



Orientación a Objetos

Conceptos y terminología (II parte)

Miguel Angel Abián

ORIENTACIÓN A OBJETOS: CONCEPTOS Y TERMINOLOGÍA (PARTE 2)

Miguel Ángel Abián
mabian AT aidima DOT es

Copyright (c) 2003, Miguel Ángel Abián. Este documento puede ser distribuido sólo bajo los términos y condiciones de la licencia de Documentación de javaHispano v1.0 o posterior (la última versión se encuentra en <http://www.javahispano.org/licencias/>).

Abstract: This paper, divided in two parts, provides an introduction to Object-Oriented (OO) terminology and its concepts. It introduces terms and core concepts to Object-Oriented from a conceptual point of view, giving formal, intuitive and manageable definitions. There is a comparison with the structured paradigm. In this second part, concepts like class, object, message, Abstract Data Type, polymorphism, inheritance and relationships are explained.

Keywords: Object Oriented Terminology, Object Oriented Programming, Object Orientation, Classes, Objects, Software Analysis, Software Design, Learning Object Orientation, Object Orientation Paradigm, Structured Paradigm, Modular Programming, ADT

"[...] el hombre que realmente cuenta en el mundo es el que hace cosas, no el mero crítico: el hombre que realmente hace el trabajo, incluso aunque sea ruda e imperfectamente, no el hombre que sólo habla o escribe sobre cómo deberían hacerse."

"La crítica es necesaria y útil; a menudo es indispensable; pero nunca puede tomar el lugar de la acción, ni siquiera ser un pobre sustituto para ella. La función del mero crítico es de una utilidad muy subordinada. Es el que actúa quien cuenta realmente en la batalla por la vida, y no el hombre que mira y dice cómo debería librarse la lucha, sin compartir él mismo la tensión y el peligro."

Theodore Roosevelt

"Una buena herramienta en manos de un mal ingeniero de software produce software de mala calidad con muchísima rapidez."

Alan Davis

"Si yo tuviera que vender mi gato (al menos a un informático), no diría que es amable y autosuficiente y que se alimenta de ratones: más bien diría que está orientado a objetos"

Roger King

7. Conceptos básicos de la orientación a objetos.

Tras haberse expuesto en la primera parte de este artículo los fundamentos de la orientación a objetos (abstracción, modularidad, encapsulación y jerarquía), el siguiente paso consiste en explicar los conceptos básicos de la OO:

- Objeto
- Clase
- Mensaje
- Herencia
- Polimorfismo
- Relaciones

8. Objetos.

De manera informal, un objeto representa una entidad, ya sea física, conceptual o de software. Por ejemplo, un objeto puede representar un avión (entidad física, del mundo real), una reacción nuclear (entidad conceptual, derivada de la abstracción de un proceso real) o una lista anidada (entidad de software, abstracción de entidades existentes sólo en software). En cierto modo, el mundo, o la manera como lo contempla el ser humano, está orientado a objetos.

A pesar de la falta de rigor del párrafo anterior, nos pone sobre la pista de las características relevantes de los objetos: un objeto puede corresponder a una cosa material, con existencia independiente del observador; a una abstracción de carácter conceptual, también procedente del mundo físico; o a una abstracción de carácter conceptual, proveniente del software. Cuando nos restringimos al estudio de objetos que representan entidades de software, recalamos en el estudio de sistemas de software, y viceversa; pero ésta limitación viene impuesta por nuestra área de interés, no porque no puedan usarse objetos en otros campos del saber (a menudo se usan sin ser llamados objetos).

Una definición más formal de objeto aparece en *UML User's Guide* ([Rumbaugh, J. et al, 1999]): *“Una manifestación concreta de una abstracción; una entidad con una frontera e identidad bien definidas que encapsulan estado y comportamiento; una instancia de una clase”*. De forma similar, Booch (*Object Oriented Analysis and Design with Applications* [Booch, G., 1994]) escribe *“Un objeto posee estado, comportamiento e identidad: la estructura y el comportamiento de objetos similares están definidos en su clase común; los términos instancia y objeto son intercambiables”*.

La definición oficial de la OMG [*OMG Unified Modeling Language Specification. Version 1.4* [OMG, 2001]] prosigue en una línea de pensamiento similar a la de las dos anteriores: *“Una entidad delimitada precisamente y con identidad, que encapsula estado y comportamiento. El estado se representa mediante sus atributos y relaciones, el comportamiento mediante sus operaciones, métodos y máquinas de estados. Un objeto es una instancia de una clase”*.

Una definición sin referencia explícita al término “clase” aparece en *Dictionary of Object Technology: The Definitive Desk Reference* ([Firesmith E.,

1995]: “*Un objeto representa un elemento, unidad o entidad individual e identificable, ya sea real o abstracta, con un papel bien definido en el dominio del problema*”.

Las tres primeras definiciones mencionan las palabras *estado*, *comportamiento* e *identidad*.

La **identidad**, según [Booch, G., 1994], es “*aquella propiedad de un objeto que lo distingue de todos los demás*”. Consideremos como ejemplo un proceso físico: la fisión del Uranio 235. Esta reacción nuclear, tal y como se estudia en los libros de texto, modela un fenómeno físico presuntamente bien conocido: la rotura del núcleo de un cierto elemento. Un objeto “Fisión de uranio enriquecido en la central nuclear de Vandellós, el 12 de marzo de 1980” presenta identidad, estado y comportamiento. La identidad de este objeto particular la determina ya su propio nombre: no describe ningún otro tipo de reacción nuclear o proceso físico, ni otras reacciones del mismo género que hayan ocurrido o vayan a ocurrir en otros lugares o fechas.

El **estado** de un objeto comprende todas sus características. Dentro de estado del objeto que nos sirve de ejemplo existen características o propiedades, como la energía de activación o la velocidad media de los neutrones incidentes contra los nucleidos; algunas permanecen constantes durante el proceso; otras varían, pero deben tener un valor mínimo a lo largo de la vida del objeto (si no es así, el objeto -la reacción- se destruye). Por ejemplo, un cambio en la velocidad de los neutrones que inciden sobre los nucleones de uranio provocaría un cambio de estado en el objeto. La reacción proseguiría, bien de forma más rápida (más nucleones fisionados por unidad de tiempo), bien de forma más lenta (menos nucleones fisionados por unidad de tiempo). En un caso límite, la reacción se extinguiría por falta de neutrones con la velocidad adecuada; sería como si un incendio se parara al borde de un cortafuegos por falta de árboles que propaguen sus llamas.

El **comportamiento** de un objeto es simplemente el modo como actúa y reacciona, dependiendo de sus cambios de estado y del intercambio de mensajes con otros objetos. El comportamiento del objeto que nos sirve de ejemplo (emisión de residuos, de neutrones lentos, etcétera) variará según los estímulos externos: los controladores de la reacción pueden aumentar el número de neutrones emitidos por unidad de tiempo, pueden disminuirlo interponiendo barras de grafito entre las barras de uranio enriquecido, etc.

Esquemáticamente, puede escribirse:

Objeto = Identidad + Comportamiento + Estado

Desde un punto de vista utilitarista (al fin de cuentas, queremos que los objetos trabajen para nosotros, y no al revés), suele considerarse que los objetos ofrecen servicios (u operaciones) a sus clientes. El comportamiento de un objeto se expresa dando su conjunto de operaciones. Un objeto ejecuta una operación cuando recibe –y acepta– una petición, por medio de un mensaje (estímulo).

9. Clases. Tipos de datos y tipos abstractos de datos. Clases y tipos abstractos de datos en los lenguajes de programación: los TADs como pilar de la programación modular y de la programación orientada a objetos. Clases abstractas.

En los dos primeros subapartados se van a considerar por separado los tipos abstractos de datos y las clases, y en el tercero se verán las relaciones entre ambos conceptos. La presentación separada viene dictada porque los tipos abstractos de datos (TAD) aparecieron en la teoría de programación, mientras que las clases (con ese nombre u otros) siempre han formado parte del pensamiento humano.

9.1. Clases.

Informalmente, una clase es la definición abstracta de un objeto. Consideremos algunas definiciones formales: *“Una clase es una descripción de un conjunto de objetos que manifiestan los mismos atributos, operaciones, relaciones y la misma semántica” (Object-Oriented Modeling and Design [Rumbaugh et al., 1991]); “Una clase es un conjunto de objetos que comparten una estructura y un comportamiento comunes” [Booch G., 1994].*

Toda clase puede verse desde tres perspectivas distintas pero complementarias:

- 1) Como conjunto o colección de objetos.
- 2) Como plantilla para la creación de objetos.
- 3) Como definición de la estructura y comportamiento de una clase.

La primera perspectiva apunta a la columna vertebral del concepto de clase: una clase implica clasificación y abstracción. En realidad, una clase no deja de ser la formalización y verbalización de unos procesos que los seres humanos realizamos continuamente, a menudo de forma inconsciente y preprogramada. El ser humano piensa con ideas, con abstractos; y, a medio camino entre la percepción y la cognición, se halla la función de clasificar, es decir, la función de poder afirmar que aquello percibido pertenece a un grupo de cosas (clase) o a otro. Pensar consiste en buscar semejanzas y olvidar diferencias; consiste en abstraer, en generalizar. En el ejemplo del apartado anterior, podríamos considerar a nuestro objeto como un caso particular de una abstracción más general –clase–, cuyo nombre podría ser “Fisión del uranio enriquecido”.

Por la propia naturaleza de nuestro sistema nervioso, moldeado por la selección natural mediante un lento y azaroso proceso de prueba y error, nos resulta difícil razonar con conceptos que rasgan clasificaciones establecidas. La historia de la ciencia está plagada de ejemplos: muchos biólogos durmieron inquietos cuando se descubrió un simpático bichejo, no muy agraciado (no se puede sobrevivir a la época de los grandes lagartos y ser un Adonis del reino animal), llamado ornitorrinco. El ornitorrinco rompía una clasificación fundamental de los zoólogos: la de mamífero y no mamífero. Este curioso animal pone huevos: está en su derecho, por supuesto; pero rompía la tradicional clasificación según la cual los mamíferos tienen pelo, o algo similar,

en alguna parte de su cuerpo, son lactantes siendo crías, son de sangre caliente y NO ponen huevos.

Todos los seres humanos sabemos que un pájaro volando en vertical no deja de ser un pájaro. En contraste, las máquinas y los sistemas de software (incluso las inteligencias artificiales) carecen de la capacidad de abstracción y de esa extraña capacidad informe llamada “sentido común”. En el campo del reconocimiento digital de imágenes, por ejemplo, resulta difícil que los sistemas de reconocimiento *comprendan* que un ave –o un avión, o cualquier forma geométrica– volando en vertical, en horizontal o en cualquier otra orientación continúa siendo la misma ave que en reposo. Es decir, resulta difícil que comprendan que se encuentran ante distintos *estados* de un mismo *objeto*.

Pese a los importantes avances logrados en el campo de las inteligencias artificiales (dejo al lector la cuestión de si vale la pena invertir tanto dinero en IA mientras tantas inteligencias naturales desfilan por la cola del paro), poco se ha avanzado en el desarrollo de sistemas de IA generales. Existen IAs especializadas (*Deep blue*, sistemas expertos) de gran eficacia, pero no se han construido IAs de carácter general. En consecuencia, el análisis OO, basado en la abstracción y que incluye la identificación de las clases del dominio del problema, continúa siendo una actividad humana. Se dará una breve visión del análisis y diseño OO en el Apdo. 15.

Curiosamente, las tareas más simples o inmediatas para nosotros son las más complicadas de analizar e implementar por medio de ordenadores. Incluso los robots mejor diseñados tienden a confundirse ante situaciones elementales para las personas. Así, a duras penas un robot identificará una puerta cerrada con la misma puerta mientras se abre, pese a que las imágenes captadas por aquella son óptimamente mucho más perfectas que las formadas por el ojo humano emétrope. Es más, si la puerta se va a cerrar por un golpe de aire, mejor será taparse los oídos, pues la identificación de las sucesivas posiciones de la puerta, el cálculo de su velocidad y la parametrización de los movimientos necesarios para impedir el cierre brusco provocarán una suerte de parálisis y perplejidad algorítmica en el robot. Perplejidad semejante a la que sufriría al leer *El ser robótico* y *la nada* en una sentada.

En la segunda perspectiva se destaca que la relación entre una clase y los objetos derivables de ella puede considerarse como la existente entre una fabrica y las cosas producidas por ésta. Un ejemplo: una fábrica de automóviles produce automóviles del mismo modo que una clase *Automóvil* crea objetos *automóviles*. Una clase *Automóvil* solamente producirá objetos *automóviles*, del mismo modo que una fabrica real de automóviles sólo produce coches, no televisores o aviones.

La tercera hace hincapié en que la definición de una clase permite definir una sola vez la estructura común, así como usarla cuantas veces sea necesario.

A la acción de crear objetos de una clase se le llama instanciar; y a los objetos así creados, instancias de la clase. Por así decirlo, una clase equivaldría a los planos de un edificio; los objetos serían los edificios que pueden construirse con los planos; y el programador se encargaría de cobrar el alquiler.

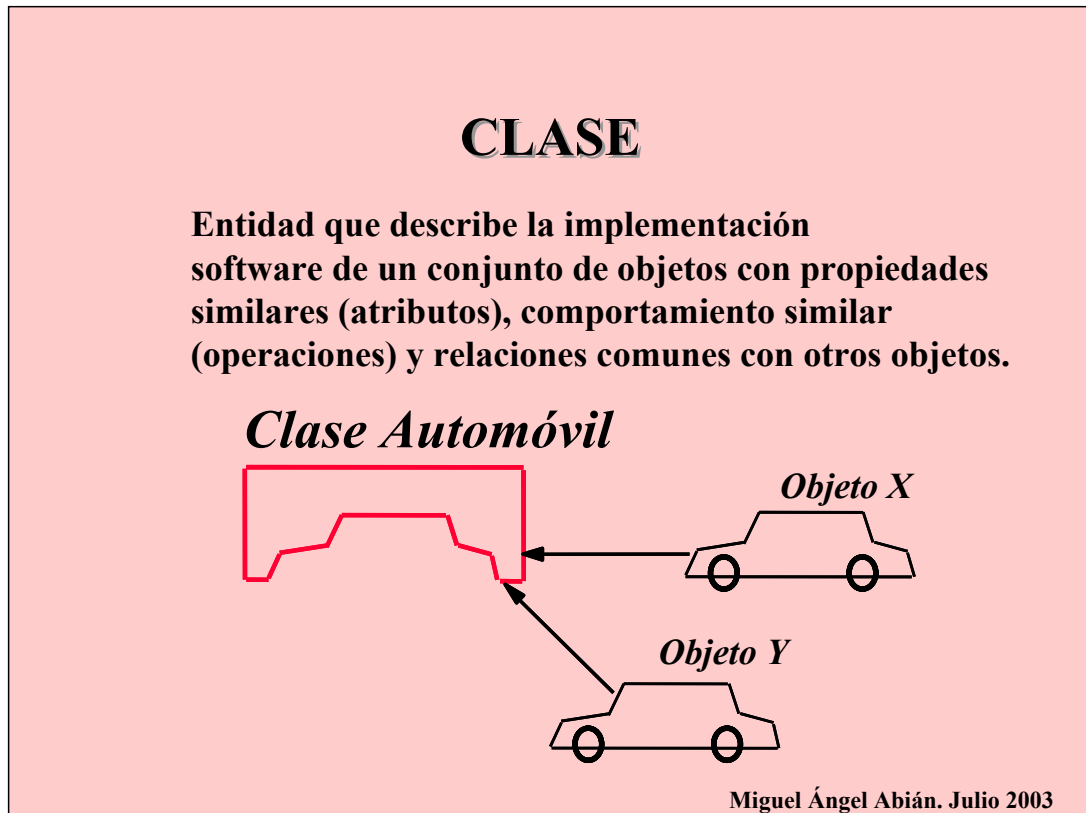


Figura 7. Las clases como fábricas de objetos

Las clases definen *atributos* (vinculados con el **estado** de los objetos derivados de ellas) y *operaciones* (las cuales establecen el **comportamiento** de las instancias). Esquemáticamente:

Clase = Atributos + Operaciones + Instanciación

Se llama atributo a la abstracción de una característica común para todas las instancias de una clase, o a una propiedad o característica de un objeto. Las operaciones de una clase son servicios ofrecidos por ésta, que llevan (o pueden llevar) a cambios en el estado de un objeto de dicha clase. Se dice que una clase *A* es *cliente* de una clase *B*, si *A* contiene una declaración en la cual se establezca que cierta entidad *E* (atributo, parámetro, variable local) es de tipo *B*. Conceptualmente, una operación puede considerarse como una petición a un objeto para que haga algo.

Cuando las clases se implementan en lenguajes OO, se habla de *variables de instancia* (implementaciones software de atributos) y de *métodos* (implementaciones software de operaciones). La diferencia entre método y operación resulta importante: véase el Apdo. 10. Cada instancia (objeto) de una clase (u objeto) contiene su propio conjunto de variables de instancia, cuyos valores pueden variar con el tiempo. El conjunto de los valores tomados por las variables de instancia de un objeto, en un instante dado, representa el estado del objeto. O dicho en sentido inverso, el estado de un objeto viene representado por los datos almacenados en sus variables de instancia.

Cuando se consideran sistemas de software, enseguida se percibe la tendencia evolutiva de los lenguajes de programación hacia la representación y reproducción de la manera como contemplamos el mundo que nos rodea. Muy pocas personas comprenden a la primera la secuencia de código ensamblador que permite sumar dos números enteros; algunos individuos –extremadamente escasos– incluso pueden pensar directamente en binario; casi todos, sin embargo, entendemos que una clase *Entero* representa el conjunto de los números enteros, y que éstos tienen su propio comportamiento, en el cual se incluye la operación suma.

Con la evolución y el desarrollo de lenguajes de programación reproducimos el modo de pensar al cual nos ha conducido la evolución, acompañado a veces de extrañas mutaciones, pasos hacia atrás y atavismos. Nuestro acervo de lenguajes de programación, todavía muy primitivo, tiende a ver el mundo tal y como nosotros los vemos. ¿Para qué reinventar lo que ya funciona?

9.2. Tipos de datos y tipos abstractos de datos.

Las clases, los tipos de datos y los tipos abstractos de datos (TADs) son conceptos estrechamente relacionados.

Un **tipo de datos** (o, abreviadamente, un tipo) consiste en una colección de valores, con una representación asociada (llamada, en programación, **estructura de datos**), y un conjunto de operaciones definidas sobre ellos; al conjunto de valores se le llama **dominio del tipo**. Por ejemplo, el tipo de datos *int* representa valores como -2, -1, 0, +1, +2,... sobre los cuales están definidas operaciones como suma, multiplicación, etc. Los tipos de datos describen unidades elementales de información en un sistema particular de software.

Se dice que una variable o identificador es de un tipo cuando sólo puede contener valores pertenecientes al dominio de dicho tipo. Del mismo modo, un objeto pertenece a un tipo (o es instancia de un tipo) si los valores de sus atributos se encuentran incluidos dentro de la colección de valores permitidos por el tipo y si el objeto cuenta con el conjunto de operaciones (interfaz) establecida por el tipo.

Cuando declaramos una variable como *int* (en C, Java o C#, por caso), ¿qué estamos haciendo en realidad? Bajo las bambalinas, estamos indicando al compilador correspondiente que la variable

- a) Sólo puede tomar valores de un conjunto (el dominio del tipo) de valores posibles. En este caso, un intervalo acotado del conjunto de los números enteros.
- b) Sólo admite como aceptables un conjunto de operaciones: aquellas definidas para los valores del dominio del tipo.

Cualquier incumplimiento de a) o b) derivará en un error en tiempo de compilación. La declaración de la variable reserva espacio en la memoria disponible para el programa, de modo que pueda albergar en ese espacio algún valor extraído del dominio de *int*. El tipo de datos *int*, al igual que los tipos primitivos, no dispone de una implementación en el ámbito del código fuente: el

compilador proporciona una implementación.

Hoy día, cualquier programador ha trabajado con lenguajes con tipos; pero cuando se introdujeron los tipos se sentó el *Aleph* de la programación moderna. Gracias a ellos, los programadores pudieron –y pueden– olvidarse de las máquinas sobre las cuales trabajaban y de las representaciones internas de los datos. Si por algún motivo se cambia la representación interna de los tipos, basta con recompilar el programa; no es necesario reconstruir en la nueva representación todo el código ya escrito.

Los tipos proporcionan un modo de eliminar errores de memoria, pues un sistema de tipos asegura que todos los accesos a memoria son compatibles en cuanto a tipo. Basta con combinar un sistema de tipos con una gestión automática de la memoria y ciertas comprobaciones en tiempo de ejecución para que la implementación de un lenguaje quede completamente exenta de ciertos errores de memoria.

Con los tipos pueden realizarse comprobaciones imposibles en código máquina o en ensamblador y evitar, por consiguiente, muchos errores de memoria. Internamente, los ordenadores procesan y manipulan secuencias de bits. Que una cadena de bits represente un número entero, de coma flotante, un carácter alfanumérico o una instrucción nativa de la arquitectura sobre la cual se ejecuta el programa no importa mucho en ese ámbito. Cualquier operación de la máquina podría aplicarse a cualquier dato. En el lenguaje ensamblador tampoco existen comprobaciones de tipos.

Los sistemas de tipos constituyen una gran ayuda para el programador, si bien no pueden proporcionar una seguridad completa. Ciertos resultados de la teoría de autómatas apuntan a que ningún sistema de tipos puede especificar la totalidad de las entradas válidas para un sistema de software. Consecuentemente, un diseñador de lenguajes debe alcanzar un compromiso entre simplicidad y seguridad. Visual Basic, por ejemplo, permite asignar una fecha a una variable de tipo *String*.

Los lenguajes de programación suelen clasificarse en lenguajes de tipos (de datos) estáticos o dinámicos. Los primeros (COBOL, Fortran, Pascal, C/C++, Object Pascal, Java, Ada, etc.) comprueban en tiempo de compilación el tipo de cada variable; los segundos (Smalltalk, Lisp, Snobol, Self, CLOS, etc.) comprueban el tipo de las variables en tiempo de ejecución. En estos últimos, los tipos se asocian con valores; en los otros, los tipos se asocian con variables.

Cualquier lenguaje de tipos estáticos dará error durante la compilación del siguiente código (o uno equivalente):

```
int i;  
i = "Mi primer error"; //error
```

pues el compilador detectará al instante la invalidez de asignar un valor de tipo *float* o *double* a una variable declarada como de tipo *int*. Un lenguaje de tipos dinámicos, en cambio, sólo detectará el error en tiempo de ejecución (donde de nada sirven las lamentaciones y los arrepentimientos).

Los lenguajes de tipos estáticos –que son la mayoría– presentan dos ventajas fundamentales: a) aumentan la seguridad, pues permiten detectar

muchos errores durante la fase de compilación; b) suelen generar código ejecutable más veloz y de menor tamaño que el de los lenguajes de tipos dinámicos, pues la comprobación de tipos no ocupa tiempo ni memoria durante la ejecución (ya se ha realizado en la etapa de compilación). Su desventaja principal reside en la rigidez: no permiten la importación de nuevos tipos por parte de programas en ejecución y complican la implementación de estructuras dinámicas por naturaleza (como árboles, pilas, listas, etc.).

Pascal nos brinda, desafortunadamente, un buen ejemplo de inflexibilidad. En este lenguaje no pueden programarse rutinas generales de ordenación de *arrays*, listas o árboles. El sistema de tipos de Pascal necesita que los tipos de aquellos datos dentro de colecciones como *arrays*, listas o árboles formen parte de las declaraciones de los procedimientos o funciones que manipulan los datos. Así pues, deben escribirse procedimientos de búsqueda, ordenación, etc., para cada par colección/tipo de datos. Imposibilidad de reutilización del código, falta de flexibilidad para el programador y confusión conceptual para los programadores inexpertos: éstas son las consecuencias de la rigidez del sistema de tipos de Pascal. ¿Por qué esa tercera acusación? Pues porque Pascal obliga al programador a fijarse en los elementos dentro de las colecciones, cuando en realidad debería centrarse en la colecciones (métodos, eficacia, etc.) y manipular los datos de forma genérica, no específica. Las colecciones pueden perfectamente entenderse y usarse sin conocer su implementación interna ni los datos que pueden contener.

Los lenguajes de tipos dinámicos son el polo opuesto: ofrecen una flexibilidad total, a cambio de lentitud (todas las comprobaciones se hacen en tiempo de ejecución) y de una disminución considerable de la seguridad.

Los lenguajes también se clasifican en lenguajes de tipos fuertes (o *fuertemente tipados*) y de tipos débiles (o *débilmente tipados*). Se llama tipificación o tipado al proceso de declarar qué tipo de información puede contener una variable o identificador. En el primer caso nos encontramos ante lenguajes en los que el tipo de cada variable ha de ser definido. En los lenguajes de tipos débiles no se comprueba el tipo de cada variable o no resulta obligatorio definirlo; por ejemplo, en Visual Basic se puede concatenar la cadena de texto "23" con el entero 7, y después tratar la concatenación como un entero, sin que se precise efectuar ninguna conversión explícita de tipos.

Los lenguajes de tipos estáticos no están obligados a ser también de tipos fuertes. Tampoco existe ninguna restricción por la que los lenguajes de tipos dinámicos no puedan ser también fuertemente *tipados*. Python, por caso, cae dentro de ambas clasificaciones.

Un **tipo abstracto de datos** es la noción matemática que define un tipo de datos. Como su propio nombre indica, constituye una descripción abstracta de un tipo de datos. En esta descripción abstracta y formal (matemáticamente bien definida), las propiedades de las operaciones se especifican solamente con axiomas. La especificación por axiomas de un tipo se compone de una **especificación sintáctica** (en la cual se definen los nombres, dominios y rangos de las operaciones sobre el tipo), y de una **especificación semántica**. Esta especificación suele darse mediante un conjunto de axiomas, escritos en forma de ecuaciones, que establecen cómo opera cada operación sobre otras (aproximación algebraica). También puede usarse, pese a ser menos

frecuente, lo que llamaré aproximación abstracta o de modelado. Consiste en describir el significado de las operaciones en términos de las operaciones sobre otros tipos abstractos, los cuales pueden estar especificados algebraicamente. En suma, un TAD se define por un número de operaciones aplicables, el modo como puede invocarse cada operación (la sintaxis) y sus efectos (la semántica).

Los TADs se definen únicamente por su comportamiento exterior (operaciones), no por su estructura. En los tipos de datos, por el contrario, las operaciones sobre los valores suelen estar implícitas. En un TAD, los valores están definidos implícitamente en términos de las operaciones. Resulta posible construir distintas versiones de un mismo TAD: el tipo *INTEGER* de Pascal y el *int* de Java implementan, de formas muy distintas, un mismo TAD. Una implementación de un TAD es una *instancia* del TAD.

Un TAD no se circunscribe a ningún hardware, ni a determinadas implementaciones de software; su naturaleza es matemática, y su propósito reside en especificar la estructura, las reglas y las aserciones sobre un concepto de datos, así como las funciones que puede proporcionar ese concepto. No deben confundirse los tipos abstractos de datos y los tipos de datos, pues **un TAD no es un tipo de datos**: no contiene ninguna representación del tipo que define (o, dicho de otra forma, de los valores del tipo). Podríamos considerar que la palabra abstracto, para un TAD, significa “sin relación con ninguna cosa real” o “concebido aparte de realidades concretas”.

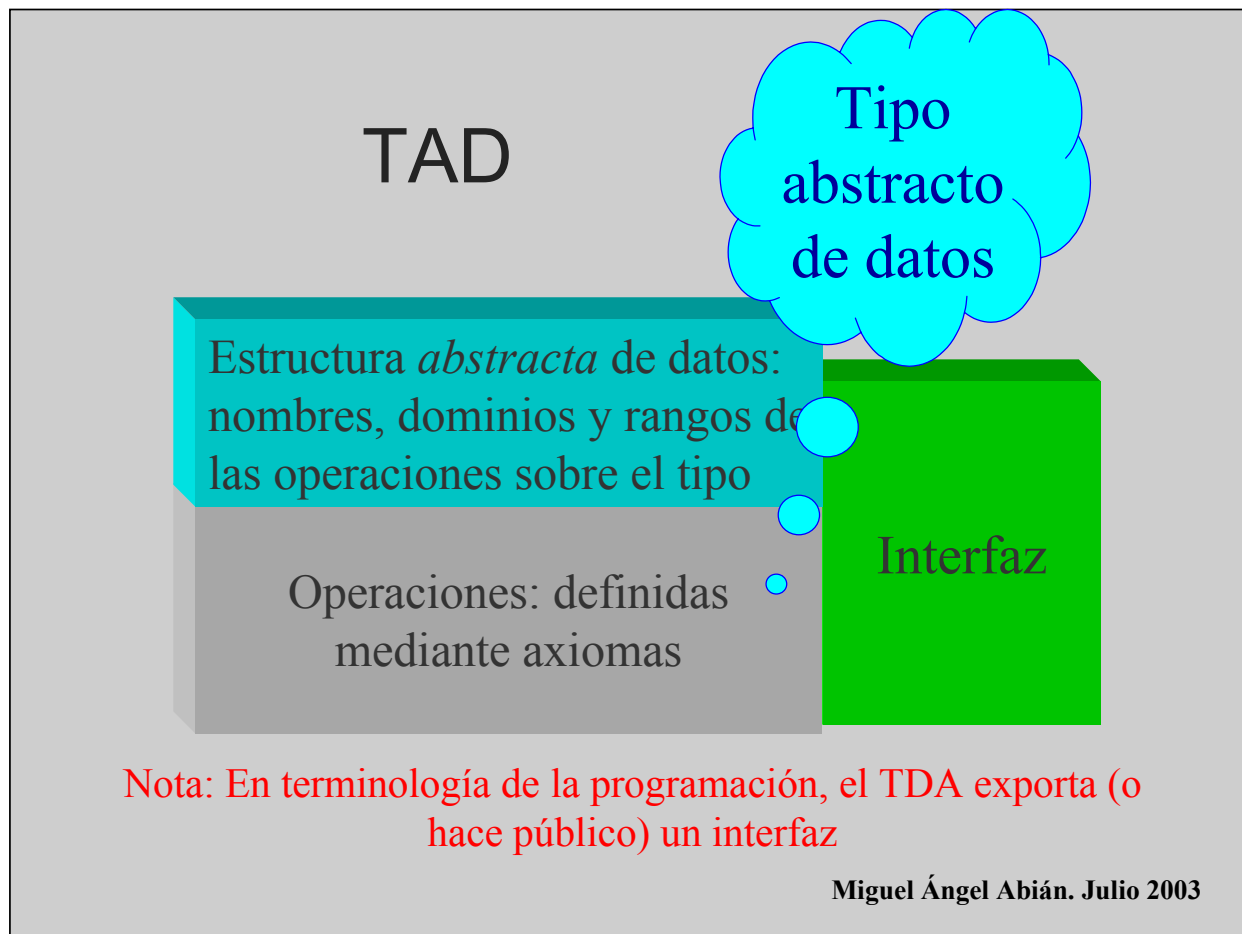


Figura 8. Esquema de un tipo abstracto de datos

Todo TAD cumple dos principios:

- **Ocultación de la información (encapsulación):** Sólo puede accederse a los valores del tipo que define, o modificarlos, mediante las operaciones abstractas definidas sobre ellos. La encapsulación hace referencia al ocultamiento de la información y a la capacidad para expresar la unidad formada por los valores y las operaciones.
- **Abstracción de datos:** Se separan las propiedades lógicas de los datos de su representación o implementación.

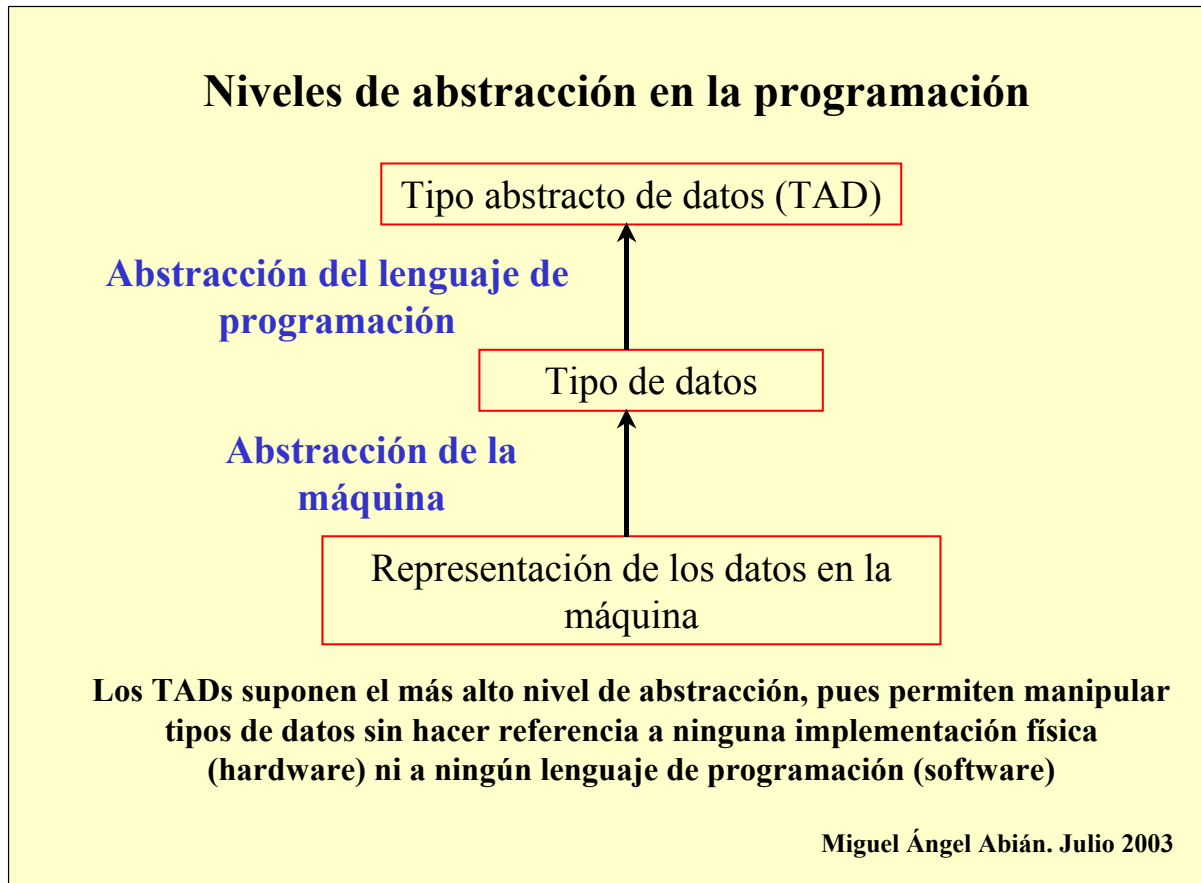


Figura 9. Diagrama de los sucesivos niveles de abstracción

Como ejemplo sencillo, veamos cómo puede especificarse formalmente un TAD *Natural* (representa el concepto de número natural):

TAD Natural

Nombre: Natural

Dato: Secuencia de dígitos precedida de un signo positivo

Operaciones:

crear \rightarrow Natural
 $=$: (Natural, Natural) \rightarrow Booleano (verdadero, falso)
 $<$: (Natural, Natural) \rightarrow Booleano (verdadero, falso)
 $+$: (Natural, Natural) \rightarrow Natural
 \times : (Natural, Natural) \rightarrow Natural

Axiomas y operaciones:

0 –cero– es un número natural.
Si n es un número natural, también lo es $S(n)$ [S es la operación Siguiente]
 $(S(n) = 0) = \text{falso}$
si $S(n) = S(m) \Rightarrow m = n$
 $(n + 0) = n$
 $(m + S(n)) = S(m + n)$
 $(n \times 0) = 0$
 $(m \times S(n)) = ((m \times n) + m)$
 $0 < S(0)$
si $m < n$ entonces $m < S(n)$ y $S(m) < S(n)$

donde m, n son números naturales (datos del TAD *Natural*)

Como vemos, el TAD se presenta en la forma de una especificación algebraica, la cual se basa en una notación formal (es decir, matemáticamente bien definida) y un sistema de razonamiento e inducción. Un álgebra consiste en un dominio de valores (naturales, booleanos, etc.), junto con un conjunto de operaciones (funciones) definidas sobre el dominio. El símbolo "=" es parte del álgebra y del metalenguaje matemático, pero guarda relación con el operador asignación de muchos lenguajes de programación. Podría haber usado una operación *igual* para diferenciar entre símbolo del álgebra y del metalenguaje, pero eso complicaría innecesariamente la notación, amén de no corresponder con la práctica matemática.

No cabe impresionarse por la especificación algebraica; sólo define rigurosamente conceptos por todos conocidos: no hay un número natural cuyo siguiente natural sea 0, el número natural siguiente a uno dado siempre es mayor que éste último, la suma de un natural y cero da como resultado el natural... Nada nuevo bajo el sol, en realidad; pero utilizar una notación formal evita las ambigüedades que suelen poblar el lenguaje natural y permite –siguiendo unas normas de razonamiento– extraer conclusiones no evidentes, además de posibilitar la verificación formal de programas. El caso más extremo de lenguaje no formal que conozco aparece en *Alicia en el país de las Maravillas*: “*De otro modo –continuó Tweedlede– si fuera así, podría ser y si así fuera, así sería; pero como no es así, no lo es. Es lógico.*”

Nuestros bien conocidos 1, 2, 3,... no son más que los nombres (nuestra representación de los naturales) que asignamos a $S(0)$, $S(S(0))$, $S(S(S(0)))$... Usamos esta representación porque tenemos diez dedos, lo cual no deja de ser una casualidad: si fuéramos ciempiés inteligentes nuestras tablas de multiplicar

no cabrían en un lápiz, y aprender los nombres de todos nuestros números sería ardua tarea, casi posible sólo para ciempiés con bachillerato superior.

Nuestro ineludible antropocentrismo no sólo se aprecia en nuestra aritmética, sino también en las máquinas que construimos. El primer ordenador digital que se construyó contaba –qué casualidad– con aritmética decimal. La complejidad de la circuitería necesaria para realizar operaciones con notación decimal resultó tan alta que nuestro antropocentrismo tuvo que abandonarse bajo la alfombra, y hubo que adoptar la aritmética binaria (no conozco, en la actualidad, ningún ordenador que use internamente aritmética decimal). Supongo que a un circuito un número decimal debe parecerle algo así como una gárgola patituerta, por grotesco, extraño y antinatural. Pero ¿cómo va a caber algo decimal por circuitos que sólo admiten encendido/apagado? A empujones, claro está.

Por otro lado, e independientemente de la representación, los axiomas resultan fundamentales para definir cómo deben comportarse los números naturales. Sin ellos, $2 + 3$ podría dar perfectamente 0, en lugar de 5. Los axiomas **obligan** a que $S(S(0)) + S(S(S(0)))$ sea igual a $S(S(S(S(S(0)))))$; es decir, a que $2 + 3 = 5$, usando la notación a la cual estamos acostumbrados desde niños.

Este ejemplo tan simple ya muestra dos características esenciales de los tipos abstractos de datos, vinculadas a los principios de ocultación de la información y de abstracción de datos:

- 1) Las operaciones proporcionan una especificación del tipo independiente de la representación de éste: indican qué hacen en términos de valores y tipos, pero no cómo implementarlas. Los axiomas establecen cómo se comportan unas operaciones con respecto a otros; pero no aseveran nada acerca de cómo manipular o almacenar los valores del tipo. En nuestro ejemplo, un TAD *Natural*, no se considera la forma de representar o manipular los números naturales: podrían representarse mediante cuentas de ábaco, dedos, o ceros y unos en celdas de memoria RAM. Estas tres formas de representación, con sus limitaciones, son estructuras de datos del tipo *Natural*. Lo importante es que, tal y como están definidas las operaciones, la suma de dos y tres dará cinco, independientemente de que usemos dedos, canicas o números binarios para representar los sumandos y el resultado.
- 2) Los TADs se definen operacionalmente, no estructuralmente. Es decir, se definen en función de las operaciones (o funciones) que permiten, no de su estructura interna.

Otro ejemplo de TAD, más complejo que el TAD *Natural*, es el de una pila. De acuerdo con la intuición, una pila es un contenedor de elementos que se introducen y se eliminan siguiendo el principio “último en entrar-primero en salir” (LIFO en inglés). Los elementos pueden añadirse en cualquier momento, pero sólo el último –aquél introducido más recientemente– puede eliminarse. A insertar un elemento en la pila se le llama apilar; a eliminar uno, desapilar.

Un TAD Pila puede especificarse así:

TAD Pila

Nombre: Pila

Dato: Pila de elementos del mismo tipo (tipoElemento, para abreviar)

Operaciones:

crear -> Pila

apilar: (Pila, tipoElemento) -> Pila

desapilar: (Pila) -> Pila

cima: (Pila) -> tipoElemento

estaVacio: (Pila) -> Booleano (verdadero, falso)

Axiomas:

Para toda P: Pila y todo j: tipoElemento se cumple que:

EstaVacio (crear) = verdadero

EstaVacio (apilar (P,j)) = falso

cima (crear) = error

cima (apilar (P,j)) = j

desapilar (crear) = error

desapilar (apilar (P,j)) = Pila

Los axiomas del TAD *Pila* constituyen, como sucedía con el TAD *Entero*, la especificación de qué debe hacer una pila, no de cómo debe hacerlo. Vienen a decir que una pila recién creada está vacía; que cuando una pila tiene al menos un elemento, no está vacía; que intentar extraer un elemento de una pila que se acaba de crear produce un error... Lo que se espera de una pila, en definitiva. Cualquier descripción de la implementación de un TAD *Pila* enturbiaría las propiedades algebraicas de las pilas (la *semántica* de la pila, común a toda pila), además de añadir la complejidad de considerar todas las posibles implementaciones de aquéllas (*arrays*, listas enlazadas, etc.).

La descripción algebraica de una pila resulta muy interesante debido, precisamente, a su alto nivel de abstracción. Fijémonos en dos detalles: **a)** Al considerar un *tipoElemento* genérico, no particularizamos para ningún tipo de dato concreto: muy bien podríamos considerar en la pila números naturales, enteros, complejos, neumáticos u ornitorrincos (suponiendo que se dejen); **b)** En ningún momento se especifica cómo implementar las operaciones –sólo lo que deben hacer en cuanto a valores–, ni se menciona cómo deben manipularse o almacenarse los datos. En el mundo real, los datos pueden representarse en posiciones de memoria de un ordenador, pero también en un papel o mediante manipulaciones mecánicas en el espacio (imaginemos una pila de neumáticos de la cual extraemos elementos).

Al igual que podemos escribir, a partir de los axiomas del TAD *Entero*, ecuaciones como ésta para los naturales:

$$a + b + c = a + c + b,$$

los axiomas del TAD Pila también permiten escribir ecuaciones como la siguiente:

$$\text{desapilar}(\text{desapilar}(\text{apilar}(\text{apilar}(P, a), b), c))) = \text{desapilar}(\text{desapilar}(\text{apilar}(\text{apilar}(P, a), c), b)))$$

Muchas veces se denominan **estructuras de datos** a las pilas, colas, árboles, etc., sobre todo en libros de texto de los años 60 y 70. Cuando se usa este término, suele darse por sentado que se está tratando la implementación de aquéllos. Las colas, los árboles, etc., son tipos abstractos de datos; el uso del término “estructura de datos” resulta confuso, pues “estructura” parece indicar una organización de la memoria, cuando los TADs no necesitan representación ni –por tanto– memoria. Incluso los tipos primitivos no necesitan memoria: son las declaraciones de las variables de tipos las que precisan memoria, pero no las definiciones de los tipos.

Al considerar lenguajes de programación, pueden matizarse las relaciones entre TADs y tipos de datos:

- a) Un TAD es un modelo puramente matemático, y carece de la noción imperativa de estado (variables que cambian durante el tiempo). En un TAD no existen bucles, *if*, u otras estructuras propias de los lenguajes de programación.
- b) Los TADs son herramientas abstractas para el diseño de tipos de datos.
- c) Para usar un ADT en un programa se precisa implementarlo mediante tipos de datos (quizás predefinidos en el lenguaje, y quizás no). Lenguajes como Fortran no permiten la creación de tipos de datos que no figuren en el lenguaje.
- d) Un tipo de datos es un concepto de los lenguajes de programación. Muchas veces, los TADs no pueden representarse usando tipos de datos, debido a las limitaciones inherentes a los ordenadores (memoria finita, representaciones discretas de la información). Un tipo de datos *float* o *double* no puede representar exactamente a un TAD *Real*, cuyo dominio es infinito.
- e) La representación formal de un TAD emplea un lenguaje, el matemático, más abstracto y bien definido que el de cualquier lenguaje de programación. Aunque no hay pruebas formales, se tiende a pensar que los problemas de un cierto nivel de abstracción se solucionan mejor usando niveles superiores de abstracción.
- f) Al igual que puede pensarse en los tipos como en materializaciones software de los TADs, puede pensarse asimismo a la inversa: un tipo abstracto de datos puede concebirse como un tipo cuya representación como tipo concreto ha sido abstraída, y a cuyos datos (los valores del dominio del tipo) no puede accederse si no es mediante un conjunto de operaciones.

Para cotejar un ADT con algo que no lo es, consideremos el siguiente código escrito en C:

```
struct nodo_lista_enlazada {  
    void *dato_nodo;  
    nodo_lista_enlazada *proximo_nodo_en_lista;  
};
```

Este es un posible modo de definir una lista enlazada en C, pero no es el único modo; ni siquiera es necesario utilizar un lenguaje con punteros (Java no los tiene) para programar listas enlazadas. El código arriba escrito podría escribirse de muchas maneras distintas en diferentes lenguajes de programación (incluso dentro de uno en concreto podría escribirse también de otros modos: usando arrays, etc.), y seguiría siendo una descripción software de una lista enlazada. No es un TAD lo que tenemos arriba, pues un TAD posee una existencia abstracta, ideal; **la idea** que podemos extraer del código –lista enlazada: estructura consistente en una serie de nodos donde cada nodo almacena algún tipo de información, y donde puede encontrarse la posición del próximo nodo en la lista– sí conforma (debidamente formalizada) un TAD.

Cuando se implementa un TAD, deben tenerse en cuenta los tres componentes de la implementación:

- a) La especificación del TAD: operaciones, axiomas (a veces también se utilizan los axiomas con el nombre de precondiciones y poscondiciones).
- b) Alguna representación interna (estructura de datos) para los valores del tipo (datos), que especifique cómo se almacenan en memoria los valores. Por ejemplo, si intentamos representar datos del tipo coma flotante, resulta adecuado recurrir a la normalizada representación definida por el estándar IEEE 754 (*Standard for binary floating point arithmetic*). Aun así, el programador goza siempre de libertad para elegir la representación que estime oportuna, si bien debe ser compatible con la implementación elegida para los procedimientos.
- c) Un conjunto de subprogramas (funciones o procedimientos): cada uno implementará una de las operaciones de la especificación para el TAD. Las implementaciones no podrán ser arbitrarias, ya que las funciones se encontrarán restringidas por los axiomas del TAD. Para el caso de la pila, existirán funciones como *cima*, *apilar*, *desapilar*, etc.

Cualquier implementación software correcta (o completa) de un TAD debe seguir cumpliendo los principios de ocultación de la información y de abstracción de datos, además de cumplir con los axiomas especificados por el TAD. Así, la implementación deberá ocultar la representación del tipo (su estructura de datos), y sólo podrá accederse a sus valores (datos) mediante las funciones (interfaz).

Los lenguajes OO permiten, en líneas generales, la implementación de tipos abstractos de datos mediante la estructura *class*, es decir, por medio de

clases. Las ventajas de implementar correctamente TADs son muchas. Por un lado, las estructuras de datos y los algoritmos quedan escondidas tras las operaciones, obteniéndose así una abstracción que simplifica la programación de los clientes usuarios de la implementación. Como solamente la interfaz del TAD es visible desde el exterior, una implementación de un TAD puede conceptualizarse mediante un conjunto de operaciones.

Por ejemplo, puede considerarse que un hospital se caracteriza por medio de dos operaciones *ingresarPaciente* y *darAltaPaciente*. Cualquier programa cliente que use un tipo *Hospital* y sentencias como *ingresarPaciente*("Javier Pérez", "01/07/03") y *darAltaPaciente*("Javier Pérez", "15/07/03") será mucho más legible que un programa que acceda directamente a las estructuras de datos descriptoras del hospital.

Por otro lado, cualquier modificación de los algoritmos y estructuras de datos que subyacen bajo el hospital será invisible para los clientes usuarios del interfaz (*ingresarPaciente* y *darAltaPaciente*). Si se cambiara todo el sistema de gestión de los ingresos y altas (en el mundo real) y las estructuras de datos y algoritmos que sustentan el hospital (en el modelo de software del hospital), los programas clientes que usaran su interfaz seguirían funcionando correctamente. Siempre, desde luego, que la interfaz no hubiese sido modificada.

El lector habrá percibido ya las dos caras de la abstracción matemática: resulta más difícil de comprender al principio, pero aporta una generalidad que permite avanzar mucho más de lo que uno imaginaba al principio. Pensemos por un momento en los números complejos; inicialmente un número complejo resultaba absurdo: ¿qué número real resulta de hacer la raíz cuadrada de -1 ? Ninguno. Pero a alguien con espíritu explorador se le ocurrió aplicar a los números imaginarios las mismas operaciones aplicables a los reales (suma, resta, multiplicación, potencias, etc.). ¿Resultado? Los números imaginarios se utilizan hoy en óptica, electricidad y electrónica, hidrodinámica, física cuántica, y en matemática pura y aplicada. Incluso algunos espíritus demasiado exploradores se salieron por la tangente: el psiquiatra Jacques Lacan escribió sin sonrojarse que el órgano sexual humano es "equivalente a la raíz cuadrada de menos uno". Desconozco si esta inverosímil línea de investigación ha llevado a resultados empíricos... Una vez que el genio sale de la botella, resulta imposible volver a meterlo en ella.

TADs, tipos de datos y estructuras de datos

- **TAD**: Modelo matemático, abstracto, de grupos de datos junto con las operaciones que los manipulan

- Las operaciones definen el TAD.
- Por tanto, el TAD se define operacionalmente, no estructuralmente.
- La especificación del TAD no incluye consideraciones acerca de su implementación
- Algunos autores llaman, a veces, TADs a las implementaciones de los TADs.
- Aun no recurriendo a una especificación formal del TAD, suele resultar muy útil para el programador especificar en lenguaje natural lo que se espera de aquél.

- **Tipo de datos**: Implementación software del modelo matemático que especifica el TAD.

- Los tipos primitivos (*int*, *char*, *long*, etc.) son implementaciones de TADs proporcionada por los lenguajes de programación.
- Se habla propiamente de tipos de datos cuando son implementaciones correctas de TADs. Es decir, cuando cumplen los principios de encapsulación (ocultación de la información) y abstracción.
- Un tipo de datos tiene asociada una representación en memoria de los datos (**estructura de datos o de almacenamiento**) y la implementación de las operaciones del TAD al cual implementa.
- La representación de los tipos de datos definidos por el usuario se realiza de acuerdo con los tipos de datos existentes en el lenguaje.
- Si un tipo de datos del lenguaje resulta soportado directamente por el hardware subyacente, suele usarse la misma representación de memoria de la arquitectura hardware para poder usar las operaciones proporcionadas por el hardware. En caso contrario, estas operaciones se realizan por software.
- Las operaciones suelen implementarse como operaciones de hardware (la manera más rápida y eficaz) o como subprogramas.
- Una variable de un tipo de datos es una instancia de un tipo de datos, y los valores que puede tomar corresponden a los del dominio del tipo. La ligadura entre una variable y un valor puede ser modificada en tiempo de ejecución mediante operaciones de asignación (como $k = k+1$).

Figura 10. Cuadro resumen de las características de los tipos abstractos de datos, los tipos de datos y las estructuras de datos

9.3. Clases y tipos abstractos de datos en los lenguajes de programación: los TADs como pilar de la programación modular y de la programación orientada a objetos.

Al considerar lenguajes de programación o sistemas de software, todo lo expuesto acerca de los tipos abstractos de datos puede resumirse, usando vocablos de programación, en que un TAD se caracteriza por las siguientes propiedades:

- a) Define un tipo de datos (**tipo**).
- b) Define (y hace visible desde el exterior) una serie de operaciones (**métodos**). Al conjunto de todas las operaciones se le llama **interfaz**.
- c) Las operaciones de la interfaz son el único mecanismo de acceso a las **estructuras de datos** del tipo.
- d) Los axiomas definen el **dominio de aplicación** del tipo.

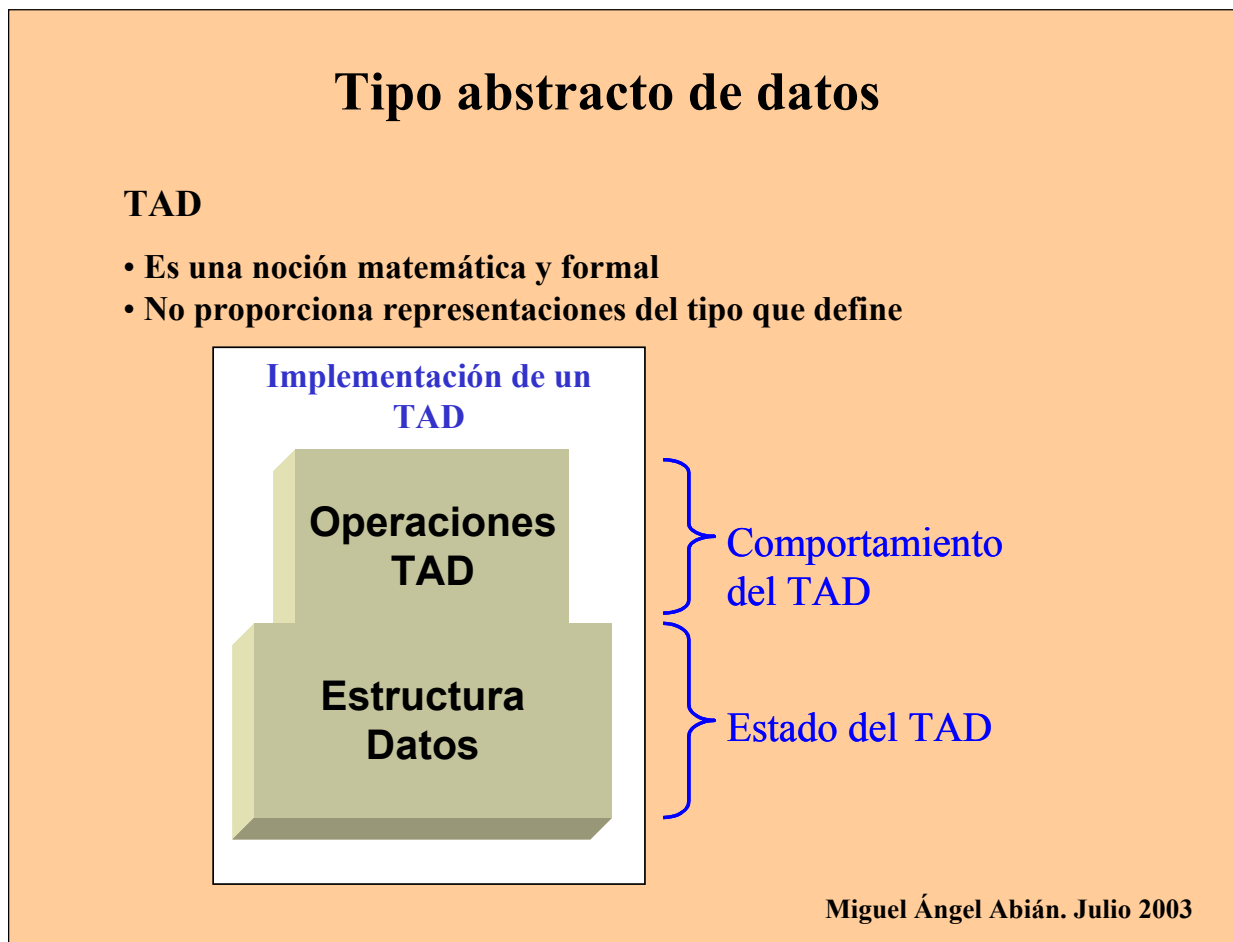


Figura 11. Representación de la implementación por software de un TAD

Todos los tipos de datos predefinidos –también conocidos como tipos primitivos, básicos,...– en los lenguajes de programación (como *int*, *long*, *float*, *char*... incluso el poco añorado *INTEGER* de FORTRAN I) se basan en TADs. Muchos autores –algunos influyentes– y textos consideran que los tipos predefinidos son TADs; en mi opinión, esta identificación proviene de un lamentable (pero persistente) error conceptual: un tipo predefinido es una **implementación** de un TAD, no un TAD. A nadie se le ocurriría afirmar que un pliego de condiciones para amueblar un aeropuerto es –o será– el mobiliario del aeropuerto, o que el plano de un edificio es el edificio; en ingeniería del software, sin embargo, parece habitual confundir las especificaciones de las cosas con las cosas mismas. Démosles a los objetos la dignidad de llamarlos por su propio nombre.

Creo interesante remarcar, por última vez, la diferencia entre tipos predefinidos y TADs:

1) Un TAD es una noción matemática: no depende de ningún lenguaje de programación ni de ningún hardware.

2) Un tipo predefinido es una implementación concreta (y generalmente no única) de un TAD; depende del lenguaje de programación y del hardware empleados. El tipo predefinido *int*, por poner un ejemplo, puede representarse como números binarios de complemento a dos o como números binarios en signo y magnitud. Distintos lenguajes de programación y arquitecturas hardware pueden asignar distinto número de bytes a un mismo tipo.

En síntesis: un TAD especifica el qué, no el cómo; un tipo predefinido especifica el cómo, basándose en el qué, y lo oculta al programador.

Consideremos lo que sucede cuando escribimos un código tan inofensivo como éste:

```
int k;  
k= 7;  
k = k + 1;
```

Al compilar y ejecutar el programa, el compilador reserva memoria para una variable de tipo entero, a la que llama *k*. Cuando nos referimos a la variable *k* nos referimos, en realidad, a la zona de memoria reservada en el momento que se creó. Si aplicáramos el test de Voight-Kampff a *k* nos diría que es una réplica material del tipo definido por el TAD *Entero*, aunque *k* no lo sepa. El compilador oculta al programador todos los detalles de las zonas de memoria, de la representación de valores, etc. Al declarar la variable *k* y asignarle el valor 7, en términos de un TAD *Entero*, se ejecuta la operación crear (véase el ejemplo del TAD *Natural*); y al ejecutar *k= k + 1*, se realiza la operación + del TAD. Utilizar idénticos símbolos en el lenguaje matemático y en el de programación puede resultar confuso, pero es una práctica habitual. A la vista queda, por medio de este ejemplo trivial, que cualquier operación abstracta de un TAD tiene su correspondencia en operaciones del tipo durante la compilación y ejecución de programas que lo usen, y viceversa. El compilador proporciona implementaciones para las operaciones de los tipos

primitivos.

En contraposición con la relativa antigüedad de los tipos predefinidos, el concepto de **tipos de datos definidos por el usuario** o por el programador es relativamente reciente (algunos textos usan TAD para referirse a los tipos de datos definidos por el usuario; no me parece una práctica recomendable, pues un tipo definido por el usuario puede no ser una *buena implementación* de una TAD).

Un lenguaje que permita crear tipos definidos por el usuario debe proporcionar una unidad sintáctica que pueda encapsular la definición del tipo y el código para implementar las operaciones sobre los valores del tipo: ahí es donde aparece el concepto de **módulo**. Los primeros lenguajes estructurados, aparecidos en la década de los sesenta, permitían crear tipos definidos por el usuario (usando *struct* en C o *type* en Pascal, por ejemplo), pero eran malas implementaciones de tipos abstractos de datos.

Consideremos el siguiente código en C:

```
struct Complejo {
    double parte_real;
    double parte_imaginaria;
    // En C no es posible definir métodos en el interior de un struct
};

struct Complejo crear(double r, double i){
    struct Complejo c;
    c.parte_real=r;
    c.parte_imaginaria=i;
    return c;
}

struct Complejo sumar(struct Complejo com1, struct Complejo com2){
    struct Complejo suma;
    suma.parte_real = com1.parte_real + com2.parte_real;
    suma.parte_imaginaria= com1.parte_imaginaria + com2.parte_imaginaria;
    return suma;
}
```

El código define un tipo de datos definido por el usuario (*Complejo*) y unas operaciones que actúan sobre él. Ahora bien, no es una correcta implementación de un TAD *Complejo*. Los motivos saltan a la vista: se viola el principio de ocultación de la información, pues no existe encapsulación (los métodos *crear* y *sumar* se definen fuera de la estructura de datos) y no se oculta la estructura interna.

Basta con escribir dos líneas como

```
Complejo c1 = crear(3,2);
c1.parte_imaginaria= 4;
```

para darse cuenta de que la estructura de datos usada para implementar el

TAD *Complejo* resulta visible a los clientes (programas o subprogramas usuarios del nuevo tipo). Un cliente puede alterar los valores de los *Complejos* sin utilizar las funciones definidas por el programador para éstos: no hay restricciones para su manipulación externa. Cuesta mucho inventar una implementación de un TAD peor que ésta.

Una consecuencia lógica –y casi inevitable en programas de una cierta extensión– es que los programas o subprogramas clientes pueden cumplir sus cometidos accediendo directamente a las estructuras de datos de los nuevos tipos (en este caso, las variables de tipo flotante x e y).

Si algún día se decidiera cambiar la representación usada para los números complejos (por eficiencia algorítmica, por ejemplo), podrían representarse los números complejos mediante un array de dos componentes (la primera componente para la parte real; la segunda para la imaginaria) o en forma polar (r [distancia del número complejo al origen del plano complejo], θ [ángulo formado entre el punto y el eje X positivo del plano complejo]). Al cambiar la representación a una de las otras dos alternativas, todos los clientes que accedieran directamente a las variables x e y (representación cartesiana de los números complejos) dejarían de funcionar correctamente. Sería necesario modificar el código de todos ellos.

Un problema derivado de permitir la manipulación directa de las variables del tipo por parte de los clientes es la posibilidad de la destrucción de la integridad de los valores de los tipos definidos por el usuario. Hablar de pérdidas en la integridad de los datos ante administradores de bases de datos suele ponerlos nerviosos y provocarles reacciones (sudoración excesiva, autismo sensorial, hiperventilación, desaparición súbita del apetito, encogimiento del iris: las típicas reacciones de lucha o huida). Esta práctica es cruel e innecesaria, y, si bien no está penada, es recomendable evitarla por llana humanidad. La idea de integridad es la misma tanto para datos de BDs como para valores de tipos: un tipo (o un dato en una BD) no debe tomar valores que violen la definición del mismo o den lugar a situaciones ilógicas o imposibles. Veamos un sencillo ejemplo en Pascal:

```
type fecha = record
    dia : 1..31
    mes: 1..12
    anyo: 1..2003
end;
```

Es posible definir variable del nuevo tipo fecha:

```
var fecha1;
```

A pesar de que pueden usarse funciones o procedimientos para comprobar que no se obtengan valores inválidos (como 31 de febrero) al crear fechas o realizar operaciones sobre ellas, nada puede impedir que un cliente escriba directamente sentencias como éstas:

```
fecha1.dia:=31;
fecha1.mes:=2;
fecha1.anyo:=2003;
```

Por otro lado, cualquier programador, por poco avezado que esté, se da cuenta enseguida de las grandes diferencias entre los nuevos tipos *Complejo* y *Fecha* y los tipos predefinidos en C y Pascal. Tipos como *int*, *long*, *short*, etc., sí son implementaciones correctas de TADs, y existían ya antes de la aparición de lenguajes de programación capaces de permitir tipos definidos por el programador. Un programador no puede (casi nunca) acceder a la representación interna de un tipo primitivo, ni manipularla directamente. El programador no manipula directamente las cadenas de bits mediante las cuales se representan los datos a bajo nivel. Los valores de estos tipos primitivos sólo pueden modificarse mediante las operaciones definidas con ellos (+, -, etc.). Si sumamos en C dos variables *x* e *y* del tipo *int* mediante el operador +, el resultado siempre será la suma de los valores enteros almacenados en las variables, independientemente de la representación interna –y oculta– de sus valores en la máquina; la implementación del ADT *Entero* en C asegura que la suma siempre se comporta del mismo modo. Si la representación interna cambiara, los clientes no percibirían ningún cambio, pues seguirían usando *int* mediante la misma interfaz (sus operaciones: suma, producto, etc.).

Un lenguaje de programación soporta el uso de la abstracción de datos si posee construcciones sintácticas que puede usarse para construir implementaciones correctas de TADs sin excesivas complicaciones. C y Pascal son ejemplos de lenguajes que no cumplen esta condición.

Existe, en la bibliografía, bastante confusión entre los TAD y las clases. No existe en informática ningún Tribunal Supremo al cual pueda uno dirigirse para saber si una clase es o no un TAD, o para pregunta cuál es la relación entre TADs y clases; es más: la distinta terminología usada en cada lenguaje OO añadiría ruido y confusión a la respuesta. Siendo coherente con todo lo expuesto, sí pueden justificarse estas inferencias (NOTA: Utilizo ahora *tipo* como abreviatura de tipo abstracto de datos y *clase* como construcción sintáctica de muchos –pero no todos– lenguajes OO; cuando escribo que un objeto corresponde a un tipo (o que es de un tipo) me refiero a que el objeto posee la interfaz definida por el tipo abstracto de datos; un objeto corresponde a una clase cuando es instancia de ella):

- Las clases siempre implementan tipos.
- Para un analista, las clases y los TADs son una misma cosa.
- Para un programador, las clases y los TADs son cosas distintas.
- No todo TAD corresponde -o puede corresponder- a una clase. En Modula-3, por ejemplo, un TAD puede implementarse mediante la construcción *module*.
- Un mismo tipo puede ser implementado por distintas clases (cada una con distintas implementaciones de las operaciones que aquel especifica). En consecuencia, instancias de distintas clases pueden corresponder a un mismo tipo.
- Una clase puede ser simultáneamente la implementación de distintos

tipos. Un objeto puede, por tanto, corresponder a distintos tipos sin dejar de ser una instancia de una sola clase.

- Una clase define, al menos, un tipo de datos. Un tipo de datos puede no corresponder a ninguna clase (los tipos primitivos en C++ y Java, por ejemplo).

¿Suficientemente confuso? Antes de que el hipotético ingeniero de software descrito en el Apdo. 2 olvide el autocontrol Zen y recurra a licores de alta graduación y cigarrillos sin filtro, estimo conveniente dejar, como resumen, estas cuatro ideas clave:

- a) Una clase es un tipo abstracto de datos **equipado con una posible implementación**. Lo contrario no es necesariamente cierto.
- b) Una clase define implícitamente un tipo de datos. Lo contrario no es necesariamente cierto.
- c) No debe confundirse una clase con el tipo de los objetos generados por la clase (esta afirmación se completará en el Apdo. 12.1).
- d) La programación OO no consiste en la escritura de clases (en el sentido de “construcciones sintácticas de algunos lenguajes”), pues existen lenguajes OO que carecen de la construcción *class* (los lenguajes orientados a objetos, pero no a clases, sino en la implementación de verdaderos TADs, junto con algunas características adicionales.

Las clases no son más que un modo de los lenguajes de programación para implementar TADs. Como bien dice James O. Coplien en *Multi-Paradigm Design for C++* [Coplien, J., 1996]: “La esencia de la programación orientada a objetos es que la implementación de una operación de un tipo es elegida de acuerdo con la forma del objeto en tiempo de ejecución. [...] Desde una perspectiva de lenguaje de programación, la programación orientada a objetos es **programación modular** (encapsulación), más **instanciación** (la habilidad de crear múltiples instancias de cualquier módulo dado), más **herencia** y **polimorfismo**”. Amén.

La mencionada programación modular, basada en la implementación (a menudo incompleta; a veces inconsciente) de TADs, no surgió de la nada, sino que fue una evolución lógico de la programación estructurada. Ésta última, tal y como se explicó en el Apdo. 3 de este artículo (primera parte) se basa en la descomposición de los programas en procedimientos. La programación modular fue usada, sin saberlo, por muchos programadores que usaban lenguajes estructurados (como Algol, C o Pascal), pues su sustento teórico apareció más tarde que sus primeros usos en programación. En 1972, David Parnas publicó un importante artículo (*On the criteria to be used in decomposing systems into modules* [Communications of the ACM, 15:1053-1058, 1972]). En él mostró la utilidad de la descomposición de un sistema en una colección de módulos que soportaran una interfaz procedural y mantuvieran escondido su estado local:

Cada módulo [procedimiento] debería ser diseñado teniendo en cuenta la ocultación de la información, así como para esconder una decisión de

diseño del resto del programa.

Su concepto de módulo tuvo un gran impacto en el desarrollo de lenguajes como Modula-2 y condicionó buena parte de la investigación teórica en cuanto a modularidad. J. B. Morris señaló acertadamente en *Types are not Sets* ([**Proceedings ACM Symposium of Principles of Programming Languages, POPL, 1973**]) los problemas de integridad ocasionados por la falta de restricción de acceso a los valores de los tipos de datos definidos por el usuario –violación del principio de ocultación de la información–, un problema común a los primeros lenguajes estructurados (C, Pascal, etc.). El concepto formal de tipo abstracto de datos fue introducido a mediados de los años setenta por investigadores como J.A. Boguen, J.W. Thatcher, J.V. Guttag, E.G. Wagner y J.B. Wright.

Las razones prácticas para usar la modularidad las resumió muy bien Bjarne Stroustrup, creador de C++:

Cualquier parte de un programa depende de un gran número de otras partes, pero estas dependencias son tan complicadas y no intuitivas que incluso el programador que escribió una programa particular no siempre puede seguirlas. Cuando una parte se cambia, no es siempre sencillo ver todas las implicaciones del cambio en el resto del programa.

Una definición práctica –y codificable– de módulo sería que es un conjunto de uno o más estamentos contiguos en un programa, que tienen un nombre mediante el cual otras partes del sistema pueden invocarlo, de manera que cada módulo sea compilable por separado; pero esta definición resulta demasiado simplista: una buena programación modular se debería basar en la correcta implementación de tipos abstractos de datos, facilitada por las estructuras sintácticas del lenguaje empleado.

Generalmente, un TAD se implementa mediante módulos que contienen un conjunto de declaraciones de tipos y un conjunto de procedimientos/funciones encargados de manipular los valores de los tipos. Cuando se sentaron las bases teóricas de la programación modular, no existían lenguajes que implementaran de forma pura y cómoda verdaderos TADs (implementaciones conformes a los dos principios de los TADs); pero poco a poco se fueron desarrollando lenguajes que sí los implementaban.

Aun cuando pueden diseñarse “módulos” (y así lo hicieron muchos programadores antes de la programación modular) sin que el lenguaje de programación usado los permita directamente, se precisa, por parte de los programadores, un alto grado de disciplina para conseguirlo. Un chiste muy popular entre los programadores de la década de los setenta decía: “Los buenos programadores, antes de la programación modular, construían buenos programas modulares; los malos, programas monolíticos. Con la programación modular, los buenos programadores construían buenos programas modulares; los malos, malos programas modulares”. Bueno, así progresa el mundo...

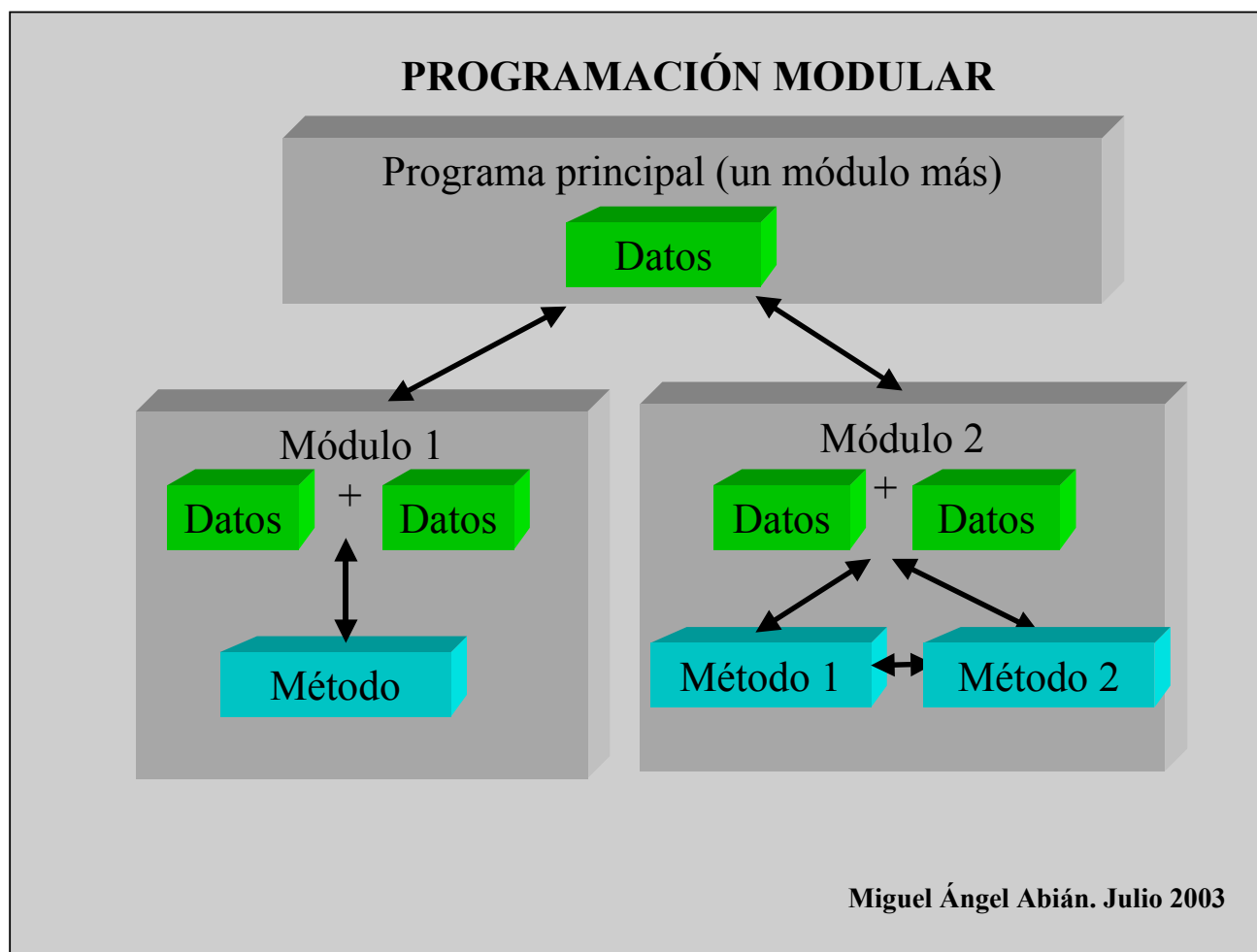


Figura 12. Esquema de la programación modular

Como el pasado siempre se entiende mejor desde el futuro, ojeemos la todavía no escrita *Historia crítica de la programación (Tomo 1: 1900-2050)*:

Con el advenimiento de la programación modular, la infancia de la programación empezó a poblarse de metodologías y metodólogos. Los últimos años de la década de lo sesenta se llenaron de métodos en los cuales aparecía la palabra “modular”, aunque su significación solía ser arbitraria, dependiendo de la creatividad, originalidad y espíritu de contradicción del metodólogo.

Consciente o inconscientemente –en algunos casos, incluso contra su voluntad–, todas las metodologías modulares se basaban en la implementación de tipos abstractos de datos (incluso antes de que se les designara con esas palabras –hacia 1974, aproximadamente–). La programación modular, con ciertos añadidos, se convirtió en la base que condujo a la programación orientada a objetos. Programando mediante módulos era más difícil (pero no imposible, como demuestra la bibliografía) crear programas ilegibles y demasiado dependientes de implementaciones concretas. [...] En resumen, la situación podría esquematizarse así:

***Orientación a objetos =
Objetos (Abstracción) + Clases (TADs + Clasificación) + Herencia
(Reutilización) + Polimorfismo (Flexibilidad)***

La proliferación de metodologías en programación se volvió una contagiosa y virulenta epidemia a finales del siglo XX y principios del siglo XXI. Se desconocen los motivos exactos del crecimiento exponencial de las metodologías en esos períodos, pero se barajan motivos como la contaminación del medio ambiente, la radioactividad artificial, el elevado número de canales de televisión y el manido *fin de siècle*. Existían importantes diferencias entre los metodólogos de la programación de los años 60 y los que les sucedieron. Entre 1960 y 1970, un metodólogo solía ser un programador o un académico, frecuentemente de mediana o avanzada edad, que inventaba o formulaba una metodología, basada en su larga experiencia y sus errores, para contar cómo hubiera diseñado y construido programas si hubiese sabido desde el principio todo lo que sabía cuando formuló la metodología (quizás motivado por la lógica idea de que su trabajo hubiera resultado mucho más sencillo si siempre hubiera programado o teorizado del mismo modo; así, en caso de error, al menos siempre hubiera errado igual).

En las décadas de 1980 y 2020, sin embargo, las metodologías de la programación se convirtieron en una enfermedad de rápida propagación. Cualquiera podía formular su propia metodología —a menudo inconsistente con el resto o directamente enfrentada con ellas—, y para ser considerado como metodólogo bastaba con tener algún adepto fuera de la oficina o del despacho universitario. La experiencia y la edad ya no eran una ventaja, más bien constituían un obstáculo, pues hacían más difícil encajar con las nuevas generaciones. Incluso se produjo un fenómeno nunca visto antes: muchos teóricos de la programación eran más (re)conocidos por sus trabajos inéditos, sobre los que ellos mismos contaban maravillas, que por sus trabajos publicados. Convulso siglo el XX, sin duda.

Las metodologías de programación aparecidas entre 1980 y 2020 eran tan variadas como sus autores; algunas parecían intentos desesperados y maniáticos por llevar el constructivismo francés a la programación (al grito de “En la programación todo son estructuras”; los escépticos solían replicar: “Las estructuras no programan”); otros optaban por el lema “el programa es el diseño”, el cual no contaba con mucho respaldo teórico; algunos programadores optaron por adoptar una metodología distinta para cada día laborable de la semana (los sábados elaboraban la suya propia y los domingos pensaban en maneras de promocionarla).

En 1995, unas 80 metodologías se denominaban a sí mismas “orientadas a objetos” (otras 35 se denominaban “híbridas”), y su número crecía tan rápidamente que parecía que iba a superar, en pocos años, al número de átomos de hidrógeno en el universo conocido.

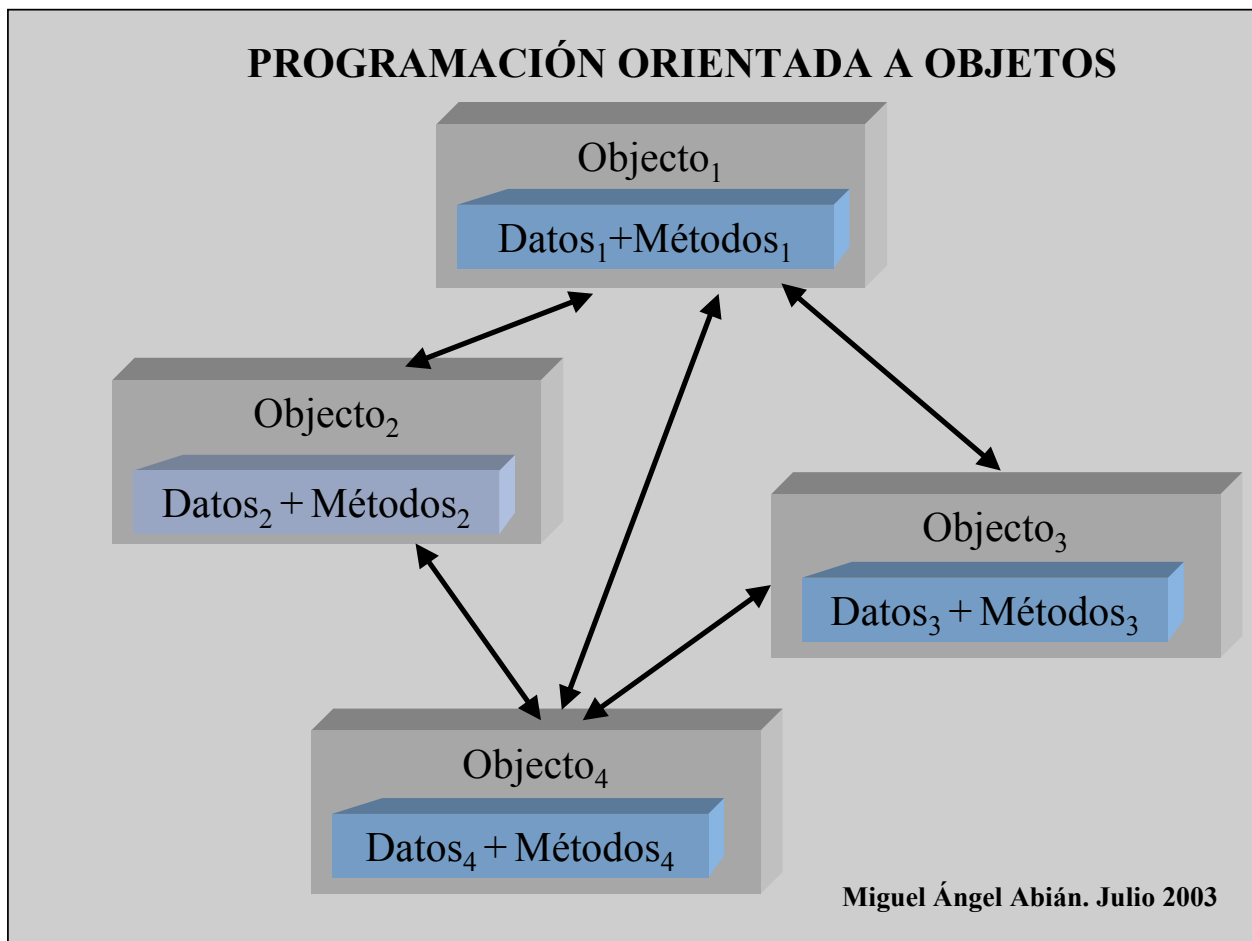


Figura 13. Esquema de la programación OO

Las ventajas de la programación modular son muchas:

1. La división en módulos separa físicamente los componentes software de un programa, permitiendo así desarrollar arquitecturas coherentes de software (con el código correctamente organizado y estructurado).
2. Se permite la encapsulación y la ocultación de datos y código.
3. Permite establecer la visibilidad o no visibilidad de datos y código.
4. Los módulos permiten separar claramente la interfaz de la implementación.
5. Si se cambia la implementación de un módulo, los clientes – siempre que no se cambie la interfaz– no notarán ninguna diferencia.
6. El creador de módulos puede centrarse en la obtención de módulos que cumplan las especificaciones.
7. El programador puede ensamblar módulos dentro de sus aplicaciones (como si fueran componentes electrónicos, véase el Apdo. 4.3 de la primera parte), sin necesidad de conocer su interior. Es un primer paso hacia la reutilización.
8. Se facilita la prueba y depuración del software.
9. Se facilita el trabajo en grupo de los desarrolladores.

10. Cuando el lenguaje usado permite la utilización de verdaderas implementaciones de TADs, los programas pueden entenderse en términos de cómo se usan los tipos, sin considerar los detalles de manipulación de la estructura de datos subyacente ni su implementación.

9.4. Clases abstractas.

La noción intuitiva de clase abstracta resulta muy fácil de comprender. Supongamos que vamos a un concesionario de vehículos y que decimos a un vendedor: “Deseo un vehículo”. Lo primero que nos diría es: “¿Qué tipo de vehículo desea?”. Todo el mundo sabe lo que es un vehículo, pero “vehículo” no es una descripción completa –ni útil– de ningún medio de locomoción. “Vehículo” viene a ser el conjunto de características y propiedades que hacen de un objeto físico un vehículo. Forzando un poco el lenguaje, *vehículo* sería la esencia de la *vehiculidad*. Vehículo sería una clase abstracta.

Según [Booch, G., 1994], una clase abstracta es

[...] una clase que no tiene instancias. Una clase abstracta se escribe con la intención de que sus subclases concretas añaden elementos nuevos a su estructura y comportamiento, normalmente implementando sus propiedades abstractas.

Las clases abstractas agrupan conceptos coherentes y cohesivos, pero incompletos, de forma que no pueden representar objetos. Habitualmente, las características establecidas en las clases abstractas se especializan en las clases concretas derivadas de aquéllas por herencia. Cada lenguaje OO utiliza distintas construcciones (*abstract classes*, métodos y variables virtuales, etc.) para las clases abstractas.

10. Mensajes.

Consideremos, antes de entrar en materia, un brazo mecánico neumático, con electroválvulas, conectado a un autómatas.



Figura 14. Brazo neumático conectado a un autómatas

Desde el punto de vista del análisis estructurado, cuando programamos 10 ciclos en el autómatas sucede lo siguiente:

- a) El autómatas envía una señal eléctrica al brazo mecánico.
- b) El brazo mecánico comprueba que la electroválvula de comienzo de carrera está activada. Si no es así, envía una señal eléctrica de error al autómatas.
- c) El brazo mecánico abre el orificio de entrada del aire a presión.
- d) El brazo sale (se extiende) completamente
- e) La electroválvula de final de carrera capta que el brazo se ha extendido de forma completa y cierra el orificio de entrada del aire comprimido.
- f) El brazo retrocede a su posición inicial. Al alcanzarla, activa la

- electroválvula de comienzo de carrera.
- g) La electroválvula envía una señal eléctrica al autómata.
 - h) El autómata convierte la señal eléctrica recibida en una electrónica, e incrementa el número de ciclos realizados en una unidad.
 - i) Mientras el número de ciclos realizados sea inferior a 10, se repiten los pasos a) – h).

En un análisis orientado a objetos sucede lo siguiente:

- a) Un objeto *Autómata* envía un mensaje *hacerCiclos(10)* a un objeto *BrazoMecánico*.

El ejemplo anterior se encuentra deliberadamente fuera del ámbito de la programación, para que no se vincule sólo a ésta el envío de mensajes. Pero podemos considerar otro ejemplo, típico en programación: supongamos que vamos a dibujar en el monitor de un ordenador diversos objetos (una manzana, un coche, etc.). En una aproximación estructurada, vendríamos a tener algo así, en pseudocódigo:

```
para cada x de la lista de cosas dibujar(x);
dibujar(x):
    según tipo de x:
        si tipo A → dibujar_A(x)
        si tipo B → dibujar_B(x)
        ...
```

En una aproximación OO, a cada objeto x de la lista se le enviaría el mensaje *dibujar(x)*.

En el paradigma de la orientación a objetos, éstos se comunican mediante el intercambio de mensajes. Según [Rumbaugh, J. et al, 1999], un mensaje es “una llamada a una operación o a un objeto, en la cual se incluye el nombre de la operación y una lista de valores de argumentos”. De acuerdo con [Booch, G., 1994], es “una operación que un objeto realiza sobre otro”. Booch clasifica las operaciones en cinco grupos básicos:

- **Modificador:** Operación que cambia el estado de un objeto.
- **Constructor:** Operación que crea un objeto o inicializa su estado.
- **Destructor:** Operación que destruye un objeto o su estado.
- **Selector:** Operación que accede al estado de un objeto sin modificarlo.
- **Iterador:** Operación que permite acceder a los atributos de un objeto en un orden bien establecido.

Algunos textos definen los métodos como implementaciones de operaciones (o sea, como algoritmos codificados); pero la mayoría suele usar de forma intercambiable los términos *mensaje*, *método* y *operación* (a veces

también se usa *función*). Esta equivalencia me parece errada: las operaciones son los elementos definidos del comportamiento de una clase; los métodos, sus implementaciones.

Una operación –estimo– yace en terrenos más conceptuales, pues se centra en la especificación de los requisitos (o del contrato) que debe satisfacer el mensaje para ambas partes: emisor y receptor. Por ejemplo, *public String toString()* es –siendo coherente con lo expuesto– una operación definida en una clase X. El método (o implementación) asociado ni siquiera tiene por qué definirse en X: puede definirse en una subclase de X. Basta con considerar que X tenga varias subclases para darnos cuenta de que a **una sola operación** le pueden corresponder **muchos métodos**. Incluso dentro de una misma clase, una operación puede implementarse de distintas maneras (polimorfismo).

La importancia de usar una terminología precisa no se discute en muchos campos; nadie pide en una ferretería “unos tornillos”, sino “unos tornillos Allen de métrica 4”, o algo similar, más exacto. La diferencia entre usar **operación** y **método** no es –a mi modo de ver– un asunto baladí, sólo interesante para académicos, ni enlaza con preguntas tan inútiles como preguntarse cuántos ángeles caben en un alfiler o si se podrá fumar después de morir.

Los programadores neófitos tienden a considerar, cuando hay métodos – en distintas clases– y con distinto código, que corresponden a distintas operaciones; lo cual es un error a la hora de analizar y planificar código. Clases distintas pueden tener métodos distintos (es decir, con código distinto) y corresponder, en realidad, a la misma operación. Por ejemplo, distintas clases puedan usar diferentes métodos para escribir en un archivo local, en uno remoto o en un *socket*; sin embargo, todas implementan una misma operación *escribir()*. Del mismo modo, una misma clase puede ofrecer distintos métodos a distintos clientes, aun cuando todos los clientes soliciten la misma operación. Por otro lado, me parece asimismo más natural considerar que los objetos invocan métodos como reacción a estímulos (recepciones de mensajes) que identificar mensaje y método.

En todo caso, el lector puede optar por una visión bucólica de la programación OO; todo programa OO en ejecución es una ruidosa colmena donde las abejas son los mensajes, que zumban de celdilla en celdilla (equivalentes a los objetos). La miel suele quedársela el ingeniero de software o el programador; su jefe es quien cata de la jalea real.

INTERCAMBIO DE MENSAJES

El intercambio de mensajes como única vía de comunicación entre objetos es una de las características de la OO



Miguel Ángel Abián. Julio 2003

Figura 15. Intercambio de mensajes entre dos objetos

11. Herencia. Subclases y subtipos: cuando un círculo no es una elipse. Algunos consejos prácticos.

11.1. Herencia.

Un artículo sobre la OO que no mencionara la herencia sería como resumir *Romeo y Julieta* sin mencionar a los dos enamorados. La significación de la palabra “herencia” en programación no difiere de la empleada en el lenguaje ordinario (*Los hijos heredan rasgos de los padres*).

En la POO, la herencia permite que unos objetos puedan basarse en otros objetos ya existentes. En términos de clases, la herencia es el mecanismo por el cual una clase *X* puede heredar propiedades de una clase *Y* (*X* hereda de *Y*), de modo que los objetos de la clase *X* tengan acceso a los atributos y operaciones de la clase *Y*, sin necesidad de redefinirlos. Sin embargo, las propiedades de una clase no son necesariamente la suma de las propiedades de todas sus superclases.

La herencia crea automáticamente una jerarquía de especialización-generalización (las jerarquías se vieron en el Apdo. 4.4 de la primera parte. No obstante, **la herencia no es generalización**. La primera es un mecanismo de los lenguajes OO mediante el cual puede implementarse o materializarse la generalización. Algunos lenguajes OO (como Self) no tienen herencia; lo cual no es obstáculo para que sí tengan jerarquías de generalización-especialización.

Herencia. Ejemplo de jerarquía de clases

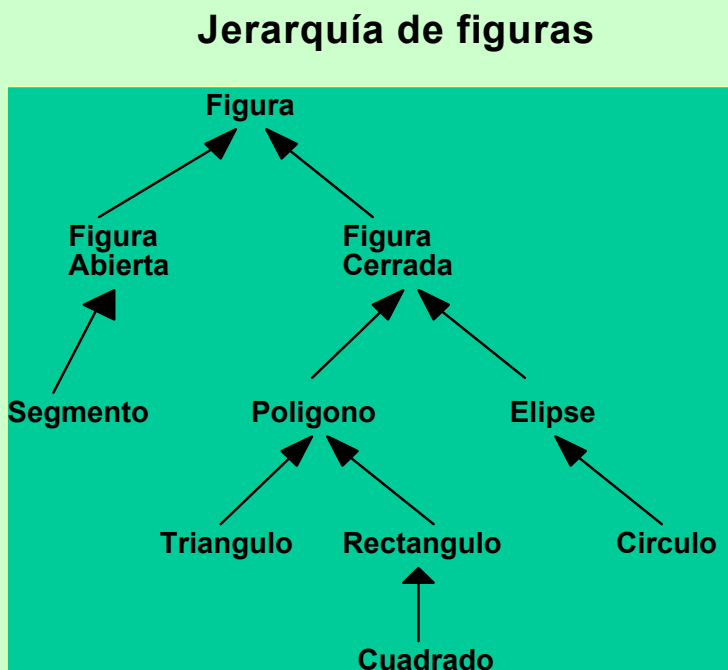


Figura 16. Ejemplo de jerarquía de clases ocasionada por la herencia

Suelen usarse los pares de términos **hija/padre** y **subclase/superclase** para designar al par *clase que hereda de otra/clase de la cual hereda*. A veces se dice que las subclases son especializaciones de la superclase; y que la superclase es la generalización de las subclases.

La herencia presenta dos cualidades contradictorias entre sí. A saber: una clase hija extiende o amplía el comportamiento de la clase padre, pero también restringe o limita a la clase padre (una subclase se encuentra más especializada que su clase padre). Existe cierta tirantez esencial, constitutiva, entre los dos conceptos (**herencia como extensión** y **herencia como especialización**); a veces se olvida esta tensión, pero siempre sigue ahí.

Suele identificarse la herencia mediante la regla “es un” o “es un tipo de” (se volverá sobre esta norma en el subapartado 11.2). Por ejemplo, toda *Motocicleta* es un *Vehículo*, luego la clase *Motocicleta* es una subclase de la clase *Vehículo*. En general, no resulta recomendable emplear la herencia cuando no funcione la regla “X es un Y” o, más precisamente, cuando no pueda justificarse que todo instancia de la clase X es también una instancia de Y. Veamos algunos ejemplos: una clase *ConductorMotocicleta* que heredase de dos superclases *Persona* y *Motocicleta* constituiría un mal uso de la herencia: no toda instancia de *ConductorMotocicleta* es una instancia de *Motocicleta*. Del mismo modo, no conviene establecer una relación de herencia entre una clase *Motor* y una clase *Coche*, pues un coche no es un género de motores. Tampoco sería correcto vincular mediante herencia un *array* de objetos *Coche* y la clase *Coche*, pues una colección o conjunto de coches no es un tipo de coche.

El uso de la herencia para reutilizar código entre clases que incumplen la regla “es un” suele considerarse incorrecto, aunque a veces se permite (**herencia de implementación o de funcionalidad**). John K. Ousterhout critica con agudeza la herencia de implementación en *Scripting: Higher-Level Programming for the 21st Century* ([IEEE Computer, Marzo 1998]):

Otro problema con los lenguajes OO es su hincapié en la herencia. La herencia de implementación, en la cual una clase aprovecha código que fue escrito para otra clase, es una mala idea que hace el software más difícil de manejar y reutilizar. Liga la implementación de las clases juntas de modo que ninguna clase puede comprenderse sin la otra. Una subclase no puede ser comprendida sin conocer cómo los métodos heredados han sido implementados en la superclase, y una superclase no puede ser comprendida sin saber cómo se heredan sus métodos en las subclases. En una jerarquía compleja de clases, ninguna clase individual puede comprenderse sin comprender todas las otras clases en la jerarquía. Y lo que es peor, una clase no puede separarse de su jerarquía para reutilizarse. La herencia múltiple hace estos problemas incluso peores. La herencia de implementación causa el mismo entrelazamiento y fragilidad que se ha observado cuando se abusa de las sentencias *goto*.

La relación “es un” ilustra una característica crucial de la herencia: **un objeto de una subclase puede usarse en cualquier lugar donde se admita un objeto de la superclase**. Lo contrario no es cierto (esto también ocurre en

el mundo real: los padres, por ejemplo, no heredan los rasgos de los hijos ni pueden intercambiarse por ellos).

La noción de que un objeto de una clase hija puede sustituirse por un objeto de la clase padre conduce inexorablemente a que se incorporen, cuando hablemos de los objetos pertenecientes a una clase, los objetos pertenecientes a todas las subclases. Por claridad, se usa “la clase” de un objeto para referirse a la clase más especializada de la cual es instancia el objeto. Lo dicho en el párrafo inmediatamente superior puede escribirse un poco más precisamente: ***un objeto de una clase especializada puede substituirse por un objeto de una clase más general en cualquier situación donde se espere un miembro de la clase general, pero no al revés.***

Veámoslo con un ejemplo, escrito en Java:

```
public class Persona {
    // Código en Java

    private String nombre;
    private String apellido;

    public Persona (String nombre, String apellido) {
        this.nombre=nombre;
        this.apellido=apellido;
    }

    public void obtenerNombre() {
        System.out.println(nombre);
    }

    public void obtenerInfo() {
        System.out.println("Nombre: " + nombre);
        System.out.println("Apellido: " + apellido);
    }
}

public class Estudiante extends Persona{
    // Código en Java

    private String codEstudiante;

    public Estudiante(String nombre, String apellido, String codEstudiante) {
        super(nombre, apellido);
        this.codEstudiante=codEstudiante;
    }

    public void obtenerInfo() {
        super.obtenerInfo();
        System.out.println("Codigo de estudiante: " + codEstudiante);
    }
}
```

```
        public void obtenerCodEstudiante() {  
            System.out.println(codEstudiante);  
        }  
    }
```

Las siguientes líneas se compilarán y ejecutarán correctamente:

```
public class Test {  
    public static void main(String args[]) {  
        Persona p1= new Persona("Luis","Garcia");  
        p1.obtenerInfo();  
        Persona p2= new Estudiante("Javier","Perez", "AC001");  
        p2.obtenerInfo();  
    }  
}
```

pues todo estudiante es una persona (recordemos: *un objeto de una subclase puede usarse en cualquier lugar donde se admita un objeto de la superclase*).

Sin embargo, la siguiente línea dará error:

```
p1.obtenerCodEstudiante(); //dará error
```

pues este método no existe en Persona (no toda persona es un estudiante: *un objeto de la superclase **no** puede usarse en cualquier lugar donde se admita un objeto de la subclase*).

La herencia suele clasificarse así:

- **Herencia de especialización:** Una clase hija es un caso particular de la clase padre. Las subclases, cuando redefinen los métodos heredados de la superclase, especializan o concretan la clase padre. Esta forma de herencia es, con diferencia, la más usada.
- **Herencia de especificación:** Una clase padre define el comportamiento de sus subclases, pero no proporciona ninguna implementación por defecto. Por consiguiente, la superclase –una clase abstracta o un *interface*– se encuentra incompleta. Esta forma de herencia es la segunda más común. Resulta muy útil para definir una interfaz común para clases relacionadas.
- **Herencia de implementación o construcción:** Se usa la clase padre sólo por su comportamiento, a pesar de que no exista relación entre la clase padre y las subclases. En este caso, la herencia se usa para aprovechar código ya escrito, pero no implementa una generalización (“es un” o “es una clase de un”). Su uso suele considerarse incorrecto y desaconsejable.

- **Herencia de generalización:** Las clases hijas modifican o redefinen algunos métodos de su superclase, de modo que extienden el comportamiento de aquélla. Comportamiento, por tanto, exactamente opuesto al de la herencia de especialización. Se recomienda evitarla; pero a veces se hace imprescindible. Cuando se tiene un conjunto de clases de modificables y se desea conseguir clases más generales, resulta casi inevitable no usarla. Consideremos una biblioteca software con una clase *MonitorRGB*; si se quiere crear una clase *Monitor* genérica, que use otras coordenadas cromáticas, habrá que usar la herencia de generalización.
- **Herencia de extensión:** Las subclases añaden nuevas funciones con respecto a su superclase, pero no se modifica ni redefine ningún método heredado. Esta forma de herencia permite a los programadores desarrollar nuevas bibliotecas a partir de las ya existentes.
- **Herencia de limitación o restricción:** Las clases hijas limitan el comportamiento de su superclase. Eiffel permite directamente la herencia de limitación; pero puede simularse indirectamente en Java y C++. Está desaconsejada, pues las clases hijas ya no pueden usarse en lugar de la clase padre: un objeto de la superclase puede ser substituido por un objeto superclase y encontrarse con que algunos métodos se han suprimido.
- **Herencia de variación:** Las clases hijas y su superclase comparten código. A diferencia de la herencia de implementación, aquí sí existe una relación entre las clases, pero no es del tipo “es un”. Por ejemplo, una clase *Video* puede usar métodos de una clase *Audio*. En general, no se recomienda el uso de esta técnica, y casi siempre puede evitarse. Continuando con el ejemplo, podría definirse una clase padre más general (*MedioAudioVisual*).
- **Herencia de combinación:** También conocida como herencia múltiple. Se explica un poco más adelante.

La herencia se clasifica asimismo en herencia simple y herencia múltiple. En la herencia simple, una clase **sólo** hereda (es subclase) de una superclase. En la herencia múltiple, una subclase admite **más de una superclase**. Entendamos que heredar de más de una superclase no quiere decir que la herencia múltiple consista en que una subclase pueda heredar de una clase que sea, a su vez, subclase de otra clase.

Una subclase que herede, mediante herencia múltiple, de dos o más superclases puede mezclar las propiedades de las superclases, en ocasiones de forma ambigua o poco recomendable.

Dos problemas fundamentales se presentan con la herencia múltiple:

- **Herencia repetida.** (Figura 19). La clase *A* hereda de *B* y *C*, que a su vez derivan de *D*. Consecuencia: la clase *A* hereda dos veces de *D*.
- **Conflictos de nombres.** Si una clase *A* hereda simultáneamente de dos superclases *B* y *C*, aparecerá un conflicto de nombres si usan el mismo nombre para algún atributo o método. ¿Qué definición usará *A* del atributo o del método con el mismo nombre? ¿La de la superclase *B* o la de *C*? ¿O ninguna de ellas? Generalmente este problema se soluciona redefiniendo en la subclase la propiedad o método con el mismo nombre en las superclases.

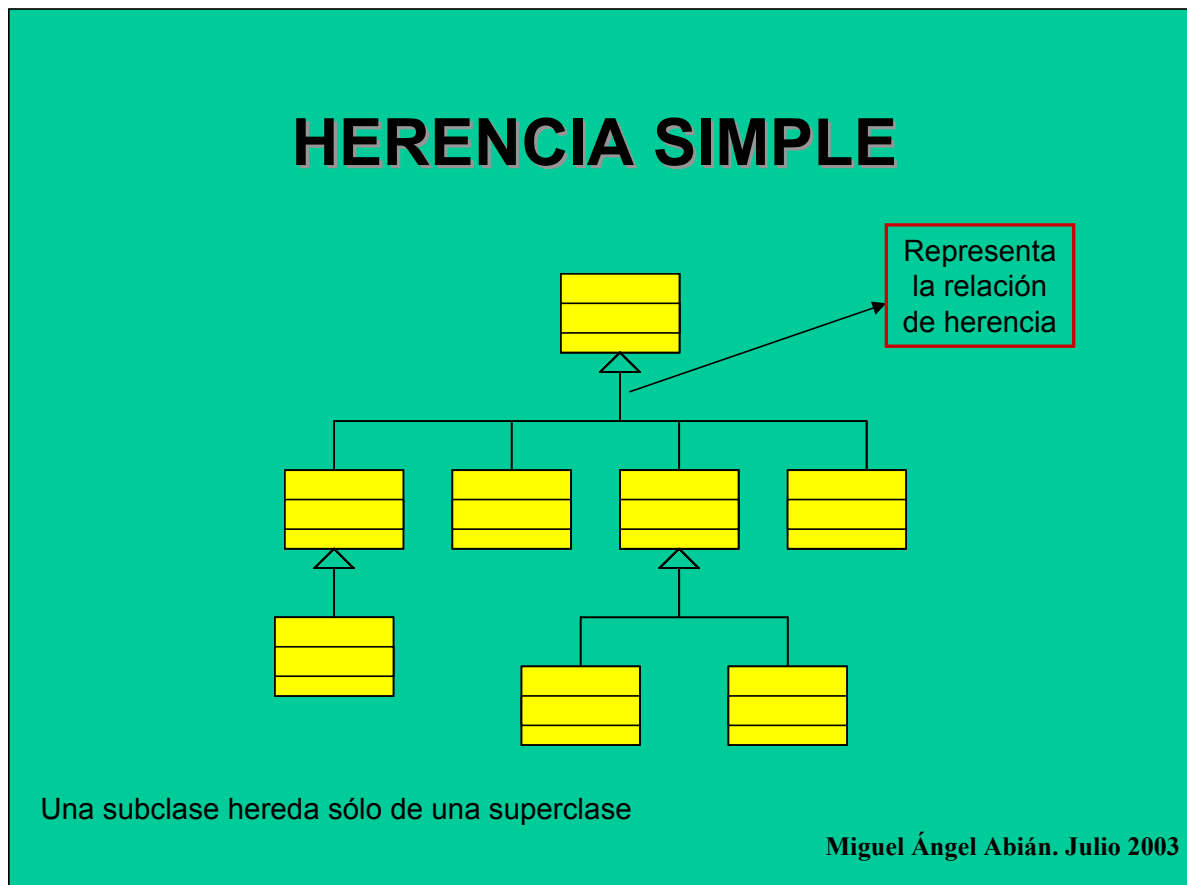
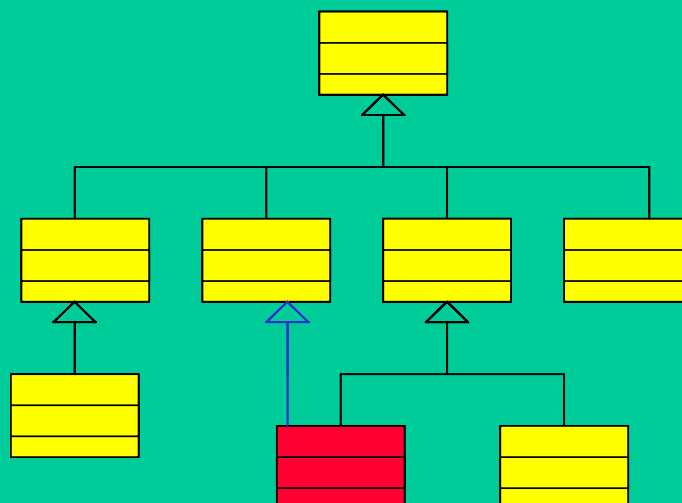


Figura 17. Esquema de la herencia simple

HERENCIA MÚLTIPLE

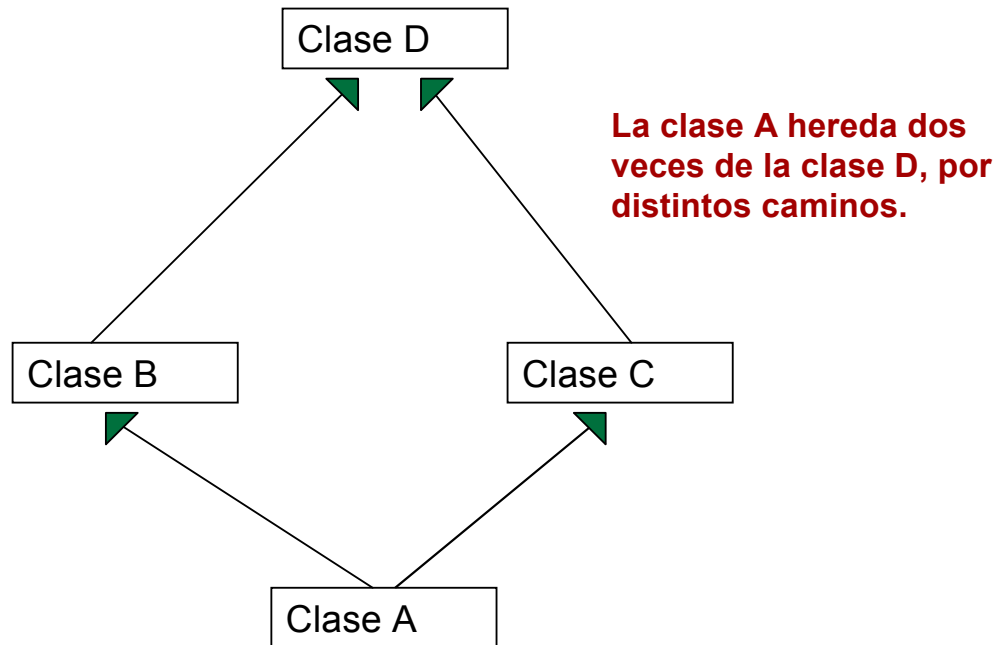


Una subclase hereda de dos o más superclases

Miguel Ángel Abián. Julio 2003

Figura 18. Esquema de la herencia múltiple

Un ejemplo del problema de la herencia repetida



Miguel Ángel Abián. Julio 2003

Figura 19. Ejemplo del problema de la herencia repetida

Cuando se consideran lenguajes OO, la herencia simple no plantea problemas conceptuales o de compilación (aunque sí prácticos: las clases hijas pueden redefinir los métodos del padre, de manera que no se obtengan los resultados esperados); las dificultades surgen de la herencia múltiple, que cada lenguaje soluciona de distintas maneras.

Eiffel, por ejemplo, no compila ninguna subclase en la que aparezca el problema del conflicto de nombres o el de herencia repetida. C++, sin embargo, sí permite compilar –bajo ciertas condiciones– con estos problemas. Así, una subclase en C++ puede plantear el problema del conflicto de nombres, pero el compilador exige que cualquier referencia al nombre repetido (ya corresponda a un miembro de datos o a una función miembro) especifique la procedencia de éste. Una subclase que herede más de una vez de un superclase se entiende que hereda, en C++, de la misma superclase; y, en consecuencia, esta situación no genera errores de compilación.

Cada lenguaje incorpora sus propias reglas para el control de la herencia: en algunos son las superclases las que deciden qué métodos o atributos podrán ser redefinidos en las subclases, en otros la decisión recae en las subclases; algunos otros permiten que el control lo ejerzan tanto unas como otras. Por ejemplo, C++ permite que la visibilidad de lo heredado (métodos y

atributos) sea determinada bien por la superclase, bien por la subclase. En Smalltalk sólo las subclases pueden ocultar métodos o atributos heredados. Java hace recaer el control de la herencia en las superclases, que deciden qué características serán inaccesibles para sus subclases, las cuales no pueden ocultar nada de lo heredado.

Java no permite la herencia múltiple, en un intento de evitar los problemas intrínsecos a ella; pero la simula mediante la construcción sintáctica *interface* y la herencia simple. La construcción *interface* de Java está muy próxima a lo que es un TAD, pues no existe implementación de las operaciones y constituye una expresión de diseño puro. Sí está permitida la herencia múltiple de *interfaces* (un *interface* puede extender, por medio de la palabra reservada *extends*, a más de un *interface*). El conflicto de nombres entre *interfaces* sólo se producirá cuando haya métodos con iguales argumentos, pero con distintos tipos de retorno (el compilador no lo permitirá).

La originalidad de Java en cuanto a la herencia resulta grande y práctica, pues se separa de forma explícita la interfaz y la implementación. Un programador que se restrinja a usar herencia simple no necesita los *interfaces* de Java, pues cada clase ya tiene implícita una interfaz (conjunto de métodos visibles desde el exterior de la clase), sin que sea necesario utilizar la construcción *interface*. En términos de TADs, un *interface* Java es una forma de declarar un tipo compuesto sólo por métodos abstractos. Muchas clases distintas pueden implementar un mismo interfaz Java. Las bibliotecas de Java hacen un uso intensivo –y mayoritariamente afortunado– de los *interfaces* para construir clases con idéntica interfaz, pero bajo las cuales se ocultan implementaciones absolutamente distintas. En C++ pueden simularse los *interfaces* de Java mediante el uso de clases abstractas puras.

Resulta importante estudiar la manera como un lenguaje implementa la herencia, sobre todo la herencia múltiple, y no dar nada por sentado, pues pueden obtenerse resultados inesperados.

La herencia constituye una característica definitoria de la programación orientada a objetos y ha sido, por su alto apoyo a la reutilización del código, un factor primordial en su éxito.

Gracias a ella, se facilita el cumplimiento del principio Abierto/Cerrado (***Una entidad de software –clase, módulo, etcétera– debe estar abierta para su extensión, pero cerrada para su modificación***). Las nuevas características, si se sigue este principio, se incluyen añadiendo código nuevo, en lugar de modificando el código ya existente.

El mal uso de la herencia puede desembocar en programas caóticos, de difícil lectura, y que exhiban comportamientos inesperados. Un programa (o subprograma) que funcione siempre mal resulta enseguida identificable, pero un programa que a veces se comporte bien y a veces mal puede pasar baterías de pruebas sin revelar sus problemas.

El lector habrá notado que casi siempre empleo expresiones como “la clase del objeto”, y que rara vez escribo “el tipo del objeto”. Con ello, intento distinguir entre *tipo* y *clase*, por motivos que se aclararán más adelante.

11.2. Subclases y subtipos: cuando un círculo no es una elipse

La herencia de clases, que es la vista hasta ahora, no implica siempre herencia de tipos. Es más, la herencia, o mejor, el mecanismo de la herencia, puede ser usada de manera que impida la herencia de tipos. Aceptaremos provisionalmente que un tipo de datos es **subtipo** de otro si cualquier método que pueda ser aplicado a los elementos o valores del supertipo puede aplicarse también a los valores del subtipo (más adelante se definirá subtipo formalmente). Un ejemplo muy sencillo nos permitirá apreciar que la herencia de clases no siempre implica herencia de tipos:

El conjunto de los números reales es una subclase de los números imaginarios. Todo número real es un número imaginario (con la parte compleja nula). Sin embargo, los números reales no son un subtipo de los complejos, pues la raíz cuadrada de un número imaginario es una operación bien definida; pero la raíz cuadrada de un número real no está bien definida para todos los reales. No existe, en consecuencia, herencia de tipos para este caso.

Vemos, por consiguiente, que la herencia de clases difiere de la de tipos. Los mecanismos implicados en cada caso son sutilmente distintos. La herencia de tipos restringe el conjunto de valores aceptables para el subtipo. Dicho más precisamente: el conjunto de valores que satisface un subtipo coincide con el subconjunto de los valores que satisfacen el tipo del cual procede. La herencia de subtipos actúa pues como un mecanismo limitante.

La herencia de clases sigue otro cauce. En principio, todo número real es imaginario, luego se le puede aplicar las operaciones de los números imaginarios. La herencia de clases no limita **por sí** el comportamiento de las subclases.

En algunos lenguajes, como Java o C#, se asume automáticamente que las subclases definen subtipos del tipo de la superclase. Para que las subclases definan subtipos no basta con usar los mecanismos de los lenguajes OO. El TAD *Pila* nos proporciona un buen ejemplo:

```
interface Pila {
    // Código en Java
    public void apilar(Object valor);
    public Object cima();
    public boolean estaVacio();
}

class NoPila implements Pila {
    // Código en Java
    private Object v = null;

    public void apilar(Object valor) {
        v = valor;
    }
}
```

```
        public Object cima() {
            return v;
        }
        public boolean estaVacio() {
            if (v == null)
                return true;
            else
                return false;
        }
    }
```

La clase *NoPila* **no es** una representación software de una pila, aunque herede del *interface* Pila, pues no se comporta como una pila. El principio de “último en entrar-primero en salir” no se cumple en esta implementación del TAD *Pila*.

En algunos lenguajes (verbigracia, C++) sólo puede definirse subtipos a partir de subclases. En otros (Smalltalk, Objective-C, p.ej.), las subclases y los subtipos van por caminos independientes: se puede tener subtipos sin subclases, y viceversa. En Java y C#, los subtipos se derivan de subclases o *interfaces* (se detallarán en el Apdo. 12.2). En estos dos lenguajes, las jerarquías de tipos y clases pueden construirse independientemente.

Parece lógico que un subtipo deba conservar el significado del tipo original y comportarse de una forma compatible o coherente con el tipo del cual es subtipo. En otras palabras, deben cumplirse algunas condiciones para que una subclase defina un subtipo.

Barbara Liskov describió esas condiciones en *Data Abstraction and Hierarchy* ([SIGPLAN Notices 23(5), Mayo de 1988]):

[...] Si para un objeto O_1 de tipo S hay un objeto O_2 de tipo T tal que, para todos los programas P definidos en términos de T , el comportamiento de P no cambia cuando O_1 es substituido por O_2 , entonces S es un subtipo de T .

Esta frase se conoce ahora como el enunciado del principio de substitución de Liskov. Pese a su elegante y formal título, no afirma nada extraordinario: cualquier función o método que use referencias o punteros a un objeto de una superclase debe poder usar objetos de sus subclases sin necesidad de saberlo. En resumidas cuentas: no puede haber privilegios para la superclase o las subclases. Sus consecuencias, sin embargo, no resultan triviales, como ahora estudiaremos.

Veamos con un ejemplo qué puede ocurrir cuando se incumple el principio de substitución de Liskov:

```
public class Elipse {

    // Código en Java
    // Ejemplo donde se emplea la herencia
    // para violar el principio de substitución

    private double semiejeA;
    private double semiejeB;
```

```
public Elipse (double a, double b) {
    semiejeA = a;
    semiejeB = b;
}

public void setSemiejeA(double a) {
    semiejeA = a;
}

public void setSemiejeB(double b) {
    semiejeB = b;
}

... // Métodos de acceso, etc.
}

public class Circulo extends Elipse {

    // Código en Java

    public Circulo(double r) {
        semiejeA = r;
        semiejeB = r;
    }

    public void setSemiejeA(double r) {
        semiejeA = r;
        semiejeB = r;
    }

    public void setSemiejeB(double r) {
        semiejeA = r;
        semiejeB = r;
    }

    ... // Métodos de acceso, etc.
}
```

Nota: siempre que me refiera a clases, objetos o variables procedentes de código omito las tildes y uso cursiva.

Aunque *Circulo* parece una clase normal y corriente, no se comporta de la manera, en comparación con *Elipse*, como uno podría pensar:

```
Elipse e= new Circulo (0.0);
e.setSemiejeA(2);
e.setSemiejeB(3);
```

El área de e ($9 * \pi$ unidades arbitrarias de superficie) no es la que uno esperaría de una elipse ($6 * \pi$ u.a.s). Un programa cliente que usara la interfaz de *Eclipse* obtendría resultados inesperados si se substituyera un objeto *Eclipse* por un *Circulo*. Por supuesto, si cambiamos *new Circulo (0.0)* por *new Eclipse (0.0, 0.0)* todo funcionará como se espera.

Pese a que un objeto *Circulo* (tal y como se define aquí) pertenece a una subclase de *Eclipse*, un objeto *Circulo* no es un objeto *Eclipse*, pues su comportamiento no es el lógico de una elipse. **La subclase *Circulo* no es un subtipo de *Eclipse*, aun cuando sí es una subclase.** Un círculo no extiende el concepto de elipse; en realidad, lo restringe: en un círculo ambos semiejes son idénticos. Nos hallamos ante un caso de herencia de limitación.

Desde luego, existe cierta similitud entre círculos y elipses. Matemáticamente, un círculo es un caso especial (o degeneración) de una elipse. ¿Cómo incorporar esta semejanza al diseño software sin violar el principio de Liskov? Una posible solución consistiría en diseñar *Circulo* y *Eclipse* como subclases de una superclase común, como *Paralelogramo* o *FormaGeometrica*.

¿Nos está fallando la regla “es un” en un ejemplo tan trivial como éste? No, en realidad no. En cuanto a comportamiento, un objeto *Circulo* no **es un** objeto *Eclipse*. La norma “es un” hace referencia al comportamiento de objetos, no al comportamiento de entidades reales (del mundo material que nos rodea) o matemáticas. Aun cuando un círculo es una elipse, matemáticamente hablando, un objeto *Circulo* no es un objeto *Eclipse*: su comportamiento difiere. En la POO sólo importa el comportamiento de los objetos.

Podría sobrescribirse la clase *Circulo*, prohibiendo el uso de *setSemiejeA* y *setSemiejeB* mediante el lanzamiento de excepciones *IllegalOperation* cuando se las invocará y la creación de métodos *setRadio*. Ahora bien, esto seguiría impidiendo que *Circulo* fuera un subtipo de *Eclipse*. Además, complicaría innecesariamente la situación: recuerda un poco a la técnica de golpear con el puño un televisor averiado para que funcione. A veces resulta, pero nadie se gana la vida reparando así televisores...

De los tipos de herencia vistos en el Apdo. 11.1, las herencias de implementación, generalización, limitación y variación violan casi siempre el principio de substitución de Liskov.

En líneas generales, conviene desconfiar cuando la incorporación de una clase hija obliga a modificar la clase padre. Cuando uno se descubre dando golpecitos –en sentido figurado, espero– a su código, es el momento de replantearse el diseño de la jerarquía de clases desde el principio. Por supuesto, es posible continuar adelante y resolverlo todo por medio de soluciones *ad hoc*; incluso el programa así obtenido puede funcionar correctamente y no revelar al exterior fallos de diseño. Al fin y al cabo, un gato muerto también rebotará en el suelo, como si estuviera vivo, si se le arroja desde una altura lo bastante elevada. ¿Problemas? Se compromete la reutilización o ampliación del código, la documentación se vuelve más voluminosa si se quiere poder entender lo hecho, y el programa cada vez es más inestable ante nuevos cambios.

Podemos ver otro ejemplo de las consecuencias de la violación del principio de Liskov con el siguiente ejemplo:

```
public class Vector3D {

    // Código en Java

    private double x,y,z; // Coordenadas espaciales

    public Vector3D(double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public double moduloCuadrado() {
        return (x*x + y*y + z*z);
    }
}

public class VectorEspacioTiempo {

    // Código en Java

    private double t; // Coordenada temporal

    public VectorEspacioTiempo(double x, double y, double z, double t) {
        super( x,y,z); //Llamada al constructor de la superclase
        this.t = t;
    }

    public double moduloCuadrado() {
        return (t*t – super.moduloCuadrado());
    }
}
```

Las instancias de la clase *Vector3D* representan vectores en el espacio tridimensional habitual; las de *VectorEspacioTiempo* designan vectores tetradimensionales del espacio-tiempo. Un programa que use una función como ésta:

```
public double longitud(Vector3D v) {
    return Math.sqrt(v.moduloCuadrado());
}
```

obtendrá los resultados esperados cuando le pase un objeto *Vector3D* (la

función devolverá un número real no negativo); pero su comportamiento cuando se le pase como argumento un objeto *VectorEspacioTiempo* puede ser distinto del esperado (lanzamiento de excepciones), pues la raíz cuadrada de un número negativo no es un número real, es un número imaginario.

El álgebra de los vectores del espacio-tiempo no coincide con el álgebra de los vectores tridimensionales: la longitud de un tetravector puede ser un número negativo, situación no permitida para los vectores tridimensionales.

De una manera más práctica, el principio de Liskov fue reformulado por Bertrand Meyer ([**Meyer B., Construcción de software orientado a objetos (2ª Ed.), 1999**]) Meyer, una de las pocas personas que ha realizado grandes contribuciones tanto a la teoría como a la práctica de la OO, fue el primero en difundir el **diseño por contrato**. Un contrato es una especificación de lo que una operación debe realizar o lograr. Los contratos suelen formularse por medio de **precondiciones** y **poscondiciones**. Una precondición describe condiciones que deben ser verdaderas cuando se invoca una operación. Una poscondición describe condiciones que deben ser verdaderas tras la invocación de una operación; la implementación de la operación (método) será errónea si su poscondición resulta falsa después de invocar la operación con la precondición cumplida. Las precondiciones y poscondiciones suelen hacer referencia a los **invariantes de clase**: afirmaciones acerca de los valores de los atributos que deben cumplir todos los objetos de una clase.

Si la llamada a una operación satisface las precondiciones de un contrato, éste nos asegura que las poscondiciones serán verdaderas tras la ejecución de la operación.

Consideremos una función matemática $d(X, Y)$ que nos da la distancia euclídea al cuadrado entre dos puntos X e Y del espacio bidimensional. Una precondición de $d(X, Y)$ es que X e Y deben ser vectores reales del espacio 2D; es decir, sus componentes deben ser números reales, no imaginarios. Una poscondición de esta función es que la distancia al cuadrado entre dos vectores es no nula (≥ 0). Si se cumple la precondición al llamar a la función $d(X, Y)$, se cumplirá la poscondición tras haberse ejecutado ésta. Supongamos que incumplimos la precondición:

$$d((2, +3j), (1,0)). \quad j^2 = -1$$

Entonces también se incumplirá la poscondición: la distancia al cuadrado será de -8 unidades de longitud; la distancia será pues un número imaginario.

A continuación se expresa el principio de Liskov en el lenguaje de precondiciones y poscondiciones.

Una subclase define un subtipo del tipo de la superclase (tipo de la superclase = supertipo) cuando se cumplen las siguientes condiciones:

- ▶ La subclase incluye todos los métodos de sus superclases, con las mismas declaraciones.
- ▶ Si existe una conexión entre resultados de métodos de la superclase, debe mantenerse ese vínculo también en su subclase.
- ▶ No deben endurecerse, respecto a la superclase, las precondiciones

que afecten a un método. Dicho de otra forma, pueden exigirse requisitos más débiles a los métodos de la superclase.

- ▶ No deben suavizarse, respecto a la superclase, las poscondiciones que afecten a un método. Es decir, no pueden existir más excepciones que en el método de la superclase; pero pueden exigirse más requisitos.
- ▶ Los invariantes de una clase deben mantenerse en las subclases.
- ▶ Cualquier restricción, explícita o implícita, sobre los valores de las variables de instancia de la superclase debe ser satisfecha también por la subclase. La clase puede endurecer estas restricciones, pero no suavizarlas.

Cabe destacar que las anteriores condiciones sólo se aplican a métodos redefinidos en la subclase, no a nuevos métodos (lo cual incluye también métodos con el mismo nombre pero distintas declaraciones o firmas).

Estas seis reglas nos aseguran que cualquier subclase que las verifique cumplirá el principio de substitución en la forma expresada en la página 46. Se pueden resumir brevemente en la frase de Meyer: *“Un subtipo no debe requerir más y no debe prometer menos que su supertipo”*.

Java, al igual que C++, Delphi o C#, no satisface necesariamente el principio de substitución. Tal y como vimos al principio del subapartado, es posible tener subclases y no subtipos. Aún peor: podemos declarar definiciones de subtipos que son rechazadas por el compilador de Java.

Los programadores de Java nunca hemos tenido que preocuparnos de la primera regla, pues en Java se cumple automáticamente. En un lenguaje como Eiffel, en cambio, una subclase puede eliminar métodos de su superclase.

Con respecto a la segunda regla, en Java (al igual que en Smalltalk), todas las clases descienden de una superclase común: ***java.lang.Object***; esta superclase tiene dos métodos relacionados entre sí (*equals()* y *hashCode()*). El método *equals* (*Object obj*) permite averiguar si dos objetos son iguales, basándose en los valores de sus códigos *hash*. Si en las subclases, es decir, en cualquier clase de Java, se redefine *equals()*, pero no *hashCode()*, el método *equals()* sobrecargado –resultado de implementar *equals()* en la subclase– no funcionará correctamente, o sea, no dará una evaluación correcta de la igualdad entre dos objetos. Java implementa *hashCode()* y *equals()* para los tipos primitivos y las clases predefinidas, pero es el programador quien debe implementarlas para las nuevas clases que defina. En tanto que clases como ***java.util.HashMap***, ***java.util.Hashtable*** y ***java.util.TreeMap*** usan indirectamente *hashCode()* para evaluar la igualdad de objetos, no será posible encontrar objetos en instancias de estas clases si sólo se ha redefinido *equals()*, y no *hashCode()*.

El siguiente código ayudará a ejemplificar los problemas que pueden derivar el incumplimiento de la segunda regla:

```
public class NIF {  
  
    // Código en Java  
    // Esta clase representa números de identificación fiscal  
  
    private long numero;  
    private char letra;  
  
    public NIF (long numero, char letra) {  
        this.numero = numero;  
        this.letra = letra;  
    }  
  
    public boolean equals(Object o) {  
        if (o == this)  
            return true;  
        if (!(o instanceof NIF))  
            return false;  
        NIF temporal = (NIF) o;  
        return ((temporal.numero == numero) &&  
            (temporal.letra == letra));  
    }  
  
    // NÓTESE QUE SE HA OMITIDO EL MÉTODO hashCode()  
}
```

Supongamos que se usa la clase NIF con un *HashMap*:

```
Map listaNIF= new HashMap();  
listaNIF.put(new NIF(73573491, 'a'), "Luis");
```

Al intentar recuperar la clave (*Luis*) que corresponde al NIF introducido, mediante `listaNIF.get(new NIF(73573491, 'a'))`, nos encontraremos que obtenemos *null*.

Por este motivo, siempre es recomendable redefinir métodos como `equals()`, `hashCode()` y `toString()`, incluso en las clases más sencillas. Nunca se sabe qué uso futuro se les puede dar. Conviene acostumbrarnos desde el principio a estas redefiniciones, aunque su esencia se comprenda más adelante. No sólo porque pueden evitar errores cuando se usen las colecciones de Java, sino también porque es una consideración a tener en cuenta cuando fabriquemos nuestras clases.

Un buen ejemplo de lo que puede pasar si se viola la tercera regla nos lo proporciona esta situación: consideremos dos subclases, *Auxiliar* y *Administrativo*, que heredan de una superclase *Persona*. *Persona* cuenta con un método `double obtenerSalario(double presupuesto)` –uso notación de Java–, debidamente implementado en sus subclases.

Se podrían los resultados de `obtenerSalario()` con un código como éste:

```

Persona personas[] = new Persona[10];
personas[0] = new Auxiliar(...); // No importan parámetros del constructor
...                               // Se rellena el array
personas[9] = new Administrativo(...);

for (int i=0; i<=9; i++) {
    personas[i].obtenerSalario(10000000); // presupuesto= 10.000.000 Euros
}

```

Imaginemos ahora que se añade al esquema una nueva subclase *Profesor*, siempre tan sujeta a congelaciones de sueldo, y cuya implementación de *obtenerSalario* necesita un parámetro adicional (*inflacion*). Nos hallamos ante un caso de endurecimiento, con respecto a la superclase, de las precondiciones para el método de la subclase. Veamos cómo quedaría el código para obtener el salario:

```

Persona personas[] = new Persona[10];
personas[0] = new Auxiliar(...); // No importan parámetros del constructor
...                               // Se rellena el array
personas[9] = new Administrativo (...);

for (int i=0; i<=9; i++){
    if (personas[i] instanceof Profesor) {

        personas[i].obtenerSalario(10000000, 2.75);
                                // presupuesto= 10.000.000 Euros
                                //inflación anual = 2,75% (un tanto optimista...)
    }
    else
        personas[i].obtenerSalario(1000000);
}

```

El endurecer las precondiciones en una de las subclases, conduce a que el código no sea verdaderamente orientado a objetos (¿dónde quedan el polimorfismo y la reutilización del código ya escrito?). Esta violación lleva a que debemos plantearnos si *Profesor* no es una subclase de *Persona* (lo cual no parece el caso) o si la declaración de la operación *obtenerSalario()* en *Persona* no es lo bastante general. Resultados semejantes se obtienen cuando se viola la cuarta regla.

Hemos visto ya dos ejemplos de lo que sucede cuando se viola la quinta regla. En el caso de los vectores espacio-temporales se viola el invariante de los vectores tridimensionales: la longitud de un vector es un número real no negativo. En el caso de los círculos y eclipses, el invariante es un poco más sutil, pero aparece implícito: los semiejes de una elipse son independientes entre sí. Un círculo viola este invariante.

Las consecuencias del incumplimiento de la sexta regla saltan a la vista: objetos de las subclases tendrán estados no permitidos por la superclase.

11.3. Algunos consejos prácticos.

Cuando uno comienza a construir jerarquías de clase, bastante consigue si identifica correctamente las superclases y las subclases; al igual que, cuando uno está aprendiendo a conducir, atinar con la marcha que corresponde es todo un logro.

Con el tiempo, empero, uno va descubriendo qué jerarquías de clases devienen más útiles y cuáles no. Para mí, afirmar que una biblioteca de clases es útil equivale a decir que es flexible (permite incorporar nuevas clases sin ocasionar cambios sustanciales en las ya existentes) y, por consiguiente, reutilizable.

Por lo que sé, no existe ningún método formal para desarrollar o mejorar jerarquías de clase, así que uno suele depender de su intuición y experiencia, o de la de otros. Por mi propia experiencia (C++, Smalltalk, Eiffel y Java), he encontrado, mejor, he reencontrado, algunas reglas, fundamentalmente empíricas, que me han resultado útiles para construir jerarquías de clases. A saber:

1. Resulta conveniente construir jerarquías estrechas y profundas. Esto es, con no demasiadas clases en cada nivel de especialización y con muchos niveles.
2. Conviene evitar que cualquier clase contenga código para averiguar la clase o el tipo de un objeto. Una jerarquía de clases bien construida debe ser auténticamente polimorfa.
3. El diseño de una jerarquía de clases no debe considerarse aisladamente del tamaño de los métodos de las clases que la componen. Cuanto más extenso sea el código de un método, más difícil resultará reutilizarlo en otras clases de la jerarquía.
4. Todos aquellos métodos compartidos por las clases deben situarse en la superclase base (que puede ser una clase convencional, una clase abstracta o un *interface*).
5. Debe intentarse seguir siempre el principio de substitución de Liskov. Así, las subclases serán también subtipos y extenderán la conducta de la superclase. Si esto no resultara posible, conviene que las clases incumplidoras del principio se sitúen en el nivel inferior de la jerarquía, para así no romper la homogeneidad de las clases de niveles superiores.
6. El uso de patrones (el patrón Estado, por caso) puede llevar a la violación de algunas de las reglas anteriores o al crecimiento excesivo del número de clases. Interesa pues valorar el uso de patrones.
7. El encapsulamiento, fundamental en la OO, resulta ser el cemento de las jerarquías de clases. El acceso a los atributos sólo mediante la interfaz de las clases implica desacoplamiento entre las clases y fomenta la abstracción.
8. Los métodos no utilizados por la mayoría de las subclases no deben ubicarse en una superclase, sino en las subclases que los usen. Se evita así la propagación a lo largo de la jerarquía de métodos poco utilizados, lo cual puede desencadenar pérdida de eficacia (se desperdicia memoria; véase el Apdo. 12.3). Por otro lado, la superclase adquiere así un carácter más abstracto, más genérico y,

por ende, más reutilizable.

9. El empleo de clases abstractas o *interfaces* como superclases presenta ventajas: todas las subclases se amoldan a una interfaz común, ampliable en el futuro.
10. Conviene evitar (o al menos limitar) el uso de las herencias de implementación, generalización, extensión y limitación (descritas en el Apdo. 11.1).

12. Polimorfismo. Interfaces. Implementación del polimorfismo en los lenguajes OO: algunos ejemplos.

12.1. Polimorfismo.

La palabra polimorfismo (“muchas formas”) no se vincula sólo a la programación; se ha usado antes en otras disciplinas. Así, Freud –con su característica actitud de sospecha ante el fluctuante protoplasma infantil– ya escribió que “los niños son polimorfos perversos”. Así, en biología se usa a menudo la expresión “genes polimorfos” para referirse a genes que expresan características distintas dependiendo del ambiente donde se hallan.

Por otro lado, cualquier lenguaje natural se encuentra repleto de verbos o expresiones polimorfas; consideremos –por ejemplo– expresiones como “abrir los ojos”, “abrir la puerta” o “abrir la mente”. En cada una de ellas el verbo “abrir” expresa, dependiendo de su complemento directo, situaciones completamente distintas: los ojos no se abren como se abren las puertas, ni las mentes se abren en otro sentido que no sea metafórico (salvo que la desdichada mente se encuentre en una mesa de operaciones y bajo el bisturí de un neurocirujano...).

En programación OO, el polimorfismo representa, según [Booch G., 1994]:

[...] un concepto en la teoría de tipos, según el cual un nombre (declaración de una variable) puede denotar objetos de muchas clases diferentes, relacionados mediante alguna superclase común; de este modo, todo objeto denotado por este nombre es capaz de responder a algún conjunto común de operaciones de diferentes maneras. Cualquier objeto denotado por este nombre puede responder a algún conjunto común de operaciones. Lo contrario del polimorfismo es el monomorfismo, el cual se encuentra en todos los lenguajes que están a la vez fuerte y estáticamente ligados, como Ada.

En [Rumbaugh J. et al., 1991] podemos encontrar: “*Polimorfismo: Toma de varias formas; propiedad que permite a una operación tener distintos comportamientos en diferentes clases*”. Esta definición me parece incompleta; las variables y los atributos también pueden ser polimorfos.

Una visión más teórica la proporcionan *Object-Oriented Systems Analysis: Modeling the World in Data* ([Sally Shlaer y Stephen J. Mellor, 1988]): “

En tiempo de diseño, una invocación polimórfica es una invocación de una de las operaciones basadas en un conjunto de instancias, donde todas las operaciones publicadas en el conjunto tienen el mismo nombre de módulo, pero diferentes nombres de clases. Cuando indica una invocación polimórfica, el diseñador establece (1) las operaciones publicadas [visibles] que se invocarán, y (2) que la selección se basará en el tipo de la instancia.

El polimorfismo suele clasificarse en **universal** y **ad hoc**, de acuerdo con la clasificación establecida por Luca Cardelli y Peter Wegner en *On Understanding Types, Data Abstraction, and Polymorphism* ([Computing Surveys, Diciembre de 1985]). Dentro del polimorfismo universal puede

hablarse de polimorfismo paramétrico y de inclusión; en el polimorfismo *ad hoc*, de polimorfismo por sobrecarga de métodos y de operadores.

Cuando un método (o función) se define por una combinación de su nombre y la lista de sus parámetros (o argumentos), hablamos de **polimorfismo por sobrecarga** de funciones. Con este polimorfismo podemos usar el mismo nombre para distintas funciones (correspondientes, generalmente, a una misma operación: véase la distinción entre función/método y operación en el Apdo. 10), con diferentes listas de parámetros. También puede hablarse de sobrecarga de operadores. Internamente, los compiladores traducen los métodos y operadores sobrecargados en métodos distintos.

En un lenguaje sin polimorfismo de sobrecarga, como C, para calcular el valor absoluto de números enteros y reales sería preciso escribir código semejante a éste:

```
int valor_absoluto_entero(int a) {
    if (a>=0)
        return a;
    else
        return (-a);
}

int valor_absoluto_float(float a) {
    if (a>=0)
        return a;
    else
        return (-a);
}

int valor_absoluto_double(double a) {
    if (a>=0)
        return a;
    else
        return (-a);
}
```

y, dependiendo del argumento, llamar a una u otra.

En un lenguaje con polimorfismo de sobrecarga, los distintos métodos pueden tener el mismo nombre, y para saber el valor absoluto de un número (sea entero o real) bastaría con invocar a *valor_absoluto (numero)*. En vista del tipo de datos del número (es decir, del argumento), el compilador decidiría que método usar.

Incluso en la suma de un número entero y uno de coma flotante se percibe la larga sombra del polimorfismo. El **polimorfismo de coerción** permite que una operación aritmética pueda producir resultados de distintos tipos, dependiendo del tipo de los operandos. Por ejemplo, cuando se efectúa una operación aritmética con dos miembros de distintos tipos (*int*, *short*, *long*, *float*, *double*, etc.) el compilador realiza encubiertamente una conversión automática

entre los tipos. Mediante la conversión implícita de tipos, una sola abstracción (la operación producto, v. g.) proporciona varios tipos. En general, el polimorfismo de coerción posibilita que un argumento sea convertido al tipo esperado por un método, evitando así un error de tipos.

En el **polimorfismo paramétrico**, el mismo objeto o función puede usarse uniformemente como parámetro en distintos contextos o situaciones, sin cambios. Las clases genéricas o paramétricas se usan para facilitar la definición de clases de ámbito genérico, no específicas de un determinado tipo de datos. Una clase genérica permite que la clase concreta de la implementación en software de un TAD permanezca sin especificar hasta que sea instanciada en tiempo de ejecución por un mensaje.

En estas clases se definen todos los métodos que se precisarán, pero el tipo de datos real que manipularán se especificará como un parámetro en el momento en que se creen los objetos de la clase. Podemos imaginarnos una clase paramétrica como una colección de clases, fantasmales casi, que flotan en el aire del compilador; cuando una de ellas se instancia con el tipo de datos que va a manipular, se materializa la clase necesaria. Así, una clase genérica *Pila* (más exactamente, *Pila-de*) puede usarse para general pilas contenedoras de números enteros, de coma flotante, de objetos del sistema o definidos por el usuario. De forma pseudoaritmética:

Clase genérica + argumentos = clase concreta

Los tipos genéricos definidos por las clases genéricas son variables de tipos que pueden ser instanciadas con un tipo específico para cada instancia de la clase.

No todos los lenguajes OO admiten el polimorfismo paramétrico. En Java y C#, verbigracia, no está implementado. Sí existen extensiones de Java que lo admiten –la más antigua es *Pizza*, ahora *General Java*–, y se espera que figure en la versión 1.5 de Java (*Tiger*). También se anuncia como una de las nuevas características de la segunda versión de C#. C++ sí lo admite, mediante el uso de *templates* (plantillas). Para declarar una clase genérica en C++ se usa:

```
Template <class Ttipo> nombre_clase;
```

donde *Ttipo* es el marcador del lugar del nombre del tipo que se especificará cuando se instancie la clase. Para crear una instancia específica se emplea la sintaxis:

```
Template <class Ttipo> nombre_clase;
```

donde *Ttipo* es el nombre del tipo de datos sobre el que operará la clase.

Veamos cómo se implementaría nuestro sufrido TAD *Pila* usando clases paramétricas:

```
// Código en C++
#include <iostream.h>

template <class TipoPila> class Pila {

private:
    TipoPila pl[100] // Pila de 100 elementos como máximo
    int cima;

public:
    Pila() {
        cima=0;
    }

    void apilar(TipoPila tp) {
        if (cima==100) {
            cout << "Pila llena. El limite es de 100 elementos.\n"
            return 0;
        }
        else {
            pl[cima]=tp;
            cima++;
        }
    }

    TipoPila desapilar() {
        if (cima==0) {
            cout << "Pila vacia. Introduzca antes algún elemento.\n"
            return 0;
        }
        else {
            return pl[cima - -];
        }
    }
}
```

Si quisiéramos crear pilas de enteros y de caracteres alfanuméricos, bastaría con escribir código como éste:

```
Pila <int> p1;
Pila <char> p2;
p1.apilar(3);
p1.apilar(7);
p2.apilar('c');
p2.apilar('z');
```

Puede ser que algún lector se haya extrañado al leer que Java o C# no permiten, por ahora, el polimorfismo paramétrico. A fin de cuentas, resulta factible escribir esto:

```
public class Pila {  
  
    // Código en Java  
    private Object pl[100] // Pila de 100 elementos como máximo  
    private int cima;  
  
    public Pila(){  
        cima=0;  
    }  
  
    public void apilar(TipoPila tp) {  
        if (cima==100) {  
            System.out.println("Pila llena. El limite es de 100 elementos.");  
        }  
        else {  
            pl[cima]=tp;  
            cima++;  
        }  
    }  
  
    public Object desapilar() {  
        if (cima==0) {  
            System.out.println("Pila vacia. Introduzca antes algún elemento");  
            return null;  
        }  
        else {  
            return pl[cima - - ];  
        }  
    }  
}
```

Como en Java (y también en C# o Smalltalk) todas las clases derivan de una superclase común *Object* (los tipos primitivos no), las instancias de la clase *Pila* arriba expuesta pueden almacenar distintos tipos de objetos. Los problemas surgen cuando se intenta añadir objetos distintos, no vinculados por herencia, a una misma instancia de *Pila*. Si intentamos añadir un *String* a una pila de objetos *Ministro*, el compilador de Java no dará error o aviso alguno. Lo peor vendrá poco después: el programa fallará en tiempo de ejecución. El compilador no puede informarnos de si el código intenta realizar operaciones no válidas sobre *Ttipo*, ya que sólo la máquina virtual Java conocerá, en tiempo de ejecución, el verdadero tipo de cada objeto. Y como bien sabe cualquier programador, un algoritmo que se comporte de manera impredecible es tan fiable como las promesas de un objeto *Político* en campaña electoral.

Por otro lado, para luego utilizar los objetos almacenados en una instancia de *Pila* se necesitarán hacer conversiones (*casting*) de tipos de *Object*, lo cual implica –además del riesgo de excepciones *ClassCastException*– una pérdida de velocidad en tiempo de ejecución, ya que las conversiones dinámicas de tipos son muy ineficaces.

En un verdadero polimorfismo paramétrico, como el de C++, el compilador comprueba en tiempo de compilación que todos los objetos de una instancia de una clase genérica son de tipos compatibles.

Código como éste producirá un error en tiempo de compilación:

```
Pila <int> p1;  
p1.apilar('s'); // Error
```

El **polimorfismo de inclusión o de substitución** se produce cuando un método definido en una clase se redefine en alguna de sus subclases manteniendo la misma declaración (nombre y lista de parámetros). Se soslaya, pues, el método original, sustituyéndolo por otro en la subclase. Recordemos: ***un objeto de una subclase puede usarse en cualquier lugar donde se admita un objeto de la superclase***. En consecuencia, el polimorfismo de inclusión permite establecer referencias a objetos de cualquier clase en una jerarquía de clases como si fueran instancias de la superclase.

Consideremos un ejemplo, utilizando las clases *Persona* y *Estudiante* del Apdo. 11.1:

```
Persona p=new Persona ("Javier", "Perez");  
Estudiante e=new Estudiante("Jose", "Garcia");
```

Invoquemos los métodos *obtenerInfo()* de cada uno de los objetos:

```
p.obtenerInfo();
```

El intérprete de Java buscará el método *obtenerInfo()*, sin argumentos, en la clase *Persona* (es el objeto cuyo método ha sido invocado), lo encontrará y lo invocará.

Lo mismo sucederá con

```
e.obtenerInfo();
```

sólo que buscará en la clase *Estudiante*.

Sin embargo, la situación ahora será completamente distinta:

```
e.obtenerNombre(); //Método que está sólo en la clase Persona
```

Ahora, el intérprete de Java buscará el método *obtenerNombre()* en la clase *Estudiante*, y no lo encontrará. El intérprete, incansable, seguirá buscando en la superclase de *Estudiante* –*Persona*–, lo encontrará y lo invocará.

El siguiente código ilustra muy bien el polimorfismo de inclusión en Java:

```
Persona p1=new Estudiante("Javier", "Perez", "MC002");
```

Como vimos en el apartado dedicado a la herencia, puede usarse un objeto de la subclase en cualquier lugar donde pueda usarse un objeto de la superclase. En consecuencia, podemos escribir:

```
p1.obtenerInfo();
```

En este caso, el intérprete de Java empezará buscando algún método coincidente en la clase del objeto sobre el cual se ha enviado el mensaje (objeto receptor). Como *p1* hace referencia a un objeto de la clase *Estudiante*, el método invocado será el de esa clase, no el de la clase *Persona*. O, lo que es lo mismo, la llamada a un método se amolda al tipo real de objeto (*Estudiante*) y no al tipo de la variable que hace referencia a él (*Persona*). Esta conclusión es general para todos los lenguajes OO, no está limitada a Java.

La división del polimorfismo en universal y *ad hoc* no se debe al azar. Visto lo anterior, se aprecia que el polimorfismo universal cuenta con más quilates que el *ad hoc*. El polimorfismo de sobrecarga representa una propiedad sintáctica de los lenguajes que lo implementan, pero conceptualmente no aporta novedad: podríamos utilizar nombres distintos para las funciones sobrecargadas y nada cambiaría. Con todo, permite que se ahorre tiempo y esfuerzo en definir los nombres de las funciones. De resultas, el programador puede dedicar más tiempo a desarrollar las aplicaciones. El polimorfismo de coerción discurre por un camino conceptual muy similar.

El polimorfismo universal es más *puro* y requiere que los lenguajes que lo implementan cuenten con ligadura dinámica (se verá más adelante).

Cuando se consideran lenguajes OO, la noción de tipos de datos aceptada hasta ahora (es decir, colección homogénea de valores junto a una interfaz) muestra sus grietas y desconchones conceptuales (ausentes en lenguajes sin polimorfismo). En los lenguajes polimórficos, una variable de un tipo puede denotar instancias de muchas clases distintas, cada una con su propia estructura. Un tipo, por tanto, puede referirse a colecciones heterogéneas de objetos, si bien no totalmente independientes.

Una operación *Operacion(x: X)* que devuelva objetos de tipo *Y* funcionará de distintas maneras en lenguajes polimórficos o monomórficos. En lenguajes polimórficos (Pascal, Ada, C), *x* e *y* harán referencia a valores o instancias de los tipos *X* e *Y*. En los polimórficos, *x* e *y* pueden referirse a objetos de distintos tipos.

Una variable en un lenguaje con polimorfismo puede referirse a instancias de la clase asociada en su declaración o de cualquiera de sus subclases. Así pues, se precisa considerar que una variable tiene un tipo estático y un conjunto de tipos dinámicos.

El **tipo estático** es el tipo asociado a la variable cuando es declarada; el **tipo dinámico** es el tipo correspondiente a la clase del objeto vinculado a la variable en tiempo de ejecución. El conjunto de tipos dinámicos de una variable no puede ser cualquiera: viene condicionado por la herencia. Por consiguiente, el polimorfismo –mejor dicho, el polimorfismo de inclusión; se verá su definición más adelante– no es libre, está restringido por la herencia.

Los lenguajes de tipos estáticos (descritos en el Apdo. 9.2) garantizan, dado un mensaje *x.Operacion()*, que **al menos** existirá una implementación apropiada de *Operacion*; en caso contrario, el programa no compilará. Por otro lado, la ligadura dinámica (descrita en el siguiente subapartado) garantiza que se escogerá la implementación adecuada.

En cambio, los lenguajes de tipos dinámicos no pueden garantizar nada acerca de si existe alguna implementación de *Operacion* para el objeto *x*. *Operacion* puede perfectamente no pertenecer al protocolo del objeto *x*. Si es así, obtendremos en tiempo de compilación un error del tipo “*x’ no entendió Operacion*” u “*Operacion no es un método válido de x*”. Un lenguaje de tipos dinámicos exige comprobar los tipos en tiempos de ejecución o controlar los errores que pueden producirse en tiempo de ejecución.

Así las cosas, nuestra noción inicial de tipo debe ser reemplazada por una noción que asuma que todas las instancias con un mismo comportamiento (aun de distintas clases) pertenecen al mismo tipo. Si lo preferimos en palabras de un teórico como Peter Wegner (*Dimensions of object-based language design* [OOPSLA, 1987]):

[...] un tipo constituye una especificación de comportamiento que puede usarse para generar instancias con el mismo comportamiento. [...] en sistemas de tipos polimórficos [...] perdemos la simple noción intuitiva de tipo como clasificador de colecciones de valores, y debemos reemplazar esta noción por una noción más rica, pero menos intuitiva, de los tipos como clasificadores de contextos para evaluación.

En la POO, un tipo corresponde a la implementación de un TAD; da cuenta de las características comunes de un conjunto de objetos que exhiben el mismo comportamiento (también llamados instancias del tipo).

Muchos autores y textos consideran “tipo” y “clase” como términos intercambiables; no es así: existen algunas diferencias importantes, un tanto sutiles.

Como se vio en el Apdo. 9.2, los tipos son herramientas que aumentan la productividad del programador. Al forzar al usuario a declarar los tipos de las variables y expresiones que manipula, los compiladores pueden realizar las comprobaciones en tiempo de compilación. En síntesis: los tipos se usan principalmente para comprobar si los programas son correctos.

Una clase, en cambio, constituye un objeto más relacionado con el tiempo de ejecución. Tal y como se mostró en el Apdo. 9.1, las clases pueden entenderse como plantillas o fábricas de objetos y como colecciones de objetos. Esta dualidad no existe en los tipos. Éstos proporcionan información de la interfaz y determinan qué operaciones están permitidas, mientras que las clases proporcionan información referente a la implementación, incluyendo los cuerpos de los métodos y valores iniciales para las variables de instancia. Las clases no se usan para comprobar la corrección de los programas, sino para crear y manipular objetos. En tiempo de ejecución, un objeto tiene clase, no tipo.

Conclusión: pese a estar relacionados, no son términos equivalentes: *tipo* hace hincapié en la existencia de un comportamiento común, mientras que

clase se refiere a la existencia de una estructura y un comportamiento común.

Muchas veces las clases se utilizan como tipos, es decir, para especificar o declarar los valores que se consideran aceptables en un cierto contexto. Aun así, la diferencia persiste: **una clase no sólo define los métodos que un objeto puede entender (su protocolo), que sería lo único verdaderamente imprescindible y necesario para averiguar si éste puede aceptarse en cierto contexto, sino que también define cómo reacciona el objeto en respuesta a un mensaje.**

Algunos lenguajes OO permiten definir tipos, clases (en el sentido de estructura sintáctica), tipos y clases o ninguno de los dos. Por ejemplo, Java y C# permiten definir tipos usando la construcción *interface*, además de clases. Un *interface* declara un tipo, pero sin implementación. Los *interfaces* suelen compararse con las clases puras abstractas de C++ (clases en las que todas las funciones miembro se han definido abstractas), pero aquéllos no sólo separan la declaración del tipo de su implementación, sino también la herencia de clases y la de tipos.

Una clase suele corresponder directamente a un tipo (no se confunda *corresponder* con *ser*), pero no siempre sucede lo contrario. En Java, una variable puede ser del tipo designado por una clase, un tipo primitivo, una matriz o un *interface*. Por ejemplo:

```
// Código en Java
List lista; // Se declara lista como del tipo List
lista = new ArrayList(); // Se asocia a un objeto de una
                        // clase que implementa el interface List
```

En Eiffel o Ruby, en cambio, cada tipo debe estar basado en una clase. Todos los tipos, incluso los primitivos y las matrices, derivan de clases. En dichos lenguajes, cada variable o valor con nombre representa una referencia a un objeto, no el propio objeto.

Por último, y visto lo visto, echemos una ojeada a lo que distintos personajes, imaginarios, por supuesto, responderían a la pregunta “¿Qué entendería usted por un tipo?”:

- **Programador con unas semanas de experiencia:** Un tipo es un nombre o identificador para un conjunto de valores aceptados por el lenguaje de programación... ¿Estoy en lo cierto? Algo así ponía en mi manual de C.
- **Programador con unos meses de experiencia:** Un tipo es un conjunto de valores y operaciones.
- **Programador con unos años de experiencia:** Un tipo constituye una estrategia de los lenguajes de programación con tipos para clasificar valores por su comportamiento y para disminuir la aparición de errores... En realidad, cada día que pasa entiendo menos los tipos...
- **Informático con alma de matemático:** ¡Por la santísima trinidad (léase von Neumann, Turing y Gödel)! ¿Cómo no habré caído antes? Un tipo, es decir, un conjunto de valores y operaciones, es un álgebra que admite homeomorfismos y define un cuerpo de valores. Trivial. Q.e.d.

- **Informático con alma de filósofo:** No debemos confundir, como hacen el resto de mis colegas, las palabras y las cosas que son designadas por las palabras. Un tipo es conceptual... [*pausa dramática*] es una entidad conceptual a cuyos valores sólo puede accederse por medio de la interfaz del tipo. La interfaz también resulta conceptual... [*suspiro dramático*] es el velo que oculta el interior de las operaciones... [*expiración dramática*] entendiendo por interior un interior conceptual... un interior casi exterior, claro es.
- **Ingeniero de software alejado del mundanal ruido y con necesidad de publicar artículos como sea:** Poco importa que los tipos sean conceptuales o patatas fritas. Lo único importante es que ayudan a clasificar los lenguajes de programación y a estudiar la evolución de las técnicas de desarrollo de software, tal y como se verá en mi próxima publicación.
- **Individuo enamorado de los ceros y unos (programador de compiladores o jinete del silicio):** Los sistemas operativos, cuyos corazones son los *kernels*, no entienden de entidades conceptuales. Ellos sólo entienden ceros y unos, uno detrás de otro, siguiendo una misteriosa melodía, la melodía de la CPU. Los tipos son un medio para especificar los requisitos de almacenamiento para las variables declaradas del tipo.

12.2. Interfaces.

En lo que sigue, tal y como he hecho hasta ahora, utilizo la palabra **interfaz** (en castellano y sin cursiva; la considero de género femenino), para designar el conjunto de mensajes al que un objeto puede responder. Esto es, para referirme al aspecto exterior que un objeto presentar al resto de los objetos. Empleo, en cambio, la palabra **interface** (en inglés y con cursiva; la considero de género masculino) para referirme a la construcción sintáctica presente en lenguajes como Java y C#.

La introducción en Java de la palabra reservada *interface* fue un golpe de originalidad, luego aprovechado por Delphi (Object Pascal) y C#. Los *interfaces* no son inventos teóricos, antes al contrario: además de permitir la distinción entre tipo y clase, resultan muy prácticos para los programadores. Así como las clases implementan tipos en el código fuente mediante la implementación de sus métodos, los *interfaces* no implementan tipos en el código fuente, sólo los definen.

Esta aparente incompletitud no deviene gravosa para el programador, antes bien, aporta flexibilidad a su código. Supongamos que definimos un *interface Leer*. Podrá ser utilizado para leer disquetes, CD-ROMs, DVDs, libros electrónicos, documentos HTML, etc. Si el tipo *Leer* no se implementara como un *interface*, sino como una clase, permanecería atado a una implementación –la escogida para la clase–, y ya no resultaría útil para una gama de situaciones como la presentada antes.

Un *interface* de Java, de C# o de Delphi resulta ser, al fin y al cabo, un mero **contrato**. Cualquier clase de Java que declare –mediante *implements*– un *interface* debe respetar el contrato.

En Java, un *interface Leer* se introduciría así:

```
public interface LeerMedio {  
    public void Leer (Medio medio);  
}
```

Cualquier clase que implementara Leer lo haría así:

```
public class DVD implements Leer {  
    ...  
    public void LeerMedio (Medio medio) {  
        //Código del método para leer un DVD  
    }  
    ... // Resto de métodos  
}
```

La introducción de los *interfaces* conlleva importantes consecuencias para los programadores. Si declaramos una variable *miDVD* como de tipo *DVD*, se tendrá pleno acceso, mediante esa variable, a los métodos no privados del objeto. Sin embargo, si la declaramos como de tipo *Leer*, sólo podrá accederse con esa variable a los métodos que *Leer* declara; en este caso, *LeerMedio*. Cualquier petición de otro método declarado público en la clase *DVD* será rechazado.

Por otro lado, los *interfaces* permiten factorizar el código y extraer comportamientos comunes de las jerarquías de clases; todo ello redundando en la producción de código más compacto e inteligible.

Para darnos cuenta de la flexibilidad de los *interfaces* basta con que consideremos el siguiente código:

```
// Código en Java  
ArrayList lista = new ArrayList();
```

Nota adicional para los lectores principiantes en Java: Las clases *ArrayList* y *LinkedList* forman parte del paquete *java.util* y son dos de las colecciones que proporciona Java. Ambas descenden de la superclase abstracta *AbstractList* e implementan el *interface List* (que declara los siguientes métodos: *containsAll*, *equals*, *hashCode*, *iterator*, *listIterator*, *listIterator*, *remove*, *removeAll*, *retainAll*, *subList*). Dependiendo del uso que vayamos a dar a la lista (inserción y eliminación de elementos en posiciones arbitrarias, o al principio y final de ésta) conviene emplear una u otra.

Sun recomienda el uso de *ArrayList* o *LinkedList* en lugar de *Vector*, pues no se encuentran sincronizadas (distintos hilos pueden acceder concurrentemente a una misma instancia) y su eficacia es mayor. *Vector* se mantiene por compatibilidad con las primeras versiones de Java.

Compilará sin problemas. Con todo y con eso, no resulta muy flexible. ¿Y si el programador necesita en algún momento, por motivos de eficacia, cambiar de un *ArrayList* a un *LinkedList*, o viceversa? Podría probar con este código:

```
// Código en Java; no compilará
ArrayList lista = new ArrayList();
lista = new LinkedList();
```

Como la clase *LinkedList* cuenta con métodos ausentes en *ArrayList*, el código anterior no será aceptado por el compilador de Java. La variable *lista*, tal y como se ha definido, no puede pasar de referirse a un objeto *ArrayList* a uno *LinkedList*.

Los *interfaces* nos proporcionan una solución muy elegante y versátil:

```
// Código en Java
List lista = new ArrayList();
lista = new LinkedList();
```

Al declararse *lista* como de tipo *List*, sobre ella podrán actuar todos los métodos de *List*, comunes también a la clase *LinkedList* (véase el recuadro azul de esta misma página). El programador podrá, por tanto, usar según le convenga la variable *lista*, bien apuntando a un objeto *ArrayList*, bien apuntando a un objeto *LinkedList*, con la seguridad de que responderá a un conjunto común de métodos. Obrando así, fija la interfaz (en el sentido de conjunto común métodos visibles externamente), pero deja libre la implementación.

En los libros de introducción a la programación en Java suelen presentarse los *interfaces* como una solución a la falta de herencia múltiple de Java (lo mismo es válido para C# y Delphi). Prácticamente, heredar de una sola superclase e implementar múltiples *interfaces* simula la herencia múltiple; pero sin los problemas de implementación y de sintaxis que ocasiona esta última. Un *interfaz* puede compararse a una clase abstracta pura, pero hay algunas diferencias: una clase abstracta puede incluir implementaciones parciales de métodos, mientras que un *interface* sólo puede declararlos.

Los *interfaces* separan comportamiento y reutilización del código. Permiten establecer jerarquías de comportamiento (léase jerarquías de tipos) independientes de las jerarquías de generalización/especialización derivadas de la herencia.

12.3. Implementación del polimorfismo en los lenguajes OO: algunos ejemplos.

He dudado entre incluir o no este subapartado en el artículo. Pienso que la implementación de los lenguajes OO no deberá interferir en el correcto manejo de la POO. Sin embargo, interfiere, y de gran manera. En C++ o Delphi, por ejemplo, no puede entenderse por qué se usan las palabras reservadas *virtual* o *dynamic*, sin comprender cómo implementan el polimorfismo. Podría pensarse que esto es una cuestión reservada a los lenguajes híbridos, pero la evidencia demuestra lo contrario: Java usa *final*, y las ventajas en cuanto a eficacia que puede representar su uso sólo pueden comprenderse entendiendo el modo como se implementa el polimorfismo en Java; C# usa *virtual* y otras palabras, cuya importancia sólo pueden entenderse si comprendemos como se compilan las llamadas a métodos.

En un mundo perfecto, un compilador identificaría todas las llamadas a métodos, comprobaría si hay métodos redefinidos y optimizaría el código en función de ello, sin que el programador tuviera que preocuparse de nada. Desgraciadamente, no existe un mundo perfecto. Nunca existirá.

Podría muy bien haber omitido este subapartado, pero me hubiera parecido engañoso, tanto para mí como para el lector. ¿Cómo programar de manera OO si casi todos los lenguajes OO reflejan en su sintaxis la manera como materializan el polimorfismo?

Puedo, por supuesto, errar al incluir este subapartado, pero espero que el lector tenga en cuenta que he preferido no refugiarme bajo la gruesa y cómoda manta de las excusas educadas: “la variedad de lenguajes y compiladores es tan amplia que no puede abordarse aquí la implementación del polimorfismo”, “como el lector puede buscar en la bibliografía”, “como es bien sabido”...

El polimorfismo de inclusión, derivado de la herencia, conlleva importantes consecuencias en la compilación o interpretación de lenguajes OO, pues exige consideraciones ausentes en los lenguajes no polimórficos.

Pese a que soy consciente de la diferencia entre compiladores y editores de enlaces (*linkers*), usaré –por simplicidad– *compilador* para denotar a ambos, y *compilación* para designar al proceso global que acaba con la creación de un archivo ejecutable.

El polimorfismo por sobrecarga no implica ninguna novedad con respecto a los lenguajes de tipos estáticos tradicionales (Pascal, C, Fortran, etc.), pues el compilador o intérprete escoge el método correcto durante la compilación: a pesar de tener el mismo nombre, son distintos por tener distintos argumentos. Por ello, en todo este subapartado usaré “polimorfismo” para denotar polimorfismo de inclusión.

En compilación se llama enlace o ligadura (*binding*) a la conexión que un compilador establece entre una llamada a una función y el cuerpo de ésta. Dependiendo de cuando se realiza el enlace (antes de la ejecución o durante ella), se habla, bien de enlace estático, bien de enlace dinámico. Lenguajes como Ada, Fortran, Cobol, C y Pascal usan siempre ligadura estática. En cambio, Smalltalk, CLOS, Objective-C, Java y C# son de ligadura dinámica. En

Simula, C++ y Delphi (Object Pascal) se usa el enlace estático, con la excepción de las funciones virtuales y dinámicas.

El enlace estático o temprano (*early binding*) significa que el compilador, con la llamada a un método de un objeto, selecciona el método en función del tipo de la variable o puntero que apunte o haga referencia al objeto, independientemente de la clase del objeto. La ligadura dinámica o tardía (*late binding*) es una característica común de casi todos los lenguajes OO, aun cuando puede encontrarse en lenguajes no OO, como Lisp y Prolog, en los cuales no hay declaraciones de tipos.

Implementar llamadas a funciones en lenguajes de ligadura estática resulta relativamente fácil. En cada llamada a una función, el compilador incluye un puntero a la dirección de memoria donde comienza la función (en ensamblador se realiza un CALL a la dirección donde comienza el código de la función). Para la ligadura dinámica, el proceso se torna mucho más complejo.

Los lenguajes OO exigen generalmente enlace dinámico (no hay acuerdo en la bibliografía acerca de si es imprescindible: algunos autores afirman que constituye un requisito indispensable; otros opinan que es recomendable, pero no obligatorio). Como un objeto de una subclase puede usarse en cualquier lugar donde se admita un objeto de la superclase, una variable o un puntero declarado del tipo de una superclase puede hacer referencia (apuntar) a objetos instancias de sus subclases. Es más: una misma variable puede hacer referencia durante su tiempo de vida a objetos de subclases distintas. En consecuencia, un mensaje con un nombre dado puede conducir a acciones distintas, dependiendo de la clase del objeto al cual se envíe.

Un compilador o un intérprete de un lenguaje OO se ve obligado a aguardar al momento inmediatamente anterior a la ejecución de un método polimórfico en el código para comprobar si el método recibe el número correcto de argumentos y si son del tipo adecuado (*dynamic type checking*), y para vincularlo con el código adecuado. La ligadura de un mensaje a un método concreto se realiza necesariamente en tiempo de ejecución, pues los objetos se crean dinámicamente (generalmente mediante *new*); sólo los tipos de las variables están fijadas en compilación. Tal y como hemos visto en el ejemplo anterior, la llamada a un método –en un lenguaje OO– se amolda al tipo del objeto y no al tipo de la variable o puntero que hace referencia (o apunta) a él.

El enlace dinámico hace posible que el método llamado de un objeto no dependa del tipo con el cual se declaró la variable o puntero, sino de la clase del propio objeto. Dicho de otro modo, permite relacionar en tiempo de ejecución un nombre o identificador (como la declaración de una variable) con una clase, de manera que la asociación identificador-clase no se lleva a cabo hasta que el objeto referenciado por el nombre es creado.

Cuando se envía en tiempo de ejecución un mensaje a un objeto y se encuentra disponible –por herencia– más de una versión del método al que invoca el mensaje, la ligadura dinámica garantiza que se escogerá la versión del método más adecuada para el objeto.

Resumiendo:

Enlace dinámico	<p>El tipo de una variable con un cierto nombre se asocia con el contenido de la variable.</p> <p>En los lenguajes con punteros, la decisión sobre qué función debe llamarse depende del objeto apuntado y no del tipo del puntero.</p>
Enlace estático	<p>El tipo de una variable con un cierto nombre se asocia con la declaración de la variable.</p> <p>En los lenguajes con punteros, el compilador utiliza el tipo del puntero para enlazar el objeto en tiempo de compilación.</p>

Cualquier lenguaje que permita el enlace tardío debe contar con alguna estrategia para averiguar en tiempo de ejecución a qué clase pertenece el objeto cuyo método se llama por medio de un puntero o una referencia del tipo de alguna de sus superclases y para llamar luego al método correspondiente.

Aunque pueda parecer contradictorio, existen dos formas de enlace dinámico: **estático** y **dinámico**.

El enlace dinámico de tipo estático se puede encontrar en C++ –más adelante se explicará cómo lo implementa–, Delphi y Eiffel. En estos lenguajes, tras compilarse un programa, se ha fijado ya qué método (o versión de una operación) corresponde a cada objeto y variable del programa. Esta resolución no resulta trivial, como en los lenguajes monomórficos, porque las subclases pueden sobrescribir o redefinir los métodos de las superclases.

Distinta se presenta la situación en el caso del enlace dinámico *puro*. En esta variedad de ligadura, la búsqueda y selección de los métodos que deben ejecutarse se realiza en tiempo de ejecución. El enlace dinámico-dinámico, usado en Java y C#, permite añadir nuevos objetos y tipos a un programa ya en marcha e interaccionar con objetos que no existían cuando comenzó la ejecución del programa. Esto no puede ocurrir en los lenguajes con enlace dinámico-estático; pues se requeriría que el compilador contara, ora con una bola de cristal, ora con alguna máquina del tiempo que cupiera en la carcasa del ordenador.

Hay situaciones en las que resulta imprescindible la ligadura dinámica pura: bases de datos, programación distribuida (CORBA), programación cliente-servidor, etc.

En los lenguajes de tipos dinámicos (obligados a contar con enlace dinámico-dinámico), la búsqueda del código de un método se realiza en tiempo de ejecución. En Smalltalk, p. ej., un procedimiento (*Lookup*) recorre las superclases del objeto receptor del mensaje hasta encontrar el método apropiado. El mecanismo se representa en la Figura 20.

Esta manera de implementar el polimorfismo presenta dos inconvenientes fundamentales:

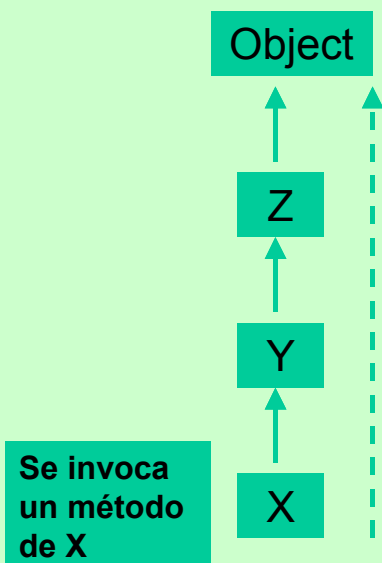
- Sólo en tiempo de ejecución se sabrá si el mensaje enviado a un objeto forma parte de su protocolo. Como puede esperarse, esto conlleva, en tiempo de ejecución, una gran cantidad de errores del género “no coinciden los tipos” o la necesidad de escribir código que compruebe los tipos en tiempo de ejecución y procese las excepciones generadas.
- Todas las operaciones, incluso las más simples (sumas de números, etc.), se ligarán dinámicamente, como se haría con cualquier mensaje que sí fuera polimorfo. Esto redundará en un tiempo de procesamiento adicional en tiempo de ejecución. En jerarquías de clases muy profundas y con muchos métodos, el tiempo necesario para localizar el método adecuado puede ser significativo.

¿Cómo funciona la ligadura dinámica en un lenguaje fuertemente tipado?

El caso de Smalltalk

¿Qué método se ejecuta cuando un objeto recibe un mensaje?

- 1) Se busca en la clase del objeto receptor un método con la misma declaración (nombre, lista de argumentos y tipo de retorno) que el método invocado.
- 2) Si lo encuentra, lo ejecuta. En caso contrario, se busca en la superclase inmediatamente superior.
- 3) Se repite el paso 2 hasta encontrar el método en alguna superclase o llegar a Object, la clase de la que descienden todos los objetos en Smalltalk.
- 4) Si se llega a Object y no se ha encontrado el método, se produce un error en tiempo de ejecución.



Miguel Ángel Abián. Julio 2003

Figura 20. Funcionamiento de la ligadura dinámica en Smalltalk

Para los lenguajes de enlace dinámico-estático (como C++, Delphi, etc.) la situación deviene más compleja: ¿cómo realizar el enlace dinámico? Esto es, ¿cómo compilar métodos polimórficos en lenguajes donde todo queda fijado en tiempo de compilación? Como el compilador –a diferencia de un intérprete– no puede averiguar el tipo del objeto en tiempo de ejecución, debe ser previsor e insertar código que sea capaz de determinarlo y que, después, llame al cuerpo del método correcto.

¿Cómo conseguirlo? La religión de los compiladores es politeísta: no existen tablas de mandamientos al respecto. El modo exacto de implementar el enlace tardío o dinámico cambia de un lenguaje a otro. Incluso dentro de un mismo lenguaje puede variar según el compilador. Existen distintas soluciones: se puede asignar un identificador a cada objeto, un puntero para cada método o un único puntero a una tabla virtual de métodos.

La solución que se va a explicar es la basada en tablas virtuales, que se usa en la mayor parte de los lenguajes compilados, además de usarse (con cambios) en muchos lenguajes interpretados. El primer lenguaje abordado es C++, porque sus compiladores han influido en la mayoría de los compiladores actuales para lenguajes OO.

C++ usa punteros a tablas virtuales y funciones virtuales (utilizo la terminología usual de C++: funciones miembro, funciones virtuales, etc., pero la idea subyacente no cambia para otros lenguajes, aunque lo haga la terminología). Esta estrategia no es original de C++, sino que procede de Simula, el ancestro y patriarca de todos los lenguajes OO.

Para conseguir el enlace dinámico en C++, toda función miembro redefinida (*overriden*) en una subclase debe definirse como una función virtual (usando la palabra reservada *virtual*) con un cierto código predeterminado por si el método no se redefine en alguna subclase. Cuando una función se declara virtual, se mantiene virtual en todas las subclases, las subclases de las subclases, y así sucesivamente. La palabra *virtual* se comporta como una bengala que avisa al compilador de que, si no tiene otro remedio, realice un enlace dinámico.

Un compilador estándar de C++ trabaja así: para cada clase con funciones miembro virtuales genera una tabla de punteros a las direcciones de memoria donde comienzan las funciones virtuales de esa clase (ya sean redefinidas o directamente heredadas de alguna superclase).

Estas tablas se denominan tablas de funciones miembro virtuales, tablas de funciones virtuales o –simplemente– tablas virtuales (*v-tables* en inglés). Cada puntero de una tabla virtual apunta a una función virtual de esa clase, haya sido redefinida o no. Las tablas virtuales se implementan habitualmente como vectores de punteros; a veces se usan listas enlazadas, pero la idea subyacente se mantiene igual.

Con las tablas virtuales el puzzle aún no se halla resuelto. Cada objeto debería contar con algún documento de identidad –binario, por supuesto– que indicara la clase de la cual ha sido instanciado (*Nacionalidad*: Binaria; *Estado civil*: Compilado/En proceso de compilación; *Lugar de nacimiento*: Intel x86;...). La solución más inmediata consistiría en que el compilador incluyera en el código ejecutable una tabla virtual para cada objeto que procediera de alguna clase con funciones virtuales.

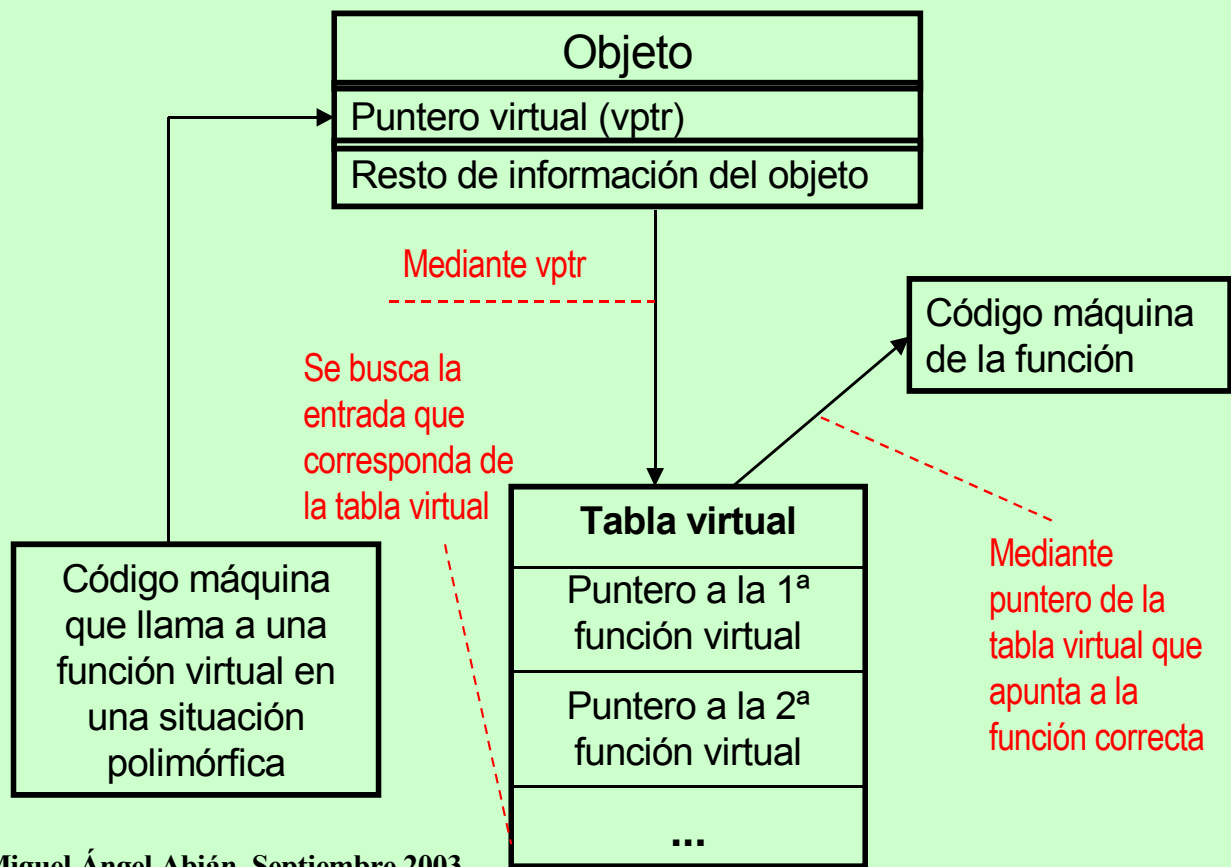
Obrar así requeriría demasiada memoria: por una parte, el fichero ejecutable podría ser muy grande (sobre todo en programas que puedan tener cientos o miles de objetos). Por otra parte, la memoria consumida por el programa en tiempo de ejecución también sería grande. Asignar a cada objeto un puntero, denominado puntero virtual (*v-pointer* o *vptr*), que apunte a la tabla virtual de la clase de donde procede el objeto resulta una solución mucho más económica en cuanto a memoria.

Con los punteros virtuales, cada vez que un compilador de C++ encuentra una llamada a una función virtual, inserta código para buscar el *vptr* en el objeto sobre el que se efectúa la llamada y con éste puntero busca la dirección de la función en la tabla virtual.

En tiempo de ejecución, cada que se llama a una función virtual en C++, sucede lo siguiente:

- 1) Se sigue el puntero virtual a la tabla virtual correspondiente a la clase a la que pertenece.
- 2) Se encuentra en la tabla virtual el puntero que hace referencia a la función que está siendo llamada.
- 3) Se invoca a la función a la que hace referencia el puntero del paso 2. En ensamblador tendremos un CALL a la posición de memoria designada por el puntero + *n* (donde *n* depende del número de funciones virtuales en la tabla virtual). Con este último paso se lleva a cabo el enlace dinámico.

La ligadura dinámica en C++



Miguel Ángel Abián. Septiembre 2003

Figura 21. Funcionamiento de la ligadura dinámica en C++ (tiempo de ejecución)

Consideremos un ejemplo muy sencillo:

```
Class Deposito {  
  
    // Código en C++  
    float volumenAgua; // litros de agua  
  
public:  
  
    virtual void ponerAgua(float cantidad) {  
        volumenAgua = volumenAgua + cantidad;  
    }  
  
    virtual void quitarAgua(float cantidad) {  
        volumenAgua = volumenAgua - cantidad;  
    }  
}
```

```

virtual void vaciarDeposito() {
    volumenAgua = 0.0;
}
...
}

```

Class DepositoMunicipal: Deposito {

```

// Código en C++
// Esta clase hereda de la clase Deposito
// Suponemos que sólo puede ponerse agua si hay menos
// de 1000 litros, y que sólo puede quitarse agua si hay más de
// 100 litros

```

public:

```

virtual void ponerAgua(float cantidad) {
    if (volumenAgua < 1000)
        volumenAgua = volumenAgua + cantidad;
}

virtual void quitarAgua(float cantidad) {
    if (volumenAgua > 100)
        volumenAgua = volumenAgua - cantidad;
}

virtual void vaciarDeposito() {
    volumenAgua = 0.0;
}
...
}

```

Las clases *Deposito* y *DepositoMunicipal* generarán sus tablas virtuales al ser compiladas:

Índice tabla virtual de la clase Deposito	Operación	La tabla virtual de DepositoMunicipal apunta a la función:
0	ponerAgua	DepositoMunicipal::ponerAgua
1	quitarAgua	DepositoMunicipal::quitarAgua
2	vaciarDeposito	Deposito::vaciarDeposito

Índice tabla virtual de la clase <i>DepositoMunicipal</i>	Operación	La tabla virtual de <i>DepositoMunicipal</i> apunta a la función:
0	ponerAgua	<i>DepositoMunicipal::ponerAgua</i>
1	quitarAgua	<i>DepositoMunicipal::quitarAgua</i>
2	vaciarDeposito	<i>Deposito::vaciarDeposito</i>

Nótese que en este ejemplo el método *vaciarDeposito* no se redefine en la clase *DepositoMunicipal*. Cada objeto de cualquiera de las dos clases incluye un puntero a la tabla virtual adecuada. Ante unas líneas como:

```
Deposito deposito;  
deposito = new DepositoMunicipal(300);  
deposito.ponerAgua(500);
```

el compilador, siempre ojo avizor, generará un código parecido, en cuanto a significado, al siguiente:

```
(deposito→devolverTablaVirtual(deposito))(0, 500);
```

La función *devolverTablaVirtual* usará el puntero virtual contenido en *deposito* para devolver la tabla virtual correspondiente a la clase a la cual pertenece el objeto.

En tiempo de ejecución, la línea anterior se convertirá en

```
deposito→TablaVirtual2[0](500);
```

donde *TablaVirtual2* corresponde a la tabla virtual de la clase *DepositoMunicipal*.

En ensamblador, el compilador introduce un código como éste:

```
(1) ld [%i0], %o1  
(2) ld [%o1], %o1  
(3) call %o1,0  
(4) nop
```

En (1) se carga la tabla virtual y el puntero *vptr*; en (2) se carga la función y *vptr*. En (3) se hace la llamada indirecta, y en (4) se retorna al punto de llamada.

Conviene dejar claro que ***todo lo dicho –tanto en tiempo de compilación como en tiempo de ejecución– acerca del enlace dinámico sucede cuando se tiene una llamada polimórfica a una función virtual***, es decir, una llamada sobre una referencia o puntero del tipo de la clase base. Si se llama a una función virtual directamente:

```
DepositoMunicipal dm;  
dm= new DepositoMunicipal(1500);  
dm.ponerAgua(200);
```

no se necesita emplear tablas virtuales ni punteros virtuales, pues el compilador conoce el tipo del objeto. Basta, pues, con realizar un enlace estático. No obstante, algún compilador podría emplear –innecesariamente– enlaces dinámicos ante situaciones similares.

El uso de tablas virtuales y punteros virtuales hace que las llamadas polimórficas a funciones virtuales resulten menos eficientes que las llamadas convencionales o estáticas (como en C) pues las aplicaciones estáticas se ejecutan sin la sobrecarga en que se incurre con las tablas virtuales.

Para cada clase con alguna función virtual se necesita una tabla virtual, cuyo tamaño es proporcional al número de funciones virtuales declaradas para esa clase. La aproximación de trabajar con tablas virtuales tiene un inconveniente añadido: debe reservarse memoria para las tablas virtuales completas de cualquier subclase de una superclase con funciones virtuales, aunque las subclases no redefinan ningún método o sólo redefinan uno. Cuando se abordan jerarquías de clases muy extensas, la memoria necesaria puede aumentar de forma considerable. Si se tienen muchas clases o un gran número de funciones virtuales en cada clase, las tablas virtuales pueden llegar a ocupar una parte importante del espacio de direcciones.

Por otro lado, el hecho de necesitar un puntero para cada objeto de una clase con alguna función virtual también aumenta la memoria necesaria. Con respecto a la velocidad, para las llamadas polimórficas se necesitan dos indirecciones y una búsqueda en un vector; las llamadas estáticas sólo precisan una indirección.

Con todo, el enlace dinámico mediante funciones virtuales es más eficaz – en cuanto a velocidad– que el de Smalltalk o lenguajes similares, pues se substituye la búsqueda en la jerarquía de clases por una indirección a una tabla de las funciones virtuales asociadas a cada clase.

Esta implementación del polimorfismo mediante tablas virtuales y funciones virtuales, aunque ingeniosa, presenta algunas pegas conceptuales, no relacionadas con la eficacia o el consumo de recursos:

- El programador debe conocer desde el principio si una función en una clase va a ser redefinida en sus clases hijas. Si no anticipa correctamente que alguna subclase puede redefinir en el futuro una función de la superclase, no podrá hacer polimórfica esa función más adelante (las funciones no declaradas como *virtual* no pueden ser redefinidas). Por tanto, los programadores que intenten sobrecargar un método no virtual no podrán. Este “poder de adivinación” del sufrido programador choca frontalmente con la orientación a objetos, pues compromete la reutilización futura de las clases. En [Rumbaugh J. et al., 1991] se formula una precisa crítica de la palabra virtual de C++:

C++ contiene facilidades para la herencia y la resolución de métodos en tiempo de ejecución, pero una estructura de datos de C++ no está automáticamente orientada a objetos. La resolución de métodos y la capacidad de redefinir una operación en una subclase están solamente disponibles si la operación se declara virtual en la superclase. Por lo tanto, la necesidad de redefinir un método debe ser anticipada y escrita en la definición original de la clase. Desafortunadamente, el programador de una clase puede no esperar la necesidad de definir subclases especializadas o puede no saber qué operaciones tendrán que ser redefinidas por una subclase. Esto significa que la superclase debe a menudo ser modificada cuando se define una subclase e implica una seria restricción en la capacidad de reutilizar bibliotecas de clases mediante la creación de subclases, especialmente si el código fuente de la biblioteca no está disponible. (Desde luego, usted puede declarar todas las operaciones como virtual, incurriendo en un ligero coste en memoria y en un exceso de llamadas a funciones)

- ▶ Se entremezcla en el lenguaje la interfaz y la implementación: el polimorfismo es el qué y el uso de *virtual* es el cómo.
- ▶ C++ permite redefinir funciones declaradas como *private virtual*. Por mucho que los manuales de C++ lo consideren normal, no deja de ser un sinsentido usar *private* y *virtual* simultáneamente. O lo uno o lo otro.
- ▶ C++, al igual que Java y muchos otros lenguajes OO similares, padece con estoicismo un mal crónico: *el problema de la superclase frágil* (se verá con detalle en el Apdo. 16).
- ▶ Una valoración personal y entrometida: algo me huele a chamusquina cuando se dejan en manos del programador tareas que debería llevar a cabo el compilador (como comprobar si una función se redefine en alguna subclase). Personalmente, siempre recomiendo a quien quiera aprender programación orientada a objetos con C++ que declare todas las funciones como *virtual*, aunque al principio no acabe de entender el porqué.

Delphi (Object Pascal) usa también funciones virtuales (declaradas con la palabra reservada *virtual*) y tablas de métodos virtuales: el mecanismo de funcionamiento es el mismo que el explicado arriba para C++. Incorpora, por razones de eficiencia, la palabra reservada *dynamic*. Los métodos declarados como *dynamic* se registran en una tabla virtual propia donde sólo se incluyen, para cada clase, los métodos virtuales propios (esto es, los que son realmente redefinidos). Las tablas virtuales de las clases pertenecientes a una misma jerarquía se unen entre sí en forma de lista.

Los métodos *dynamic* se crearon para optimizar el código. Como ya se vio, los métodos declarados *virtual* consumen mucha memoria en el caso de jerarquías muy profundas, pues se necesita un puntero para cada método virtual de las subclases (cuando un método se declara virtual, se mantiene virtual en todas las subclases), aunque no haya sido redefinido, y deben mantenerse también las tablas virtuales para todas las subclases. Esto conlleva una sobrecarga innecesaria de memoria.

El uso de *dynamic* permite usar menos memoria. ¿El precio? Las llamadas a los métodos *dynamic* suelen requerir más tiempo, pues a veces se precisa recorrer toda la lista de métodos.

Delphi necesita, cuando se redefine un método en una subclase, que se declare con la palabra reservada *override*. Así, si se sobrescribe un método inadvertidamente, el compilador dará error.

Desde el punto de vista del programador de Delphi, los métodos virtuales y los dinámicos hacen lo mismo; dependiendo de la situación, unos son más rápidos o consumen menos memoria que otros.

Java cuenta con una implementación del polimorfismo muy distinta de la de C++ o Delphi, más original y mucho más potente y flexible. En Java no existe la palabra reservada *virtual*, pues por defecto todos los métodos son virtuales. Solamente si un método se declara *final* se podrá impedir que pueda ser redefinido en alguna subclase. Java no es un lenguaje de tipos dinámicos, como Smalltalk, sino un lenguaje de tipos estáticos y fuertemente tipado. Durante la compilación, el compilador de Java detecta si se producen asignaciones no válidas o si se llama a métodos de un objeto que no forman parte de su protocolo. Java usa enlace estático –en tiempo de compilación– cuando trabaja con situaciones donde no tiene cabida el polimorfismo y cuando un método se declara *final*, *static* o *private*. En el resto de los casos, Java usa enlace dinámico.

Java emplea *tablas de métodos*, muy similares a las tablas virtuales de C++ o Delphi. ¿Dónde reside, por tanto, la diferencia con C++ o Delphi? Para contestar necesitamos repasar antes una característica indeleble de Java: la independencia de la plataforma. Java, como Smalltalk, no es sólo un lenguaje, más bien es un entorno de programación. Sería más apropiado hablar usar “plataforma Java” o “entorno de programación de Java”, pero suele usarse –supongo que por brevedad– “Java” a secas; en lo que sigue usaré los dos primeros términos cuando quiera destacar la idea del entorno.

El entorno de programación de Java consta, muy simplificada, de dos partes:

- 1) El entorno de programación.
- 2) El entorno de ejecución.

El primero no es más que un compilador de Java, que convierte el código Java en *bytecodes* almacenados en ficheros con la extensión *class*. Estos *bytecodes* no son nativos de ninguna plataforma, pero pueden ser ejecutados en una imaginaria MVJ (Máquina virtual Java), una computadora abstracta definida por la especificación de la MVJ, disponible por parte de Sun Microsystems para todos los desarrolladores. Cualquier implementación de la MVJ acorde con la especificación de Sun será compatible con todas las demás. Cada plataforma donde puede programarse en Java cuenta al menos con una implementación de la MVJ; o, dicho de otro forma, permite que funcione el JRE (*Java Run Environment*). En los chips Java, los *bytecodes* son el código nativo; vendrían a ser máquinas Java implementadas por hardware, petrificadas en silicio.

Los *bytecodes* se refieren a la memoria mediante referencias simbólicas que pasan a ser direcciones de memoria real cuando cargan en una MVJ. En un fichero binario Java, se hace referencia a otras clases e *interfaces*, y a sus campos, métodos y constructores; estas referencias son simbólicas, en ellas se usan los nombres completos de las otras clases e *interfaces*. Para los métodos, una referencia simbólica es un conjunto de información que identifica unívocamente a un método, incluyendo el nombre de la clase, el nombre del método, el número de argumentos que requiere, el tipo de cada uno y el tipo que retorna. La MVJ debe resolver las referencias simbólicas para poder llamar a los métodos adecuados.

De manera general, cuando se carga una clase Java, la MVJ usada almacena las constantes de ésta en la *constant pool* del programa, que reside en memoria. Este *pool* almacena cadenas de texto, nombres de clases y de interfaces, nombres de variables y la **tabla de métodos** de la clase. Ésta última es un *array* de referencias directas, no simbólicas, a los métodos de instancia no privados que pueden invocarse sobre una instancia de la clase, incluyendo los métodos de instancia heredados de las superclases. Más concretamente: las tablas de métodos contienen referencias a **bloques de métodos**. En un bloque de método se almacena la información concerniente al tipo del método y la dirección de memoria donde comienza. Las tablas de métodos, esencialmente equivalentes a las tablas virtuales de C++ y Delphi, se generan en tiempo de ejecución, no en tiempo de compilación, como sucede en dichos lenguajes.

Dentro de cualquier implementación por software de la MVJ se encuentra un motor de ejecución, susceptible de ser implementado de muchas maneras distintas. Un motor de ejecución que funcione como intérprete (originalmente así funcionaban todos los motores) se limitará a interpretar los *bytecodes*. Cada clase no abstracta en un fichero *.class* cargada por una implementación de la MVJ genera una tabla de métodos, que es incluida como parte de la información disponible de la clase. Con ella, la MVJ puede localizar

rápidamente un método no estático invocado sobre un objeto.

Cuando un método admite ser redefinido en las subclases (o dicho de otro modo, cuando no se define como *static*, *private* o *final*), se compila con un *bytecode* etiquetado como *invokevirtual* (otros *bytecodes* relacionados con métodos son *invokespecial*, *invokestatic* e *invokeinterface*). Tras el *bytecode* *invokevirtual* siempre aparece un índice numérico de dos bytes –en la forma *invokevirtual op1, op2*– que la MVJ utiliza para buscar la declaración del método (o firma; *signatura* en inglés) en la *constant pool* de la clase, pues el compilador no puede prever el tipo de un método en ejecución.

Cuando una implementación de la MVJ con un motor del género intérprete ejecuta un programa y encuentra un *bytecode* *invokevirtual*, coloca en la pila los parámetros del método y un puntero al objeto cuyo método se está solicitando. La MVJ usada emplea el puntero sito en la pila para localizar la tabla de métodos del objeto y buscar dentro de ella el bloque del método con la declaración obtenida del índice de dos *bytes*. En cuanto consigue la dirección de memoria donde empieza el método, la implementación de la MVJ comienza a ejecutarlo. Generalmente, este proceso de búsqueda no volverá a repetirse, pues el intérprete substituye los *bytecodes* *invokevirtual* por *bytecodes* *invokevirtualquick*. Tras los *bytecodes* *invokevirtualquick* aparecen unos *bytes* que representan punteros directos a los métodos, es decir, direcciones reales de memoria. Se ahorra tiempo, por tanto, al evitarse las dos búsquedas.

La desventaja más inmediata de este abordaje de la implementación del polimorfismo es la velocidad: Java necesita dos búsquedas para localizar –y empezar a ejecutar– un método. A esto conviene añadir que todos los métodos no declarados *static*, *private* o *final* se consideran siempre polimórficos –a pesar de que no lo sean–; ello conlleva que aumentará innecesariamente el tiempo de ejecución y la memoria usada cuando se llamen dinámicamente a métodos no polimórficos. Por su naturaleza dinámica, Java puede optimizar el código en ejecución (C++ y Delphi no pueden), pero no puede alcanzar en velocidad a los compiladores a código nativo. Por así decirlo, aunque existe enlace dinámico en C++ y Delphi, éste se ejecuta en tiempo de compilación y es, por consiguiente, una simulación del verdadero enlace dinámico. *Java sí cuenta con un genuino enlace dinámico.*

La manera descrita de implementar el enlace dinámico se usa en muchas implementaciones de la MVJ, pero no es la única: existen implementaciones en las que se usa un solo puntero y se manipula la pila de otra manera. La especificación de la MVJ sólo describe qué debe hacer, no cómo hacerlo. Pueden construirse máquinas Java que no usen ni siquiera tablas de métodos; por ejemplo, en los teléfonos móviles casi todas las implementaciones de la MVJ no las usan (no pueden permitirse la memoria que necesitan).

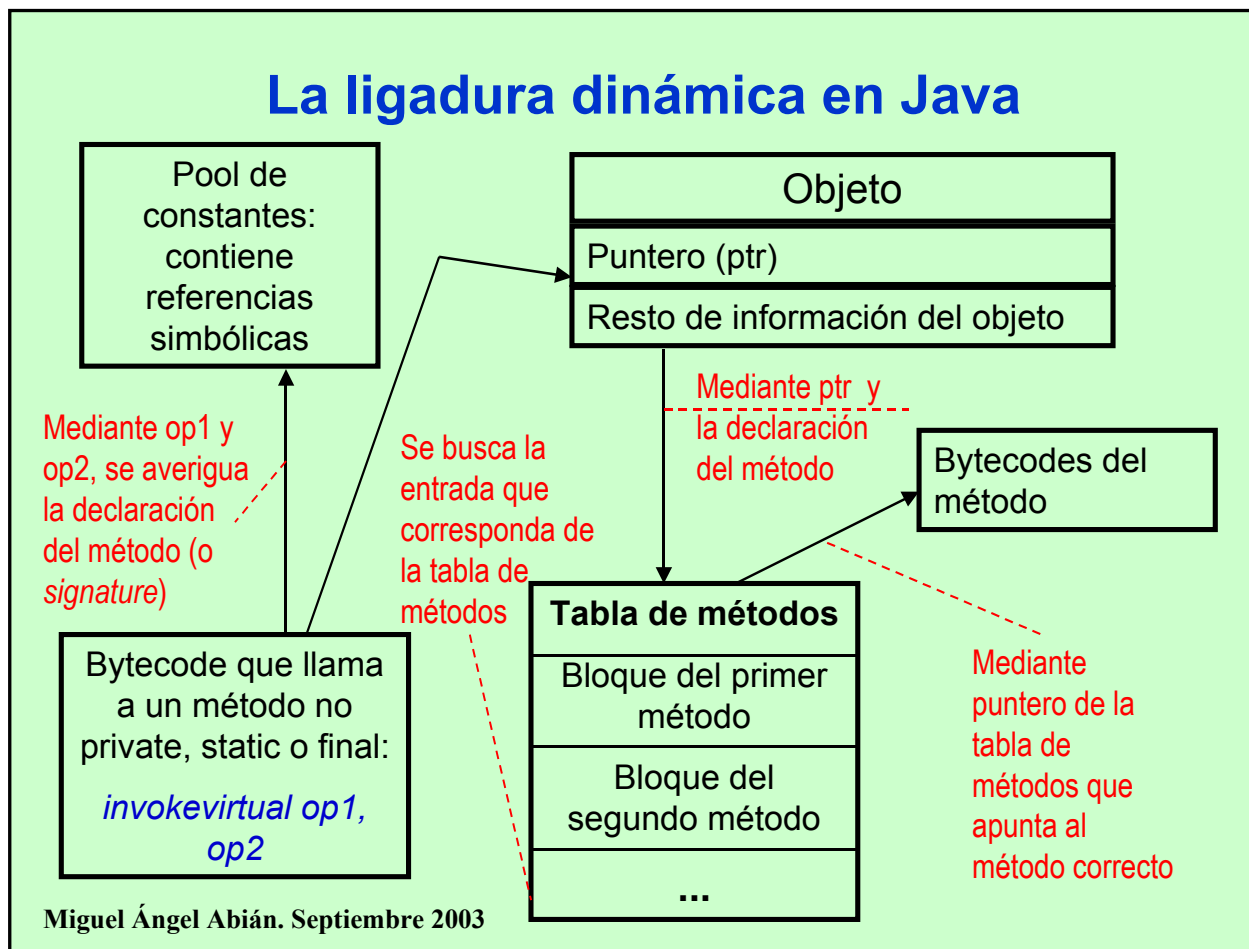


Figura 22. Funcionamiento de la ligadura dinámica en Java (tiempo de ejecución)

La implementación del polimorfismo en Java permite realizar operaciones impensables en lenguajes como C++ o Delphi: permite cargar dinámicamente nuevas clases; importar nuevas clases y tipos, y unirlos a programas ya en ejecución; recompilar *al vuelo* clases que han sido modificadas; y aprovechar mejoras futuras de las implementaciones de la MVJ para código ya compilado en *bytecodes*. Su lentitud en comparación con los lenguajes compilados a código nativo se ve compensada por su versatilidad y seguridad. Antes de que apareciera Java, ningún lenguaje proporcionaba tanta versatilidad y seguridad. Smalltalk o CLOS son los únicos lenguajes comerciales anteriores a Java que ofrecen más flexibilidad que Java, pero fallan estrepitosamente en cuanto a seguridad. En Smalltalk, por ejemplo, uno puede enviar cualquier mensaje a un determinado objeto y solamente en ejecución sabrá si el mensaje forma parte del protocolo del objeto o no; si la respuesta es negativa, el programa lanzará una excepción. Se podría pensar —con cierta ingenuidad, creo yo— que es responsabilidad del programador saber si un mensaje corresponde a un objeto o si un objeto puede convertirse mediante *casting* en uno de otro tipo, pero la realidad nos viene a decir que no somos máquinas y que dar por supuesto cosas en tiempo de ejecución suele conducir a desastres seguros.

Se pueden construir compiladores de Java a código nativo que usen exactamente la técnica de C++ para implementar el polimorfismo. Habrá, eso sí, que tener en cuenta que la acumulación de tablas de métodos en la pila necesitará bastante memoria. Los programas así compilados perderán todas las capacidades dinámicas de la plataforma Java y dependerán de la plataforma, pero ganaran mucho en velocidad en comparación con el uso de un motor intérprete en una MVJ.

Los motores de ejecución no se limitan sólo a interpretar el *bytecode*. Otro tipo de motores de implementación es el de los compiladores JIT (*Just-In-Time* o, en una traducción un poco libre, *al vuelo*). Estos *jitters* traducen en tiempo de ejecución el *bytecode* a código máquina nativo, y lo ejecutan directamente. En estos compiladores, la implementación del polimorfismo cambia bastante con respecto a los motores intérpretes. Cuando el motor de ejecución de una MVJ es un compilador JIT, también se generan tablas de métodos, pero son ligeramente distintas de las ya descritas: en ellas, cada método no *static*, *private* o *final* tiene dos direcciones o entradas en la tabla de métodos, no una. De éstas, una apunta al *bytecode* ejecutable y otra al código nativo. La MVJ usada reemplaza cada dirección del *bytecode* en la tabla de métodos por la dirección del propio compilador JIT. Y cuando la MVJ llama a un método mediante la dirección en la tabla de métodos, se ejecuta el compilador JIT en lugar del método. El compilador compila el *bytecode* en código nativo, y entonces busca la dirección del código nativo de vuelta a la tabla de métodos. Desde ese instante, cada llamada al método se convierte en una llamada a la versión nativa del método. Las dos entradas se tienen para no perder el código de *bytecodes*: los compiladores JIT suelen permitir la ejecución de código nativo y la interpretación de *bytecodes*.

C# es un lenguaje interpretado que usa, para implementar el polimorfismo, tablas virtuales muy similares a las tablas de métodos de Java. La única diferencia significativa entre la ejecución de código escrito en Java o en C# se fundamenta en que C# fue creado pensando en su uso exclusivo con motores de ejecución del tipo compilador JIT. Las primeras versiones de Java incorporaban motores de ejecución de tipo intérprete, aunque hoy en día casi todas las máquinas virtuales comerciales usan compiladores JIT. El *bytecode* de Java y el MSIL (Microsoft Intermediate Language) de la plataforma .Net son lenguajes de tipo ensamblador muy parecidos. Por ejemplo, la suma de dos números en *bytecode* es:

```
iload_1
iload_2
iadd
istore_3
```

y en MSIL:

```
ldloc.1
ldloc.2
add
stloc.3
```

Afirmar que “*Java es interpretado y C# compilado*” está tan fuera de contexto como una cabeza disecada de jabalí en una oficina bancaria o como “*el encuentro casual entre un paraguas y una máquina de coser sobre una mesa de disecciones*”. Técnicamente, esta distinción carece de sentido. Comercialmente... bueno, ésa es otra historia.

Sin embargo, entre Java y C# sí existen diferencias en el modo de declarar los métodos polimórficos por parte del programador. Como vimos ya, en Java todo método que no es *static*, *private* o *final* es automáticamente virtual. En C# existen tres tipos de declaraciones de métodos:

- 1) Estático. Por ejemplo, *static void metodo()*.
- 2) No estático y no virtual. Por ejemplo, *void metodo()*.
- 3) No estático y virtual. Por ejemplo, *virtual void metodo()*.

Los métodos no virtuales se enlazan en tiempo de compilación; los estáticos siempre son no virtuales, como en Java. Pero, por algún extraño motivo, C# admite métodos no virtuales y no estáticos. Como sucede en C++, se pueden marcar los métodos con *virtual*, pero hay que marcar con la palabra reservada *override* aquellos métodos de las subclases que vayan a redefinir el método virtual de la clase madre. Si se desea crear un método para una subclase con la misma declaración que un método de la superclase, pero sin que reemplace al método original, el método de la subclase se marca con *new*.

13. Relaciones.

Aparte de la generalización, vista ya en el Apdo. 11, podemos hablar, en cuanto a relaciones entre clases, de asociaciones, agregaciones y composiciones.

Las **asociaciones** forman el tipo más general de relación posible entre clases y, por añadidura, el de menor contenido semántico. Generalmente, las asociaciones aparecen en el análisis OO del problema en cuestión, y a medida que se avanza en el diseño se sustituyen por otras relaciones más específicas. Durante la implementación, puede simplificarse el código si se consideran que son unidireccionales.

Según [Rumbaugh J. et al., 1991], una asociación que se recorra sólo en un dirección puede implementarse mediante alguna variable que se refiera a un objeto. El número de referencias (variables) necesarias depende de la multiplicidad de la asociación: una multiplicidad de 1 sólo necesita una referencia; una de N requiere un conjunto de referencias. Esta implementación suele ser bastante útil, pero no es la única que existe: una asociación puede implementarse en ambas direcciones, pero esto conduciría a la ruptura del encapsulamiento. Si se alterase una referencia, se necesitaría cambiar también la otra para mantener actualizada la asociación.

Un ejemplo de asociación es la relación “trabaja/es trabajador de” entre un empleado y la empresa para la que trabaja.

Las **agregaciones** son una especialización de las asociaciones, vinculada a una relación de la forma todo-parte. Los objetos *parte* forman parte del objeto *todo*, pero la vinculación entre las partes y el todo no es absoluta: se pueden crear y destruir de modo independiente instancias de cada clase involucrada en la relación.

La relación todo-parte suele reconocerse porque se manifiesta en la forma de “X está compuesto por Y”. Por ejemplo: “los dibujos están formados por elementos gráficos”, “la casa está formada por muros”, etcétera.

Un ejemplo de agregación nos lo da la relación entre un ratón y un ordenador. Un ratón puede quitarse de un ordenador y colocarse en otro ordenador. Si consideramos una clase *Marina* y una clase *Barco*, correspondientes a las entidades del mundo real que todos conocemos, la relación entre ambas es una agregación. Todos los objetos *Barco* de una instancia de *Marina* pueden ser destruidos, y sin embargo la instancia de *Marina* seguirá existiendo. En el mundo real, puede haber marinas sin barcos. Es una situación rara y paradójica, pero se ha dado y puede darse. También es posible añadir nuevas instancias de *Barco* a un objeto *Marina* ya creado. En el mundo real, pueden fabricarse nuevos barcos e incorporarse a una marina, y una marina puede prestar barcos a otra.

Las **composiciones** son también una especialización de las asociaciones, vinculada asimismo a una relación del tipo todo-parte, pero con un vínculo absoluto y permanente. La composición indica que los objetos *parte* están contenidos físicamente en el objeto *todo*. Como consecuencia, los

tiempos de vida de las partes se hallan fuertemente relacionados con el tiempo de vida del todo. Cuando se crea una instancia del objeto *todo*, se crea una instancia de cada uno de sus objetos *parte*; y cuando se destruye la instancia *todo*, se destruyen todas las instancias de los objetos *parte*. Un objeto *parte* no puede asignarse a un objeto *todo* con el cual no se haya creado.

La relación entre un programa y el área de memoria reservada por el sistema operativo es un ejemplo límite de composición. Si tenemos varios programas en ejecución, ninguno puede invadir o utilizar el área de memoria reservada para otro. La ligadura entre programa y área de memoria reservada permanece mientras uno de los dos está activo. Al ejecutarse un programa, se le asigna un área de memoria que permanecerá reservada para él hasta que finalice. Una vez finalizado, el área de memoria reservada será liberada (destruida), y perderá su identidad. A un programa no se le pueden asignar zonas de memoria que estén reservadas para otros programas; y una zona de memoria no puede ser, una vez creada, asignada a otros programas hasta que finalice el programa al cual está asociada.

14. La orientación total a objetos.

Dejar conceptos en el aire guarda cierta similitud con depositar balones en el suelo: alguien acabará pegándoles un puntapié y los enviará lejos, muy lejos. En nuestro caso, el balón de la orientación a objetos aterrizó hace tiempo en el campo de la orientación total a objetos.

La orientación total a objetos (OTO) rompe la separación entre clases y objetos. Su premisa esencial deriva de esta cuestión: ¿para qué tener clases, cuando los objetos ya tienen identidad, comportamiento y estado? La OTO considera, en suma, las clases como objetos.

Hasta cierto punto, el puntapié era lógico. Los lenguajes OO *no puros* se enfrentan a algunas inconsistencias internas. A saber: la creación de los objetos y el uso de variables de clase (*variables de clase* forma parte de la terminología de Smalltalk; véase el Apdo. 17).

Crear objetos resulta engañosamente sencillo: basta con escribir sentencias como *new Estudiante()*. Ahora bien, procediendo así no se está enviando ningún mensaje a un objeto. En un lenguaje completamente orientado a objetos, éstos se crearían mediante el envío de mensajes a otros objetos. Los lenguajes OO no puros consideran las clases como fábricas o plantillas de objetos, y emplean un mecanismo separado para crear objetos mediante las plantillas.

El uso de variables de clase, existentes en C++ (métodos de datos estáticos) y en Java (variables estáticas), viola el principio de encapsulación. Con el uso de variables de clase (compartidas por todas las instancias de la clase), un objeto puede acceder a información localizada fuera de las fronteras de su cápsula.

En los lenguajes totalmente orientados a objetos (como Smalltalk-80, Spoke, Beta, CLOS, Self, etc.), las clases son objetos por derecho propio. Pueden almacenar datos, recibir mensajes, ser instanciadas y ejecutar métodos, como cualquier otro objeto. Beta es el único lenguaje TOO que no permite declarar variables o métodos de clase, lo cual limita su aplicación.

Estrictamente hablando, las clases de Simula, C++, Java, Delphi y Java no se consideran objetos. En ellos, una clase no puede ser almacenada en variables, ser pasada como parámetro de un método... Sin embargo, y tal como veremos en este mismo apartado, los lenguajes no OTO presentan diferencias importantes en cuanto al tratamiento que conceden a las clases.

Aun cuando las clases son objetos en la OTO, son objetos un tanto especiales: pueden crear nuevos objetos (instancias). Tal y como escribieron Adele Goldberg y David Robson en *Smalltalk80: The Language* ([Goldberg A. y Robson D., 1989]):

Hay dos clases de objetos en el sistema [Smalltalk], unos pueden crear instancias de ellos mismos (clases) y otros no.

Las instancias de las clases no pueden crear otras instancias; para hacerlo deben solicitárselo a alguna clase, que lo hará en lugar de ellas.

Se hace necesario pues introducir el concepto de **metaclase**, una clase especial que se instancia para producir una clase. Por lo general, una metaclase sólo cuenta con una instancia (la clase que define), y una clase sólo

es instancia de una única metaclass. Una metaclass es una clase un tanto exclusiva; como una clase es a su vez un objeto con ciertas particularidades, podríamos acordar que una metaclass es un objeto doblemente especial. Hasta en los lenguajes TOO hay privilegios de nacimiento.

La mejor manera de intuir lo que es una metaclass consiste en situarnos en el ámbito de los compiladores. Cualquier clase debe implementarse internamente de alguna manera: en algún lugar deben almacenarse sus métodos, instancias, superclases, etcétera. Toda esta información puede declararse en una clase (la metaclass de la clase). Las metaclasses son muy útiles para los programadores; pues una metaclass permite conocer el conjunto de métodos de una clase, las instancias que han sido creadas, las que han sido destruidas, sus superclases, etc. La reflexión de Java se basa precisamente en el empleo de metaclasses.

¿Acaba con las metaclasses la orientación total a objetos? Ojalá sí, pero: ¿cuál será la clase –descripción o plantilla– de una metaclass?, ¿de qué clase es instancia una metaclass?, ¿y cuál es la clase de la clase de la metaclass?... Prefiero parar aquí la cadena (recursiva) de preguntas, antes de que degeneren en un trabalenguas.

La recursividad puede quebrarse de dos maneras:

- Definiendo que la clase de una metaclass es una metaclass (en Smalltalk, *Metaclass*; esta metaclass se explicará en este mismo apartado) que es instancia de sí misma. Esta solución, absolutamente *deus ex machina*, solventa el problema de la recursividad hasta el infinito. Smalltalk emplea esta táctica.
- Permitiendo que el programador cree o defina sus propias *Metaclass* para resolver el problema de la recursividad. Cuando el programador haya escogido la *Metaclass* deseada, el lenguaje deberá encargarse de gestionar una clase *Class*, metaclass de la clase elegida como *Metaclass* y a partir de la cual pueden generarse metaclasses. CLOS y objVLisp emplean este enfoque.

Los lenguajes OO pueden encuadrarse, dependiendo del tratamiento que dan a objetos, clases y metaclasses, en los siguientes grupos:

- **Grupo 1:** Todos los objetos pueden verse como clases, y viceversa. Sólo existe un tipo de objetos; las metaclasses no son necesarias. Hay muy pocos lenguajes en este grupo; Self es el más conocido.
- **Grupo 2:** Existen clases y objetos, y son cosas distintas. Cualquier objeto es instancia de una clase, pero ni los programas ni los objetos pueden acceder a las clases. Internamente, pueden existir metaclasses, pero no son accesibles para los programas. En este grupo se halla C++.
- **Grupo 3:** Todos los objetos son instancias de clases, y todas las clases son instancias de metaclasses. Las metaclasses son clases, y también instancias de sí mismas; la clase de una metaclass es la propia metaclass. Gracias a esto, los programas y los objetos pueden acceder a las clases. Tenemos pues objetos, clases y metaclasses.

Sólo hay dos tipos de entidades distintas: los objetos y las clases. Java pertenece a este grupo.

- **Grupo 4:** Muy similar al grupo 3, pero aquí la clase de una metaclasses no es la propia metaclasses. Cada clase cuenta con una metaclasses especializada (la *Clase Class* de Smalltalk), y hay una clase que da origen a todas las metaclasses (que a su vez resulta ser instancia de ora clase). Tenemos, en definitiva: objetos, clases, clases de clases (metaclasses especializadas), la metaclasses (***Metaclass*** en Smalltalk) y la clase de la metaclasses (***Metaclass class*** en Smalltalk). Entidades distintas entre sí sólo hay tres: objetos, clases y metaclasses. Las clases de clases manejan mensajes destinados a las clases, como constructores (*new*), y las variables de clase. Smalltalk pertenece a este grupo.

Teniendo presente la anterior clasificación, vamos a comparar entre sí algunos lenguajes (Self, Smalltalk-80, C++, Delphi, Java y C#).

El lenguaje **Self** es el Príncipe de Plata de la Corte de la POO, el Walter Gropius de la escuela de la orientación a objetos, el puro entre los puros. La sencillez y sobriedad de los edificios de la Bauhaus les confería un estilo moderno, distinto del resto. Con Self ocurre exactamente lo mismo: es tan sencillo que se ha convertido en un referente para la investigación en el desarrollo e implementación de lenguajes OO y en la creación de interfaces gráficas y entornos de desarrollo.

Self viene a decir: “Basta, basta a los privilegios de las clases; que no haya objetos de primera y de segunda categoría, que no se distinga entre los humildes objetos y las prepotentes clases: ha llegado el momento de la igualdad”. Para conseguir esto, se elimina el concepto de clase: Self no tiene clases, tiene *prototipos*.

Estos prototipos no son los mismos perros con collares nuevos. Sólo son objetos de los cuales se pueden hacer copias. Si queremos crear un nuevo objeto, basta con encontrar uno ya existente, copiarlo y modificarlo. Las modificaciones afectarán únicamente al nuevo objeto. Self incorpora algunos prototipos primitivos, como *list*, *sequence*, *point*, *byteVector*, *set*, *dictionary*, *process*, *time*, *paint*, etc.

Tal vez alguien se pregunte dónde queda la herencia en un panorama como éste, sin clases. En ningún sitio: Self carece de herencia. Copiando objetos y modificándolos puede conseguirse todo lo que permite la herencia. Self es un ejemplo de lenguaje en el que las relaciones de generalización/especialización pueden implementarse sin herencia.

La relación entre Self y la investigación en interfaces gráficas y entornos de desarrollo deriva de la ausencia de clases. En los lenguajes OO con clases, si queremos crear nuevos géneros de objetos debemos crear nuevas clases o modificar las existentes. Sólo los lenguajes de enlace dinámico puro (como Smalltalk) permiten que cambiemos una clase en tiempo de ejecución y que se mantengan las instancias creadas con la vieja definición. Incluso en ellos, una vez parado y reiniciado el programa, todos los objetos se comportarán de

acuerdo con la nueva definición de la clase. Como sabe cualquier programador, esto enlentece el proceso de construcción y depuración de aplicaciones. Self permite construir aplicaciones *al vuelo*, fundiendo la construcción y la depuración de aplicaciones en una misma amalgama. Los programas pueden crearse probando modificaciones sobre objetos existentes y explorando sus consecuencias.

El lenguaje totalmente orientado a objetos más utilizado es **Smalltalk**. Fue Smalltalk-80 la primera versión del lenguaje donde se empleó el concepto de metaclasses (emplearé *Smalltalk* por *Smalltalk-80* en el resto de este apartado). Su particular manera de llamar a las cosas aún complica más el entendimiento de lo que es una clase. En la sintaxis de Smalltalk no se usa la palabra “metaclass” para designar a las metaclasses, sino la combinación del nombre de la clase a la cual se refiera la metaclass y la palabra “class”. Por ejemplo, la metaclass de una clase llamada *Coche* se declara como *Coche class*.

En Smalltalk, todas las metaclasses son instancias de la clase ***Metaclass*** (empleo otro tipo de letra, cursiva y negrita para diferenciarla). La metaclass *Coche class*, por tanto, es instancia de ***Metaclass***. Ésta, por ser una clase, debe ser instancia de algo. Siguiendo la terminología del lenguaje, la metaclass de ***Metaclass*** es ***Metaclass class***. Como ya se mencionó, Smalltalk rompe la circularidad recursiva al establecer que ***Metaclass class*** es una instancia de ***Metaclass***. Circulo cerrado.

En síntesis:

1. Una clase es un objeto que puede crear instancias.
2. Una metaclass es una clase que define una clase.
3. La única instancia de una metaclass es la clase que define.
4. Todas las metaclasses son instancias de ***Metaclass***.
5. La metaclass de ***Metaclass*** es ***Metaclass class***.
6. ***Metaclass class*** es una metaclass. En consecuencia, también es una instancia de ***Metaclass***.

Una de las principales dificultades de Smalltalk es la notación que usa, que obliga a detallar en todo momento a qué nos referimos. Lastimoso resulta que un lenguaje tan flexible y OO como Smalltalk ofrezca tantas dificultades a los principiantes –y no tan principiantes–, por una mera cuestión de terminología. A veces toda esta terminología de Smalltalk me recuerda a un diálogo marxista (de los hermanos Marx, no del otro).

Nota adicional para los lectores principiantes en Smalltalk: Si el lector se encuentra perdido en la terminología de Smalltalk (lo cual demostraría que es una persona perfectamente normal), le recomiendo que se aferre al significado del prefijo *meta-* como si fuera un salvavidas. *Meta-* es una preposición griega que suele usarse con el sentido de *después, junto a o con* en la formación de palabras castellanas. Metaclase vendría a ser literalmente algo como así como “después de la clase”. ¿Y qué hay después de una clase? Pues la definición de ésta. **Una metaclase es una clase que define o proporciona información sobre una clase.** Por analogía, también podemos hablar de metalenguajes. Un metalenguaje sería un lenguaje en el cual se habla de un objeto lenguaje.

Si es la primera vez que el lector toma contacto con el lenguaje Smalltalk, espero que éste no sea su último contacto. En realidad, el uso de las metaclases sólo es imprescindible para algunas aplicaciones avanzadas. Es perfectamente posible programar en Smalltalk durante mucho tiempo sin tener que encararse con las metaclases. La flexibilidad de Smalltalk no tiene parangón con ningún otro lenguaje comercial. Incluso en algunas versiones comerciales de Smalltalk he podido cambiar la representación interna de los números de coma flotante (a menudo con resultados fatales para el sistema).

Smalltalk es un lenguaje que vale la pena conocer, pues ha influido en el diseño de lenguajes como Eiffel, Java y C#, además de contar con un cierto triunfo comercial. Hasta hace poco, toda la estrategia WebSphere de IBM se basaba en Smalltalk; ahora se basa en Java (Eclipse).

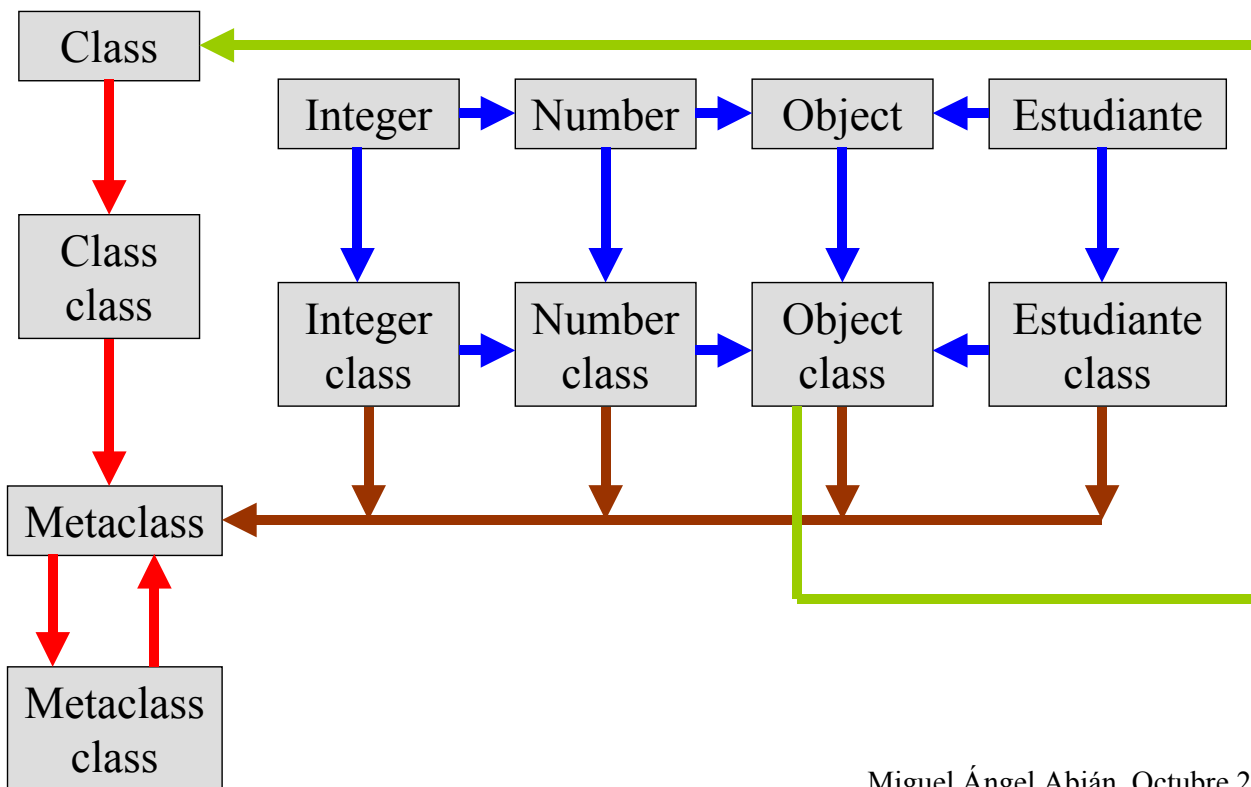
Este lenguaje fue en su momento un habano dentro de la cajetilla de cigarrillos que eran los lenguajes de programación. Se encuadra dentro de la filosofía vital, tan ignorada hoy, de poner la tecnología al servicio de los humanos, y no a los humanos al servicio de la tecnología. Que esta concepción haya quedado anticuada dice poco de nosotros, y mucho de Smalltalk y de su creador (Alan Kay).

He aquí lo que cuenta Steve Jobs, el cofundador de Apple Computer, acerca de su primer encuentro con Smalltalk (1979):

*[...] me mostraron tres cosas. Pero estaba tan enceguecido por la primera, que no alcance a tomar en cuenta las otras dos. Una de las cosas que me mostraron fue la **orientación a objetos**; me la mostraron pero no la tomé en cuenta. La otra cosa que me mostraron fue un **sistema de computadoras en red**... Tenían unas cien computadoras Alto, interconectadas usando correo electrónico, etc., pero no la tomé en cuenta. Estaba tan enceguecido por la primera de las cosas, que era una **interfaz gráfica para el usuario**. Pensé que era lo mejor que había visto en la vida. Ahora recuerdo que tenía varios defectos; estaba incompleta, hacía varias cosas mal. Pero pensábamos que tenía el germen de la idea (aunque no lo sabíamos en aquel momento), y al conocerlo me resultó obvio que algún día todas las computadoras trabajarían así...*

Cada clase de Smalltalk es una subclase de la superclase base *Object*. *Object* no tiene superclase: representa el fin de la jerarquía de clases. Todas las metaclases descienden de una superclase base llamada *Class*.

Jerarquía de clases en Smalltalk-80



Miguel Ángel Abián. Octubre 2003

Figura 23. Esquema simplificado de la jerarquía de clases de Smalltalk-80

Cuando se crea una clase, Smalltalk sigue **internamente** estos pasos:

- 1) Para crear una clase *Estudiante*, se debe crear primero la metaclasses *Estudiante class*. Para ello, se envía un mensaje de creación al objeto **Metaclass**, el cual se implementa en la clase **Metaclass class**.
- 2) Se inicializa el objeto *Estudiante class* enviándole un mensaje que es implementando en la clase **Metaclass**.
- 3) Finalmente, se envía el mensaje **new** al objeto *Estudiante class*, que es implementando en la clase **Behavior** (una de las clases predefinidas en Smalltalk), para crear el objeto *Estudiante*. El término "objeto *Estudiante*" no es una equivocación mía; al fin y al cabo, toda clase es un objeto.

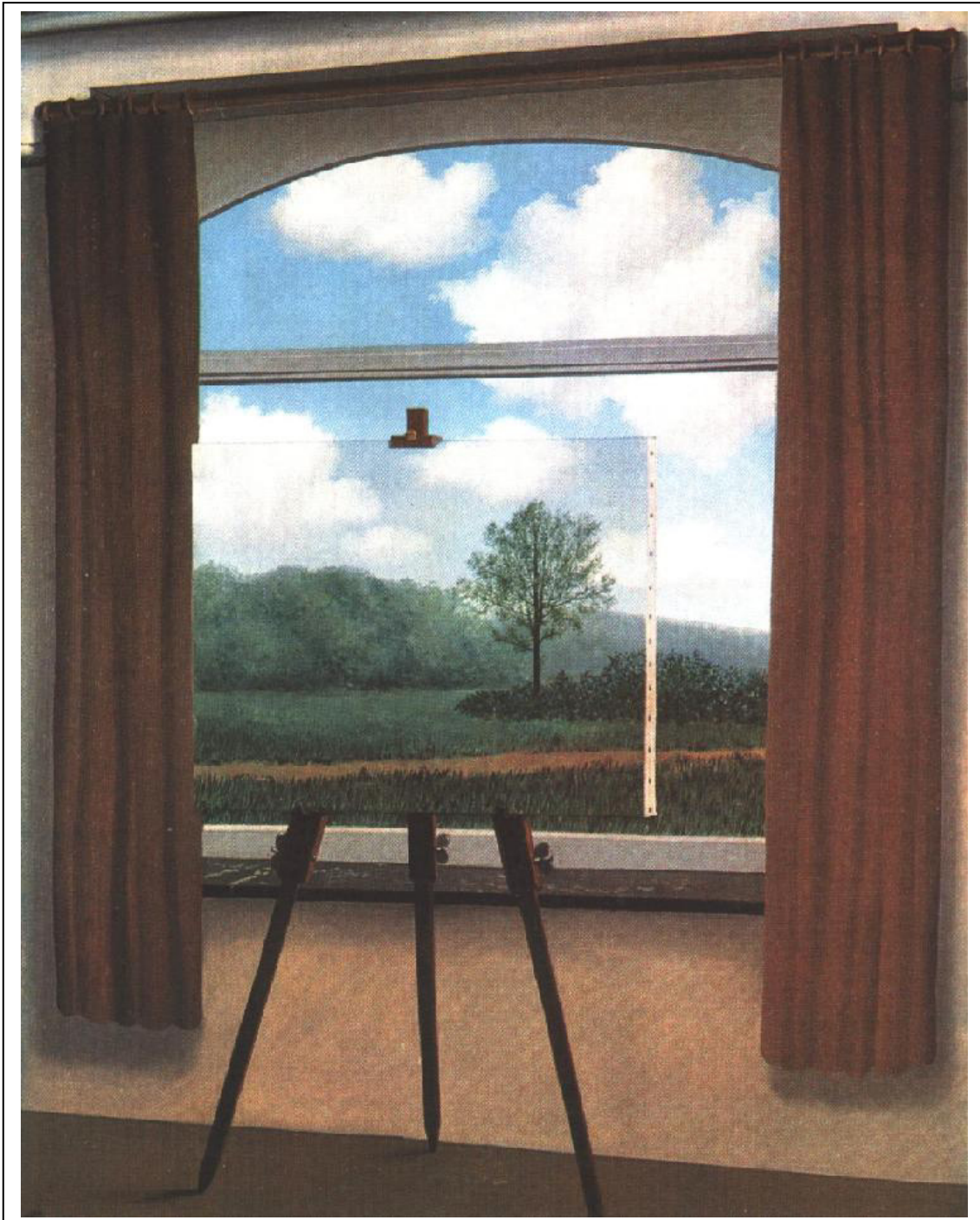


Figura 24-. La condition humaine (1933). Cuadro de René Magritte. Cuando la meta-realidad se confunde con la realidad

Todos los derechos de reproducción de las obras de Magritte están reservados. Se reproduce con fines didácticos y sin ánimo de lucro.

Comparar **C++** con Smalltalk resulta descorazonador. C++ no incorpora mataclases, ni proporciona facilidades para acceder a las clases. Para la orientación total a objetos, C++ es un panorama yermo, baldío, improductivo.

Delphi no cuenta con un concepto claro de metaclasses, pero incorpora funciones de clase, las cuales proporcionan información sobre la clase a la que pertenece un objeto. Por ejemplo,

```
Writeln( "Estudiante", self. className);
```

mostrará la cadena "Estudiante". La definición del método `className` es:

```
class function className: String
```

Java incorpora un metamodelo débil, no tan completo como el de Smalltalk, en el cual las clases pueden responder a mensajes y a peticiones de información de sus variables de clase. Cada clase Java es representada por un objeto-clase instancia de una clase llamada *Class*. La clase *Class* es una metaclasses, y proporciona servicios a los programas; por ejemplo, incluye cierto grado de reflexión, así como la capacidad de crear nuevas instancias de una clase y la de invocar genéricamente a los métodos de un objeto.

Para cada clase cargada por la máquina virtual Java, se crea una instancia de *Class*. Incluye métodos como *getName* (devuelve el nombre de la clase como un *String*), *forName* (carga la clase y la devuelve), *newInstance* (crea una nueva instancia de la clase usando el constructor sin argumentos), *getSuperClass* (devuelve la superclase de la clase), etc. Por ejemplo, *Estudiante.getName()* es un mensaje a la clase *Estudiante*, que devuelve la cadena "Estudiante". Una sentencia como *variable.newInstance()* es un mensaje dirigido a la clase ligada a la variable llamada, y devuelve una nueva instancia de esa clase. *Class* se usa intensivamente en los paquetes *java.lang.reflect* y *java.beans*.

El metamodelo de Java, pese a su sencillez en comparación con el de Smalltalk, proporciona características imprescindibles para sistemas dinámicos, en los cuales pueden añadirse nuevas clases a programas en ejecución. En tiempo de ejecución, por poner un solo ejemplo, se puede determinar la superclase de ejecución de una subclase, e incluso cambiarla por otra, dando lugar a la **herencia dinámica**. Por otro lado, tampoco implica las dificultades conceptuales del modelo de metaclasses de Smalltalk.

C# cuenta con un metamodelo idéntico al de Java, sin mejoras significativas.

La programación totalmente orientada a objetos conlleva ventajas: a) se simplifica conceptualmente la programación (sólo hay objetos); b) no se rompe el esquema objeto-mensaje-objeto cuando se crean objetos, pues enviarse mensajes a una clase (que es un objeto) para crear instancias de ésta; y c) se permite una mayor flexibilidad en cuanto al tratamiento de las clases.

Inevitablemente, la PTOO presenta también desventajas. Por una parte, la separación entre clases y objetos no es una barrera artificial: las clases son las descripciones de las cosas, y los objetos son las cosas mismas. El uso correcto de las metaclasses, por otra parte, precisa cierto tiempo de estudio, aprendizaje

y práctica, y no se encuentra exento de dificultades. En los lenguajes OO más modernos se usan las clases del mismo modo que los tipos: para especificar qué valores se aceptan, lo cual facilita la tarea del compilador. Y finalmente, no parece que el trabajo con lenguajes que permiten la OTO proporcione ventajas intrínsecas con respecto a los lenguajes sólo OO.

15. Esbozo del análisis y diseño orientado a objetos.

En este apartado se presente exponer de manera muy resumida el proceso de análisis y diseño OO. El lector no debe tomar lo que sigue como una explicación completa del análisis y diseño OO. Simplemente son unas directrices generales, un sencillo esbozo, para enfocar un proyecto OO. Han sido extraídas de mis notas para varios seminarios que impartí en el año 2002, y están basadas en mi propia experiencia personal y en el poso que me ha dejado la lectura de libros sobre esta materia. Por su naturaleza, siguen una estructura discursiva, casi oral, en la que se entremezcla la exposición con un ejemplo muy sencillo. He preferido no modificarlas, ni mejorarlas (o empeorarlas, según se mire), pues la construcción de sistemas OO también sigue un proceso discursivo y dinámico. No se proporciona una solución al ejemplo propuesto, pues es un ejemplo típico, que el lector puede encontrar en muchos textos.

Al lector interesado en encontrarse con descripciones más completas y exactas del análisis y diseño orientado le recomiendo estos textos:

- Construcción de software orientado a objetos (2ª Ed.). **Bertrand Meyer, 1999.**
- Designing Object-Oriented software. **Rebecca Wirfs-Brock, Brian Wilkerson y Lauren Wiener, 1990.**
- Simulation Model Design and Execution: Building Digital Worlds. **Paul A. Fishwick, 1995.**
- Object-Oriented programming in Eiffel. **Pete Thomas y Ray Weedon, 1995.**

El último libro, a pesar del título, da una visión del análisis y diseño OO aplicable a cualquier otro lenguaje.

El estudio de la notación de UML y de cualquier metodología concreta de análisis y diseño OO queda fuera del alcance de este artículo. Con todo, el autor no descarta, si las fuerzas le permiten seguir remando en la chalupa de la orientación a objetos, escribir en el futuro algún artículo sobre estos asuntos.

Algunas de las metodologías de análisis y diseño OO más conocidas son

- Object Oriented Design (OOD), de Grady Booch.
- Object Oriented Software Engineering (OOSE), de Ivar Jacobson.
- Object Behaviour Analysis (OBA), de Rubin y Goldberg.
- Better Object Notation (BOM), de Nerson.
- Visual Modeling Technique (VMT), de IBM.
- General Object-Oriented Design (GOOD), de Seidewitz y Stark.
- Object Oriented Design (OOD), de Coad y Yourdon.
- Object Oriented System Analysis (OOSA), de Shlaer y Mellor.
- Object Oriented Structured Design (OOSD), de Wasserman y otros.
- Syntropy, de Cook y otros.
- Desfray, de la empresa Softeam.
- Responsibility Driven Design (RDD), de Wirfs-Brock y otros.
- Object Oriented System Analysis (OSA), de Embley, Kurtz y Woodfield.
- Colbert, de E. Colbert.

- Frame Object Analysis (FOA), de Andleigh y Gretzingr.
- Semantic Object Modeling Aproach (SOMA), de Ian Graham.
- Systems Engineering OO (SEOO), de LBMS.
- Object Oriented Jackson Structured Design (OOJSD), de Jackson.
- Fusion, de Coleman y otros.
- Hierarchical Object Oriented Design (HOOD), de la ESA (Agencia Espacial Europea).
- Object Oriented Analysis (OOA), de Coad y Yourdon.
- Object Modeling Technique (OMT), de Rumbaugh y otros.
- Object Oriented Role Analysis and Modeling (OORAM), de Reenskaug y otros.

15.1 Análisis orientado a objetos.

La primera tarea a la que nos enfrentamos cuando abordamos un proyecto orientado a objetos, sea de software o no, es conseguir una definición del problema. En esta primera fase tan malo es pecar por exceso o por defecto. Una definición del problema extensa, exhaustiva, excesivamente prolífica y completa puede hacer que nos perdamos en detalles secundarios y terciarios, y que no concedamos la importancia necesaria a los factores que sí son cruciales. Imaginémonos que nos hemos perdido en camino secundario y que disponemos de un mapa de carreteras. Lo lógico es ubicarnos en primer lugar con respecto a las carreteras principales; y luego, tomándolas como referencia, buscar nuestra posición entre los caminos secundarios. Por otro lado, una definición del problema demasiado precaria y limitada de poco servirá para nuestros objetivos. En general, se busca una descripción precisa, coherente, completa –sin caer en la exhaustividad– e inteligible. La claridad constituye la piedra sobre la que edificar nuestros proyectos OO.

La tarea del analista no se centra, por supuesto, en determinar los requisitos del nuevo sistema a partir de su clarividencia. Los analistas deben deducir los requisitos del sistema por medio de la información aportada por los clientes. Un analista que no sepa escuchar difícilmente será un buen analista. Si bien las entrevistas y reuniones con los usuarios son clave para llevar a buen puerto un proyecto OO, pueden encontrarse otras fuentes de información:

- La consideración cuidadosa de las salidas (informes, ficheros, gráficas, estadísticas, etc.) que deberá generar el nuevo sistema.
- El estudio detenido de las funciones que deberá proporcionar el sistema.
- La documentación (manuales, informes de fallos y errores, etc.) de los sistemas que han estado en funcionamiento en la empresa u organización a la que va dirigida el nuevo sistema.

Quiero hacer hincapié en la importancia de mantener una relación fluida con los clientes y usuarios. El trato con usuarios expertos puede resultar más valioso (y gratificante) que el estudio minucioso de cientos de folios escritos con la precisión de un arpón atunero.

En la arquitectura contemporánea ha sido usual no tener en cuenta –y, a menudo, despreciar– la opinión de los clientes. No sólo eso: cuanto mayor ha sido la oposición del público a un edificio –la cual ha llegado en algunos casos

a proporciones numantinas—, mayor ha sido el reconocimiento del arquitecto dentro de su camarilla y de la profesión.

En la ingeniería del software, un proyecto que no tenga en cuenta al cliente ni a los usuarios lleva rotulada la palabra “FRÁCASO”. Los usuarios expertos pueden saber cosas que los analistas tardarán semanas en averiguar, o que no averiguarán jamás. Además, todos los usuarios saben muy bien cómo quieren que sea el sistema, a veces incluso tengo la sensación de que nacieron sabiéndolo. Hacer caso omiso de las opiniones de los usuarios y clientes, aunque obliguen a modificar el sistema en una etapa avanzada, es el camino más rápido para hacerse un (mal) nombre dentro de la comunidad de analistas.

Las entrevistas con los usuarios, junto con las otras tres fuentes principales de información, deben permitir a los analistas contestar preguntas como: ¿cuál será el alcance del sistema?, ¿cuáles son sus fronteras?, ¿qué tareas debe desempeñar?

La generación de casos de uso del sistema ayuda a contestar a las preguntas anteriores, además de contribuir a la comunicación entre analistas y usuarios. Si el lector no está familiarizado con los casos de uso, le recomiendo el tutorial “Casos de uso”, de Óscar Canalejo, publicado en **javaHispano**. Curiosamente, los casos de uso no son un método OO, ya existían dentro del análisis estructurado. En sí mismo, esto no plantea ningún problema esencial: casi ninguna teoría o metodología puede explicarse a sí misma en sus propios términos.

Por lo general, la tarea de los analistas concluye con la redacción de una **especificación de requisitos**. Una especificación de requisitos es un documento donde se detallan, en un lenguaje claro y sencillo, los requisitos exigibles al nuevo sistema. Si en el dominio del proyecto se manejan términos poco usuales o de significado distinto al habitual, lo más indicado es confeccionar, dentro de la especificación de requisitos, un diccionario o vocabulario. Así, clientes, programadores, usuarios, diseñadores e implementadores pueden usar los términos del proyecto con el mismo significado.

Usar en la especificación de requisitos frases como “se usará la subrutina LD110 en ensamblador para la transferencia remota de archivos” viene a ser como silbar un acorde de *Sister Ray* en medio de un concierto de música clásica. No resulta recomendable. Primero, porque los usuarios y clientes no suelen entender este tipo de lenguaje; y, al fin y al cabo, son ellos los que pagan. Y segundo, porque se introducen términos y decisiones de diseño en la etapa de análisis.

Consideremos un ejemplo muy sencillo (omito cualquier consideración con respecto al hardware), que será analizado en los siguientes subapartados:

La universidad X desea substituir la matriculación tradicional de los alumnos, de carácter presencial, por la matriculación electrónica mediante Internet. En el nuevo sistema de matriculación electrónica, los alumnos dispondrán de un *login* y un *password*, con los que podrán identificarse en la página web de la universidad. Una vez identificados, podrán elegir las asignaturas en las cuales desean matricularse (suponiendo que queden

plazas libres). También podrán consultar las calificaciones de las asignaturas en las que están o han estado matriculados. Concluida con éxito la matriculación, el sistema les permitirá imprimir un documento justificativo de la matriculación.

El personal administrativo de la universidad podrá introducir alumnos en el sistema, y modificar sus datos.

El sistema almacena los datos personales de todos los alumnos y profesores. Por otro lado, los profesores de la universidad también podrán acceder al sistema (mediante *login* y *password*). Un profesor, una vez identificado ante el sistema, podrá acceder a todas las clases que imparte en el curso académico en vigor y consultar las listas de los alumnos matriculados en ellas. Podrá buscar alumnos. Podrá asimismo asignar calificaciones, modificarlas, consultarlas e imprimirlas.

Un profesor puede impartir asignaturas distintas, e incluso clases distintas de una misma asignatura. El sistema almacenará toda esta información.

Toda asignatura cuenta con un código único (p. ej., MCR: Mecánica Cuántica relativista). Dada una asignatura, le pueden corresponder varias clases en un mismo curso académico, cada una con su código (ejemplo: MCR-A, MCR-B).

El administrador del sistema se encargará de la política de seguridad (asignación de contraseñas, etc.).

Por cierto, no conviene olvidar que la primera misión de un analista es moldear el dominio del problema. Algunos analistas son tan impetuosos que comienzan moldeando directamente la (presunta) solución.

15.2 Diseño orientado a objetos.

Una vez que el analista ha redactado la especificación de requisitos, el diseñador puede comenzar su trabajo. Resulta útil seguir los pasos de los siguientes subapartados.

15.2.1 Identificación de las entidades candidatas a clases. Clases de entidad, de frontera y de control

La forma más cómoda de identificar las clases candidatas para el sistema consiste en escribir (en singular) los sustantivos de la especificación de requisitos. En nuestro ejemplo:

Universidad, matriculación, Internet, sistema de matriculación electrónica, *login*, *password*, página web, asignatura, plaza libre, calificación, documento justificativo de la matriculación, personal administrativo, alumno, datos personales, profesor, clase, curso, lista de alumnos, código único (de asignatura), código (de clase), curso académico, administrador del sistema, política de seguridad.

Algunas de estas entidades darán lugar a **clases de entidad** (clases que contienen información que el sistema necesita conocer permanentemente). Por lo general, las clases candidatas listadas arriba que aparezcan repetidamente en los casos de uso del sistema gozan de bastantes posibilidades de acabar

siendo clases del sistema.

No conviene olvidar que no todas las clases candidatas pueden determinarse directamente a partir del análisis OO. Puede haber clases candidatas encubiertas bajo los nombres. Por ejemplo, de *datos personales* podríamos extraer las clases candidatas *DNI*, *nombre*, *dirección*, *teléfono*.

En el caso de aplicaciones de software, lo normal consiste en incluir clases que representen menús, ventanas, etc. (**clases de interfaz o de frontera**). Normalmente no se consideran los componentes de éstas (botones, listas desplegables, etc.) en el diseño. Para ellos se prefiere usar prototipos en papel o diseñados por ordenador.

En cualquier sistema de software mínimamente realista, también suelen usarse **clases de control**; se encargan de la coordinación entre los objetos procedentes de clases de frontera y los procedentes de clases de entidad. Las clases de control no tienen una contrapartida concreta en el mundo físico. Suele encontrarse una relación unívoca entre los casos de uso y las clases de control.

15.2.2 Identificación de las tareas candidatas a responsabilidades

Para identificar las funciones candidatas a responsabilidades del sistema conviene escribir una lista con los verbos (en infinitivo) de la especificación de requisitos. En nuestro ejemplo:

Identificarse (en el sistema), elegir (asignaturas), consultar (calificaciones), matricularse (en una asignatura), modificar (datos), imprimir (matrícula), introducir (alumnos en el sistema), acceder (al sistema), acceder (a las clases), buscar (alumno), asignar (calificación), modificar (calificación), consultar (calificación), imprimir (calificación) impartir (asignaturas), encargarse (de políticas de seguridad).

Abbot proporciona en *Program design by informal English descriptions* ([**Communications of the ACM**, vol. 26, núm. 11, 1983]) la siguiente equivalencia entre las partes del habla y los componentes de un modelo orientado a objetos:

Parte del habla	Componente del modelo	Ejemplos
Sustantivo propio	Objeto	Juan, Luis
Sustantivo común	Clase	Alumno, Profesor
Verbo de acción	Operación	Matricular, buscar, enviar
Verbo de ser	Herencia	Es un tipo de, es alguno de
Verbo de tener	Agregación	Tiene un , consiste en, incluye
Verbo modal	Restricciones	Debe ser
Verbo transitivo	Método	Entrar
Verbo intransitivo	Método (evento)	Depende de
Adjetivo	Atributo	Azul, 3 años de edad

15.2.3 Depuración de las entidades candidatas a clases y de las tareas candidatas a responsabilidades

En esta etapa se eliminan las clases y responsabilidades candidatas que no quedan dentro del ámbito del sistema. Las clases fuera del alcance del sistema se caracterizan porque sus nombres describen cómo funciona el sistema, pero no hacen referencia a *algo* dentro del sistema. Por ejemplo, *universidad* e *Internet* no necesitan ser modeladas en este proyecto. Por otro lado, *administrador del sistema* y *personal administrativo* corresponden a actores del sistema, externos a él, y no necesitan ser modelados como clases. Algunas clases candidatas pueden quedar aún como dudosas. Por ejemplo, *sistema de matriculación electrónica* puede considerarse o no como una clase explícita.

15.2.4 Eliminación de redundancias

Durante esta etapa se eliminan aquellas clases que son equivalentes (una misma clase tiene más de un nombre).

En nuestro ejemplo, las clases *alumno* y *profesor* equivalen, para el sistema, a *datos personales*; todas las acciones que el sistema realiza sobre los alumnos y los profesores involucran en realidad a sus datos personales. Podemos pues suprimir *datos personales*.

También podemos fundir las operaciones *imprimir (matrícula)* e *imprimir (asignatura)* en una sola operación *imprimir*, con distintas funciones (e implementaciones) según los parámetros que se le pasen.

15.2.5 Separación de los atributos y las clases

A consecuencia de las ambigüedades de cualquier lenguaje natural, casi siempre nos encontraremos con clases candidatas que pueden corresponder a atributos, y no a verdaderas clases. Las clases correspondientes a atributos suelen comportarse como datos: no realizan operaciones y sus objetos no cambian su estado por sí mismos. Empeñarse en conservar estas clases candidatas suele llevar a diseños OO degenerados, en los cuales hay objetos que se limitan a ser entidades pasivas, que actúan como meros contenedores de datos.

15.2.6 Utilización de patrones de diseño

Los patrones de diseño (estructuras de diseño que han sido probadas una y otra vez, y que han resultado ventajosas) constituyen una herramienta que no debe faltar en la caja de herramientas de ningún programador o diseñador.

Tratarlos aquí, aun superficialmente, quedaría fuera del alcance de un artículo de propósito general sobre la OO. Recomiendo al lector la lectura de *Design Patterns: Elements of Reusable Object-Oriented Software* ([Gamma et al., 1995]). En **javaHispano** se han publicado unos interesantes artículos sobre patrones, obra de Alberto Molpeceres, y un práctico tutorial sobre ellos, obra de Martín Pérez. Tanto los artículos como el tutorial son interesantes para el lector interesado en un primer contacto con el mundo de los patrones.

El programador principiante no debe pensar en aplicar patrones a cualquier código que esté escribiendo o a cualquier diseño. Utilizar patrones

por mera devoción o porque están en boca de todos constituye prueba inequívoca de que no se ha entendido el genio de los patrones. La técnica correcta es: “Tengo un problema. ¿Existirá algún patrón que pueda facilitarme el trabajo?”; no: “Tengo una lista de patrones. ¿Cuántos podré aplicar a mi código?”.

15.2.7 Asociación de las operaciones a las clases

En esta etapa se establecen qué operaciones corresponden a cada clase. En nuestro sencillo ejemplo, resulta obvio asociar las operaciones *introducir (alumnos en el sistema)* y *buscar (alumno)* a la clase *lista de alumnos*. Como los datos de los alumnos y profesores contienen información como DNI, teléfono, etc., será lógico asignar a las clases *alumno* y *profesor* métodos como *getTelefono*, *setTelefono*, *getDireccion*, *setDireccion*, etc.

A veces, el diseñador debe enfrentarse a dilemas sobre a qué clase asignar una operación. En cuanto a diseño, tan válido es asignar la operación *introducirAlumno* a la clase *alumno* como a la clase *lista de alumnos*. En el segundo caso, la implementación será más compleja, pues habrá que mantener una referencia adicional a un objeto *lista de alumnos* en cada objeto *alumno*.

15.2.8 Establecimiento de las relaciones entre clases

Para establecer las asociaciones entre clases se usa la regla “es un” (vista en el apartado dedicado a la herencia) y las pautas dadas en el Apdo. 13 para las relaciones de asociación, agregación y composición.

En nuestro ejemplo, las clases *profesor* y *estudiante* pueden heredar de una superclase *persona*.

15.2.9 Revisión del diseño. Miscelánea

Conviene revisar el diseño teniendo presente la siguiente lista de errores habituales en el diseño OO:

- a) Existencia de clases que modifican directamente a otras clases.
- b) Existencia de clases que asumen demasiadas responsabilidades. En este caso, conviene repartir las tareas entre varias clases, aunque para ello haya que introducir nuevas clases (clases de control).
- c) Existencia de clases sin responsabilidades. Una clase puede carecer de responsabilidades porque las responsabilidades que le corresponderían se han repartido entre otras clases o porque actúa como un mero contenedor de datos. En cualquiera de los dos casos, conviene revisar el diseño.
- d) Existencia de clases con responsabilidades que no se emplean en el sistema. Este caso apunta a un análisis deficiente del problema. Jamás debe permitirse que las responsabilidades inadecuadas lleguen a las etapas finales del proceso de diseño.
- e) Asignación de nombres inadecuados o carentes de sentido o precisión a las clases y los métodos. El uso de nombres inadecuados sólo despista a los diseñadores y programadores, y añade complejidad al problema sin aportar ninguna ventaja a cambio. Por ejemplo, usar *PersonaQueEstudiaEnLaUniversidad* en lugar de *Alumno* constituye

un despropósito.

- f) Existencia de clases con responsabilidades desconectadas. Casi siempre, salvo en situaciones triviales, una clase debe utilizar a otras para cumplir sus responsabilidades. La mala asignación de las responsabilidades a un conjunto de clases produce una sensación de incoherencia en el diseño.
- g) Mal uso de la herencia. Conviene intentar usar solamente las herencias de especialización, especificación y extensión. La herencia de implementación está formalmente desaconsejada.
- h) Duplicación de funciones del sistema. Generalmente se debe a una incompleta ejecución del punto 15.2.4.

En el diseño, ya sea estructurado u OO, no hay verdades absolutas: dado un mismo problema, existen infinitud de diseños posibles. Un modelo de clases es correcto y completo cuando:

- ▶ Todas las funciones que debe desempeñar el sistema son proporcionadas por los objetos instancias de las clases elegidas.
- ▶ Las clases están bien encapsuladas, y presentan acoplamiento débil y cohesión fuerte (véase el Apdo. 4.2, en la primera parte).

Cuando se desarrollan proyectos de software OO conviene no perder de vista que los objetos forman o formarán parte de sistemas de software. Los objetos son representaciones de entidades, no las entidades mismas. Por supuesto, los objetos guardan cierta relación con las entidades que describen, pero no son la misma cosa.

En un buen diseño OO, los objetos están vivos, metafóricamente hablando. Realizan acciones, devuelven información. Una lista de alumnos del mundo real no puede contestar a la pregunta “¿Figura en ti el alumno X?”. Es más: los humanos que esperan que los objetos inanimados respondan a sus preguntas suelen provocar suspicacias y recelos entre sus congéneres, y a menudo acaban en instituciones estatales poco confortables. En el mundo animado de los objetos, en cambio, un objeto *lista de alumnos* puede contestar a preguntas (mensajes). Un objeto de software *libro*, por ejemplo, puede decirnos quién es su autor, cuál es su editorial, cuándo fue publicado. Un libro real no puede aportarnos esa información; se precisa que alguien (un actor externo) lo abra y busque esa información por sí mismo. Sin faltar a la verdad, un *libro* en un sistema de software está más cerca de un dibujo animado que de su equivalente real.

Un error frecuente entre los analistas y diseñadores novatos en OO consiste en olvidar que

**LOS OBJETOS SON EL SISTEMA
EL SISTEMA SON LOS OBJETOS**

En un sistema OO no debe haber objetos monstruosos en cuanto a tamaño o responsabilidad. Construir una clase *SistemaMatriculacionElectronica* que realice por sí sola todas las funciones exigidas al sistema es un mal diseño. Los objetos deben cooperar entre sí para conseguir realizar las funciones del sistema.

Cuando se aborda un proyecto de software, no conviene olvidar la universal ***estimación de Parkinson*** (economista inglés, sin relación con el médico del mismo apellido): “*Un proyecto tarda exactamente en realizarse el tiempo que se ha estimado*”. Es decir, fijado el plazo de terminación de un proyecto, el proyecto tardará en ejecutarse como mínimo ese tiempo. Esta regla sirve para cualquier clase de proyecto: de ingeniería civil, de telecomunicaciones, de software, etc. Es una regla derivada de la naturaleza humana.

16. Principios generales del diseño OO.

Existen algunos principios generales para asegurar la calidad de cualquier diseño OO. Entre los más relevantes figuran éstos:

- El principio abierto/cerrado.
- El principio de sustitución de Liskov.
- El principio de inversión de dependencia.
- El principio de segregación de la interfaz.
- El principio de equivalencia reutilización/revisión.
- El principio del cierre común.
- El principio de reutilización común.
- El principio de dependencias acíclicas.
- El principio de las dependencias estables.
- El principio de las abstracciones estables.

Todos estos principios se formulan independientemente de cualquier lenguaje de programación. No obstante, los lenguajes OO suelen facilitar su implementación. Con todo, hay lenguajes OO con más facilidades que otros para implementar en código estos principios.

16.1 El principio abierto/cerrado.

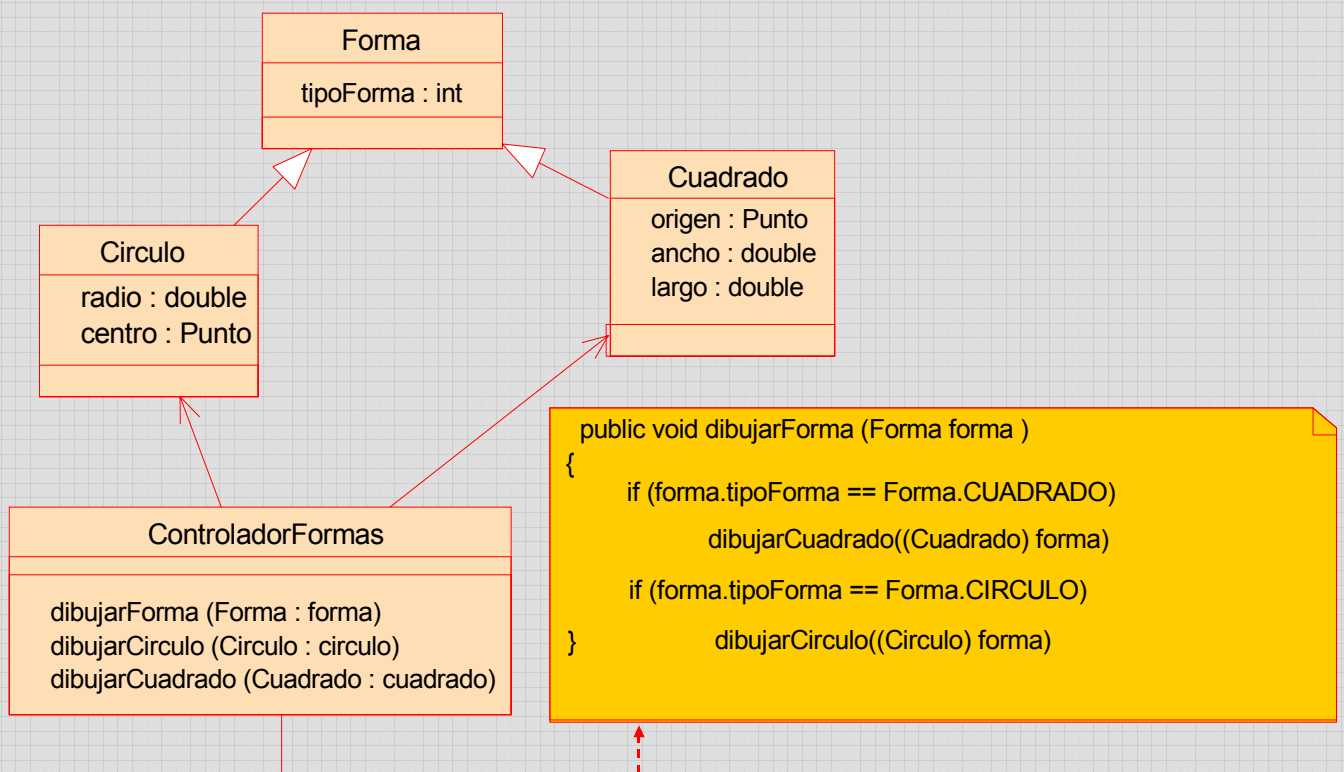
El principio abierto/cerrado fue formulado explícitamente por Bertrand Meyer:

Las entidades de software (clases, módulos, etc.) deberían estar abiertas para extensión, pero cerradas para modificación.

Este principio, ya mencionado en el apartado dedicado a la herencia, es con diferencia la mejor guía para el diseñador, sea neófito o profesional. Nos dice que debemos buscar diseños donde se pueda cambiar lo que hacen los módulos sin cambiar su código fuente. ¿Cómo hacerlo? Por los apartados anteriores ya sabemos la respuesta: mediante abstracción, herencia y polimorfismo.

Un ejemplo muy sencillo se muestra en las figuras 25 y 26.

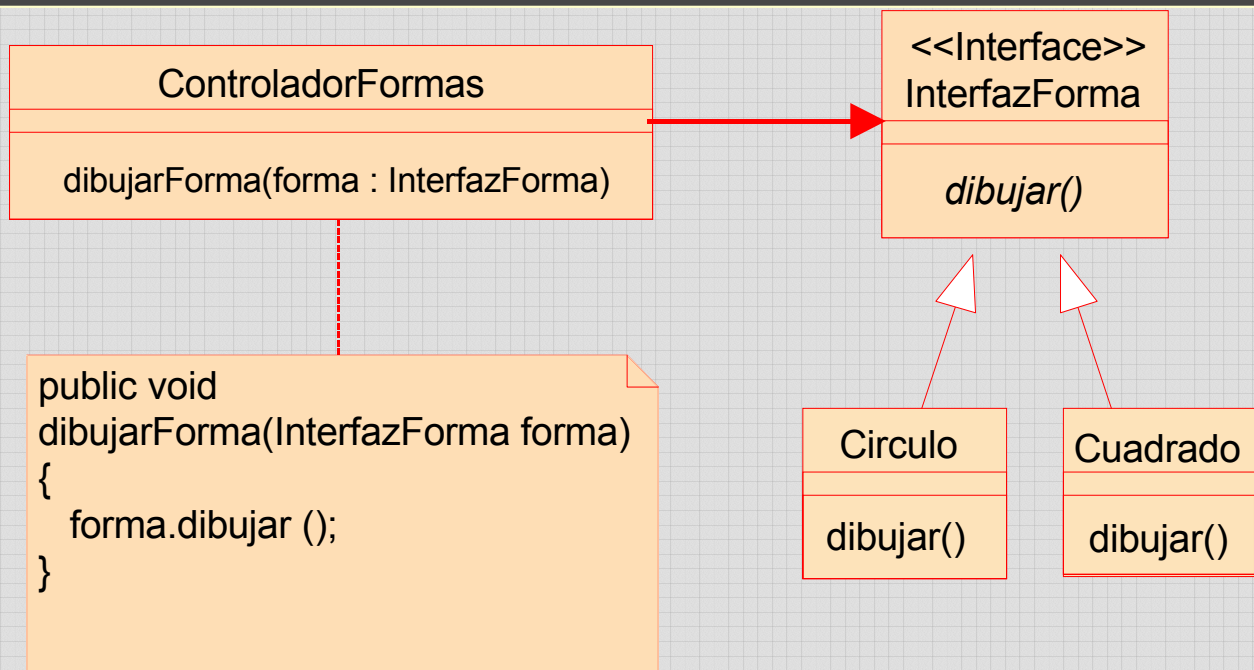
Ejemplo de incumplimiento del Principio Abierto-Cerrado



Miguel Ángel Abián, Octubre 2003

Figura 25. Ejemplo de incumplimiento del principio abierto/cerrado

Ejemplo de cumplimiento del Principio Abierto-Cerrado



Miguel Ángel Abián, Octubre 2003

Figura 26. El diseño de la figura 25 tras haber aplicado el principio abierto/cerrado

Si se quisiera introducir nuevas figuras en el esquema de la figura 25, como clases, paralelogramos, etc., debería modificarse el método *dibujarForma*. Por otro lado, las sentencias *if* o *switch/case* no ubicadas en constructores suelen emanar un olor a podrido hasta que el sano viento del polimorfismo limpia la atmósfera, dejando a su paso un aséptico olor a ozono.

Cada sentencia *if* mantiene explícitamente una dependencia del diseño hacia lo que pueda deparar el futuro: cualquier nueva forma que queramos dibujar necesitará modificar el módulo *ControladorFormas*. Con la solución mostrada en la figura 26, pueden añadirse nuevas formas sin cambiar el código que ya funcionaba; basta con añadir código nuevo.

16.2 El principio de substitución de Liskov.

Este principio reza así:

Las clases derivadas deben poder usarse mediante la interfaz de las clases base sin necesidad de que el usuario conozca la diferencia.

Como este principio se detalló con detenimiento en el Apdo. 11, no se volverá aquí sobre él. Sí me parece interesante mencionar que el principio de substitución es una consecuencia del principio abierto/cerrado. Procedamos por reducción al absurdo: supongamos que el principio de substitución es falso. Es decir, supongamos que un subtipo *S* de un tipo *T* no es sustituible como argumento de un método *M* cualquiera. Ello implica que *M* debería realizar algún tipo de prueba para determinar cuál de los subtipos está usando, y conocer así cómo tratarlo. Por lo tanto, se violaría el principio abierto/cerrado. Q.e.d.

16.3 El principio de inversión de dependencia.

Este principio fue enunciado por Robert C. Martin en *Designing Object-Oriented C++ Applications using the Booch Method* ([Martin R., 1994]):

- a) Los módulos de alto nivel no deben depender de los módulos de bajo nivel. Ambos deben depender de las abstracciones.
- b) Las abstracciones no deben depender de los detalles. Los detalles deben depender de las abstracciones.

La interdependencia de los módulos dentro de un diseño genera software con las siguientes deficiencias:

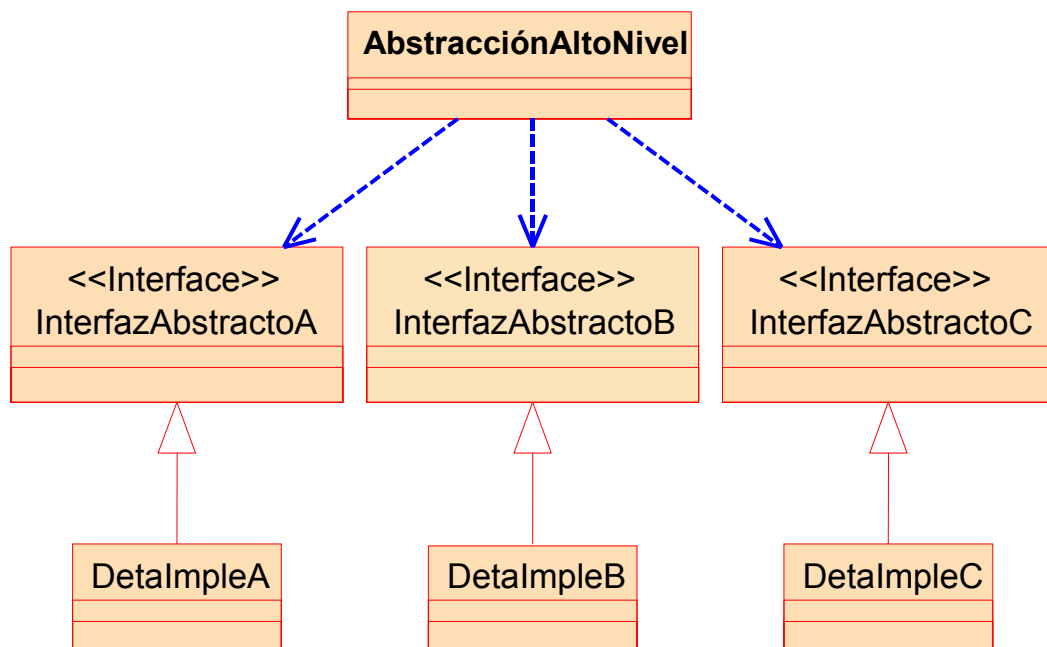
- **Rigidez.** El código se hace difícil de cambiar, pues cada modificación implica cambios en otras partes del sistema de software.
- **Fragilidad.** Cuando se realiza una modificación, pueden dejar de funcionar partes del sistema que ya habían sido probadas y consideradas correctas.
- **Inmovilidad.** Se dificulta la reutilización, a causa de que el diseño está atado al sistema.
- **Viscosidad.** Deriva de la tendencia humana, demasiado humana, a realizar apaños en el software en lugar de cambios que preserven los propósitos del diseño original. Difícil es hacer las cosas bien; fácil hacerlas mal. Tal y como dice Al Pacino en la película *Esencia de mujer*: “*en mi vida siempre tuve claro qué camino era el correcto, pero siempre elegí el equivocado*”.

El motivo de esta tendencia no hace falta buscarlo en ningún sesudo estudio acerca de la psicología humana: los cambios que preservan las intenciones del diseño original cuestan más trabajo y reflexión que los otros. Y siempre están las prisas... En general, resulta más difícil hacer las cosas que deben hacerse que las cosas que no deben hacerse (aunque ninguna tarea cuesta tanto como aquella que

no queremos hacer). Y siempre están las prisas...

Este principio propone una estrategia para evitar la rigidez, fragilidad, inmovilidad y viscosidad del software. Para ello, propone depender de clases y métodos abstractos, y de interfaces; en lugar de depender de clases y funciones concretas. En general, y no sólo en ingeniería del software, las cosas concretas varían mucho, mientras que las cosas abstractas son relativamente inmutables.

Ejemplo del Principio de Inversión de Dependencia



Miguel Ángel Abián, Octubre 2003

Figura 27. Ejemplo del principio de inversión de dependencia

16.4 El principio de separación de la interfaz.

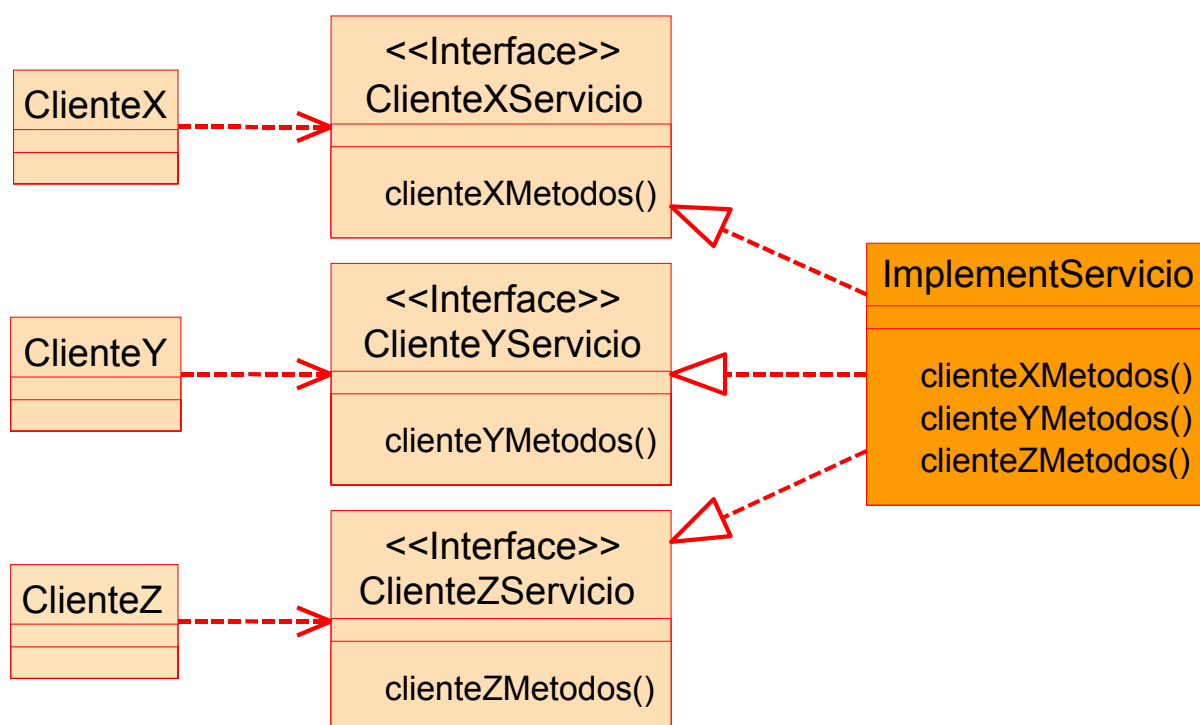
Este principio ([Martin R., 1994]) dicta que:

Los clientes no deban ser forzados a depender de interfaces que no utilizan.

Tal y como se ejemplifica en la figura 28, el principio de separación de la interfaz propugna que resulta preferible tener muchas interfaces específicas para los clientes que una sola interfaz de propósito general. Es decir: si

contamos con una clase que proporciona servicios a varios clientes, resulta más adecuado crear interfaces específicas para cada variedad de cliente, en lugar de sobrecargar la clase con todos los métodos disponibles para los clientes.

Ejemplo del Principio de Segregación de Interfaces



Miguel Ángel Abián, Octubre 2003

Figura 28. Diseño conforme al principio de separación de la interfaz

16.5 El principio de equivalencia reutilización/revisión.

Este principio ([Martin R., 1994]) dicta que:

La granularidad de la reutilización es la granularidad de la revisión. Sólo aquellos componentes que son revisados mediante un sistema de distribución pueden ser reutilizados de forma efectiva. Este grano es el paquete.

El principio de equivalencia reutilización/revisión afirma que un elemento reutilizable (ya sea una clase, un conjunto de clases o un componente) no puede ser reutilizado a menos que disponga de un sistema de control de las versiones (número de versión, conservación de las versiones anteriores

durante un tiempo, etc.). En Java, por ejemplo, la unidad de revisión o componente de revisión es el *package* (paquete); y también es la unidad de reutilización.

Este principio parece trivial, pero esto sólo es una ilusión óptica. Sin él, cualquier diseñador de software se sentiría tentado a decir que una clase es reutilizable. Y no es así: la reutilización de clases aisladas carece de sentido. Una clase reutilizable bien construida colabora con más clases para proporcionar servicios a los clientes. Por otro lado, a los clientes no nos basta con que se nos dé una clase y se nos diga que es reutilizable. Si cambiamos las clases que usamos por otras más actualizadas, queremos que se nos mantenga informados de las versiones con las que trabajamos, y que se nos proporcione documentación de las modificaciones. A veces, incluso queremos seguir trabajando con versiones anteriores; y necesitamos, por tanto, información acerca de la compatibilidad con las nuevas versiones.

16.6 El principio de cierre común.

Según este principio ([Martin R., 1994]):

Las clases dentro de un componente de revisión deben compartir un cierre común. Es decir, si una clase necesita ser modificada, todas ellas modifican ser modificadas. Lo que afecta a una, afecta a todas.

Esta versión orientada a objetos del lema de *Los tres mosqueteros* nos propone que, dado un particular tipo de cambio, todas las clases dentro de un componente deben estar cerradas a él, o estar todas abiertas a él. Así se consigue no propagar los cambios a través del sistema. Dentro de un componente, no se permite que algunas clases puedan modificarse sin que se precise modificar otras.

16.7 El principio de reutilización común

Este principio se vincula con el anterior ([Martin R., 1994]):

Las clases dentro de un componente de revisión deben ser reutilizadas juntas. Esto es, es imposible separar los componentes unos de otros para reutilizar menos que el total.

El principio de reutilización común insiste en la importancia de que las dependencias asociadas con la reutilización sean encapsuladas, si resulta posible, en un solo componente. ¿Por qué? Pues porque la dependencia de un componente (como un *package* en Java) es dependencia de todo aquello dentro del componente. Cuando se modifica un componente, y por tanto su número de versión, todos los clientes del componente deben verificar que trabajan con el nuevo componente, aunque nada de lo que ellos usaban dentro del componente haya cambiado. Por consiguiente, las clases que no son reutilizadas de modo conjunto no deberían agruparse en un componente.

Este principio también fomenta, un tanto forzadamente, la reutilización entre los clientes. Si un cliente se ve obligado, cuando reutiliza una determinada clase, a reutilizar otras clases incluidas en el mismo componente, hará un uso más extendido de la reutilización. Desde luego, esta situación no tiene porque ser perjudicial para el usuario. Al contrario: forzar a que reutilice todas las clases de los nuevos componentes permitirá que siempre se encuentre con clases más actualizadas y eficientes.

16.8 El principio de dependencia acíclica

Así se define el principio de dependencia acíclica ([Martin R., 1994]):

La estructura de dependencia entre los componentes debe ser un grafo dirigido acíclico (DAG en inglés: *directed acyclic graph*). Esto es, no debe haber ciclos en la estructura de dependencia.

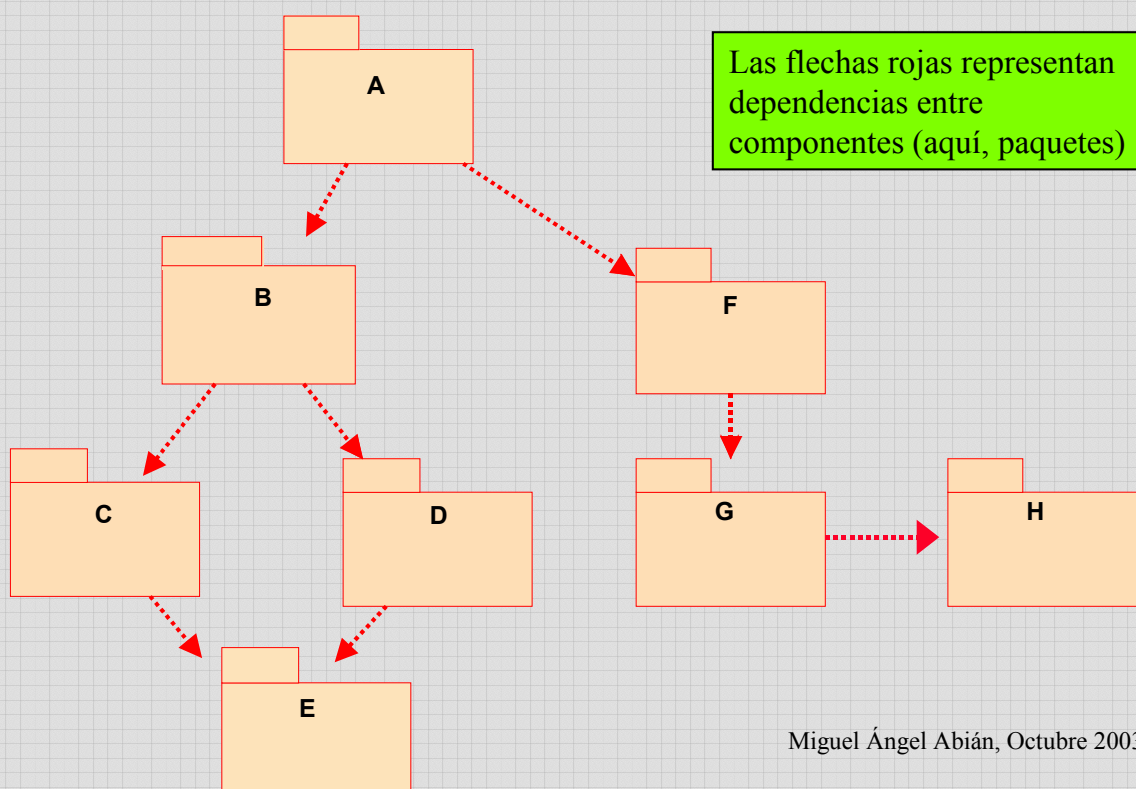
Las dependencias cíclicas constituyen un importante peligro para el diseño de software, orientado a objetos o no. Todos los componentes dentro de un ciclo deben actualizarse e incorporarse simultáneamente. Dos imágenes bastarán para aclararnos la situación. Consideremos la figura 29. El grafo de paquetes (componentes) presenta un ciclo o bucle cerrado. La situación resulta un tanto compleja: si se quiere hacer pública una nueva versión del paquete D se debe construir usando las últimas revisiones de los paquetes B, C y E, con el trabajo de depuración y pruebas que conlleva este proceso.

Si no se obra así, el sistema puede resultar tremendamente frágil e inestable. Supongamos, por ejemplo, que un desarrollador que trabaja con el paquete B escoge trabajar con alguna versión anterior de C y D (un motivo puede ser que las versiones anteriores de C y D están mejor probadas que las actuales). Las versiones anteriores de C y D dependerán a su vez de alguna versión anterior de E. Aparentemente, cuando se quiera difundir una nueva versión de B, bastará con probar el paquete con las versiones adecuadas de los paquetes dependientes y difundirlo. Entonces comenzarán los problemas.

Si E llama a un método del paquete A, B podría no funcionar o funcionar incorrectamente, pues depende de versiones anteriores de C y D, las cuales dependían de una versión anterior de E, no de la actual. Consecuencia: un *halt* 22 perfecto, el equivalente informático a una persona que intentara elevarse tirando hacia arriba de sus pies.

La moraleja salta a la vista: Todos los componentes de un ciclo deben difundirse al mismo tiempo. En sistemas muy grandes, se torna muy complejo realizar ese proceso; en ese caso, es preferible romper el ciclo, como se muestra en la figura 30. Un ciclo puede romperse mediante la adición de un componente intermedio o de nuevos *interfaces*.

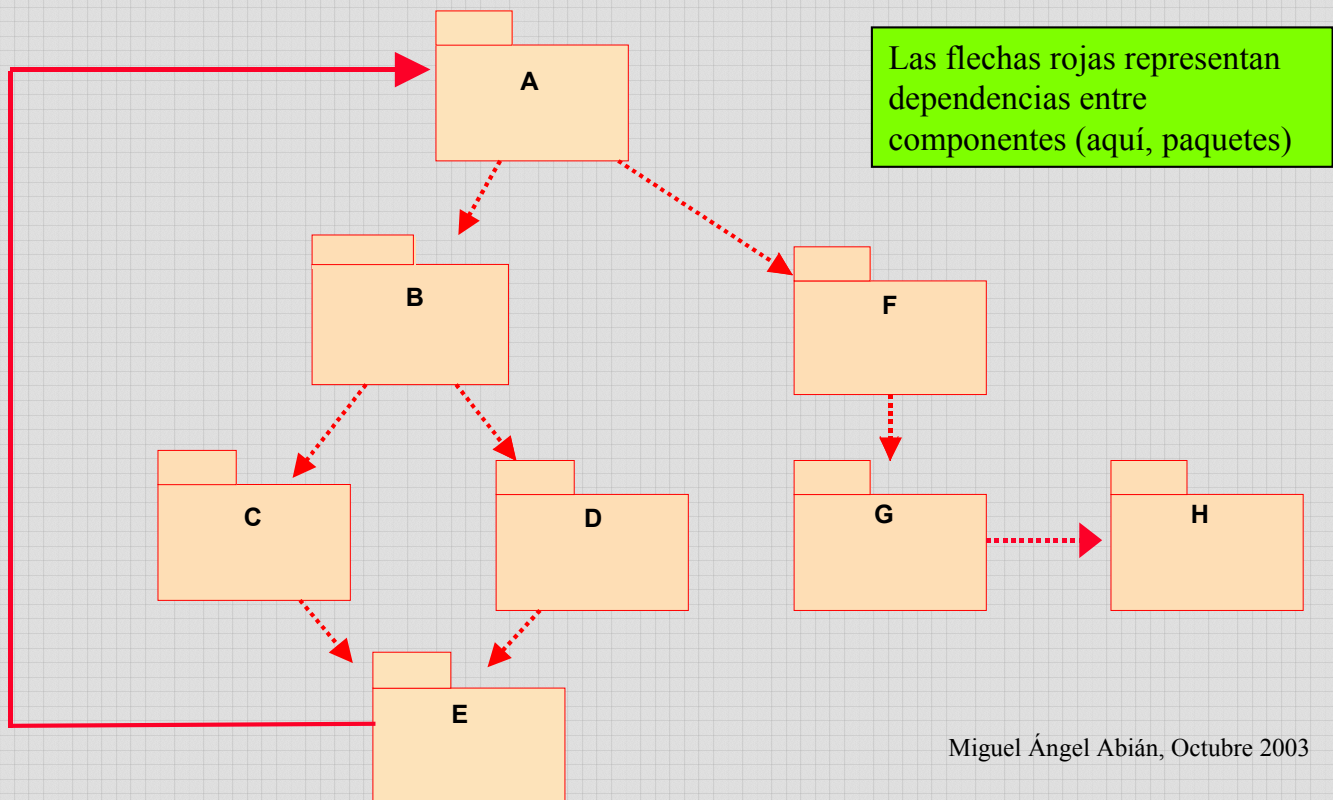
Ejemplo de incumplimiento del principio de dependencias acíclicas



Miguel Ángel Abián, Octubre 2003

Figura 29. Diseño que incumple el principio de dependencias acíclicas

Ejemplo de cumplimiento del principio de dependencias acíclicas



Miguel Ángel Abián, Octubre 2003

Figura 30. El diseño de la figura 29 tras romperse el ciclo

Con la solución de la figura 30, si se desea lanzar una nueva versión de D, bastará con asegurarse de que se dispone de la versión correspondiente de E.

16.9 El principio de las dependencias estables

Según [Martin R., 1994]):

Las dependencias entre componentes en un diseño deben correr en la dirección de la estabilidad de los componentes. El componente del que depende otro componente debe ser más estable que él mismo.

Una medida del grado de inestabilidad de un componente nos la proporciona la siguiente fórmula:

$$I = C_e / (C_a + C_e)$$

C_a es el número de clases fuera del componente que dependen de clases dentro del componente. C_e es el número de clases dentro del componente que dependen de clases externas al componente. I mide la inestabilidad del componente. $I = 0$ implica un componente completamente estable, e $I = 1$ implica un componente completamente inestable.

16.10 El principio de las abstracciones estables.

Según [Martin R., 1994]:

La categoría de clases más estable es aquella que más consiste en clases abstractas. Una categoría completamente estable debería consistir sólo en clases abstractas.

Una medida del grado de abstracción de un componente o paquete nos la proporciona la sencilla fórmula:

$$A = \text{Número de clases abstractas} / \text{Número total de clases}$$

A siempre se mantiene dentro del rango $[0,1]$. Cuanto más próximo está a la unidad, mayor es su estabilidad.

17. Breve panorámica de la historia y evolución de los lenguajes OO. Clasificaciones.

Aunque son pocos los colores en la paleta de la orientación a objetos (herencia, polimorfismo, etc.), sus mezclas han proporcionado una gran gama cromática: hay más de 150 lenguajes OO o con características de OO, y la lista continúa creciendo.

Podría proporcionar una lista de ellos (incompleta, desde luego); pero eso sería bastante inútil para el lector. Muchos de ellos han sido reemplazados por lenguajes más modernos; otros salieron de los laboratorios de software, dieron un par de vueltas al edificio y volvieron a entrar; algunos otros fueron desechados por las empresas que los fabricaron... He preferido centrarme sólo en lenguajes que hayan tenido cierta importancia histórica; que hayan representado etapas de la POO; que gocen de cierto éxito en la actualidad o, simplemente, que sean lo bastante originales (aunque quizá poco prácticos) para merecer un poco de atención.

Self, por ejemplo, es un lenguaje que borra muchos de los conceptos de la OO como un pintor borra una acuarela con una esponja. Sin embargo, es casi imposible que consiga éxito comercial.

17.1. Breve panorámica de la historia y evolución de los lenguajes OO.

La historia de los lenguajes de programación orientados a objetos comenzó con los lenguajes **SIMULA I** (*SIMple Universal Language*) y **Simula 67** (de ahora en adelante, Simula a secas). Fueron diseñados e implementados por Ole-Johan Dahl y Kristen Nygaard, en el *Norwegian Computing Centre* (Oslo). SIMULA I (1962-65) fue concebido como un lenguaje de programación destinado a aplicaciones de simulación de procesos. Simula 67 lo reemplazó en 1967. La nueva versión era un lenguaje de propósito general, pero seguía siendo útil para simular procesos (incorporaba una clase de sistema llamada *Simulation*). En comparación con los lenguajes OO actuales resulta tosco en cuanto a sintaxis, pero conceptualmente ha influido en el diseño de todos los lenguajes OO actuales. Simula no fue el típico invento pionero pronto olvidado; su influencia se extiende hasta Java y C#. Las primeras versiones Smalltalk se escribieron en Simula; y tanto Objective-C como C++ han llevado los conceptos fundamentales de Simula a C. Su uso actual es anecdótico; pero algunos profesores lo usan para enseñar a sus alumnos que lo que hacen con C++, Java o C# se podía hacer ya en los años setenta. Sin lugar a dudas, los dos profesores noruegos pueden estar contentos con su creación: pocas han resultado tan prolíficas e influyentes.

Con todo, la aparición del primer lenguaje OO no dio lugar a ninguna avalancha de nuevos lenguajes de programación ni despertó el interés de los diseñadores de software, inmersos como estaban en el paradigma estructurado (Pascal, C, etc.).

A finales de los años setenta se comenzó a desarrollar el primer lenguaje orientado a objetos puro: **Smalltalk**. En su diseño e implementación trabajaron

Alan Kay, Adele Goldberg, Ted Kaehler, Daniel Ingalls y otros, en el *Xerox Palo Alto Research Center* (PARC). La primera versión completa y estable se lanzó como Smalltalk-72.

Incluso hoy resulta difícil ponderar el impacto que tuvo Smalltalk. Demostró que un lenguaje verdaderamente orientado a objetos era posible. Este lenguaje rebosaba del vaso de los lenguajes experimentales y académicos: ofrecía un completo entorno gráfico de desarrollo (véase la opinión de Steve Jobs en el Apdo. 14) y se encontraba fuertemente ligado al sistema operativo. Con el entorno de desarrollo de Smalltalk se podían desarrollar programas rápidamente, de una manera inimaginable en la época. Por otro lado, fue el primer lenguaje de tipos dinámicos. Casi toda la teoría de la OO se desarrolló sin perder de vista a este lenguaje. Smalltalk no se compila a código máquina nativo; se compila a un lenguaje intermedio, independiente de la plataforma, que luego debe ser interpretado por una máquina virtual.

Su principal obstáculo inicial fue la velocidad: al tratarse de un lenguaje interpretado, los primeros intérpretes hacían la ejecución lenta en comparación con C o Pascal. A mediados de los años ochenta, se introdujo la compilación del código intermedio a código máquina nativo, en tiempo de ejecución. La idea fue posteriormente adoptada por Sun Microsystems (Java) y Microsoft (Plataforma .Net). Actualmente se conoce a esta compilación como compilación JIT, *Just In Time*). Este lenguaje continúa usándose, tanto en la industria como en la enseñanza, y dispone de varias versiones comerciales. Durante muchos años, fue una de las herramientas internas de desarrollo de IBM, pero acabó siendo substituido por Java.

Modula-2 fue inventado en 1978 (aproximadamente) por Niklaus Wirth (creador de Pascal). Ofrece implementaciones completas de TADs, pero no permite una característica tan importante como la herencia. Modula-3 sí incorpora herencia.

Objective-C es una extensión de C creada por Brad Cox a principios de los años 80. Es un lenguaje de tipos dinámicos, pese a ser compilado, y se usa sobre todo en entornos Mac. Permite al programador identificar la clase de cada objeto, de forma opcional. En este caso, se emplea un *tipado* fuerte. En su momento incorporó la novedad de que todas las bibliotecas se ligaban dinámicamente (algo habitual hoy), lo que implicaba programas de tamaño pequeño. Está a medio camino entre Smalltalk y C; posee gran parte de la flexibilidad de Smalltalk, y gran parte de la potencia de C (del cual toma la sintaxis).

Sus principales defectos son la ausencia de un recolector de basura, de un mecanismo similar a los *namespaces* de C++, y de la sobrecarga de operadores.

En 1980, aparecieron las primeras versiones de **C++** (entonces se conocía como C con clases; un nombre revelador, ¿verdad?). Fueron desarrolladas por Bjarne Stroustrup en los laboratorios de AT&T. C++ es una extensión del lenguaje C; aunque no es la única (existe Objective-C), es con diferencia la más usada. Con respecto a C, permite un tipado más estricto, permite definir jerarquías de clases, y admite sobrecarga de operadores y funciones. Como ya se vio en el Apdo. 14, no incluye el concepto de metaclasses, y tampoco permite que los programas accedan a las clases. La encapsulación de C++ se puede saltar fácilmente mediante punteros. Pese a ser uno de los lenguajes más empleados en la actualidad, en parte por haber atraído a muchos programadores de C, se enfrenta a los problemas derivados de nadar entre las aguas de la programación estructurada y de la POO. La gran preocupación de los compiladores de C++ es la velocidad y la eficacia, no la seguridad. No incorporan pues recolectores de basura ni gestores automáticos de la memoria. C++ ha gozado de un proceso de normalización y estandarización que ha incorporado al lenguaje muchas bibliotecas imprescindibles para los programadores.

Eiffel, desarrollado por Bertrand Meyer, apareció en 1987. Es un lenguaje con una sintaxis similar a la de Pascal, de tipos estáticos y cuenta con recolector de basura. Incorpora la idea de diseño por contrato (véase el Apdo. 11.2). Presenta bastantes mejoras con respecto a C++: el entorno de desarrollo se encarga de la gestión de la memoria, la sintaxis es más sencilla y el manejo de la herencia múltiple resulta más simple para el programador. Admite sobrecarga de operadores, pero no sobrecarga de métodos, e incluye herencia múltiple y clases genéricas (como las plantillas de C++). Es un lenguaje OO puro, pero no existen metaclasses.

Eiffel hace hincapié en la construcción de software seguro y de calidad, detalles que C++ olvidó por completo. Desgraciadamente, no ha gozado de mucho éxito comercial. Dos son los motivos principales: la interfaz de su entorno de desarrollo integrado (Eiffel Studio) se aparta mucho de la de los otros entornos, y ha sufrido la seria competencia de Java. Bertrand Meyer piensa (cito de una conversación privada, que agradezco al doctor Meyer y a Joab Jackson) que el escaso éxito de Eiffel se debe a que

“lo que hemos averiguado es que la gente no cree realmente en lo que dicen. Piensan que la programación no puede ser tan simple, que debemos estar escondiendo algo. Hay un cliché según el cual si algo parece demasiado bueno para ser verdad, debe serlo. Muchas personas razonan así. No sé realmente lo que hacer al respecto...”.

En 1995 se lanzó la primera versión de **Delphi** para entornos Windows; y en 2001 de lanzó la versión para Linux (Kylx). Oficialmente, pasó a llamarse Delphi en 2002; antes se llamaba Object Pascal. Delphi ha sido en todas sus versiones un lenguaje de programación y un entorno integrado de desarrollo. El lenguaje en sí es una extensión OO de Pascal, no tan versátil o eficiente como C++ u Objective-C. Delphi carece de herencia múltiple, pero en las últimas versiones ha incorporado la construcción *interface*.

El lenguaje **Self** fue diseñado por David Ungar y Randall Smith en 1986. La primera implementación pública de Self vio la luz en 1990 (en la Universidad de Stanford), aunque habían existido antes implementaciones internas. En 1991, Self pasó a los laboratorios de Sun Microsystems; la última versión del lenguaje (Self 4.2) apareció en junio de 2003.

Self es un lenguaje experimental, escasamente usado pero muy interesante. Como vimos en el Apdo. 14, en la POO y la PTOO se tiende a considerar a las clases como objetos que pueden crear a otros objetos, mientras que los objetos ordinarios no pueden. En Self, ese carácter especial de las clases se desvanece. En lugar de usar instancias basadas en clases, en Self se hacen copias de cada objeto existente y se cambia el código como se quiera. Se llama *prototipos* (objetos base empleados para crear copias) a los objetos base usados para hacer copias. Si se quiere, por ejemplo, construir un objeto *Estudiante*, basta con coger una copia de un objeto *Persona* y añadirle métodos como *matricularse* y *estudiar*. Así de sencillo. Su flexibilidad sorprende: en tiempo de ejecución, sin parar el programa en curso, se puede coger una copia de un objeto, cambiar sus métodos y atributos, y sólo ese objeto tendrá el código nuevo. Si los cambios son deseables, pueden llevarse al objeto del cual fue copia (su “padre”) y hacer copias de él.

La sintaxis de Self es muy similar a de Smalltalk, aunque resulta más simple y concreta. Este lenguaje funciona usando un código intermedio, que es ejecutado por una máquina virtual (como en Smalltalk, Java o C#). El lenguaje se distribuye junto con un entorno integrado de desarrollo donde destaca la sencillez, buen diseño y facilidad de uso de la interfaz gráfica. Ya quisieran otros lenguajes más asentados y comerciales contar con entornos de desarrollo de calidad tan alta como el de este curioso lenguaje.

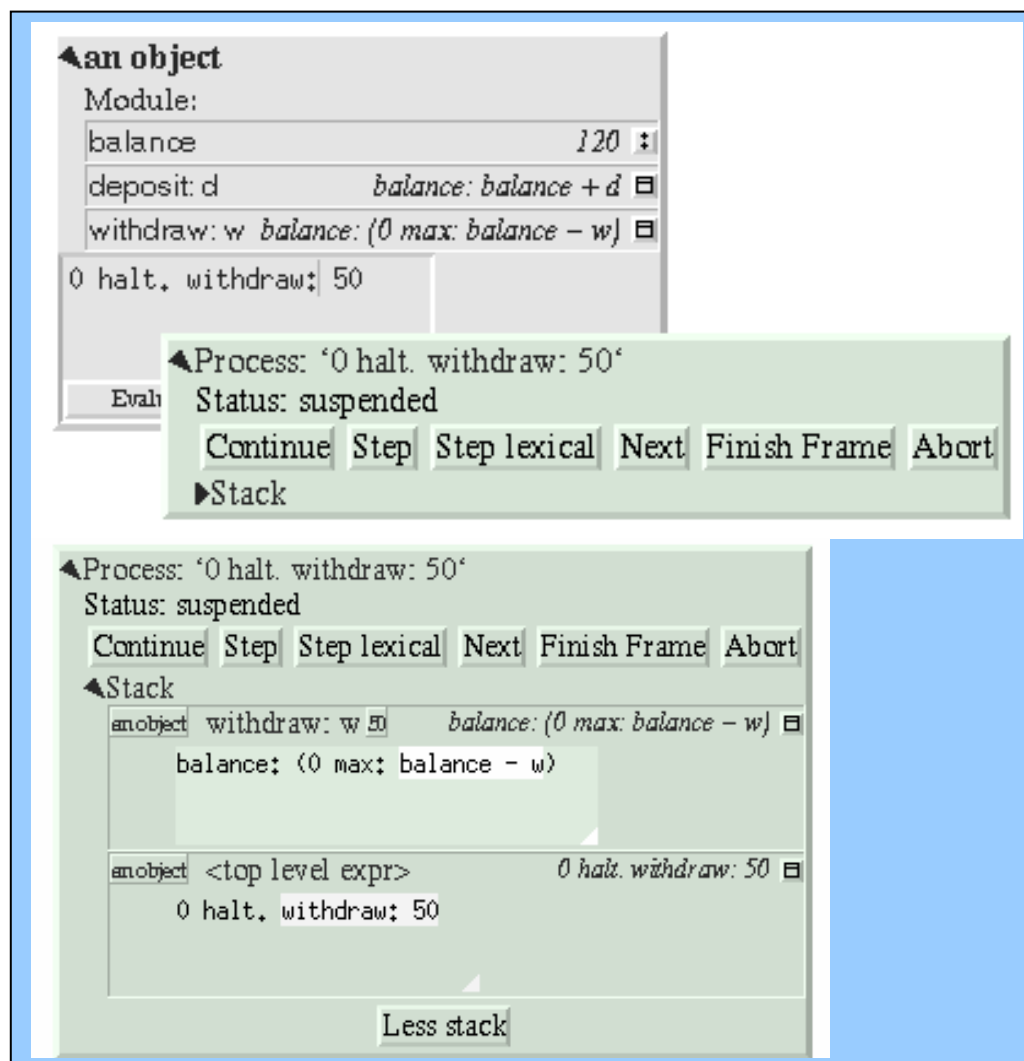


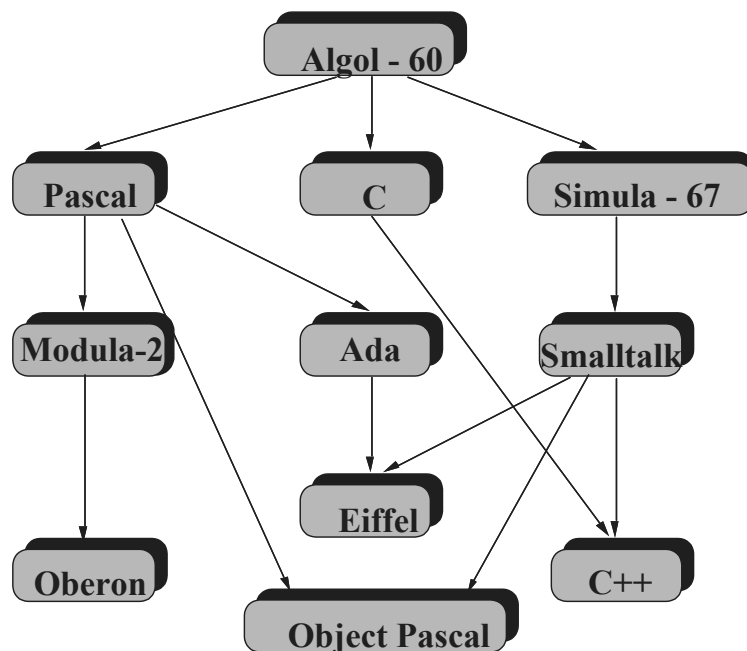
Figura 31. La programación como mecano: instantáneas del depurador del entorno de desarrollo de Self. Extraídas de la documentación oficial de Sun Microsystems.

Spoke es un lenguaje totalmente orientado a objetos, diseñado por el centro de investigación de Alcatel en Marcoussis y comercializado en 1992. Se orienta hacia la implementación de sistemas de información y de ayuda a la toma de decisiones. Los objetos se clasifican en objetos, tipos y propiedades. Mantiene separada la herencia de tipos de la herencia de clases, y permite la herencia múltiple. Es un lenguaje reflexivo, como Smalltalk, e incorpora el concepto de metaclass; sus tipos son dinámicos y es interpretado. Incluye recolector de basura y permite usar clases genéricas, además de controlar las excepciones. No ha gozado de mucho éxito ni predicamento, pero es un lenguaje muy completo

Java fue realizado por Sun Microsystems, y se lanzó al mercado en 1995. Su impacto en la comunidad de desarrolladores ha sido enorme, en parte debido a que fue el primer lenguaje que supo aprovechar las capacidades de Internet. Con todo, no ha podido desplazar a C++, aunque se ha hecho fuerte en el lado del servidor. Java es un lenguaje OO no tan puro como Smalltalk o Eiffel (véase el Apdo. 5, en la primera parte del artículo), pero es más puro que C++. Gran parte de su sintaxis es similar a la de C++, pero en su diseño influyeron en esencia Smalltalk y Eiffel. Es un lenguaje de tipos estáticos y fuertemente *tipado*, pero con ligadura dinámica. A diferencia de C++, cuenta con recolector de basura y permite desde el propio lenguaje el uso de *threads* (hilos), además de contar con numerosas facilidades para el desarrollo de aplicaciones dirigidas a Internet. Java emplea un código intermedio (*bytecode*) similar al de Smalltalk; por tanto, es independiente de la plataforma. No ha formado parte, por ahora, a ningún proceso de normalización oficial (ISO, ECMA, ANSI-BIFMA, EN, etc).

C# fue lanzado por Microsoft como el nuevo lenguaje para su plataforma .Net. De hecho, gran parte de ella ha sido escrita en C#. Fue diseñado por Scott Wiltamuth y Anders Hejlsberg, y se lanzó en 2001. Por ahora, su grado de penetración en la comunidad de programadores es bajo. C#, como todos los lenguajes de .Net, usa un código intermedio (MSIL), como Smalltalk y Java. Es pues independiente de la plataforma, aunque la estrategia comercial de Microsoft determinará si se lleva a plataformas no Windows. Sólo utiliza compiladores JIT, no intérpretes. Su sintaxis es muy similar a la de C++ y Java. Incluye propiedades, atributos, espacios de nombres y recolector de basura; permite usar hilos y XML, y también permite redefinir operadores. Ha sido normalizado por la ECMA y cuenta con el estándar internacional ISO/IEC 23270:03. Como lenguaje, no representa ningún salto revolucionario con respecto a los lenguajes anteriores, pero cuenta con un excelente entorno de desarrollo integrado (Visual Studio), que podría asegurarle un próspero futuro.

Evolución de los lenguajes OO



Miguel Ángel Abián. Julio 2003

Figura 32. Esquema de la evolución de los lenguajes OO. Nótese la fuerte relación entre los primeros lenguajes capaces de implementar TADs y los primeros lenguajes OO

A continuación se expone un breve repaso de cómo se implementan los tipos abstractos de datos en distintos lenguajes de programación:

I. Fortran 66

- Carece de de abstracción de datos.
- No proporciona ningún tipo de encapsulación de datos.

II. Simula 67

- Aparece por vez primera la abstracción de datos, mediante la construcción **class** de este lenguaje.
- En Simula, los objetos tienen existencia propia: se trata de procesos que se activan en cuanto son creados (mediante **new**, que permite inicializar los atributos).
- Todo lenguaje OO procede de Simula. La principal contribución de Simula a la abstracción de datos fue la encapsulación proporcionada por la construcción **class**.
- Ahora bien: esta encapsulación resulta incompleta, pues las variables declaradas en una clase de Simula 67 no están ocultas de los clientes que crean objetos de esa clase (se viola el principio de ocultación de la información).
- En resumen, de acuerdo con la definición específica de TAD para lenguajes de programación, las clases de Simula no son implementaciones de verdaderos TADs, debido a la violación del

principio de ocultación de la información.

III. Ada

- Proporciona características que pueden usarse para simular la implementación de verdaderos TADs.
- Las unidades de encapsulación en Ada se llaman **packages**; constan de dos partes: 1) El *package* de especificación, el cual proporciona la interfaz de la encapsulación; 2) el *package* cuerpo, que proporciona la implementación del package especificación.
- Un *package* permite agrupar tipos y procedimientos o funciones relacionados lógicamente. En Ada, una función tiene al menos un parámetro de salida; los procedimientos pueden tener parámetros de entrada o salida.
- Es potestad del programador hacer visible un *package* o proporcionar únicamente la información de la interfaz.
- Ada introduce la noción de generalización para tipos y *packages*: un tipo o *package* genérico puede parametrizarse por medio de uno o más tipos.

IV. Modula-2

- Proporciona características para simular la implementación de TADs muy similares a las de ADA. Sus unidades de encapsulación se llaman **modules**.
- La principal diferencia entre Modula-2 y Ada reside en que, en Modula-2, todas las implementaciones de tipos cuyas representaciones de datos estén ocultas en módulos deben ser punteros. Esta característica viene obligada por la manipulación que hace el lenguaje de las variables.

V. Smalltalk

- Es, posiblemente, el lenguaje OO más puro. En Smalltalk, una clase es un objeto cuyas instancias son los objetos de la clase. Una clase (definida por la construcción **class**) es una instancia de su metaclasses (**metaclass**). Para romper el bucle infinito, existe una clase que tiene como instancia todas las metaclasses del sistema, incluyéndose a sí misma.
- De su orientación pura a objetos resulta una ausencia de distinción entre clase, objeto y atributo: todos son objetos.
- Smalltalk respeta total y completamente la ocultación de la información (encapsulación): un atributo se considera igual que un parámetro de un método, y desde el exterior de una clase sólo son visibles los métodos.

Por ejemplo:	<i>saldo: aCantidad</i>	<i>“metodo set saldo”</i>
	<i>saldo:=aCantidad</i>	
	<i>saldo</i>	<i>“metodo get saldo”</i>

^saldoLos atributos, en este lenguaje, se llaman variables de instancia; y los métodos, métodos de instancia. Las variables de instancia de los *objetos clase* (variables de clase) de Smalltalk son compartidas por todas las instancias de la clase; el estado de un objeto en Smalltalk se representa por los valores de sus valores de instancia.

- Todos los métodos de instancia son públicos: forman su interfaz.
- Todas las instancias de una clase dada tienen una interfaz común.
- No existen tipos primitivos, como *int*, *short*, etc. Sólo hay objetos.
- La comunicación entre objetos se realiza exclusivamente mediante el intercambio de mensajes. Toda operación, incluso las más simples, se efectúan mediante el envío de mensajes. Por ejemplo, *x:=x1 max x2* significa que el mensaje *max* con el argumento *x2* se envía al objeto *x1*. La sintaxis es poco intuitiva (en comparación con la mayoría de los lenguajes de programación) y, en ocasiones, descorazonadora.
- En resumen: Smalltalk es un lenguaje que proporciona, a diferencia de los anteriores, implementaciones completas y cómodas de los TADs, al precio de recortar la libertad de los programadores.

VI. C++

- Es un superconjunto de C. Incorpora, por tanto, la construcción *struct*, que implementa de forma parcial los TADs, pues no proporciona control sobre el acceso a la representación de los nuevos tipos definidos. No es, por consiguiente, una verdadera implementación de los TADs.
- Proporciona la construcción ***class***, que se relaciona más directamente con los TADs que las unidades de encapsulación de Ada y Modula-2.
- Los atributos definidos en una clase se llaman miembros de datos (*data members*); las operaciones, funciones miembro (*member functions*). Los atributos en C++ son variables de tipos, bien primitivos o definidos por el usuario.
- Los miembros de datos estáticos son miembros de datos compartidos por todas las instancias de la clase; serían equivalentes, por tanto, a las variables de instancia de un objeto clase de Smalltalk.
- Las funciones de miembro estáticas son métodos que se invocan enviando un mensaje a la clase, en lugar de a las instancias de ella. Estas instancias pueden usarse independientemente de cualquier instancia.
- Las clases pueden contener miembros de datos ocultos o visibles. El programador puede decidir no esconder atributos (y, por tanto, violar el principio de encapsulación), con el fin de acceder a ellos directamente.
- Debido al uso de punteros, siempre resulta posible violar el principio de ocultación de la información, aunque los miembros de datos de una clase se declaren como privados (***private***). Veamos un ejemplo trivial:

```

class CuentaBancaria {
    private:
        char pin[4];
        ...
}

main(){
    CuentaBancaria cuenta;
    *pin_cliente = (char *)&pin;
    print("El PIN del cliente es: %s \n", pin_cliente);
}

```

- En C++ pueden sobrecargarse operadores, de modo que, definiendo adecuadamente la semántica de operadores como +, =, etc., las clases pueden integrarse completamente en el lenguaje; sin que exista diferencia entre los tipos de *primera categoría* (los predefinidos: *int*, *char*,...) y los de *segunda categoría* (definidos por el usuario). Una clase *Punto*, con los operadores sobrecargados adecuadamente, puede ser tan respetable como el venerable tipo *int*.
- C++ permite construir clases paramétricas –también llamadas plantillas–, cada una de las cuales implementa una familia de tipos.
- Permite definir clases dentro de otras clases (clases internas).
- Como C++ es un superconjunto de C, hereda su organización de módulos; lo cual puede provocar problemas al declarar variables con el mismo nombre en ámbitos distintos. La norma ANSI/ISO de C++ soluciona este problema mediante los ***namespaces*** (espacios de nombres), que definen ámbitos distintos donde se colocan las variables globales, estando prohibida la declaración de nuevos miembros de un espacio de nombres fuera de la definición de éste.



Figura 33. L'Empire des lumières (1953-54). Cuadro de René Magritte. Dos realidades complementarias (el día y la noche) se funden en el cuadro. En C++ también se funden dos realidades complementarias: la programación estructurada y la programación orientada a objetos

Todos los derechos de reproducción de las obras de Magritte están reservados. Se reproduce con fines didácticos y sin ánimo de lucro.

VII. Eiffel

- Es un lenguaje OO puro, al igual que Smalltalk.
- Al igual que ocurre con Smalltalk, las clases se declaran mediante la construcción **class**. No existe otro medio para implementar TADs.
- Pueden especificarse, usando la sentencia **feature**, qué atributos y métodos se desean que sean públicos o privados:

```

class C
    feature {all}
        ...                -- características públicas
    feature {X, Y}
        ...                -- características exportadas sólo
                           -- a X, Yy sus descendientes
    feature {NONE}
        ...                -- características privadas
end

```

- En Eiffel, las variables de instancia (implementación de los atributos) pueden hacerse públicas en forma de sólo lectura (en Smalltalk no pueden hacerse accesibles directamente –sí mediante métodos–). Como consecuencia, las variables de instancia y las funciones (así se llaman a las implementaciones de las operaciones en Eiffel) sin argumentos parecen idénticas. El cliente de la clase no necesita saber si el objeto retornado es un atributo de la instancia de la clase o si se calcula mediante una función de la clase.
- Puede haber más de una interfaz pública para una clase de Eiffel: la interfaz depende del tipo de objeto que requiera una operación en particular.

VIII. Java

- Aunque la sintaxis de Java recuerda a la de C++, está más próximo, en cuanto a diseño, a Smalltalk o a Eiffel.
- Proporciona la construcción **class**. Las clases pueden contener variables ocultas o visibles. El programador puede decidir no esconder atributos (y, por tanto, violar el principio de encapsulación), con el fin de acceder a ellos directamente.
- Existen –al igual que sucede en C++– tipos primitivos, que no son clases.
- A diferencia de C++, todos los tipos definidos por el usuario son clases.
- Por lo tanto, una implementación de un TAD en Java puede ser un tipo primitivo o una clase.
- Al igual que C++, permite definir clases dentro de otras clases (clases internas).
- Se llaman variables de instancia a las implementaciones en Java de los atributos de los objetos, y métodos de instancia a las implementaciones de las operaciones.
- En Java, una variable de clase es una variable compartida por todas las instancias de una clase. Los métodos de clase se refieren a

métodos que se invocan enviando un mensaje a la clase en lugar de a las instancias de ésta.

- En Java existen los **packages**, que vienen a ser contenedores de clases cuyo objetivo es mantener el espacio de nombres de clases dividido en secciones.
- Los **packages** de Java proporcionan un mecanismo de restricción de visibilidad que permite que ciertos elementos sean accesibles sólo desde el paquete donde se encuentran. Por tanto, las clases y paquetes de Java son los dos medios (como en C++ las clases y los *namespaces*) para encapsular y contener el espacio de nombres y el ámbito de las variables y métodos (las palabras clave **private**, **public** y **protected** funcionan de distinto modo según afecten a variables y métodos de una subclase colocada en el mismo paquete que su superclase o en uno distinto).
- A diferencia de los *namespaces* del ANSI-C++, los **packages** de Java corresponden a una jerarquía de clases en el sistema de ficheros. Un **package x.y.z** exige la existencia de un directorio llamado z dentro de un directorio denominado y, el cual se encuentra, a su vez, dentro de un directorio x.

IX. C#

- Utiliza una sintaxis muy parecida a la de C++ y Java, y un diseño muy parecido al de Java.
- Al igual que Java, emplea las construcciones **class** para implementar TADs.
- Las clases pueden contener variables ocultas o visibles. El programador puede decidir no esconder atributos (y, por tanto, violar el principio de encapsulación), con el fin de acceder a ellos directamente.
- A diferencia de Java, todos los tipos definidos por el usuario no son clases (pueden ser, por ejemplo, *structs*).
- A diferencia de Java y C++, todos los tipos primitivos son clases (como sucedía en Smalltalk).
- El uso de los **namespaces** es idéntico al de éstos en el ANSI-C++.
- Al igual que C++ y Java, permite definir clases dentro de clases (clases internas).
- C++ permite construir clases paramétricas –también llamadas plantillas–, cada una de las cuales implementa una familia de tipos.
- Permite el uso de punteros en el **unsafe code** (código inseguro), lo cual permite violar el principio de ocultación de la información, tal y como se expuso para C++.

A continuación se exponen algunos ejemplos de cómo se implementa un TAD *Cuenta* (de cuenta bancaria) en varios lenguajes de programación (pueden existir, por supuesto, muchas otras implementaciones).

EJEMPLO DE IMPLEMENTACIÓN DE UN TAD CUENTA EN SMALLTALK

```
Object subclass: #Cuenta
  instanceVariableNames: 'titular codigo saldo ultOper'
  classVariableNames: 'ultimoCodigo '
  poolDictionaries: ''
```

"Metodos de Clase"

```
abrir
  "crea e inicializa un objeto de la clase"
  ^self new abrir
```

```
inicializar
  ultimoCodigo:=0
```

"Metodos de Instancia"

```
abrir
  "Inicializar cuenta"

  titular := Persona new altaPersona.
  saldo:=0.
  codigo:= ultimoCodigo + 1.
  ultimoCodigo:=codigo.
  ultOper:= Array new:10.
```

```
ingreso: cantidad
  "Ingresa en la cuenta cantidad Euros"

  saldo:=saldo+cantidad
```

```
reintegro: cantidad
  "Retira cantidad Euros de la cuenta, si se cumplen ciertas condiciones"
```

```
(self puedoSacar:cantidad)
    ifTrue: [saldo:=saldo-cantidad]
    ifFalse: [^self error]
```

```
saldo
  "Retorna el saldo de la cuenta"
  ^saldo
```

```
verUltOper: numOper
  "Muestra las numOper ultimas operaciones en ultOper"
  ...
```

```
puedoSacar: cantidad
  "devuelve verdadero si el saldo es suficiente para sacar cantidad Euros"

  ^(saldo>=cantidad)
```

EJEMPLO DE IMPLEMENTACIÓN DE UN TAD CUENTA EN C++ (1ª parte)

// **Cuenta.h**, definición del TAD Cuenta

```
class Cuenta {
    public:
        Cuenta (Persona *persona){ saldo=0;
                                titular=persona;
                                codigo = nuevoCodigo();
                                ultOper = new lista<int>;}

        void reintegro(int suma);
        void ingreso(int suma);
        int verSaldo();
        void verUltOper(int n);
        static int nuevoCodigo(); //Devuelve el ultimoCodigo y lo incrementa

    private:
        Persona *persona;
        int saldo;
        int codigo;
        static int ultimoCodigo; //Variable de clase
        lista<int> *ultOper;

        bool puedoSacar(int suma) {return (saldo >=suma);}
};
```

EJEMPLO DE IMPLEMENTACIÓN DE UN TAD CUENTA EN C++ (2ª parte)

// **cuenta.cpp**, Definición de las funciones de la clase

```
#include "cuenta.h"

// Inicializa la variable de clase
int Cuenta :: ultimoCodigo = 0;

void Cuenta :: reintegro (int suma) {
    if puedoSacar(suma) saldo=saldo-suma;
}

void Cuenta :: ingreso (int suma) {
    saldo=saldo+suma;
}

int Cuenta :: verSaldo () {
    return saldo;
}

void Cuenta :: verUltOper(int n) {
    ...
}

static int Cuenta :: nuevoCodigo() {
    return (ultimoCodigo++);
}
```

EJEMPLO DE IMPLEMENTACIÓN DE UN TAD CUENTA EN EIFFEL

```
class CUENTA
  creation abrir
  feature {ALL}                                -- atributos públicos
    titular : PERSONA;
    saldo : INTEGER;
    codigo: INTEGER;

    abrir (persona: PERSONA) is do              -- método de creación
      saldo:=0;
      titular:=persona;
      codigo:= nuevo_codigo;                   -- nuevo_codigo
                                              -- variable de clase
      !!ultOper
    end;

    reintegro (suma: INTEGER) is do             -- método para sacar dinero
      if puedo_sacar(suma) then saldo:=saldo-suma;
    end;

    ingreso (suma: INTEGER) is do               -- método para ingresar dinero
      saldo:=saldo+suma
    end;

    ver_ult_oper (k: INTEGER) is do ... end;    -- visualiza las k ultimas
    ....                                       -- operaciones en la cuenta

  feature {NONE}                               -- atributos y métodos privados
    ultOper: LIST[INTEGER];
    puedo_sacar (suma: INTEGER): Boolean is do
      Result:= saldo>=suma
    end;
end
```

EJEMPLO DE IMPLEMENTACIÓN DE UN TAD CUENTA EN JAVA

```
package gestiondecuentas;
public class Cuenta {

    private Persona titular;
    private int saldo;
    private int codigo;
    private static int ultimoCodigo; //variable de clase
    private int ultOper[];

    public Cuenta (Persona persona) {    saldo=0;
                                          titular=persona;
                                          codigo = nuevoCodigo();
                                          ultOper = {...}

    public void reintegro(int suma)
    { ... }
    public void ingreso(int suma)
    { ... }
    public int verSaldo() { ... }
    public void verUltOper(int n) { ... }
    public static int nuevoCodigo() {return ultimoCodigo++;}
    private boolean puedoSacar(int suma) {return (saldo >=suma);}
}
```

17.2. Clasificaciones de los lenguajes OO.

En la primera parte de este artículo, en el apartado 5, se citaron las características que suelen tener los lenguajes orientados a objetos. Atendiendo a las facilidades que los lenguajes OO proporcionan a los programadores, Bertrand Meyer proporciona, en *Construcción de software orientado a objetos* (2ª Ed.) ([Meyer, B., 1999]), otra clasificación, complementaria a la de la primera parte. Sus “siete pasos hacia la felicidad basada (orientada) hacia objetos” son éstos:

1. Estructura modular basada en objetos
2. Abstracción de datos
3. Gestión automática de la memoria
4. Clases
5. Herencia
6. Polimorfismo y herencia dinámica
7. Herencia múltiple

Veamos la clasificación de algunos lenguajes según los *pasos* anteriores:

	1	2	3	4	5	6	7
Fortran 66							
Simula 67							
Ada							
C							
C++							
Delphi							(*)
Smalltalk-80							
Eiffel							
Java							(*)
C#							(*)

(*): Incluye la estructura sintáctica *interfaz*.

Además de esta clasificación, existen otras comúnmente aceptadas en el conjunto de la bibliografía:

- Clasificación de Wegner
- Clasificación de Tesler
- Clasificación en lenguajes puros/híbridos

En *The Object-Oriented Classification Paradigm in Research Directions on Object-Oriented Programming* ([Wegner, P., 1987]), Wegner clasifica los lenguajes OO en lenguajes basados en objetos, basados en clases y orientados a objetos.

Los lenguajes basados en objetos permiten objetos, es decir, disponen de componentes encapsulados que tiene identidad, comportamiento (operaciones) y estado. Ada 83 pertenece a este grupo.

Los lenguajes basados en clases disponen, además de objetos, de componentes de tipo clase, generalmente implementados con la construcción *class*. Las clases se comportan como plantillas o fábricas de objetos. CLU pertenece a esta categoría.

Los lenguajes orientados a objetos ofrecen objetos, clases y herencia. Smalltalk, C++, Delphi, Java, Ada 95 y C# pertenecen a este grupo.

Tesler, en *Object-Oriented Dynamic Languages ([Proceedings of the Object Expo Conference, Julio de 1993])*, clasifica los lenguajes de programación en lenguajes orientados a procesos y lenguajes orientados a objetos. Dentro de cada categoría distingue también entre lenguajes estáticos y dinámicos. Así, C++, Java y C# serían lenguajes estáticos OO; Smalltalk sería un lenguaje dinámico OO; Fortran, Pascal y C entrarían dentro de la categoría de lenguajes estáticos orientados a procesos; y, por último, Lisp sería un lenguaje dinámico orientado a procesos.

La clasificación en lenguajes puros e híbridos distingue entre aquellos que no proceden de ningún lenguaje OO ya existente (puros: Smalltalk, Spoke, Self, etc.) y los que sí (Delphi, Objective-C, C++). Generalmente, los lenguajes híbridos son más rápidos de aprender que los puros, resultan más eficientes y cuentan con más documentación y bibliotecas. A cambio, suelen omitir alguna característica de la POO. Por ejemplo, en C++ existen tipos de datos definidos por el usuario (*union*, *struct*) que no son objetos, y no toda comunicación entre objetos puede realizarse exclusivamente mediante el intercambio de mensajes.

18. Crítica interna de la POO.

Este artículo no pretende ser una apología ciega o condescendiente de la programación orientada a objetos. Es más: ya en la primera parte se hizo una crítica sociológica de la POO. Aquí se va a desarrollar una crítica desde dentro de la POO, expresada en sus propios términos.

El factor que más debilita a la POO es precisamente el que más firmeza le da: la herencia. El motor de su potencia coincide con su talón de Aquiles. Los problemas derivados de la herencia se vertebran alrededor de tres causas principales. A saber: la dificultad de entender aisladamente la herencia, la violación del encapsulamiento y el problema de clase base frágil.

18.1. Imposibilidad de entender aisladamente la herencia.

Entender fraccionadamente la herencia es objetivo imposible. El código de una subclase aislada suele ser ininteligible. Basta con ver este ejemplo:

```
public class ClaseHija extends ClasePadre {  
    // Código escrito en Java  
  
    public void metodo() {  
        super.metodo();  
        a = a + 3;  
    }  
    ... // Métodos y atributos de la clase hija  
}
```

Sin conocer y comprender la clase padre, no puede entenderse la clase hija. Si se desconoce el código de la clase padre, no puede saberse qué hace *metodo* o qué representa la variable *a*. Conforme nos adentremos en una jerarquía de clases profunda, más nos costará entender el interior de las subclases de los niveles más bajos.

No existen soluciones mágicas para este problema: proviene del propio mecanismo interno de la herencia. Lo más recomendable es documentar todas las clases, nombrar los métodos, atributos y clases con identificadores significativos, y no perder de vista la jerarquía a la que pertenece cada subclase.

18.2. La herencia y la encapsulación están en extremos opuestos de la OO.

En el apartado 4.3, en la primera parte, se dio cierto relieve a la encapsulación. La herencia, curiosamente, incumple parcialmente el principio de encapsulamiento. Los objetos de una subclase *conocen* cosas (atributos, métodos) de los objetos de su superclase.

Cualquier programa que pretenda reutilizar una subclase también deberá hacer uso de su superclase (o superclases). En consecuencia, cuando se usa la herencia, debe substituirse el encapsulamiento de clases aisladas por el encapsulamiento de jerarquías completas. Un componente (un *package* de Java, por caso) puede estar perfectamente encapsulado y, sin embargo, sus

clases constituyentes pueden, unas con respecto a otras, tener un encapsulamiento deficiente.

18.3. El problema de la clase base frágil: cuando construimos jerarquías sobre arenas movedizas.

El problema de la clase base frágil suele considerarse como un problema de (re)compilación. Esta visión no es completa: en realidad, para hablar con propiedad, deberían considerarse dos problemas de la clase base frágil: **el PCBF sintáctico** y **el PCBF conceptual**.

El **PCBF sintáctico** es bien conocido, y maldecido, por cualquier programador de C++. En algunos lenguajes, al añadir o modificar un método o una variable de instancia en una clase, se necesita recompilar la clase modificada y todas las que hagan uso de ella. Si no se procede así, el programa no funcionará.

En el caso de que se modifique una clase base, se necesita recompilar todas y cada una de las subclases. Basta con dejar alguna sin recompilar para que el programa acabe fallando.

En lenguajes como C++ se comprende enseguida por qué se necesita la recompilación de las subclases. En el Apdo. 12.3 se explicó cómo se implementa el polimorfismo en C++. Este lenguaje usa tablas virtuales para las clases con funciones virtuales. Cualquier introducción o eliminación de una función en una clase provoca que cambie el orden de las entradas en su tabla virtual. En consecuencia, las llamadas a la función que antes ocupaba una determinada posición tendrán consecuencias fatales, pues ya no se encontrará allí. Como las tablas virtuales se generan en tiempo de compilación, no hay posibilidad de ajustarlas dinámicamente.

Realmente, la recompilación se hace necesaria aunque sólo se haya ordenado el código o añadido algún comentario, porque las tablas virtuales almacenan punteros a funciones. Cualquier cambio en el código de una clase cambia las direcciones de memoria a la que apuntan los *vptrs* de su tabla virtual. Este problema se puede hacer menos gravoso para el programador con el uso de herramientas como *make* o de entornos de desarrollo integrado (Eclipse con el *plug-in* para C/C++, Visual Studio, C++ Builder, etc.), pero es inevitable. O lo hace una herramienta o tiene que hacerlo manualmente el programador.

Lenguajes con enlace dinámico puro, como Smalltalk, Java o C#, permiten modificar en tiempo de ejecución variables de instancia o métodos y reflejar esos cambios en el programa en funcionamiento, sin afectar al código ya existente. No obstante, una vez finalizado el programa será necesario recompilar las clases si se quiere que los cambios sean permanentes. Con todo, en Java y C# se puede evitar la complejidad de la recompilación usando *interfaces* en lugar de superclases abstractas. Por otro lado, como Smalltalk, Java y C# no permiten la herencia múltiple, las jerarquías de clases no suelen ser tan complicadas como las que pueden generarse en C++.

En Self no existe el PCBF sintáctico: como todo se construye en tiempo de ejecución, copiando y modificando prototipos, el problema de la recompilación no se plantea.

El **PCBF conceptual** se puede ver como una prolongación orientada a objetos del lema “uno elige a sus amigos y colaboradores, pero no puede elegir a sus padres”. Este problema suele manifestarse cuando se cambia el significado de la implementación de un método en la superclase o cuando se intenta introducir en la superclase un método con la misma declaración o firma que un método que ya existía en alguna subclase.

La primera situación se ilustra con este código:

```
public class Punto2D {  
  
    // Código en Java  
  
    private double x,y; // Coordenadas espaciales  
  
    public Punto2D(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public double getX() {  
        return x;  
    }  
  
    public double getY() {  
        return y;  
    }  
  
    public double moduloCuadrado() {  
        return (getX() * getX() + getY() * getY());  
    }  
  
}
```

```
public class Punto3D extends Punto2D {  
  
    // Código en Java  
  
    private double z; // Coordenada espacial z  
  
    public Punto3D(double x, double y, double z) {  
        super(x, y);  
        this.z = z;  
    }  
  
    public double getZ() {  
        return z;  
    }  
  
}
```

```

        public double moduloCuadrado() {
            return (getX() * getX() + getY() * getY() + getZ() * getZ());
        }
    }

```

Supongamos que ahora el programador decide cambiar la implementación de *Punto2D* de manera que manipule las coordenadas del punto directamente (quizás por motivos de eficacia):

```

public class Punto2D {

    // Código en Java

    protected double x,y; // Coordenadas espaciales

    public Punto2D(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double moduloCuadrado() {
        return (x * x + y * y);
    }

}

```

La clase hija ya no funcionará. Cuando se dice que la herencia es reflexiva, no debe entenderse sólo el sentido evidente de la afirmación (*si X es padre o madre de Y, Y es hijo o hija de X*). Debe tenerse asimismo en cuenta que no sólo las superclases determinan cómo serán las subclases: también las subclases establecen qué cambios pueden realizarse sobre sus superclases. El trivial ejemplo de *Punto2D* y *Punto3D* nos da una enseñanza importante: una subclase puede interferir en la implementación de su superclase. En el mundo real sería como si un hijo dijera a sus padres que deben asumir sus responsabilidades y actuar con arreglo a ellas.

La segunda situación en la que suele manifestarse el PCBF conceptual ocurre cuando se añade un nuevo método a una superclase. En este caso, los métodos con la misma declaración (nombre, argumentos y tipo de retorno) en las subclases redefinen con ansia el nuevo método. El problema se hace patente cuando el propósito del nuevo método no guarda relación con el de los métodos que ya tenían implementadas las subclases; pues las subclases usarán los métodos que ya tenían, y no el nuevo método de la superclase. C++, Delphi y Java sufren este problema de manera incurable. Por el contrario, C# está libre de él. En el nuevo lenguaje de Microsoft, si se desea crear un método para una subclase con la misma declaración que un método de la superclase, pero sin que reemplace al método original, el método de la subclase se marca con *new*. Veámoslo con un ejemplo.

Supongamos que tenemos una clase como ésta:

```
public class ClasePadre {  
  
    // Código en C#  
    ... // Métodos y atributos de la clase padre  
}
```

Consideremos que construimos una jerarquía de clases con *ClasePadre* y que una de las subclases tiene esta forma:

```
public class ClaseHija : ClasePadre {  
  
    // Código en C#  
    ... // Métodos y atributos de la clase hija  
    public virtual void miMetodo() {  
        ... // Código del método  
    }  
}
```

Por último, supongamos que se decide incluir un método *miMetodo* en la clase *ClasePadre*, con la misma declaración que la de *miMetodo* de la clase hija:

```
public class ClasePadre {  
  
    // Código en C#  
    ... // Métodos y atributos de la clase padre  
    public virtual void miMetodo() {  
        ... // Código del método  
    }  
}
```

Ante este código, el compilador de C# dará un aviso, indicando que el método *miMetodo* de *ClaseHija* sobrescribirá al de *ClasePadre*. Si no es ése el propósito del programador, puede usar la palabra *new*:

```
public class ClaseHija : ClasePadre {  
  
    // Código en C#  
    ... // Métodos y atributos de la clase hija  
    public new void miMetodo() {  
        ... // Código del método  
    }  
}
```

En lenguajes como C++, Delphi o Java, el método *miMetodo* de la clase padre sería sobrescrito de manera irremediable.

19. Algunas reflexiones finales.

¿Por qué tanta insistencia en la abstracción? ¿Para qué tanto espacio dedicado a los tipos abstractos de datos? ¿Por qué tan poco espacio dedicado a la escritura directa de código?

Mis justificaciones aparecen en los subapartados 17.1 y 17.2. En 17.3 se compara la breve historia de la construcción de software con la larga historia de la arquitectura.

19.1. El programa no sólo es el código (o “Respetemos el software”).

Una obsesión constante de los programadores es el código: a muchos sólo les preocupa generar código que funcione, aunque sea malamente y a duras penas. Siempre me ha intrigado este planteamiento. No conozco a ningún ingeniero de telecomunicaciones que piense que lo único que importa son las ondas electromagnéticas y los electrones, ni a ningún arquitecto que opine que los ladrillos, como resultan imprescindibles en cualquier construcción, son el objetivo último de su profesión, y que –por tanto–, crea que no debe prestar atención a otros aspectos de la construcción de edificios.

Algunas personas creen que para programar basta con estudiarse un libro de programación y con leerse un manual del lenguaje que se vaya a emplear. Esa forma de pensar puede servir cuando se abordan pequeñas aplicaciones e incluso cuando se abordan proyectos muy específicos, sin perspectivas de reutilización o ampliación. Cuando se abordan proyectos complejos, ese planteamiento marcha en la dirección errónea. Es como si un albañil se pusiera a construir un rascacielos... no funciona, no puede funcionar.

A veces, por simple divertimento, me gusta construir pequeños muebles. Mi punto débil son los cajones: por mucho que me esfuerce nunca salen perfectos. La caída del cajón, la escuadría, la alineación de las guías; siempre hay algún pequeño defecto. Casi inapreciable, pero aunque el cajón abre y cierra correctamente, yo sé que hay algo que no es perfecto. El mueble funciona y sirve a su propósito, pero carece de la maestría de un buen carpintero. En el fondo, me faltan los conocimientos y la experiencia del carpintero. En la ingeniería del software esos conocimientos y esa experiencia se pueden adquirir de libros de construcción de software, de diseño OO, de patrones, etc. Siempre habrá, por supuesto, una parte práctica, en la que se deberá escribir el código y para la cual se necesitarán conocer las sutilezas y la sintaxis del lenguaje de programación que se vaya a emplear. Pero lo importante es sustituir **la escritura de software como fin** por **la escritura de software como proceso**.

En los años sesenta, fallaban muchos proyectos escritos en Fortran 66, Cobol o Algol-60. En los años setenta, un buen número de proyectos escritos en Smalltalk, C ó Ada seguían fracasando. Basta con sustituir “sesenta” y “setenta” por “ochenta” y “noventa”, y Fortran, C, etc., por C++, Java y C#, para que tengamos el panorama completo (por ahora). Demasiados culpables, ¿verdad? Cuesta creer que tantos lenguajes de programación hayan resultado tan malos. ¿No fallará **el proceso de construcción de software**?

En la campaña electoral que llevó a Bill Clinton a la presidencia de los Estados Unidos, se podían ver en casi todos los centros de la campaña demócrata carteles y letreros con la frase **“IT’S THE ECONOMY, STUPID!”**. Ellos tenían claro el factor fundamental para ganar unas elecciones presidenciales; muchos programadores deberíamos aprovechar esta enseñanza y recordar qué es lo realmente importante: si las herramientas que utilizamos o lo que hacemos con ellas. Recordemos las sabias y escépticas palabras de Alan Davis: *“Un ingeniero de software indisciplinado con una herramienta de software resulta un **peligroso** ingeniero de software”* (la negrita es mía).

La escritura de buen código debe ser la consecuencia de la planificación, del diseño y del uso adecuado de la abstracción. Del mismo modo que un arquitecto muestra como resultado final de su trabajo un edificio, un programador debe mostrar su código. ¿Pero quiere eso decir que el arquitecto no ha hecho cálculos, planos y diseños previos? ¿Significa que se ha limitado a apilar ladrillo sobre ladrillo y que, a última hora, ha dibujado algún plano para enseñarlo en el Colegio de Arquitectos donde está colegiado?

Hace poco, un programador con mucha más experiencia que yo me dijo que, si un programa no funciona como él esperaba, lo toquetea (*sic*) hasta que funciona. Me pregunto si un ingeniero de canales, caminos y puertos se hubiera atrevido a decirme, sobrio, que “Primero construyo una presa; y si no funciona bien, cambio el diseño hasta que funcione correctamente”. ¿Tan poco se valora la planificación del software? ¿Por qué tantos programadores planifican mejor sus vacaciones que sus proyectos?

19.2. La abstracción es la clave (o “La representación de una pipa no es una pipa, pero al menos se parece”).

Siempre que existan formalismos matemáticos o patrones de diseño aplicables a nuestros problemas, conviene usarlos.

El uso de formalismos matemáticos, situados en un nivel superior de abstracción, permite abordar situaciones complejas y formular las soluciones por medio de un lenguaje universal.

El uso de patrones de diseño, con su alto grado de abstracción, nos permite aprender de la experiencia de otros y evitar errores ya conocidos.

No debe desecharse la abstracción matemática como algo ajeno a la programación o útil solamente en la universidad. La programación no es sólo matemática, pero se funda en ella. De hecho, es imposible entender los aspectos avanzados de la programación sin tener al menos una somera base matemática. Un ejemplo nos situará las cosas en perspectiva: la ausencia, por ahora, de una base matemática para las bases de datos OO es uno de los motivos del retraso en su aceptación. El lenguaje SQL de las bases de datos relacionales no deja de ser una aplicación de las proyecciones del álgebra relacional, una disciplina matemática bien establecida y con cierta solera. ¿Cuándo se desarrollaron los modernos sistemas de gestión de bases de datos? Una vez establecida el álgebra relacional (basada en la teoría clásica de conjuntos) y las reglas de normalización. Nadie usaría de manera general un

lenguaje de consultas no basado en principios lógicos y en un sistema deductivo formal, pues sería muy dudoso –casi imposible– que un lenguaje de consultas basado en experiencias concretas, o en pruebas y errores, diera siempre respuestas inambiguas, trazables y reproducibles.

19.3. La arquitectura del software.

La evolución de la ingeniería del software ha avanzado paralela a la de la construcción de edificios. En la Edad Media, los maestros de obras aprendían su oficio a partir de la experiencia de otros maestros y de una serie de reglas, muchas de ellas empíricas. Los conocimientos se transmitían de generación en generación, a menudo de padres a hijos, quienes heredaban el oficio. Las reglas (cómo cimentar, cómo trazar la estructura maestra, qué materiales usar, etc.) juzgadas como correctas se obtenían de las conclusiones extraídas de la construcción de las obras que no se habían derrumbado. ¿Y las construcciones que caían durante el aprendizaje de las reglas correctas? Bueno, al menos servían como ejemplos.

En aquel tiempo rara vez se tendía a la experimentación o a la innovación en cuanto a materiales, estructuras, etc., pues no existían conocimientos aplicables para nuevos diseños o materiales. Con el tiempo, la evolución de la física y la matemática proporcionó una base teórica, más abstracta, para la construcción de edificios, basada en la física del estado sólido (a veces, mal llamada física del estado sórdido), el álgebra matricial, el cálculo tensorial, etc.

A la luz de nuestros conocimientos actuales, se puede comprender por qué funcionaban algunas reglas de los arquitectos de antaño y también se pueden alegar motivos para haber descartado otras. Pero el paso dado desde la Edad Media no es cuantitativo, es cualitativo: hoy podemos saber, gracias a los conocimientos teóricos, cómo se comportarán estructuras de materiales que no se habían empleado antes (polímeros, aceros con fibra de vidrio, nanomateriales, etc.) o simular el comportamiento de edificios con formas jamás vistas antes (poliédricas, animales, etc.). Los maestros de obras del Medievo nunca hubiesen podido aventurar cómo se iban a comportar materiales y diseños no probados antes. Gracias a los avances matemáticos y físicos, se pasó de la artesanía a la ingeniería. La oruga devino mariposa.

Como puede verse, la evolución de la construcción de software, la cual apenas cuenta con medio siglo de historia, guarda fuertes similitudes con la de la construcción de edificios. La aparición de patrones, métricas, sistemas de aseguramiento de la calidad, formulaciones matemáticas y de los modernos lenguajes OO ha permitido elevar la artesanía del software a ingeniería del software.

La construcción de software también ha sufrido –y sufre– episodios semejantes a la construcción de las pirámides. Éstas se yerguen en el desierto, misteriosas, solitarias, y, por ende, hermosas. Hermosura impresionante, pero inútil para el avance de la arquitectura. Planos inclinados, poleas, palancas, andamiajes, dibujos; todo se destruía cuando se acababa la pirámide. Los faraones seguían al pie de la letra el principio de “dos pueden guardar un secreto si uno está muerto”. Ninguna de las técnicas utilizadas para su construcción sobrevivió; y –por consiguiente– las pirámides han resultado

inútiles para el desarrollo de la arquitectura. Constituyen un episodio anecdótico de la historia de la arquitectura, un legado sin futuro, una desviación del curso de la técnica arquitectónica. Representan lo mismo que los *zeppelins* a la historia de la aviación, o el psicoanálisis a la historia de la psiquiatría: ramas secundarias, prósperas por un tiempo, pero pronto abortadas.

La breve historia del software también alberga pirámides, *zeppelins*, pseudociencias, cachivaches... Señal de que la construcción de software está llegando a la edad adulta. ¿Se recordarán sus pirámides tanto como las arquitectónicas? Aún siguen allí, en medio del desierto, maravillando a turistas y estudiosos, recordándonos que el misterio es belleza; y la belleza, misterio.

El autor, por último, siente la necesidad de realizar algunas confesiones. Confiesa que no ha omitido, al menos conscientemente, ningún aspecto de la OO, ya sea práctico o teórico, que pueda ser de interés; que ha evitado utilizar la tónica habitual de referirse a citas de citas de citas y ha recurrido a los textos originales siempre que ha podido; que no ha privilegiado a ningún autor, importante o no, sobre otros y ha preferido dar una visión plural de la teoría de la OO; que la elaboración, organización y redacción del material, correcta o no, es original; que ha tratado de explicar la OO como le hubiera gustado que fuera explicada, pese a sus posibles errores y omisiones; que ha intentado, en la medida de sus fuerzas, esforzarse en evitar ambigüedades y dislates, pero no sabe si lo ha conseguido; que no cree (ni ha creído) en las verdades universales, absolutas y atemporales, y por tanto no pretende que este texto sea definitivo o completo ni que el lector lo crea así: no lo es; y que ha disfrutado mucho y de muchas maneras escribiendo este artículo, y espera que el lector así lo perciba.

Nota biográfica del Autor: Miguel Ángel Abián nació en Soria (1972). Se licenció en Ciencias Físicas en 1995 por la U. de Valencia y consiguió la suficiencia investigadora en 1997 dentro del Dpto. Física Aplicada de la U.V con una tesina acerca de relatividad general y electromagnetismo. Además ha realizado diversos cursos de Postgrado sobre bases de datos, lenguajes de programación Web, sistemas Unix, comercio electrónico, firma electrónica, UML y Java. Ha participado en diversos programas de investigación TIC relacionados con el estudio de fibras ópticas y cristales fotónicos, y ha publicado diversos artículos en el *IEEE Transactions on Microwave Theory and Techniques* relacionados con el análisis de guías de onda inhomogéneas y guías de onda elípticas.

En el ámbito laboral ha trabajado como gestor de carteras y asesor fiscal para una agencia de bolsa y actualmente trabaja en el Laboratorio del Mueble Acabado de AIDIMA (Instituto Tecnológico del Mueble y Afines), ubicado en Paterna (Valencia), en tareas de normalización y certificación. En dicho centro se están desarrollando proyectos europeos de comercio electrónico B2B para la industria del mueble basados en Java y XML (más información en www.aidima.es). Ha impartido formación en calidad, normalización y programación para ELKEDE (Grecia), CETEBA (Brasil) y CETIBA (Túnez), entre otros.

Últimamente, aparte de asesorar a diversas empresas, trabaja en la implementación Java de un emulador del microprocesador MIPS multiciclo y es investigador en el proyecto INTEROP (Interoperabilidad de software) del Sexto Programa Marco de la Comisión Europea.

Sus intereses actuales son el diseño asistido por ordenador de guías de ondas y cristales fotónicos, la evolución de la programación orientada a objetos, Java, el intercambio electrónico de datos, el surrealismo y París, siempre París.