil en el mercado y desarrollando las cien mil más específicas de la aplicación en cuestión.

Por todo esto hay quienes dicen que la orientación a objetos permite ocultar la complejidad bajo la simplicidad de los objetos.

Reutilización y extensión del código

Llamamos reutilización y extensión del código a la facilidad para emplear, en el desarrollo de una aplicación, código desarrollado y probado en otros contextos. Volveremos sobre esto en capítulos posteriores.

La orientación a objetos viene incluye dos capacidades, denominadas herencia y polimorfismo, que facilitan enormemente el uso de código escrito, ya sea por el propio equipo o por terceros, sin necesidad de copiarlo físicamente y sin violar el principio de ocultamiento de implementación.

A su vez, este código puede ser utilizado tal como está (reutilización) o se le pueden incorporar nuevas funcionalidades sin alterarlo (extensión), gracias de nuevo a la herencia y el polimorfismo.

Uso de prototipos

Como veremos más adelante, las nuevas metodologías de desarrollo de software están virando hacia un desarrollo iterativo e incremental con amplia participación de los usuarios. En este contexto, el concepto de prototipo, como versión intencionalmente incompleta de un producto, surge con mucho énfasis.

Las técnicas de orientación a objetos facilitan el uso de prototipos debido a la facilidad de implementar un sistema en forma absolutamente modular, e incluso dejar detalles de implementación para más adelante. Es la característica denominada encapsulamiento la que, al preocuparse más bien de qué hace el sistema más que de cómo lo hace, permite dejar de lado detalles fmos y concentrarse en las líneas generales del sistema. La herencia y el polimorfismo son importantes si se posee una jerarquía de clases rica, ya que favorecen el desarrollo veloz de clases similares a las ya existentes, trabajando siempre sobre comportamiento probado.

Programación en ambientes de interfaz de usuario gráfica

Las interfaces de usuario gráficas no son nuevas. Xerox lanzó el primer ambiente de este tipo con la Xerox Star en 1981. Su sucesor más famoso fue el sistema operativo de la Macintosh de Apple, en 1984. Poco después surgiría una interfaz de usuario que corría sobre DOS para PC, llamada GEM (1985), muy parecida al producto de Apple, y a ella le siguieron el ambiente Windows de Microsoft (nacido

en 1985 pero usado en forma más masiva a partir de 1990), que también corría sobre DOS, y el Warp del sistema operativo OS/2 de IBM, por nombrar sólo a los más conocidos. Sin embargo, la aparición del robusto Windows NT en 1993, la conversión del simple ambiente Windows en sistema operativo en 1995 y, finalmente, la proliferación de interfaces gráficas para sistemas Unix, han dado por tierra con las interfaces de línea de comandos que parecían inmortales hace apenas 10 años.

Estas interfaces de usuario se conocen como WIMP (windows, icons, mouse pointer) o GUI (graphical user interface). Y la interfaz web es un caso especial de las mismas.

Uno de los objetivos de la orientación a objetos es facilitar el desarrollo de software para estas plataformas –GUI y web–, ya que en este tipo de ambientes se manipulan más bien objetos y no tanto procesos.

Programación por eventos

Pero las características de las interfaces de usuario gráficas no se reducen al manejo de ventanas, iconos y el mouse, ni tampoco a su condición gráfica.

En realidad, desde el punto de vista de la programación, su mayor novedad es la forma en que el usuario interactúa con la aplicación, que modifica totalmente la forma tradicional secuencial en que la computadora ejecutaba una serie de tareas a pedido del usuario y cada tanto le pedía un dato para continuar. En estos nuevos enfoques, basados en eventos, el usuario tiene toda una gama de acciones que puede realizar y la aplicación debe estar preparada para responder con acciones previstas pero no programadas.

Aquí la programación orientada a objetos aporta su visión no imperativa de desarrollo, que permite no fijarse tanto en los procesos como en los mensajes enviados por el usuario y en las respuestas de la aplicación a esos estímulos.

PROGRAMACIÓN ORIENTADA A OBJETOS

Significado

En la programación imperativa tradicional, los algoritmos se describen en base a procesos (ya sea en forma secuencial o mediante concurrencia). Por ejemplo, se pueden escribir en Pascal o C una serie de subprogramas para el manejo de pantallas, y serán exactamente eso: procedimientos o funciones que describen cómo se abre una ventana, cómo se la cierra, cómo se la limpia, etc.. Así, los algoritmos se expresan mediante procesos y éstos como una secuencia de tareas a realizar por la computadora. Por eso este tipo de programación se llama procedimental o imperativa.

La programación orientada a objetos (**POO**) encara la resolución de cada problema desde otra óptica: la óptica del objeto. Así, para resolver el caso anterior se define un objeto pantalla y se definen todas las acciones (denominadas *métodos*) que ese objeto puede realizar (*comportamiento*), cada vez que se le envía un cierto *mensaje* o *solicitud*. Las únicas acciones que se le pueden hacer a un objeto vienen dadas por los métodos. Lo que se define es el comportamiento de los objetos frente a mensajes o solicitudes provenientes de otros objetos. Incluso los programas se disparan con el envío de un mensaje a un objeto.

Desde el punto de vista de los objetos, un programa es un conjunto de objetos colaborando, o bien, un conjunto de objetos enviando mensajes y respondiendo a otros mensajes.

Dado que esos métodos son, en la práctica, muy similares a los procedimientos de la programación imperativa tradicional y los mensajes se podrían pensar como invocaciones a esos procedimientos, se podría pensar que simplemente se ha inventado un nuevo nombre para algo que ya existía anteriormente.

Esto es cierto: las ideas de objetos provienen en gran parte de la abstracción de datos. Sin embargo, el paradigma de objetos introduce conceptos nuevos. Además, es una técnica más estructurada que los intentos anteriores de estructuración. Y es más modular y abstracta que los intentos previos de abstracción de datos y ocultamiento de implementación.

La POO es un paradigma de programación no imperativa, no procedimental.

Se basa en técnicas previas, como la abstracción y el ocultamiento de la implementación, e incorpora la herencia, el polimorfismo y otras de menor entidad.

No entraremos en mayor detalle porque dedicaremos 4 capítulos a este tema.

Sí digamos que la POO no se puede desligar de todo el paradigma de orientación a objetos. En efecto, no hay que confundir POO con el uso de un lenguaje que soporte POO. Para hacer buena POO hay que desarrollar todo el sistema utilizando el paradigma, empezando por un análisis y un diseño orientados a objetos.

Lenguajes de programación orientada a objetos

Más adelante haremos un examen más detenido de los principales lenguajes de POO, una vez que hayamos estudiado los conceptos fundamentales. Aquí solamente presentamos un esquema clasificatorio de los mismos.

Los lenguajes que se usan para la POO se pueden clasificar de muchas maneras. Nosotros hemos elegido una clasificación según su origen y afinidad con el paradigma.