



Ceci n'est pas une pipe.



Ceci n'est pas une pipe.

ORIENTACIÓN A OBJETOS: CONCEPTOS, TERMINOLOGÍA Y LENGUAJES (PARTE 1)

Versión 2.0

Miguel Ángel Abián

Ceci n'est pas une pipe.

Ceci n'est pas une pipe.



Ceci n'est pas une pipe.



Ceci n'est pas une pipe.

ORIENTACIÓN A OBJETOS: CONCEPTOS, TERMINOLOGÍA Y LENGUAJES (PARTE 1)

Resumen: Este tutorial, dividido en dos partes, proporciona una introducción a la terminología y los conceptos de la orientación a objetos (OO), así como a los lenguajes orientados a objetos.

Esta primera parte presenta los términos y conceptos de la OO desde un punto de vista conceptual, dando definiciones intuitivas y manejables, e introduce el análisis y diseño OO. Hay también una comparación con la metodología estructurada.

En la segunda parte (<http://www.javahispano.org/tutorials.item.action?id=33>), se explican ampliamente conceptos como clase, objeto, mensaje, tipo abstracto de datos, polimorfismo y relación. Además, se explican las principales características de los lenguajes OO más populares, se muestra código que implementa los conceptos de la OO (polimorfismo, herencia, etc.) en diversos lenguajes OO (Java, C++...) y se estudia la evolución de los lenguajes OO.

Abstract: This tutorial, divided in two parts, provides an introduction to Object-Oriented (OO) terminology and concepts, and to Object-Oriented languages.

This first part introduces OO terms and core concepts from a conceptual point of view, giving intuitive and manageable definitions, and introduces OO analysis and design (OOAD). There is also a comparison with the structured methodology.

In the second part (<http://www.javahispano.org/tutorials.item.action?id=33>), concepts like class, object, message, Abstract Data Types, polymorphism, inheritance and relationship are widely explained. In addition, the main characteristics of the more popular OO languages are explained, code implementing the OO concepts (polymorphism, inheritance...) is shown for several Object-Oriented languages (Java, C++...), and the evolution of OO languages is analysed.

Keywords: Abstraction, Class, CRC Cards, Encapsulation, Hierarchy, Learning Object Orientation, Modularity, Object, Object Orientation, Object Oriented Analysis, OOAD, Object Oriented Design, Object Oriented Methodology, Object Oriented Paradigm, Object Oriented Programming, Object Oriented Terminology, Software Analysis, Software Design, Structured Methodology

ÍNDICE

1. Introducción. El porqué de este trabajo	Página 5
2. Glosario de la orientación a objetos	Página 10
3. Visión general de la orientación a objetos	Página 18
3.1 Orígenes de los conceptos de la OO	Página 18
3.2 La orientación a objetos como metodología de desarrollo de sistemas.	Página 18
3.2.1 Introducción	Página 18
3.2.2 El análisis OO. Casos de uso	Página 20
3.2.3 El diseño OO	Página 28
3.2.4 La programación OO	Página 34
3.3 Comparación de la OO con la metodología estructurada	Página 35
4. Fundamentos de la orientación a objetos	Página 41
4.1 Abstracción	Página 41
4.2 Modularidad	Página 42
4.3 Encapsulación	Página 44
4.4 Jerarquía	Página 46
5. Lenguajes de programación orientados a objetos. ¿Puede utilizarse la OO en lenguajes no orientados a objetos?	Página 49
6. El paradigma orientado a objetos: éxitos y fronteras	Página 54
6.1 El éxito del paradigma orientado a objetos, ¿se basa únicamente en criterios objetivos?	Página 54
6.2 Los límites de la POO	Página 56
Nota biográfica del autor	Página 56

ORIENTACIÓN A OBJETOS: CONCEPTOS, TERMINOLOGÍA Y LENGUAJES (PARTE 1)

Miguel Ángel Abián
mabian ARROBA aidima PUNTO es

Fecha de creación: 15.01.2006

Versión: 2.0 (revisada y aumentada respecto a la primera versión; la versión 1.0 se creó en noviembre de 2002)

Copyright (c) 2002-2006, 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/>).*

Abandonadlo todo.

Abandonad a Dadá.

Abandonad a vuestra mujer, abandonad a vuestra amante.

Abandonad vuestras esperanzas y vuestros temores. Abandonad a vuestros hijos en medio del bosque. Soltad al pájaro en mano por aquellos que están volando.

Abandonad si hace falta una vida cómoda, aquello que os presentan como una situación con porvenir.

Lanzaos a los caminos.

André Breton, *Les Pas perdus* (1924)

Yo inventé el término “orientación a objetos”, y puedo decirle que no estaba pensando en C++.

Frase atribuida a Alan Kay, creador del primer lenguaje OO puro, en la OOPSLA '97

Solamente hay dos cosas equivocadas en C++: el concepto inicial y la implementación.

Bertrand Meyer, creador de Eiffel

La programación orientada a objetos es, en cierto sentido, sólo un truco de programación que usa indirección. Es un truco que los buenos programadores llevan años usando.

Bjarne Stroustrup, creador de C++

Los objetos en el sentido de la orientación a objetos han estado presentes entre nosotros desde que se desarrolló la conciencia en la especie humana, pero se ha tardado miles de años en aprovecharlos como técnica. ¿Cuánto tiempo más se necesitará para extraer el máximo rendimiento de la OO? [...] Quizá necesitemos un cambio conceptual antes que una sucesión frenética de cambios técnicos.

Howard Humphrey (2002)

Convertir C en un lenguaje orientado a objetos es como tratar de convertir un perro en un pulpo clavándole más piernas.

Steve Taylor

C te trata como un adulto consentido. Pascal te trata como un chico desobediente. Ada te trata como un criminal.

Bruce Powel Douglass

Ah, Java. Todo el rendimiento de Smalltalk combinado con la claridad sintáctica de C++.

Anónimo

Conocer la sintaxis de Java no convierte a nadie en un ingeniero de software.

John Knight

Cualquier programador lo suficientemente persistente y empeinado conseguirá escribir código al estilo Fortran o Cobol en cualquier lenguaje de programación que utilice.

Anónimo. Oído en la Universidad de Valencia

1. Introducción. El porqué de este trabajo

El propósito de este trabajo, presentado en dos partes, consiste en realizar una introducción a la orientación a objetos (OO); introducción que incluye sus conceptos, su terminología y un repaso de los lenguajes OO más conocidos y de la evolución de éstos. No pretendo sólo definir formalmente los conceptos de la OO, sino también presentarlos de una forma inteligible e intuitiva, pero no carente de precisión. Para conseguirlo no he escatimado en ejemplos ni en figuras.

Con este trabajo adquirirá una base para manejarse por las tecnologías y las metodologías OO, que dominan actualmente la ingeniería del software. Conocer los conceptos de la orientación a objetos le ayudará a saber qué puede hacer en cada lenguaje OO y cómo hacerlo, sin atarse obligatoriamente a ninguno en concreto.

La primera versión de este trabajo ganó un concurso que se celebró a raíz del tercer aniversario de javaHispano (2004). Los lectores de javaHispano lo votaron como el mejor tutorial publicado en el portal. Por no saber, yo ni sabía que existía el concurso hasta que Alberto Molpeceres me comunicó que había ganado en la categoría de "Mejor tutorial". Por cierto, como premio elegí dos libros relacionados con la OO. Para esta nueva versión, mi **mayor recompensa** consistiría en hacerle sentir curiosidad por la OO o en mostrarle algún aspecto de la OO que se le haya pasado por alto.

En esta **primera parte** del trabajo expongo de manera introductoria el análisis y diseño (general y OO), comparo la OO con la metodología estructurada, explico los fundamentos de la OO y analizo los motivos de su éxito (no siempre de carácter meramente técnico).

En la **segunda parte** (<http://www.javahispano.org/tutorials.item.action?id=33>) presento en detalle, y en ocasiones formalmente, los conceptos de la OO: clases, objetos, mensajes, polimorfismo, herencia, tipos abstractos de datos, etc. Además, expongo la evolución de los lenguajes orientados a objetos y explico cómo se materializan esos conceptos en cada lenguaje, enumerando las diferencias y similitudes entre ellos. Para algunos lenguajes (Java, C++, Smalltalk, Eiffel) incluyo código fuente. Dado que el trabajo se publica en javaHispano, la mayoría de los ejemplos están escritos en Java. En un futuro cercano, también publicaré una versión 2.0 de la segunda parte.

Deliberadamente, esta **primera parte** de *Orientación a objetos: conceptos, terminología y lenguajes* se centra en los conceptos y definiciones de la OO, y no en su aplicación a lenguajes de programación concretos. Estas son mis razones para hacerlo así:

- El análisis y diseño orientado a objetos no tiene por qué vincularse necesariamente a sistemas de software.
- En los sistemas de software, la orientación a objetos no se restringe a lenguajes de programación concretos, aun cuando se presenten como lenguajes OO. La OO es una metodología general de desarrollo de software, independiente del lenguaje utilizado, si bien hay lenguajes que cuentan con mejores mecanismos que otros para poder aplicarla eficazmente. Utilizar la OO es más que programar en uno o varios lenguajes de programación: implica manejarla como metodología de desarrollo. **Si una aplicación no incluye abstracción, encapsulación, jerarquías, mensajes y polimorfismo no está orientada a objetos, aunque esté escrita en Java, C++ u otro lenguaje OO.**
- A lo largo de diez años he visto, en programas de compañeros y clientes, clases con miles de líneas y decenas de métodos estáticos, clases compuestas sólo por métodos, clases con cohesión tan fuerte que volvían el código irreutilizable, usos de la herencia que derivaban en código confuso e inestable... Me consta que no son hechos aislados: cualquier programador o analista con cierta experiencia puede referir hechos similares, si no calcados.
El motivo de estos y tantos otros errores suele ser simple. A saber, el programador ha pasado de un lenguaje estructurado a uno orientado a objetos sin tener claros o haber madurado los conceptos de la OO. Aparentemente, estos conceptos son muy sencillos..., pero ya se sabe: el diablo está en los pequeños detalles. En los lenguajes OO puede proseguirse, conscientemente o no, con un enfoque estructurado de programación, pero resulta inadecuado.
- Este último motivo es más prosaico: en la segunda parte del tutorial se explica cómo se *traducen* a los lenguajes OO los conceptos y definiciones explicados en esta primera parte.

Sé que existe una gran diferencia entre comprender los fundamentos de la orientación a objetos y aplicarlos con éxito: entre teoría y práctica siempre hay una gruesa zanja. Por eso, aunque lea estas páginas con atención, necesitará práctica, paciencia y esfuerzo para escribir buenos programas OO. ¿Debería, pues, prescindir de los conceptos e ideas de la OO, y buscar un libro sobre la sintaxis de Java o empezar a escribir código teniendo delante algún manual de ayuda? No. Tal como dice una de las frases que abren este trabajo: “Conocer la sintaxis de Java no convierte a nadie en un ingeniero de software”. Lo crea o no, hay más sabiduría en la frase que en algunos libros de programación. Aparte, los lenguajes y las plataformas van y vienen, pero los conceptos suelen mutar poco. Es mucho más útil conocer para qué sirve la herencia, sus tipos (múltiple, de generalización, de implementación, de especialización...) y cuándo usarla que saber que Java implementa la herencia (¿de qué tipo?) mediante la palabra clave *extends*.

En el campo de la orientación a objetos, **las tecnologías han intentado suplantar a las ideas**; lo que ha producido confusión entre los desarrolladores y, al final, ha dificultado que se use una orientación a objetos normalizada y estándar. ¿Qué sentido tiene la encapsulación en un lenguaje OO donde los punteros permiten acceder a las variables privadas? ¿Es conveniente que más de cien metodologías OO vaguen por ahí, como vampiros hambrientos de sangre del cliente? ¿Por qué se permite, en un lenguaje OO, la mezcla de clases y de estructuras de datos no OO?

Hace algunos años, vi un anuncio televisivo sobre una herramienta que ha fragmentado un poco más la OO. En él salía un actor, disfrazado de Elvis Presley, que hacía de programador. Pasaba unos segundos tecleando en el ordenador, y enseguida cogía su guitarra y entonaba una canción de Elvis. El mensaje del anuncio no lleva muchas alforjas: con la herramienta se acaba enseguida el trabajo y uno puede dedicarse a lo que le gusta, aunque sea desafinar como una nutria en celo.

Bien, Johnny Carson dijo: “Si la vida fuera justa, Elvis estaría vivo; y todos los imitadores, muertos”. No sé si la vida fue justa con Elvis o no; pero estoy seguro de que no lo ha sido con la orientación a objetos. La pobre ha tenido –y tiene– más metodólogos de los que necesitaba; y muchos lenguajes y herramientas le han hecho un flaco favor. Suscribo por completo las palabras de Wolfgang Strigel cuando escribe “Los conceptos básicos de la OO se conocen desde hace dos décadas, pero **su aceptación todavía no está tan extendida** como los beneficios que esta tecnología puede sugerir” y “La mayoría de los usuarios de la OO **no utilizan los conceptos de la OO de forma purista**, como inicialmente se pretendía. Esta práctica ha sido promovida por **muchas herramientas y lenguajes** que intentan utilizar los conceptos en diversos grados” (el marcado en negrita es mío).

Antes de proseguir, debería saber qué es para mí la programación orientada a objetos (POO). Desde mi punto de vista, la POO es una herramienta para reducir la complejidad de los sistemas de software. Usando la orientación a objetos, un sistema que ocupa miles o millones de líneas de código puede organizarse como una colección de pequeñas unidades (objetos), cada una con cierta independencia y ciertas responsabilidades.

Un programa OO consiste en un conjunto de objetos que intercambian mensajes; cada objeto decide por sí mismo si debe o no aceptar los mensajes que recibe, así como la interpretación de cada mensaje. En un programa OO medianamente complicado, los objetos no son totalmente independientes: unos heredan propiedades y métodos de otros; algunos necesitan consultar a otros para desempeñar sus tareas; otros llevan dentro de sí más objetos...

La principal ventaja que percibo en la POO estriba en que un programa de objetos se extiende y mantiene de manera más sencilla que uno sin ellos. Si el programador necesita objetos que las empresas de software no “fabrican” o comercializan, puede construir sus propios objetos tomando un objeto ya existente y personalizándolo conforme a sus necesidades. De la misma manera, si un programa OO no funciona como se desea, encontrar el fallo resulta más sencillo que en uno estructurado o funcional: el error está muy localizado; es decir, se encuentra en la pequeña porción de código de un objeto, y no repartido entre decenas o cientos de líneas.

La POO se apoya firmemente en el análisis y diseño OO. Considero que escribir un programa OO sin hacer antes el análisis y diseño OO se parece a construir un mueble sin haber analizado para qué servirá y cuáles serán las dimensiones de sus componentes. Al final se tendrá un mueble, pues la persistencia humana carece de límites y, a menudo, de sensatez; pero las puertas no cerrarán bien, costará abrir los cajones, el mueble se tambaleará con cualquier golpe... En fin, **si uno no sabe adónde**

quiere ir, cualquier lugar vale. Por el interés que merece el análisis y diseño OO, le dedico un cierto espacio en ambas partes del tutorial. En la primera, uso una perspectiva general e intuitiva; en la segunda, vistos ya todos los conceptos de la OO, profundizo un poco más (desgloso una serie de pasos para el diseño OO y expongo los principios generales de éste).

El lector crítico se preguntará si la OO no es algo pasado ya, superado por nuevos enfoques y técnicas. No: **la OO sigue siendo la metodología universalmente aceptada para construir grandes sistemas de software.** Los conceptos de la OO siguen presentes en proyectos tan actuales y ambiciosos como la **Web semántica**: una red, prolongación de la actual, que contendrá información comprensible para las máquinas, de modo que las aplicaciones podrán sacar conclusiones de los datos (razonamiento automático) y realizar para nosotros tareas impensables hoy día (negociación automática entre empresas, búsquedas que tengan en cuenta el contexto de las palabras, integración de información procedente de fuentes arbitrarias).

La Web semántica codificará los recursos de la Web mediante objetos, clases y relaciones entre ellas (dicho en forma técnica, mediante *ontologías*). Si desea saber más sobre la Web semántica y sobre cómo utiliza los conceptos de la OO, puede consultar *El futuro de la Web* (<http://www.javahispano.org/tutorials.item.action?id=55>).

Observo con tristeza que el mundo se encamina rápidamente hacia una **sociedad del 20%-80%**. Es decir, una sociedad donde el 20% de las personas se dedicará a trabajos estables, bien remunerados y socialmente reconocidos, relacionados con la gestión, la economía y las tecnologías de la información; y donde el otro 80% se dedicará a tareas poco especializadas, mal remuneradas y carentes de estabilidad laboral y de seguridad social. Especialmente cínica me resulta la actitud de algunas instituciones internacionales que siempre recomiendan recortar o suprimir el gasto público en seguridad social, educación y pensiones; cuando sus trabajadores gozan de buenos sueldos, pensiones vitalicias y seguros médicos privados para ellos y sus familias. Lo mismo puedo decir de cierta superpotencia que se dedica a preservar como sea los derechos de autor y las patentes de sus ciudadanos, olvidando convenientemente que cuando era una nación en desarrollo (hasta bien entrado el siglo XIX) no respetaba los derechos intelectuales de los autores extranjeros. Puesto a tirar balones hacia dentro, también me parecen sumamente reprochables las medidas de liberalización que cierta comunidad de países occidentales exige ya a los países subdesarrollados o en vías de desarrollo, cuando esa comunidad ha tardado 50 años en adoptar esas medidas, y eso que está compuesta por países sumamente industrializados. La medicina que el médico receta a los demás no es la que él toma, desde luego. Por eso ha llegado a ser médico. Dicho de otro modo, **quien pone la soga no quiere poner el cuello.**

¿Ha mejorado el mundo gracias a la OO? ¿Ha contribuido la OO a que se cree la sociedad del 20%-80%? Por una parte, como la OO ha acelerado y abaratado la construcción de grandes sistemas de software, imprescindibles para las tecnologías de la información, ha contribuido –y contribuye– a formar esa sociedad. Por otra, el hincapié de la OO en la reutilización del código y su asociación con entornos visuales de programación y metodologías iterativas han favorecido que los lenguajes orientados a objetos hayan salido fuera del círculo de los iniciados; y han permitido también que se generen muchas herramientas libres y de código abierto (algunas de calidad desigual, que no todo el monte es orégano). Estas herramientas son usadas gratuitamente por desarrolladores que no podrían adquirirlas, aunque quisieran, si fueran de pago, y han

contribuido a difundir la POO entre mucha gente sin relación profesional con la informática. Ahora que le he mostrado la cara y la cruz de la OO, le toca a usted quedarse con la que prefiera.

En esta **segunda versión** de la parte 1 de *Orientación a objetos* he revisado y aumentado el material que había en la primera versión. Sigo escribiendo en javaHispano sobre la OO por varios motivos: primero, sigo reflexionando sobre ella; segundo, continúo leyendo libros y artículos que tratan esta materia; tercero, sigo probando lenguajes y herramientas OO. Mientras escribía esta introducción, tenía sobre la mesa unas frases de Richard P. Feynman que pronuncié una vez en una charla sobre electrodinámica clásica. Los rostros de los asistentes hace tiempo que se esfumaron de mi memoria, pero aún conservo el folio donde anoté las frases (no recuerdo ya de dónde las saqué):

Nunca más volveré a cometer el error de leer opiniones de expertos. Pero, evidentemente, uno solamente tiene una vida. Comete en ella todos los errores y aprende qué cosas no debe hacer. Y cuando lo sabes, es que has llegado al final.

Tanto en la OO como en tantas otras cosas, sigo cometiendo errores y, lo que es mucho peor, sigo leyendo opiniones de expertos. Cuanto más sé sobre la OO, más dudas tengo y más escurridizos me parecen sus conceptos y técnicas. Cuanto más intento atrapar las ideas de la OO y fijarlas en el papel, más intangibles se vuelven. Cuanto más hablo con expertos en modelado OO, más sutileza y artesanía veo en su trabajo. Con todo, considero que esta segunda versión resulta más exacta y completa que la primera. Si usted cree que no es así, me ayudaría mucho, y también ayudaría a los lectores de javaHispano, que me señalara los posibles errores o ambigüedades (mi correo es **mabian ARROBA aidima PUNTO es**). Muchas gracias.

2. Glosario de la orientación a objetos

En esta sección se definen unos cuantos conceptos que aparecerán en el trabajo. Resulta necesario contar con un glosario sobre la OO; tanto más cuanto que existen muchos lenguajes de programación OO, muchas metodologías, y muchos autores que usan los mismos términos para referirse a distintos conceptos. Con el glosario, evitaremos confusiones y ambigüedades.

En la definición de cada entrada marco en azul las palabras que, a su vez, son otras entradas del glosario (sólo la primera vez que aparecen). Algunas definiciones –como *clase*, *objeto*, *mensaje* y *operación*– son provisionales y se concretan más en la segunda parte de *Orientación a objetos* (<http://www.javahispano.org/tutorials.item.action?id=33>).

- **Análisis:** Proceso que genera el **modelo conceptual** de un **dominio** de interés. Cuando se desea construir un **sistema** (sea de software o no) para un dominio, el análisis también “captura” los **requisitos** que el sistema debe cumplir.
- **Clase (de objetos):** En un **modelo de dominio**, define un conjunto de propiedades compartido por un determinado grupo de entidades (cosas materiales, conceptos, ideas, sucesos); es una abstracción de los **objetos** similares de un **dominio**. Por ejemplo, una clase *Compra* representa el conjunto de las transacciones comerciales que son compras, y tiene propiedades o atributos como *fecha* y *hora*.
En la **programación orientada a objetos** (POO), una clase es un fragmento de código que describe los atributos y operaciones de los objetos (**instancias**) que pueden generarse a partir de ella. Por ejemplo, en una clase *Cliente*, los atributos podrían ser *nombre* y *dirección*, y las operaciones podrían ser *añadir()*, *borrar()* y *actualizar()*. Las clases pueden entenderse como plantillas para “fabricar” objetos.
- **Diseño:** Proceso que usa los resultados del **análisis** para generar una especificación de la implementación de un **sistema** o producto. Los diseños suelen usar dibujos, iconos, modelos o planos. A menudo, la palabra *diseño* se utiliza también para referirse a la descripción lógica del funcionamiento del sistema o producto.
- **Dominio:** Parte del mundo real bajo estudio. Cuando consideramos **sistemas** de software, el dominio es el campo o ámbito para el cual se construye el sistema. Por ejemplo, una aplicación de contabilidad cae dentro del dominio financiero.
- **Ingeniería del software:** Disciplina cuyo propósito es la producción de software libre de fallos, dentro del plazo previsto, cumpliendo el presupuesto inicial, y que satisfaga las necesidades del usuario o cliente.
- **Ingeniero/a de software:** Persona que aplica las técnicas de la **ingeniería del software** y que a menudo tiene que afrontar con tranquilidad de espíritu, mirada resignada y autocontrol Zen las reducciones de los plazos de entrega, los recortes en el presupuesto original y las modificaciones sustanciales y sin previo aviso de los **requisitos** iniciales.

- **Instancia:** Materialización de un **objeto** durante el tiempo de ejecución de un programa. En términos computacionales, una instancia corresponde directamente a un bloque contiguo de memoria, que comienza en una dirección de memoria y ocupa cierto espacio. Las instancias de una misma **clase** se diferencian entre sí porque cada una tiene su propio bloque de memoria. Por lo general, los términos *instancia* (de una clase) y *objeto* se usan indistintamente.
- **Mensaje:** Estímulo enviado a un **objeto** con un nombre y unos argumentos adecuados, que provoca que el objeto comience cierto comportamiento al activarse la **operación** asociada al mensaje. Los mensajes modifican el estado del objeto o devuelven información sobre su estado. Por ejemplo, un mensaje *devolverAltura* enviado a un objeto *Persona* devuelve la altura del objeto.
- **Metodología:** Colección de técnicas repetibles para resolver una familia de problemas. Por ejemplo, un libro de bricolaje contiene una metodología para hacer, en la propia vivienda, obras de carpintería, fontanería y electricidad. En software, el **análisis** y el **diseño** suelen realizarse siguiendo una determinada metodología.
Tradicionalmente, en francés, inglés y español siempre había existido la diferencia entre metodología (“conjunto de métodos que se siguen en una investigación científica o en una exposición doctrinal”) y **paradigma** (“modelo, patrón”). Actualmente, en informática se ha perdido esa diferencia: los **ingenieros de software** usan “paradigma” y “metodología” en el sentido de “colección de técnicas para resolver problemas”. En este trabajo mantendré la diferencia entre ambos términos.
- **Metodología de desarrollo de sistemas:** **Metodología** para producir **sistemas** de forma organizada, empleando una colección predefinida de técnicas y convenciones de notación. En el caso de los sistemas de software se usan metodologías de desarrollo de software (también llamadas metodologías de **ingeniería del software**).
- **Modelo:** Esquema teórico de un **sistema** o de una realidad compleja, que se elabora para facilitar su comprensión y el estudio de su comportamiento. Un modelo suele representarse mediante diagramas más los textos, notaciones o aclaraciones necesarias para entenderlos. Por ejemplo, en arquitectura, el plano de un edificio es un modelo.
- **Modelo de dominio o conceptual:** Representación de los conceptos presentes en un **dominio** de interés. Por ejemplo, en un dominio empresarial, el modelo de dominio incluye conceptos como “compra”, “venta”, “gestión de almacenes”, etc., así como las relaciones entre ellos. El modelo de dominio suele incluir aclaraciones y explicaciones sobre los conceptos. En ocasiones comprende un glosario.
- **Modelo de software:** Representación de un componente de software (una **clase** o un módulo, por ejemplo) o un **sistema** de software. Los **modelos** de software suelen representarse en **UML**.
- **Objeto:** En un **modelo de dominio**, abstracción de alguna entidad presente en el **dominio** de interés. Por ejemplo, el objeto *compra101*, con atributos o propiedades *fecha=15/01/2006* y *hora=12:30*, corresponde a la abstracción de una operación de compra realizada en esa fecha y hora. Los objetos y las **clases** se hallan muy relacionados: una clase abstrae el

conjunto de propiedades que comparte un grupo de cosas tangibles, conceptos, ideas o sucesos. Preguntarse qué viene antes, si el objeto o la clase, es similar a preguntarse qué fue antes, si el huevo o la gallina.

En la POO, estructura de datos encapsulada con un conjunto de **operaciones** que operan sobre los datos. Todo objeto tiene identidad propia, comportamiento (conjunto de operaciones) y estado (definido por los valores de sus atributos o propiedades y por sus **relaciones** con otros objetos). Para activar una **operación** de un objeto, basta enviarle un **mensaje**.

- **Operación:** Descripción de la habilidad de un **objeto** para responder a un **mensaje**, así como de los requisitos de éste. La implementación de una operación para una **clase** (es decir, el código que define la acción asociada al mensaje) se denomina *método* o *función*. Las operaciones válidas para un objeto se definen en la clase de la que procede.
- **Orientación a objetos (OO): Metodología** para desarrollar sistemas mediante **clases** y **objetos**. En el campo del software, la OO es una metodología de **ingeniería del software** que se basa en estos fundamentos: abstracción (clases), encapsulación (clases), modularidad (clases) y jerarquía (herencia y polimorfismo).
- **Paradigma:** Estrategia o punto de vista para realizar tareas o resolver problemas. Marco conceptual.
- **Programación orientada a objetos (POO): Metodología** de programación basada en **objetos** y en el envío de **mensajes** entre éstos. Los fundamentos de la POO son los de la **OO**.
- **Relación o asociación:** Abstracción de un conjunto de interrelaciones semánticas concretas que se dan sistemáticamente entre distintos tipos de **objetos**. Por ejemplo, si se abstrae el hecho de que todos los automóviles tienen ruedas, se obtiene una relación *lleva* entre las **clases** *Automóvil* y *Rueda*.
Las relaciones o asociaciones describen un conjunto de posibles interrelaciones, del mismo modo que las clases describen un conjunto de posibles objetos.
- **Requisito:** Descripción de lo que debe hacer un **sistema** o producto (requisito funcional) o de cómo debe implementarse (requisito no funcional o técnico). Verbigracia, en un sistema de software empresarial, un requisito podría ser éste: “Las compras que excedan los 1.000 € se guardarán en la base de datos de mejores clientes”.
- **Sistema:** Grupo de elementos, componentes o dispositivos que se integran para conseguir ciertos propósitos o funciones. Por ejemplo, un sistema informático se compone de hardware y software.
- **UML:** Lenguaje gráfico que se usa principalmente para visualizar, especificar, construir y documentar componentes de software (una clase de Java, por ejemplo) y **sistemas** de software (una aplicación de gestión empresarial, p. ej.). Aunque UML se ha convertido en el lenguaje estándar para los **modelos** de software **OO**, es un lenguaje de modelado de propósito general: puede usarse para modelar sistemas que no son de OO ni de software (por ejemplo, empresas).

Mundo real	Orientación a objetos
Cosa	Objeto
Concepto, idea, abstracción	Clase
Interacción	Operación

Tabla 1. Equivalencias entre algunos conceptos del mundo real y de la orientación a objetos.

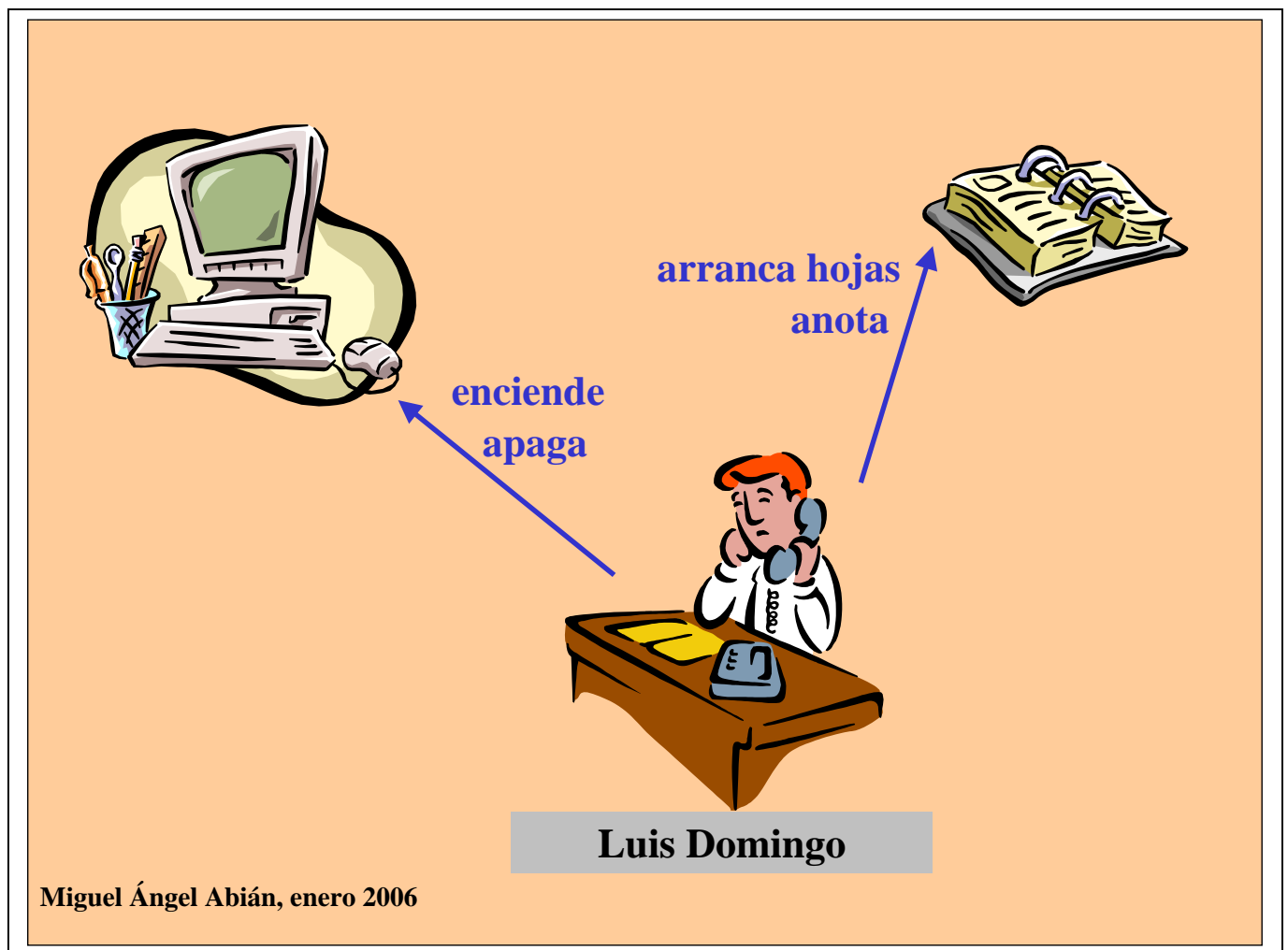


Figura 1. Ejemplo de modelo de dominio o conceptual. Los conceptos y las relaciones se han extraído de un sencillo análisis del dominio de una oficina. Si los términos del dominio son poco habituales, el modelo conceptual suele incluir un glosario con los términos que aparecen en el dominio.

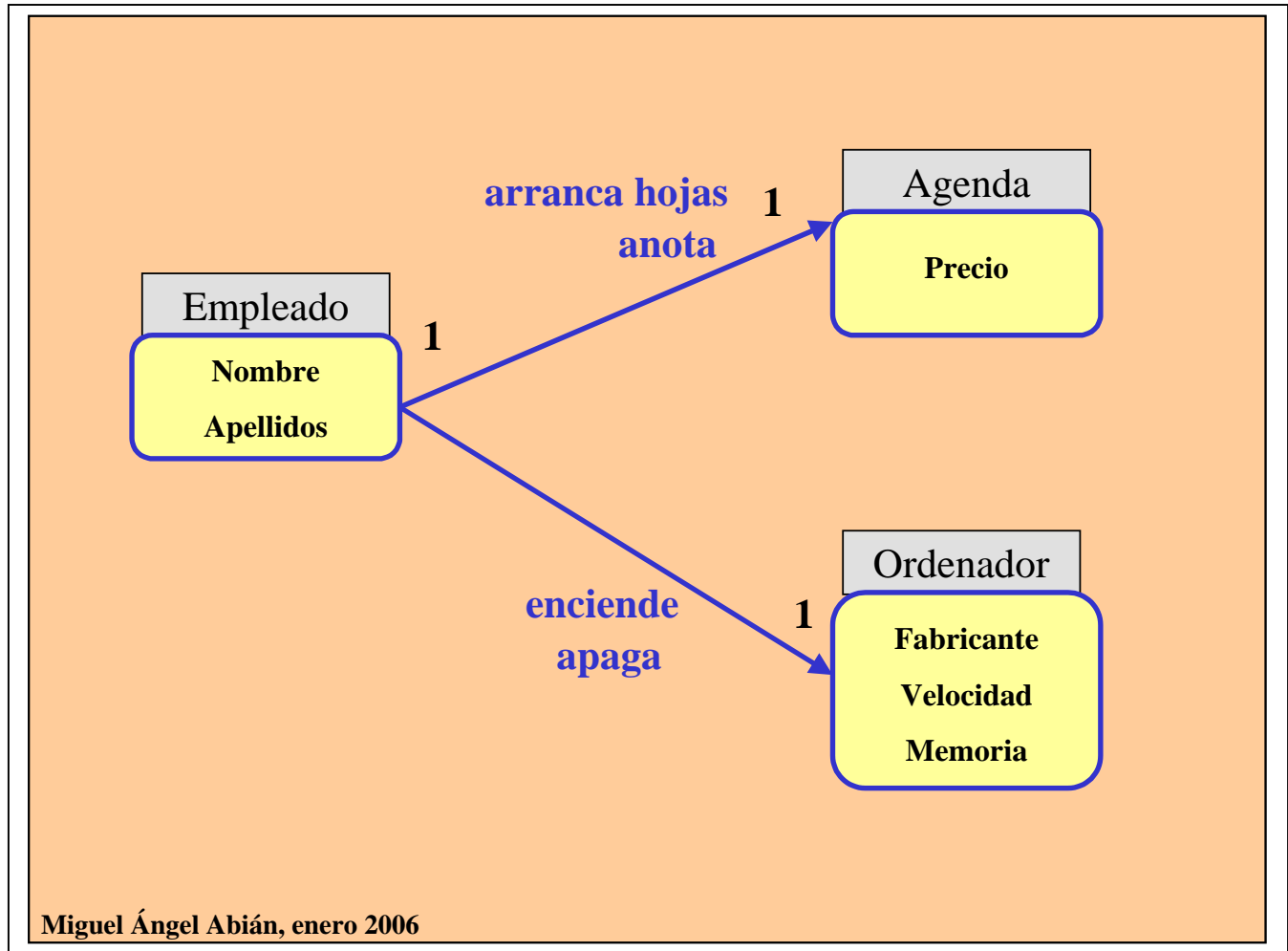


Figura 2. Ejemplo de modelo conceptual (o de dominio) orientado a objetos. Los conceptos que aparecen en la figura 1 se han convertido en clases. Este modelo conceptual OO es una representación un poco más formal del mostrado en la página anterior. Ninguno de los dos tiene nada que ver con el software. El número 1 hace referencia a la multiplicidad: un empleado arranca hojas de una agenda (la suya), anota en una agenda, enciende y apaga un ordenador (el suyo). Del mismo modo, las hojas de una agenda son arrancadas por un empleado, sólo un empleado escribe en ellas, y un ordenador es encendido y apagado por un empleado.

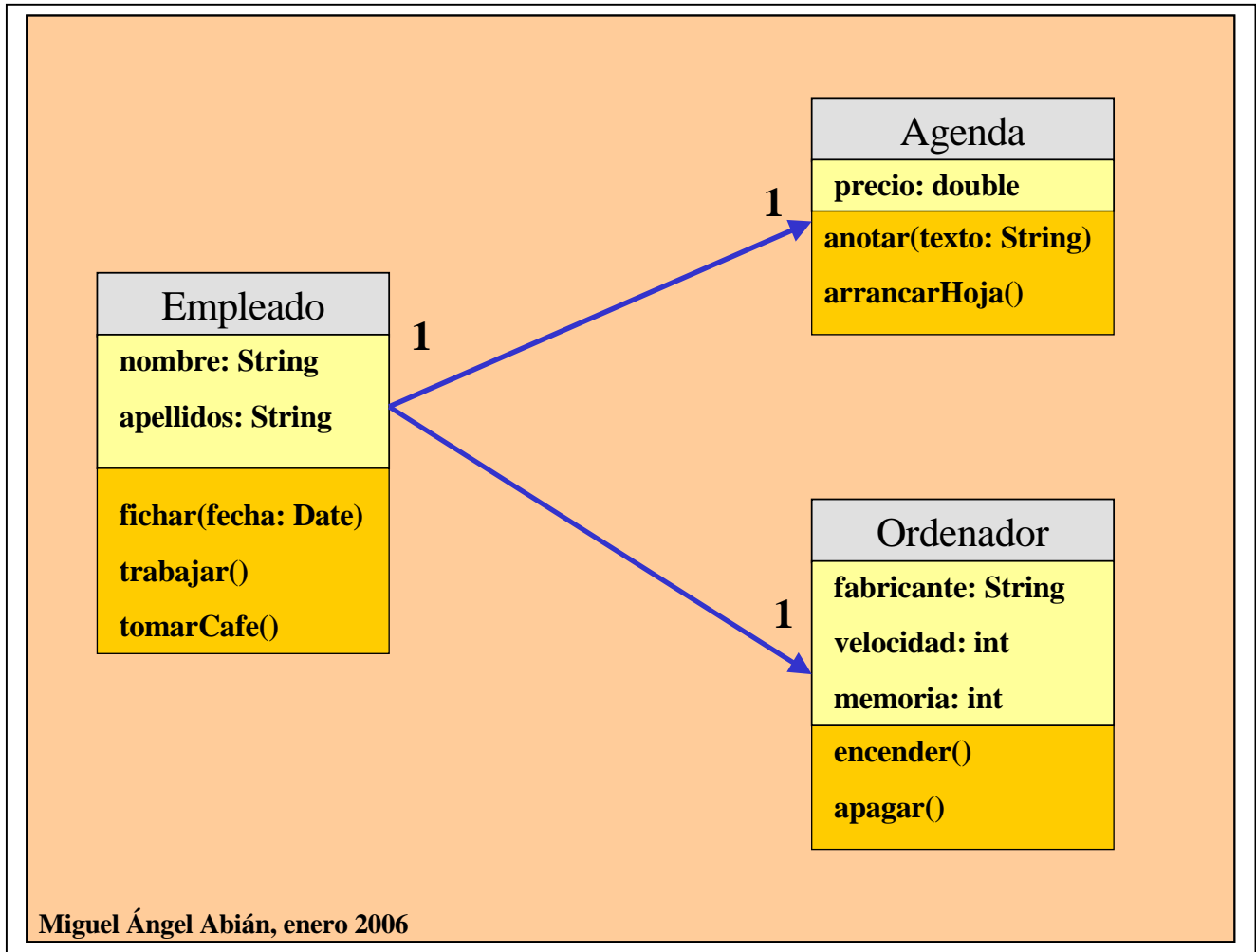


Figura 3. Modelo de software correspondiente al dominio de la figura 2 (he añadido algunos métodos para *Empleado*). Establece cómo deben ser las clases de software, sin especificar en qué lenguaje (Java, C++, etc.) van a ser implementadas. Este modelo, orientado a objetos, contiene tres clases de software, cada una con sus atributos y operaciones. El modelo de software recuerda al modelo conceptual o de dominio, pues los conceptos del mundo real se transforman en clases del modelo de software. Ahora bien, el modelo de software no es una representación exacta del mundo real. En las figuras 1 y 2, por ejemplo, era el empleado quien encendía el ordenador; en la figura 3, el ordenador dispone de una operación *encender()*.

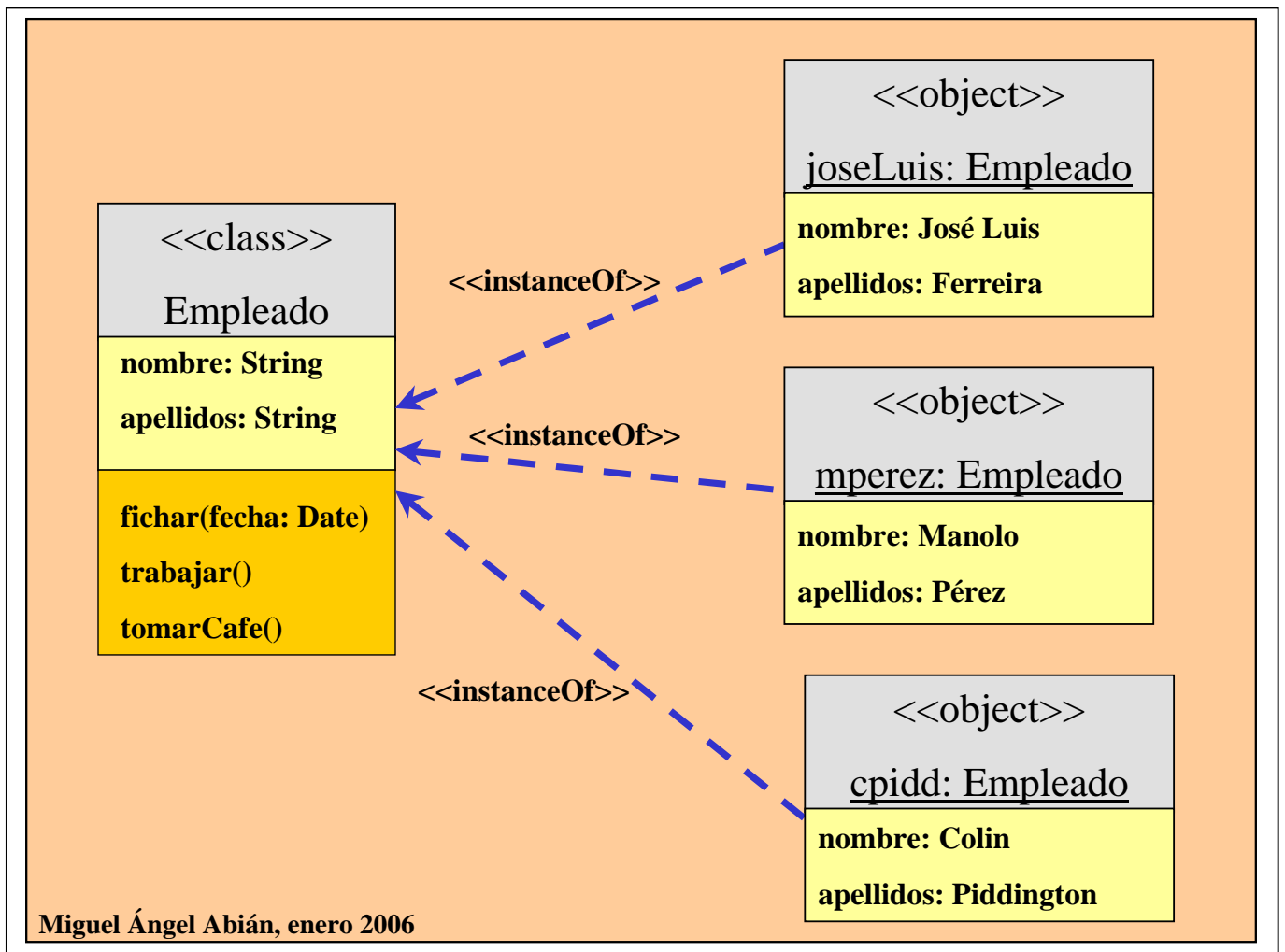


Figura 4. Modelo de software de una clase y tres instancias suyas. Utilizo los estereotipos de UML. La clase actúa como “plantilla” de instancias u objetos. Cada objeto posee identidad (cada uno tiene su propio bloque de memoria), estado (valores de los atributos o propiedades + relaciones con otros objetos) y comportamiento (todo lo que puede hacer).

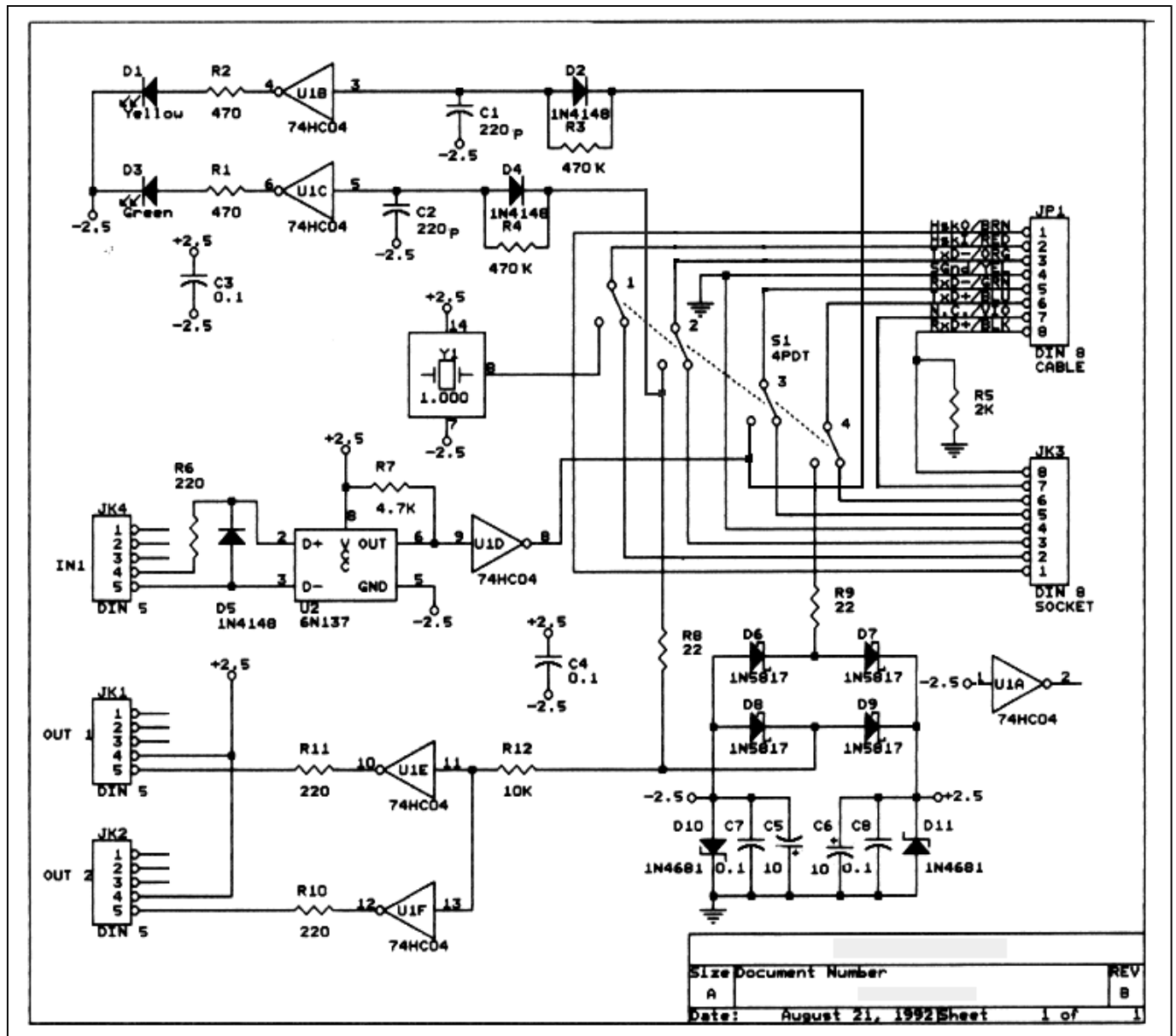


Figura 5. Este modelo describe una interfaz MIDI para un ordenador Apple Macintosh. Cada símbolo tiene un significado preciso, que puede encontrarse en cualquier manual de electrónica. Los modelos de sistemas de software no difieren mucho de los modelos de sistemas eléctricos o electrónicos. En un mundo ideal, los sistemas de software se diseñarían ensamblando componentes prefabricados, tal como se hace con los circuitos electrónicos.

3. Visión general de la orientación a objetos

3.1 Orígenes de los conceptos de la OO

Pese a que el término “orientación a objetos” ejerce una innegable fascinación para muchos ingenieros de software recién salidos de la universidad, gran parte de sus conceptos distan mucho de ser nuevos. Algunos, incluso, forman parte de la tradición cultural de Occidente: Platón y Aristóteles usaron en sus escritos términos como “objetos”, “clases”, “subclases”, “clasificaciones”, etc. (Posiblemente, Aristóteles llevó demasiado lejos el concepto de clase: clasificó dentro del mismo grupo a la cotorra y la lechuga, pues ambas son verdes.)

Ya a principios del siglo XX, Alfred North Whitehead (1861-1947) y Bertrand Russell (1872-1970) formalizaron y ampliaron los anteriores conceptos, tratando de proporcionar –infructuosamente– una base lógica autoconsistente para la matemática. Las definiciones lógicas y filosóficas de esos conceptos han influido mucho en la terminología OO. Por ejemplo, en los *Principia Mathematica* [Bertrand Russell & Alfred N. Whitehead, 1910] se define “clase” como una colección de objetos a los que se aplica un concepto, y esa misma definición inspiró el término “clase” en la OO e influyó en el desarrollo de los lenguajes de programación SIMULA I (1962-65) y Simula 67 (1967). Ambos lenguajes, desarrollados en el Centro Noruego de Computación (Oslo) por Christian Nygaard y Ole-Johan Dahl, son considerados los primeros lenguajes OO; su influencia conceptual subyace en todos los actuales lenguajes OO.

Lo anterior no tiene un interés únicamente anecdótico o histórico: los conceptos matemáticos, lógicos y filosóficos permiten expresar con exactitud y sin ambigüedades lo que hoy se considera “orientación a objetos”. Además, la OO (y, por tanto, el software OO) evolucionará con el tiempo hacia caminos aún inciertos, y son los conceptos lógico-matemáticos y filosóficos los que seguirán usándose para elaborar lo que denomino “orientación a objetos no estándar” e incluso para elaborar nuevos métodos de análisis y diseño, de lejano parentesco con la OO. Si existe algo parecido a la inmortalidad, los conceptos matemáticos lo tienen: las ideas de Pitágoras pervivirán más que las obras de Cervantes, y seguirán vigentes cuando usted y yo seamos cenizas alrededor de un sol moribundo.

3.2 La orientación a objetos como metodología de desarrollo de sistemas.

3.2.1 Introducción

Aunque todavía suele asociarse la orientación a objetos a determinados lenguajes de programación (Java, C++, Smalltalk, etc.), es mayoritaria la opinión de que la OO es una **metodología** de desarrollo de sistemas (informáticos o no). La orientación a objetos puede aplicarse a la ingeniería de procesos, a la gestión empresarial, etc.

La orientación a objetos considera los sistemas como colecciones de objetos capaces de interactuar, que trabajan conjuntamente para realizar tareas. Como toda metodología, incluye actividades de **análisis** y **diseño**. En el caso de los sistemas de software, comprende también actividades de **programación**. En los siguientes subapartados veremos en qué consisten dichas actividades.

Dentro del marco general de la OO, hay muchas metodologías OO. En rigor, deberían llamarse “submetodologías”; pero ningún metodólogo que se precie usaría el prefijo “sub” para su criatura, ni siquiera cuando ésta se parece al monstruo de

Frankenstein, con despojos tomados de aquí y allá. Cada metodología OO detalla con precisión las técnicas que deben usarse en cada actividad y emplea una o más notaciones, generalmente gráficas, para describir los modelos que se van generando.

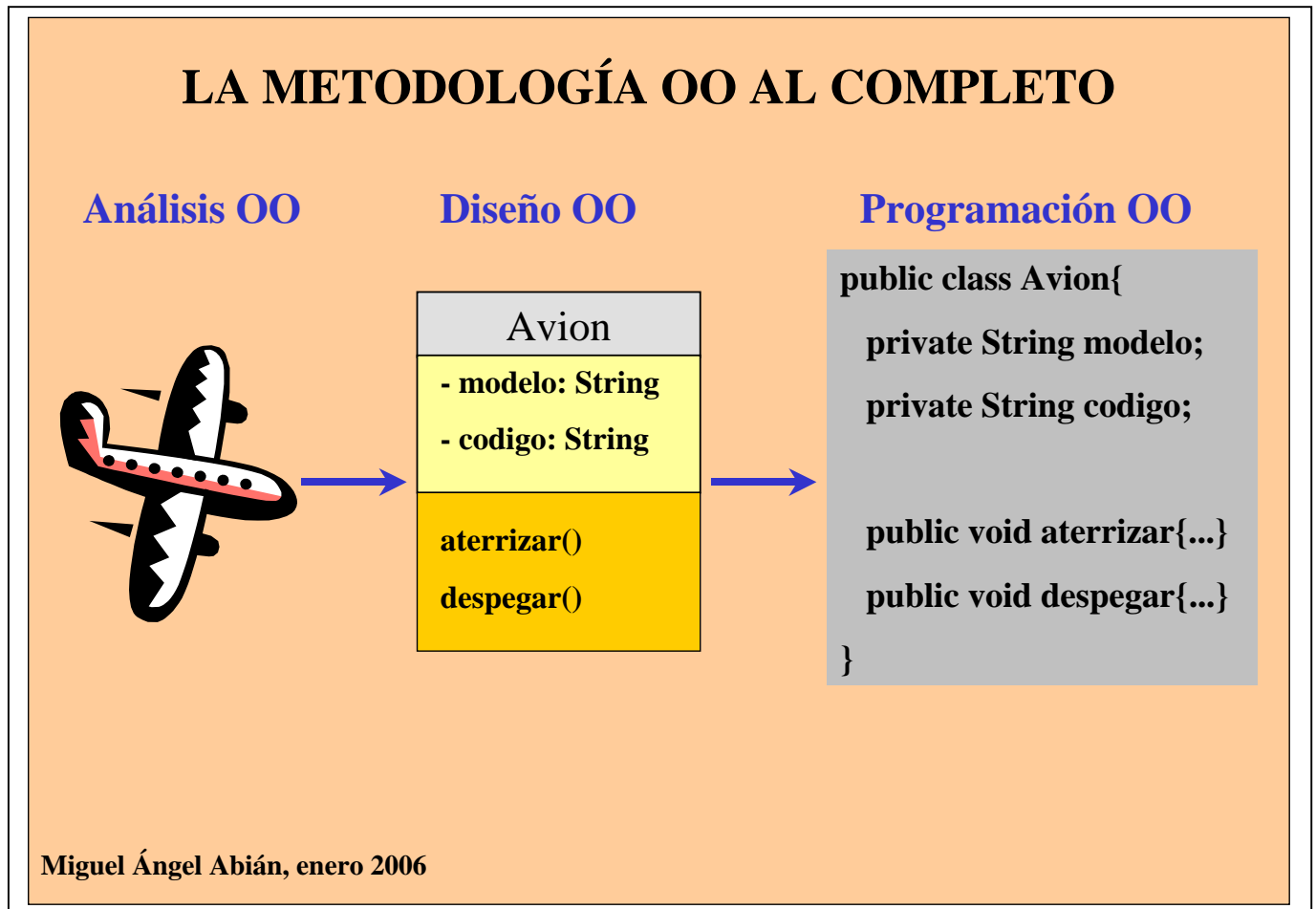


Figura 6. Las tres etapas de la OO (aplicada a sistemas de software).

Advertencia para el lector: En los apartados 15 y 16 del trabajo (están en la segunda parte: <http://www.javahispano.org/tutorials.item.action?id=33>) retomo con mucho más detalle el análisis y diseño OO, y explico los principios generales del diseño OO. Mi enfoque ha consistido en explicar primero aquí el análisis y diseño (general y OO), de una forma intuitiva. Una vez ya he explicado en la segunda parte todos los conceptos básicos de la OO (clases, objetos, polimorfismo, herencia, relaciones, mensajes) y he examinado los lenguajes OO, vuelvo allí al análisis y diseño OO desde un punto de vista más organizado y heurístico (principios de diseño OO, por ejemplo). En la medida de lo posible, he evitado las repeticiones (uso distintos ejemplos, organizo la materia de otro modo, lo que resalto en un sitio no lo hago en el otro).

El lector notará que, en el apartado 15, las clases se identifican en el diseño, no en el análisis (tal es la opinión de algunos autores). Por coherencia con lo que sigue aquí (apartado 3.2.2), cuando prepare la versión 2.0 de la segunda parte colocaré en el análisis la identificación de clases.

3.2.2 El análisis OO. Casos de uso

El **análisis** descubre y modela los aspectos relevantes de un dominio donde hay algún problema que interesa resolver; a este dominio se le llama dominio del problema o de interés. Por ejemplo, en el dominio de la física de altas energías, el problema consiste en averiguar cómo interaccionan las partículas elementales; en el dominio de una empresa, el problema podría consistir en gestionar la contabilidad.

En general, el análisis parte de una definición del problema, casi siempre imprecisa y ambigua, y genera dos cosas: a) una descripción precisa y exacta del problema; b) un modelo preciso, conciso, comprensible y exacto del dominio del problema.

En el campo de la ingeniería, el problema suele consistir en crear un sistema (eléctrico, mecánico, informático, arquitectónico, etc.) que satisfaga las necesidades de clientes y usuarios: gestionar un almacén, levantar un muro, evitar las sobrecargas de tensión, mejorar la eficacia de un motor... No resulta infrecuente que el problema consista en sustituir un sistema ya existente por otro. En ambos casos, el dominio del problema coincide con la parte del mundo real para la cual se desarrolla el sistema (un departamento, una empresa...).

Dentro del análisis, el **análisis de requisitos** se encarga de descubrir los requisitos que el sistema debe cumplir: “La aplicación permitirá registrar los envíos que lleguen”, “El muro deberá tener una sección máxima de 0,5 metros cuadrados”, “El circuito impedirá el paso de cualquier corriente con más de 100 miliamperios”, “El motor no disipará más del 40% de la energía que recibe”...

Por medio del análisis de requisitos, el analista descubre y formula precisamente los requisitos que debe cumplir un sistema para satisfacer las necesidades de clientes y usuarios. Partiendo de la descripción del problema proporcionada por éstos, obtiene una lista de requisitos para el sistema. Los requisitos no corresponden a una propuesta de solución: reflejan lo que el sistema debe hacer, pero no cómo debe hacerlo. El análisis de requisitos se revela crucial para establecer lo que realmente se pide al sistema: las descripciones de clientes y usuarios suelen ser incompletas, ambiguas e incluso inconsistentes. Al lector interesado en saber más sobre el análisis de requisitos le recomiendo *Requirements Engineering* [Ian Sommerville & Pete Sawyer, 1997].

Una vez se han obtenido los requisitos del sistema, el analista los usa (junto con su conocimiento del mundo real y sus entrevistas con especialistas en el dominio) para fabricar un modelo conceptual del dominio del problema, es decir, una representación de los conceptos más relevantes del dominio para el que se desarrolla el sistema.

A partir de la descripción del problema (requisitos), del conocimiento de los expertos en el dominio y del conocimiento general sobre el mundo real, el **análisis OO** describe el dominio del problema mediante clases y objetos. En otras palabras, construye un modelo de dominio orientado a objetos. En la figura 2 se muestra un modelo de dominio OO; en la figura 1 hay un modelo de dominio no orientado a objetos.

El análisis de requisitos y el OO están fuertemente vinculados (véase la figura 7): el primero describe el sistema deseado (el problema) mediante una lista de requisitos; el segundo usa los requisitos, junto con el conocimiento mencionado en el párrafo anterior, para generar un modelo de dominio OO, correspondiente al dominio del problema.

Un modelo de dominio se representa mediante diagramas de clases, diagramas de objetos, o mediante ambos. Los primeros presentan de forma estática las clases del dominio y sus relaciones; los segundos muestran interacciones entre objetos concretos. Si los términos del modelo son poco habituales (por ejemplo, términos médicos), el modelo de dominio suele incluir un breve glosario con todos ellos.

Las etapas del análisis OO son éstas:

- 1) Identificación de las clases del dominio. Más adelante veremos dos técnicas para identificarlas.
- 2) Elaboración del glosario de términos procedentes del dominio (esta etapa suele omitirse si los términos son de uso común).
- 3) Identificación de las relaciones o asociaciones entre las clases.
- 4) Identificación de los atributos o propiedades de las clases. Desde la perspectiva del análisis, un atributo de una clase indica que todos los objetos de esa clase tienen ese atributo.
- 5) Organización de las clases (mediante jerarquías).
- 6) Perfeccionamiento del modelo obtenido.

El modelo obtenido por el análisis OO se expresa en el lenguaje del cliente y del usuario, no depende de ningún entorno informático y no considera las restricciones de implementación. Sobre estas restricciones, llamadas **requisitos no funcionales**, volveré más adelante. Las clases del análisis son clases conceptuales, no de software.

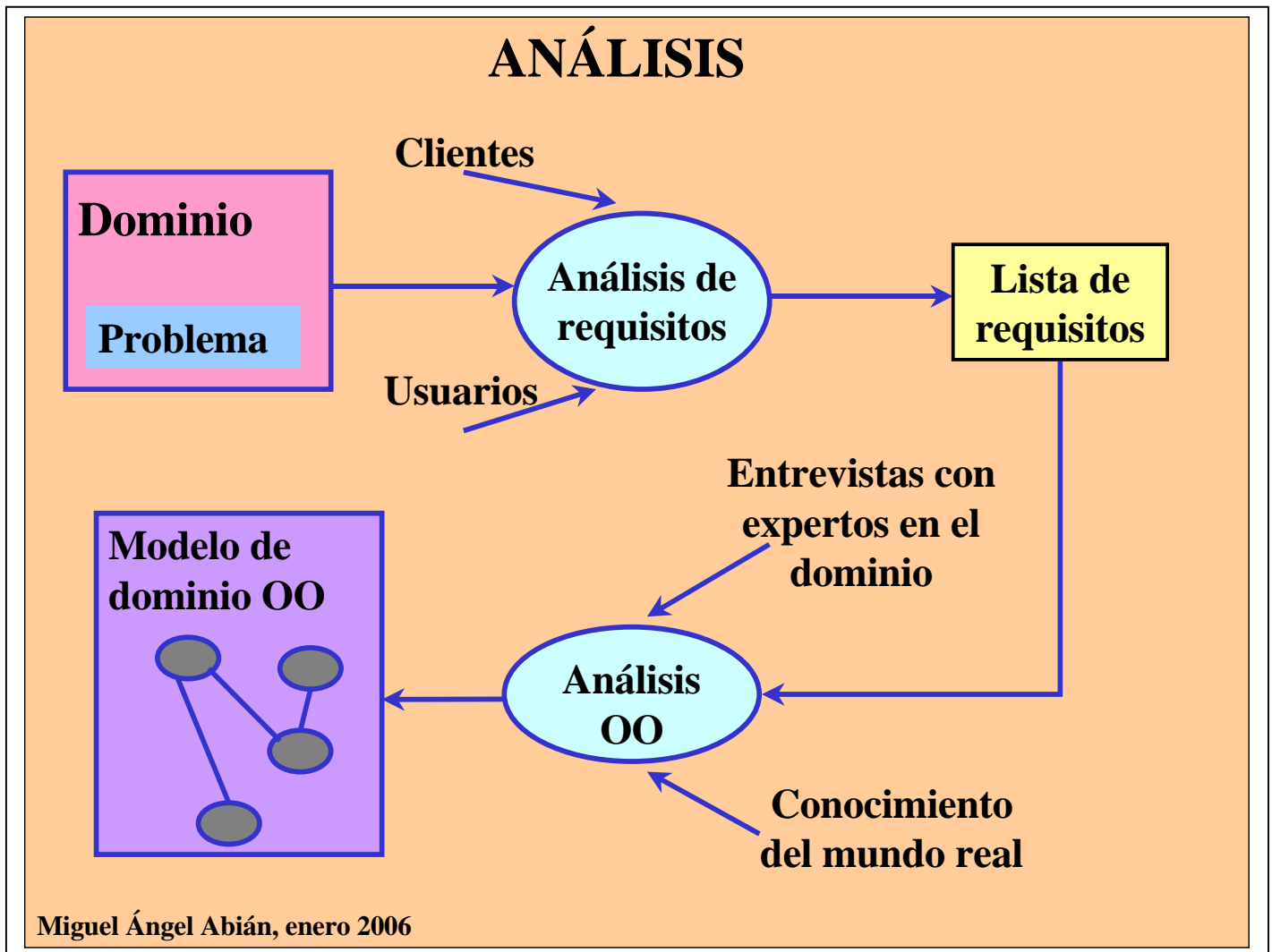


Figura 7. En ingeniería del software, el proceso de análisis corresponde a esta figura.

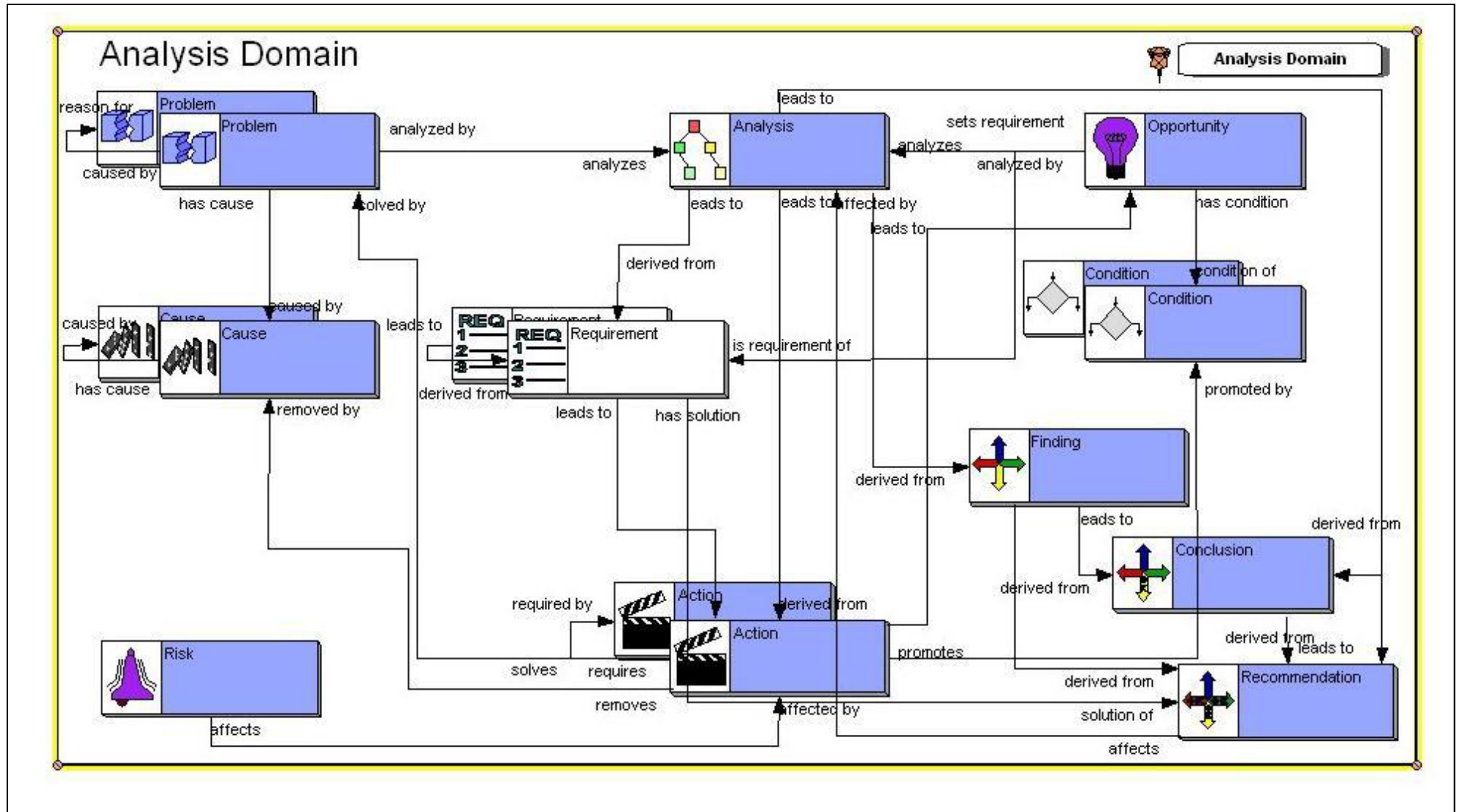


Figura 8. Metamodelo del análisis de problemas empresariales: se usan los conceptos del análisis para describir el análisis. Aparecen algunos conceptos no mencionados antes (riesgo, recomendación, oportunidad) porque este metamodelo corresponde a problemas empresariales, no de software. El diagrama se ha realizado con la herramienta Metis 3.6.

Nota: Aunque los términos “diagrama” y “modelo” se usan casi siempre como sinónimos (yo incurro a menudo en esa identificación), no significan exactamente lo mismo: un diagrama representa, a menudo parcialmente, un modelo. Un modelo puede describirse con varios diagramas y, a la vez, varios diagramas pueden representar un mismo modelo (siempre que sean consistentes).

La diferencia de significado resulta importante si se trabaja con UML, pues éste especifica que sólo hay un modelo de clases (incluido en el “modelo estructural estático”), que puede describirse con varios diagramas de clases. Por lo tanto, aunque en UML una clase sólo figura una vez en el modelo de clases, puede aparecer en varios diagramas de clases.

Para identificar las clases conceptuales de un dominio, suelen usarse dos técnicas: **la identificación de sustantivos** y **la comparación con una lista de categorías de clases**.

La técnica de la identificación de sustantivos extrae los sustantivos (nombres y grupos nominales) que aparecen en la descripción del problema y considera que corresponden a clases candidatas.

La técnica de la comparación con una lista de categorías de clases determina las clases candidatas de un dominio basándose en una lista de categorías de clases como la mostrada en la tabla 2, que procede de *APPLYING UML AND PATTERNS: An Introduction to Object-Oriented Analysis and Design and the Unified Process, Second Edition* [Craig Larman, 2002]. Asignando los conceptos del problema a las categorías de clases, se obtiene una lista de clases candidatas.

Ambas técnicas exigen un proceso de depuración, que se acelera con la experiencia del analista. Dada una lista de sustantivos o clases candidatas, convendrá aplicar las siguientes reglas de eliminación:

- 1) **Redundancia.** Si varios sustantivos se refieren a la misma entidad, se debe elegir uno de ellos (el más representativo). Por ejemplo, no tiene sentido mantener tres clases como *Trabajador*, *Empleado* y *Asalariado*.
- 2) **Atributo.** Los sustantivos que describen la estructura de las clases corresponden a atributos, no a clases. Por ejemplo, el código de un libro es un atributo de una clase *Libro*, no una clase por sí misma.
- 3) **Irrelevancia.** Los sustantivos sin relación con el problema o el dominio no son relevantes. Por caso, las clases candidatas *Mostrador del Hospital* y *Máquina de Café* resultan irrelevantes si se está elaborando una aplicación para un hospital. Siempre debe evitarse la introducción de clases no asociadas a los conceptos del dominio bajo estudio.
- 4) **Acción.** Los sustantivos correspondientes a acciones no originan clases. Por ejemplo, la clase candidata *Cálculo del IVA* no es una clase. En todo caso, *calcularIVA* sería una operación de alguna clase.
- 5) **Estado.** Los sustantivos que describen el estado de una entidad no corresponden a clases. Por ejemplo, *Automóvil Veloz* no es una clase (la clase es *Automóvil*).

- 6) Frecuencia temporal.** Los sustantivos que describen frecuencias de tiempo no corresponden a clases. Por ejemplo, en la frase “Al cliente se le informa de su saldo cada semana”, *Semana* no es una clase.
- 7) Entidad de hardware o de software.** Los sustantivos que describen entidades de hardware o de software no generan clases (salvo que se modele un sistema de hardware o de software). Por ejemplo, en el dominio de una empresa, aunque haya un requisito como “El cliente seleccionará mediante el teclado el producto que desea”, *Teclado* no será una clase. Sin embargo, una clase candidata *CPU* será clase si el dominio corresponde a componentes de hardware o a sistemas operativos.

La identificación de las relaciones entre las clases de un dominio se realiza a partir de las frases verbales presentes en la descripción del problema y de nuestro conocimiento de éste. Las relaciones pueden aparecer explícitas (“El usuario *posee* una tarjeta de crédito”) o implícitas (“asiento de avión” significa que el avión *contiene* asientos; “itinerario de vuelo” significa que un vuelo *sigue* un itinerario”).

Categoría de clase	Ejemplos
Objetos tangibles o físicos	Registro, Avión
Especificaciones, diseños y descripciones de las cosas	Especificación del Producto Descripción del Vuelo
Lugares	Tienda
Transacciones	Venta, Pago, Reserva
Líneas de las transacciones	Línea de Pedido
Papeles desempañados por las personas	Cajero, Piloto
Contenedores de otras cosas	Tienda, Lata, Avión
Cosas dentro de un contenedor	Artículo, Pasajero
Otros sistemas informáticos o electromecánicos externos al sistema	Sistema de Autorización del Pago por Tarjeta de Crédito Control de Tráfico Aéreo
Conceptos abstractos	Ansia Acrofobia
Organizaciones	Departamento de Ventas
Hechos	Venta, Pago, Reunión, Vuelo, Colisión, Aterrizaje
Reglas y políticas	Política de Reintegro Política de Cancelación
Catálogos	Catálogo de Productos Catálogo de Piezas
Registro financieros, laborales, contratos y asuntos legales	Recibo, Contrato de Empleo, Registro de Mantenimiento
Manuales, documentos, artículos y libros	Manual del Programa, Manual de Reparaciones

Tabla 2. Lista de categorías de clases. Se utiliza para proponer clases candidatas para un dominio (los ejemplos corresponden a los dominios de las tiendas y las reservas de vuelo). La tabla se ha traducido del libro de Craig Larman *APPLYING UML AND PATTERNS, Second Edition*.

En lugar de continuar hablando de análisis, dominios, modelos y demás abstracciones, prefiero mostrarlas con un ejemplo. Se basa en mis estancias veraniegas, cuando niño, en la pequeña biblioteca infantil de Soria que está junto a la rosaleda de la Alameda de Cervantes (<http://www.ayto-soria.org/html/laciudad/rutas/>). (Mi tía y mi abuela veían pasar por allí a Don Antonio Machado, siempre tras la silla de ruedas en la que iba su mujer, Doña Leonor, tísica e impedida.) El ejemplo dice así:

Una pequeña biblioteca infantil necesita un sistema informático para gestionar los préstamos de libros (cada libro tiene un código único) y periódicos. El sistema controlará los préstamos y permitirá buscar usuarios. Los socios de la biblioteca pueden sacar en préstamo hasta 3 libros, durante un tiempo máximo de 15 días (no pueden sacar periódicos). Los dos trabajadores de la biblioteca pueden, sin ser socios, sacar hasta 6 libros (por un máximo de 15 días) y 1 periódico (por un máximo de 1 día). Por motivos legales (Ley de Protección de Datos) no se conservará información sobre los libros sacados por cada lector cuando se hayan devueltos. Dadas las magras subvenciones que reciben las bibliotecas municipales –pese al extraordinario trabajo que desempeñan y a la amenaza del pago de un canon por sus libros (<http://www.derecho-internet.org/node/282>)–, el sistema deberá ser barato y podrá ejecutarse en un Pentium II.

Como puede suponer, ni la Ley de Protección de Datos (oficialmente, en España es la Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de Carácter Personal) ni el canon de libros existían cuando yo andaba con pantalones cortos: permítame estas pequeñas libertades cronológicas.

En este caso, el dominio del problema es la biblioteca; los expertos en el dominio, los trabajadores de la biblioteca. Tras hablar con ellos, averiguamos que desean sobre todo que el sistema registre los préstamos y devoluciones de libros y periódicos. La política de sanciones consiste en que, si el socio o el trabajador de la biblioteca no devuelve los libros o los periódicos en la fecha fijada, se le castiga con un día de sanción por libro/periódico y día de retraso; durante los días de sanción, no se pueden sacar libros ni periódicos. También descubrimos que los ejemplares de un mismo libro tienen códigos correlativos, y que sólo reciben un ejemplar de cada periódico. Por último, averiguamos que para hacerse socio se precisa una fotocopia del DNI, dos fotos actuales (no valen fotocopias en color) y rellenar una ficha con los datos personales (nombre, apellidos, dirección, teléfono); en el carné de socio figura un número de socio (único) y los datos personales de la ficha. Que el sistema sea barato y se ejecute en un Pentium II es un requisito de implementación, así que queda fuera del análisis OO.

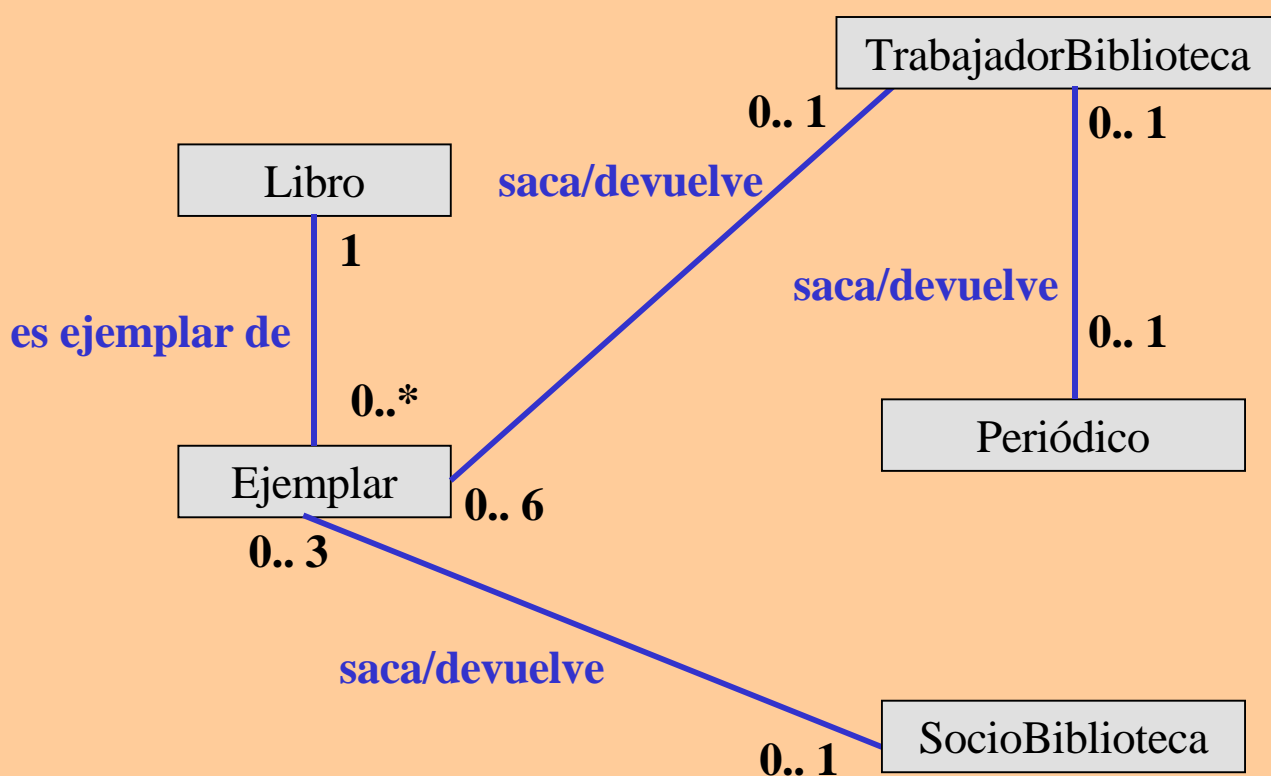
Tras estudiar la descripción del problema, aplicar una de las dos técnicas de identificación de clases y utilizar las reglas de eliminación de clases candidatas, extraemos clases como *SocioBiblioteca*, *TrabajadorBiblioteca*, *Libro*, *Ejemplar* (un libro puede tener varios ejemplares) y *Periódico*. Se podrían conservar las clases candidatas *Sanción* y *Préstamo*, pero consideramos que la información que contienen se encuentra en *SocioBiblioteca* y *Ejemplar*. Cada clase tiene sus correspondientes atributos. Por ejemplo, los atributos de *SocioBiblioteca* son *DNI*, *Nombre*, *Dirección*, *Teléfono* y *NúmeroSocio*. Las relaciones entre esas clases las obtenemos a partir de los siguientes hechos, extraídos de la descripción y de nuestro conocimiento sobre bibliotecas:

- Los libros tienen ejemplares.
- Los socios de la biblioteca sacan en préstamo ejemplares.
- Los socios de la biblioteca devuelven ejemplares.
- Los trabajadores de la biblioteca sacan en préstamo periódicos y ejemplares.
- Los trabajadores de la biblioteca devuelven periódicos y ejemplares.

Para el dominio de la biblioteca, el modelo de análisis (o conceptual) OO se representa con el diagrama de clases que aparece más abajo. El modelo de análisis muestra cómo se relacionan entre sí conceptos como “socio de la biblioteca”, “periódico”, “trabajador de la biblioteca”, “libro” y “ejemplar”. Los modelos de análisis no son únicos: a un mismo dominio le pueden corresponder varios modelos válidos. Dado un dominio, no existe un modelo de análisis universal, perfecto o indiscutible (eso sí, hay modelos más útiles y completos que otros). Por ejemplo, en el caso de la biblioteca, podríamos haber creado un modelo de análisis donde existiera una clase *Préstamo* (clase de asociación) para la relación *saca* entre *Libro* y *SocioBiblioteca*.

La **multiplicidad** indica cuántos objetos de una clase pueden vincularse, a través de una asociación, a un objeto de la clase asociada. El símbolo 0..* indica una multiplicidad de cero o varios. Un objeto *Libro* puede tener asociados cero o más objetos *Ejemplar* a través de la asociación *es ejemplar de*. La multiplicidad 0 correspondería al caso en que hubieran desaparecido todos los ejemplares de un libro (por extravío o hurto, p. ej.). Vista la asociación *es ejemplar de* desde el lado *Libro*, la multiplicidad 1 indica que todo *Ejemplar* corresponde a un *Libro*, y sólo a uno. La multiplicidad en la relación *saca* entre las clases *Ejemplar* y *TrabajadorBiblioteca* describe que un trabajador puede tener sacados, en un momento dado, hasta 6 ejemplares y que un ejemplar puede estar sacado, en un momento dado, por un trabajador o por ninguno.

MODELO DE DOMINIO DE LA BIBLIOTECA



Miguel Ángel Abián, enero 2006

Figura 8. Modelo de dominio producido al aplicar el análisis OO a la biblioteca. En este caso, el modelo consta de un solo diagrama de clases. Por motivos de espacio, omito los atributos de las clases.

El ejemplo que he escogido es deliberadamente simple: el análisis de un problema medianamente complejo requiere muchas entrevistas con los usuarios; y los modelos de análisis OO tienen decenas o centenares de clases, lo que hace necesario emplear varios diagramas de clases. Así y todo, ejemplifica bien en qué consiste el análisis OO.

Muchas metodologías OO (RUP y UP, por ejemplo) usan **casos de uso** para especificar los requisitos del sistema que se quiere construir. Un caso de uso es una narración que describe los pasos que el sistema realiza para dar a un usuario, sea persona u otro sistema, un resultado de interés. Los casos de uso “capturan” el comportamiento del sistema, sin entrar en detalles de implementación. Consideremos, vaya por caso, un caso de uso correspondiente a la aplicación de la biblioteca:

CASO DE USO: SACAR EN PRÉSTAMO LIBROS

- 1) Un socio de la biblioteca llega al mostrador del bibliotecario, donde deposita su carné de socio y varios ejemplares que desea sacar en préstamo.
- 2) El bibliotecario introduce en el sistema el número de socio que figura en el carné.
- 3) El sistema verifica si el número de socio es válido. En ese caso, comprueba si tiene ejemplares en préstamo y si está sancionado.
- 4) El sistema muestra los datos personales del socio y emite un aviso si tiene más de tres ejemplares prestados o si está sancionado. Si es así, el socio no puede llevarse nada en préstamo.
- 5) El bibliotecario introduce en el sistema los códigos de los ejemplares. Cuando termina, informa al sistema de que ha acabado con los préstamos al socio en curso.
- 6) El sistema registra los ejemplares como prestados al socio.
- 7) El bibliotecario entrega los ejemplares al socio.

El caso de uso anterior permite extraer algunos **requisitos funcionales** del sistema (por ejemplo, “El sistema emite un aviso si el usuario tiene más de tres ejemplares prestados o si está sancionado” y “El sistema registra los ejemplares como prestados al socio”). Los requisitos funcionales son aquellos que expresan, desde la perspectiva de un usuario, qué debería hacer el sistema. Por ejemplo, un requisito funcional bastante típico es “El sistema generará listados ordenados por orden alfabético”. Los **requisitos no funcionales (o técnicos)** definen o restringen cómo se implementa el sistema; es decir, especifican cómo hay que construir internamente el sistema para cumplir los requisitos funcionales. Los requisitos no funcionales pueden ser de muchos tipos: concurrencia, sistema operativo, memoria disponible, velocidad, persistencia, distribución del sistema... Veamos varios requisitos no funcionales bastante comunes:

- La aplicación funcionará con menos de 2 MB de memoria RAM.
- El sistema usará TCP/IP para enviar y recibir mensajes.
- El sistema se ejecutará en una estación de Sun con el sistema operativo Solaris 8.0.
- La aplicación se programará en un lenguaje OO.
- Los datos se guardarán en Oracle.

Los casos de uso suelen usarse, además de para capturar requisitos funcionales, para identificar clases conceptuales del dominio (en el caso de uso anterior podríamos extraer clases como *Ejemplar* y *SocioBiblioteca*). Las dos técnicas que vimos antes para identificar clases se pueden emplear con los casos de uso.

Los casos de uso no están orientados a objetos: expresan las funciones del sistema (requisitos funcionales) sin emplear objetos y clases. Si los casos de uso fueran una técnica OO, los ejemplares del paso 6 se registrarían a sí mismos como prestados cuando recibieran un mensaje de algún objeto. No obstante, son una técnica muy habitual y útil para extraer las clases conceptuales del dominio.

Debo señalar que unos pocos autores expresan sus dudas sobre la eficacia de los casos de uso en el análisis OO. Por ejemplo, Meyer escribe en *Object-Oriented Software Construction, Second Edition* [Bertrand Meyer, 1997]:

Excepto en el caso de un equipo con mucha experiencia en diseño (que haya construido con éxito sistemas de varios miles de clases, cada uno en un lenguaje OO puro), no confíe en los casos de uso como una herramienta para el análisis y diseño orientado a objetos.

En mi opinión, resulta adecuado trabajar con casos de uso si se quiere identificar clases y objetos; pero uno no debe confiar sólo en ellos: en el análisis OO hay que prestar atención también a los requisitos no expresados en forma de casos de uso, y a las conversaciones con expertos, clientes y usuarios. Con los casos de uso, se hace muy difícil saber cuándo se han descubierto todas las clases del dominio. Las clases conceptuales de un dominio, que provienen del análisis OO, siempre son más estables que los casos de uso, puesto que las funciones de un sistema suelen cambiar.

3.2.3 El diseño OO

El **diseño** es un proceso que usa los resultados del análisis para generar una especificación que implemente un sistema o un producto. El análisis trabaja en el “espacio del problema”; el diseño, en el “espacio de la solución”. Durante la etapa de análisis, lo fundamental es *qué* necesita hacer el sistema; cuando se aborda el diseño, lo importante es *cómo* construirlo.

En el caso del software, el diseño genera –dado un entorno informático y una lista de requisitos– un modelo de software (modelo de diseño) que detalla cómo debe hacer las cosas el sistema para cumplir los requisitos y, en definitiva, solucionar el problema que se plantea en el dominio de interés.

En el diseño se consideran las restricciones derivadas de la implementación (requisitos no funcionales o técnicos). Por tanto, los modelos de diseño dependen del entorno informático –ordenadores personales, estaciones de trabajo, entornos distribuidos, agendas electrónicas, sistemas operativos, lenguajes de programación– donde se vaya a implementar el sistema. En el diseño hay que contestar preguntas como éstas:

- ¿Cómo se puede distribuir entre varios ordenadores la aplicación, con el fin de que se distribuyan equitativamente las peticiones que recibe cada ordenador?
- ¿Qué componente de acceso a bases de datos es más conveniente para la aplicación?

- ¿Cómo puede ejecutarse este método en un tiempo máximo de 30 milisegundos?
- ¿Cómo puede utilizarse más eficazmente Java en la aplicación?

El **diseño OO** usa los resultados del análisis OO y del análisis de requisitos para producir un modelo de diseño basado en clases y objetos **de software**; a diferencia de lo que sucede en el análisis OO, los objetos y clases del diseño OO no modelan entidades del mundo real, sino de software.

La diferencia fundamental entre análisis y diseño OO es el nivel de detalle: el diseño refina las clases conceptuales del análisis hasta convertirlas en clases de software implementables en un lenguaje OO.

Durante el diseño OO, hay que completar pasos como éstos:

- a) Representación de las clases.** Para cada clase, hay que decidir si se representa mediante tipos primitivos (*int*, *float*, *double*...) o mediante otras clases más simples. Por ejemplo, un objeto *Línea* puede representarse mediante cuatro datos de tipo *float* o *double* (*x1*, *y1*, *x2*, *y2*) o mediante dos objetos *Punto*.
- b) Diseño de los algoritmos que implementen las operaciones de las clases.** Para ello hay que a) elegir los algoritmos, considerando factores como la complejidad computacional, la flexibilidad, la facilidad de implementación y la legibilidad; b) seleccionar las estructuras de datos pertinentes para los algoritmos (pilas, colas, árboles, etc.); y c) definir las clases y operaciones internas que se necesitarán para almacenar los datos intermedios generados por los algoritmos.
- c) Reorganización de las clases y operaciones para aumentar, si es posible, la herencia.** Véase 4.4 para una introducción a la herencia.
- d) Diseño de las asociaciones entre clases.** En la fase de diseño hay que establecer cómo se implementarán las asociaciones o relaciones (punteros, conjuntos de punteros, diccionarios...).

Todo modelo de diseño OO especifica lo siguiente:

- Los tipos de objetos (clases) que necesita el sistema para resolver el problema. Deben especificarse sus atributos, operaciones y relaciones.
- Las interacciones entre los objetos (instancias), las cuales cambian con el tiempo. Por ejemplo, un objeto *Mecánico* puede estar arreglando un objeto *Automóvil* y luego puede preguntar a un objeto *Cliente* sobre la forma de pago que desea.

El primer punto corresponde al modelo estático; el segundo, al dinámico.

Nota: Tal como he escrito al principio de este subapartado, el diseño OO toma los resultados del análisis OO con el fin de generar un modelo lo bastante detallado como para ser implementado en un lenguaje OO. Ahora bien, en la práctica, la separación entre análisis y diseño OO no siempre resulta tan tajante: a menudo se solapan ambas actividades. Abordaré las ventajas de ese solapamiento en 3.3.

La frontera entre análisis y diseño OO es sumamente borrosa en las metodologías iterativas de desarrollo de software. En una metodología iterativa, el desarrollo se estructura en una serie de pequeños proyectos cortos, de duración fija (2-3 semanas, por ejemplo). Estos proyectos se llaman iteraciones; cada iteración produce un sistema o subsistema que puede probarse, ejecutarse e integrarse en un sistema mayor. Como cada iteración tiene sus tareas de análisis, diseño e implementación, en cada una se mezcla el análisis con el diseño.

En estas metodologías también se mezcla el diseño con la implementación, lo cual no es un inconveniente, más bien al contrario; pues la implementación desarrollada en una iteración se usa para encontrar errores en el diseño, que se corrigen en la siguiente iteración.

Por regla general, cuanto más grande es el proyecto de software que se aborda, más nítida aparece la separación entre análisis y diseño, y entre diseño e implementación.

Las clases conceptuales del modelo de dominio generado por el análisis OO suelen continuar en el diseño OO como **clases de entidad** (clases de software que contienen información persistente); también las relaciones entre las clases del modelo de dominio OO suelen permanecer en el diseño OO. Además de las clases de entidad, en el diseño OO hay que añadir a menudo **clases de interfaz** (clases de software que modelan la relación entre el sistema y los usuarios externos, ya sean éstos personas o sistemas) y **clases de control** (clases de software que se usan para representar la coordinación entre clases).

Las clases de interfaz suelen ser abstracciones de ventanas, formularios, paneles, sensores, teclados, ratones, impresoras, tarjetas de red...

Las clases de control suelen encapsular el control de un caso de uso o el sistema completo; especifican qué mensajes deben intercambiar los objetos, y en qué orden, para que el sistema cumpla una o más funciones. Por ejemplo, al caso de uso “Sacar en préstamo libros” se le podría asignar una clase de control *SacarLibro* que coordinara las acciones que deben llevar a cabo los objetos *Libro*, *Ejemplar* y *SocioBiblioteca* para que el usuario pueda llevarse en préstamo un libro. La clase *Biblioteca* de la figura 10 es una clase de control: representa el sistema informático de la biblioteca.

Además de las clases de interfaz y de control, un diseño OO suele necesitar clases definidas “de serie” en el lenguaje donde vaya a implementarse el diseño. Por ejemplo, los objetos *Libro* de la biblioteca se podrían guardar en una clase *Vector* o *ArrayList* de Java, clases ausentes en el modelo de análisis. Estas clases se emplean para implementar colecciones de objetos en la memoria de un ordenador o computador y, por ende, sus propiedades no derivan del dominio del problema ni del mundo real.

A veces, en el análisis y diseño OO se usan las **tarjetas CRC** (Control-Responsabilidad-Colaborador). Estas tarjetas son pequeños trozos de papel o cartulina donde se describen las clases del sistema que se desea construir. Cada tarjeta incluye el nombre de una clase, las responsabilidades de ésta y las clases con las que colabora. Una responsabilidad es una descripción breve de lo que hace la clase; y corresponde a algo que la clase necesita conocer (p. ej., números de teléfono) o a una acción que debe ejecutar (p. ej., dar de alta a un cliente). En caso de duda sobre si algo es o no una responsabilidad, se aplica el criterio de que una responsabilidad debe usarse en otras partes del sistema.

Una tarjeta CRC debe contener más de una responsabilidad (en caso contrario, habría que rechazar la clase que contiene) y no más de tres o cuatro responsabilidades (si no es así, habría que repartir las responsabilidades de la clase entre más clases). Las clases colaboradoras son aquellas que ayudan a una clase a cumplir sus responsabilidades. Cada clase colaboradora tiene su propia tarjeta CRC.

La principal función de las tarjetas CRC consiste en establecer cómo se coordinan las clases del sistema para cumplir los requisitos. Estas tarjetas ofrecen la ventaja añadida de que la información sobre colaboraciones entre clases puede enseguida modelarse mediante diagramas UML de secuencia y de colaboración.

Las tarjetas CRC se usan normalmente en sesiones donde participan analistas, usuarios y desarrolladores; son una buena manera de romper el hielo con la gente sin experiencia en la OO. El libro *The CRC Card Book* [David Bellin & Susan S. Simone, 1997] es la mejor referencia para esta técnica.

EJEMPLO DE TARJETA CRC

Nombre de la clase: SocioBiblioteca

Responsabilidades:

Almacena los datos personales del socio

Almacena la información sobre los préstamos en vigor

Almacena información sobre sanciones

Colaboradores: Ejemplar

Miguel Ángel Abián, enero 2006

Figura 9. Ejemplo de una tarjeta CRC. Las responsabilidades de una clase y sus clases colaboradoras resultan muy útiles a los diseñadores OO. El objetivo de las tarjetas CRC no es entrar en detalles de implementación, sino capturar el propósito de la clase en unas pocas líneas. Por eso son pequeñas, para que no se pueda escribir mucho. Una tarjeta CRC no debe asignar a una clase más de tres o cuatro responsabilidades.

Además de ayudar en el análisis OO, los casos de uso resultan útiles para el diseño OO, pues las funciones representadas por los casos de uso se modelan como colaboraciones entre los objetos del sistema.

La figura siguiente muestra un diagrama de diseño correspondiente al problema de la biblioteca. Está orientado a objetos, pues muestra objetos que se envían mensajes para satisfacer los requisitos del sistema.

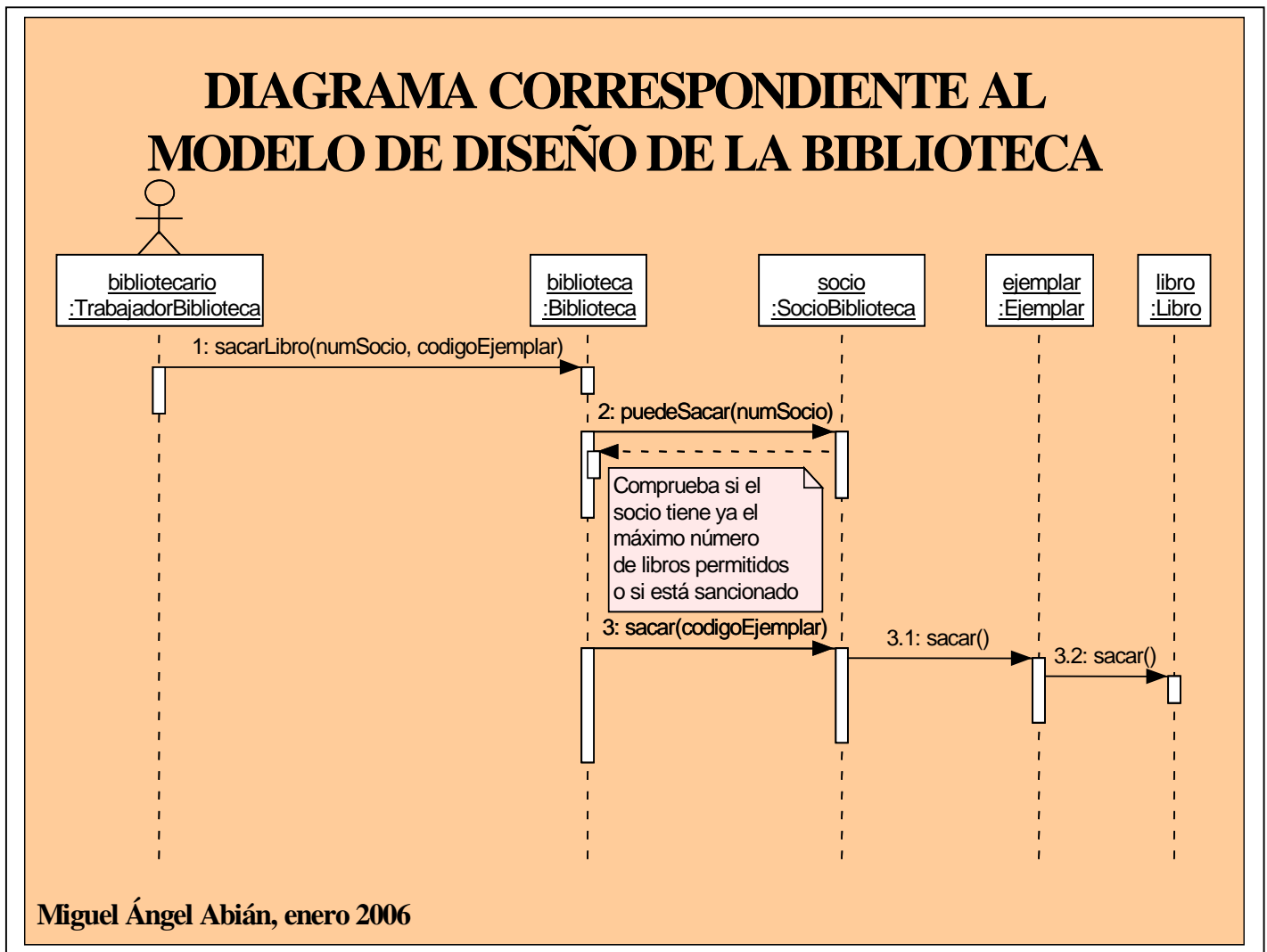


Figura 10. Diagrama parcial de un modelo de diseño OO (es un diagrama de secuencia UML). La figura modela los pasos necesarios para que un socio de la biblioteca saque en préstamo un libro. Cuando un programador experto en UML ve un diagrama de diseño como éste, tiene una idea bastante clara de cómo traducirlo a código. La clase Biblioteca es una clase de control: representa el sistema que se desea construir para gestionar la biblioteca. Este diagrama de secuencia representa sólo una pequeña parte del modelo de diseño correspondiente a la aplicación completa.

3.2.4 La programación OO

Cuando se consideran sistemas de software, la orientación a objetos consiste en el **análisis OO**, el **diseño OO** y la **programación OO**. La programación OO es el proceso que genera una aplicación a partir del diseño OO. Dicho en otras palabras, la POO materializa físicamente los diseños OO, que a su vez proceden del análisis OO. La fuerza de la POO radica en que la comprensibilidad y la facilidad de mantenimiento de los programas aumentan cuando se descomponen en clases.

El resultado de la POO es una aplicación capaz de ejecutarse en un entorno informático. A veces se dice que el resultado es el código fuente de la aplicación. Esto no es del todo cierto: sus clientes no quedarán muy satisfechos si les da un listado con el código de la aplicación o un archivo de texto con el código. En realidad, el código fuente es un modelo que usan los programadores para comunicarse entre ellos, y que puede ser traducido fácilmente (con un interprete o un compilador) a un lenguaje inteligible para los ordenadores.

Comentarios impopulares (no diga que no avisé): Conservar la documentación generada durante el análisis y diseño de un sistema (glosarios, diagramas de clases, de objetos, etc.) reduce los gastos cuando hay que modificarlo o sustituirlo. En el mundo del software, las plataformas y los lenguajes cambian mucho, pero los requisitos de los sistemas son más permanentes. Cuando hay que cambiar un sistema por otro, suele empezarse de cero (se tira toda la documentación del sistema anterior, suponiendo que la hubiera). Así se desperdicia el análisis y diseño hecho para el sistema anterior, y se sigue a pie juntillas la inmortal frase **“Parece que jamás hay tiempo ni dinero para hacerlo bien a la primera, pero siempre hay tiempo y dinero para volver a hacerlo”**.

En los sistemas empresariales –sistemas ERP, aplicaciones de gestión, de contabilidad– se calcula que, cuando se cambia un sistema por otro, se mantiene el 70-80% de los requisitos (dar de alta a clientes, registrar pedidos, guardar transacciones, comprobar pagos, etc.). En el caso de los modelos conceptuales, el 95% de las clases no cambia.

Algunas metodologías de ingeniería del software dedican poco tiempo a la documentación, pues es costosa. El resultado final es que, cuando hay que modificar sustancialmente un sistema o sustituirlo por otro, hay que empezar de cero el análisis y diseño, y el cliente debe volver a pagarlo, aunque sus requisitos apenas hayan variado. Esto no molesta, más bien al contrario, a las consultoras de software; pero sí a los clientes y a quienes intentamos mejorar la competitividad de las pequeñas y medianas empresas europeas.

De esas metodologías, algunas reducen al mínimo el análisis y diseño, y se centran en la programación. Este enfoque puede funcionar en proyectos pequeños, pero no en proyectos grandes o medianos, ni en aquellos en que se exigen características como escalabilidad o velocidad. Estas propiedades no surgen directamente de la programación, sino que deben ser tenidas en cuenta desde el principio.

3.3 Comparación de la OO con la metodología estructurada

Antes de comenzar con los fundamentos de la orientación a objetos, conviene mencionar a su oponente en el desarrollo de sistemas: la **metodología estructurada** (también conocida como funcional o algorítmica). Esta metodología toma una tarea general que se necesita llevar a cabo (como “permitir que el usuario gestione un almacén”) y la divide en subtarefas (como “retirar un pedido” y “dar entrada a los nuevos productos”). Una persona que utilice la metodología estructurada dividirá el problema bajo estudio identificando una serie de procesos, manipulaciones o tratamientos de datos (llamados funciones o procedimientos en las fases de diseño e implementación) que, organizados de modo que puedan llamarse unos a otros, proporcionen la solución. En la metodología estructurada existe siempre una separación entre datos y procesos: los procesos manipulan y usan datos, pero no se integran con ellos.

La metodología estructurada aplicada a los sistemas de software (análisis estructurado + diseño estructurado + programación estructurada) se basa en la abstracción por descomposición funcional o por procedimientos: el problema estudiado se descompone en una serie de capas sucesivas de procesos, hasta que finalmente se descompone en procesos relativamente fáciles de implementar y codificar (desarrollo *top-down*). El programa se divide en unidades lógicas (módulos) mediante el uso de funciones o procedimientos; los detalles más internos del programa residen en los módulos de más bajo nivel, y los módulos de más alto nivel se encargan del control lógico del programa.

Consideraré aquí un ejemplo de diseño estructurado. Imagine, vaya por caso, que debe escribir un programa para mostrar por pantalla un catálogo de muebles, teniendo en cuenta que las descripciones de los muebles (dibujos con las cotas correspondientes) se almacenan en una base de datos. El usuario puede seleccionar qué muebles desea ver en la pantalla. Para mostrar las descripciones de los muebles, el diseño estructurado consideraría la siguiente secuencia de pasos:

- 1) Se accede a la base de datos.
- 2) Dentro de la base de datos, se buscan los muebles que desea ver el usuario.
- 3) Se abre una lista de muebles.
- 4) Se añaden a la lista todos los muebles encontrados en el paso 2.
- 5) Se ordena la lista (por tipo de mueble, p. ej.).
- 6) Se muestra por pantalla, uno detrás de otro, el dibujo de cada mueble incluido en la lista.

Cada paso podría dividirse, a su vez, en otros. Por ejemplo, el paso 6 podría escribirse así:

- 6.1) Se lee de la base de datos la posición (x, y) de cada punto del dibujo del mueble.
- 6.2) Se llama a la función gráfica encargada de dibujar los muebles, dándole como argumento las coordenadas del punto antes leído.
- 6.3) Se repiten los pasos 6.1 y 6.2 hasta que se hayan leído todos los puntos de la descripción del mueble.

Este ejemplo muestra por qué la metodología estructurada se llama también funcional: divide el problema en pasos, cada uno con una función clara (en el ejemplo, acceder a la base de datos, buscar en ella los muebles seleccionados, ordenarlos, dibujarlos, etc.).

De una manera orientada a objetos, el problema del catálogo electrónico se solucionaría así:

- 1) El programa crea una instancia del objeto que representa a la base de datos.
- 2) El programa pide al objeto base de datos que encuentre los muebles que quiere ver el cliente.
- 3) El programa crea un objeto lista con los muebles.
- 4) El programa crea instancias de los muebles encontrados en el paso 2 y los añade al objeto lista.
- 5) El programa pide al objeto lista que se ordene (al hacerlo, ordena los objetos que contiene).
- 6) El programa pide a la lista que se muestre por pantalla.
- 7) La lista pide a cada objeto mueble dentro de ella que se muestre a sí mismo por pantalla.
- 8) Cada objeto mueble se dibuja a sí mismo en la pantalla, con las cotas correspondientes.

El uso de objetos y la asignación de responsabilidades a éstos son características inexistentes en la metodología estructurada o funcional.

La metodología estructurada resulta útil para producir pequeños programas y módulos. Muchas aplicaciones informáticas de gestión la utilizan, pues usan mayoritariamente pseudobjetos u objetos que no pueden considerarse objetos de pleno derecho (por ejemplo, objetos sin atributos o sin operaciones), generalmente asociados a las estructuras tradicionales de datos.

La principal diferencia entre la programación OO y la programación estructurada radica en sus representaciones del dominio bajo estudio. La segunda (C, Pascal) obliga al desarrollador a trabajar con abstracciones procedentes del mundo de la informática: *if/else*, *do/while*, *loop*, etc. En cambio, la POO utiliza el vocabulario del dominio; por ejemplo, si implementáramos la aplicación para gestionar la biblioteca aparecerían clases de software como *Libro*, *Ejemplar*, etc.

La conversión de los términos del análisis OO en construcciones de lenguajes de programación OO es bastante directa, lo que constituye una importante ventaja sobre la metodología estructurada. Resulta innegable que el salto conceptual entre el mundo real y la programación es menos brusco en la POO que en la programación estructurada, basada en algoritmos.

Como puede apreciarse, la POO **simula** en software las entidades del mundo real. *Simular* es la palabra exacta: los primeros lenguajes OO (SIMULA I y Simula 67) eran lenguajes de simulación. SIMULA I (1962-65) y Simula 67 (1967) surgieron con el fin de desarrollar herramientas que sirvieran para la descripción y simulación de sistemas físicos (por ejemplo, motores) y de sistemas persona-máquina. A diferencia de SIMULA I, Simula 67 era un lenguaje de programación general, si bien podía especializarse para muchos dominios, incluyendo la simulación de sistemas.

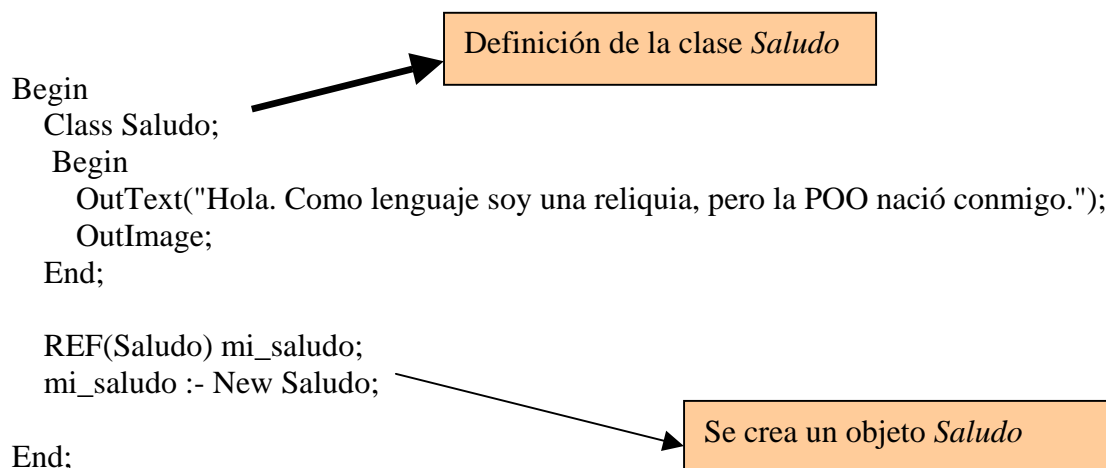
A la hora de simular sistemas físicos, los creadores de los lenguajes Simula (Ole-Johan Dahl y Kristen Nygaard, quienes ganaron en 2002 el premio Turing) se enfrentaban a dos grandes problemas. De un lado, los programas que necesitaban escribir eran muy largos y, por ende, difíciles de entender. De otro, había que modificarlos sustancialmente cuando se cambiaba el sistema bajo estudio, e incluso cuando había que probar varias simulaciones de un mismo sistema (con distintas piezas, p. ej.) para elegir la óptima.

Para solucionar ambos problemas, Dahl y Nygaard decidieron diseñar sus programas de simulación inspirándose en los objetos físicos del sistema real. Así, si en un motor había 20 componentes, el programa que lo simulaba contaba también con 20 módulos, uno por cada componente.

Esta forma de proceder resolvía los dos problemas planteados antes. Por un lado, los problemas podían descomponerse en componentes sencillos (clases), lo que aumentaba su legibilidad. Por otro, si se quería simular cómo reaccionaba el sistema físico ante el cambio de alguna pieza o componente, en el programa debía modificarse sólo un componente, no todo el programa, como sucedía en la metodología estructural. Más aún: los componentes generados para la simulación de un sistema podían usarse como “ladrillos” en simulaciones de otros sistemas.

La estrategia de asignar un objeto de software a cada elemento o pieza real parece hoy algo natural e inmediato; pero fue revolucionaria en una época en que la metodología estructurada campaba a sus anchas. Mientras que ésta generaba programas mediante llamadas sucesivas entre funciones o procedimientos, en los que los datos y las funciones permanecían separados, la naciente orientación a objetos se centraba en diseñar los programas como colaboraciones entre objetos, donde se agrupaban datos y operaciones.

Como curiosidad, muestro aquí el código correspondiente a una clase *Saludo* de Simula 67. Por mucho que creamos que Eiffel, Java o C#, o el lenguaje que uno prefiera, son la cumbre de la orientación a objetos, todas las características fundamentales de los lenguajes OO están en los lenguajes Simula.



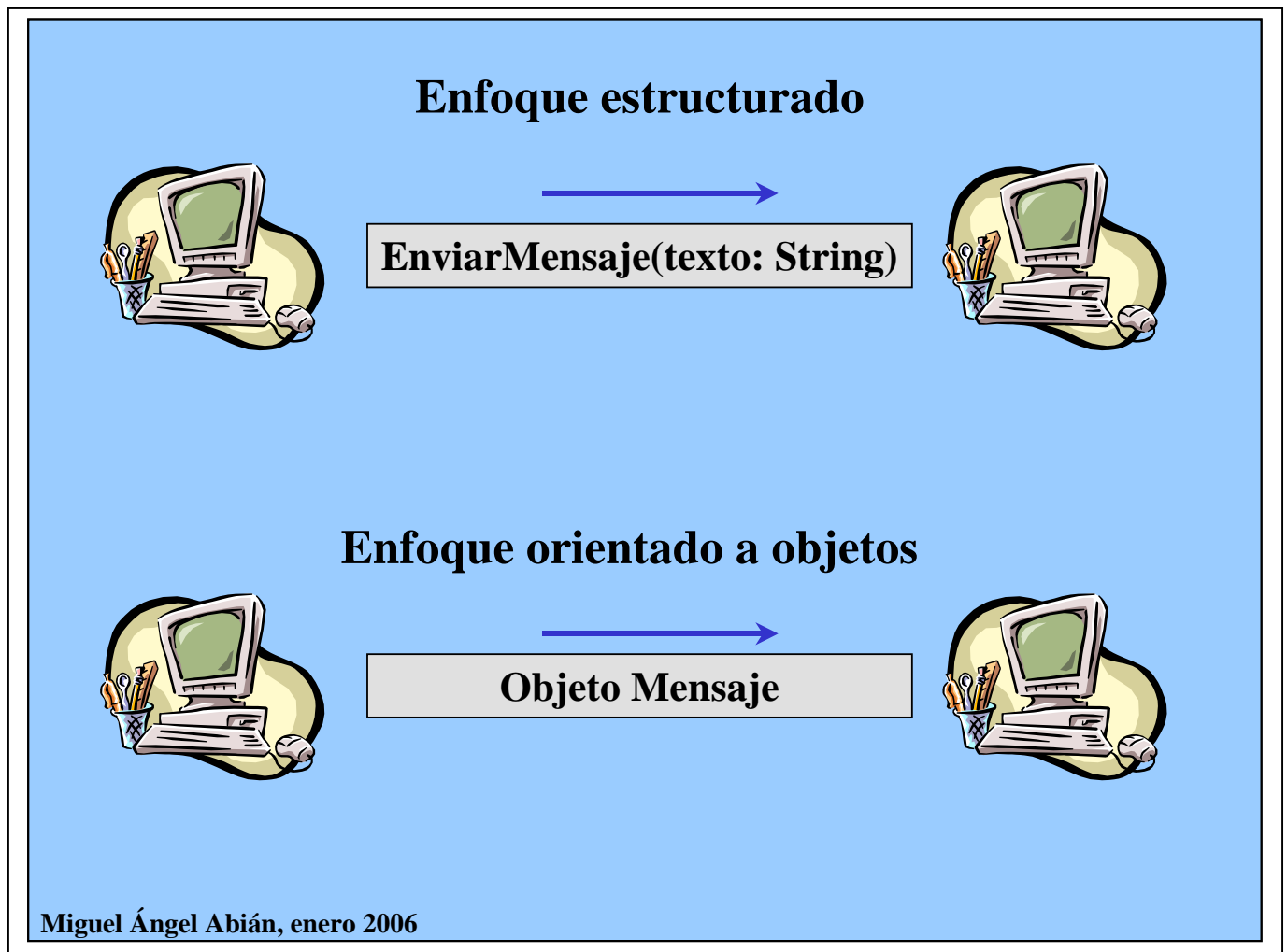


Figura 11. El dibujo muestra el envío de un mensaje de un ordenador a otro. En el enfoque estructurado, el mensaje se envía como una cadena de texto; para mostrarlo, el ordenador receptor debe saber cómo procesar el texto. En el enfoque OO, se envía un objeto Mensaje, el cual sabe qué debe hacer para mostrarse en el ordenador receptor. El objeto Mensaje lleva consigo datos (texto) y operaciones. Cuando el objeto llegue al ordenador receptor, será él mismo el que ejecute las operaciones encargadas de mostrar el texto de mensaje. En el primer caso, se envían sólo datos; en el segundo, datos y código.

En la tabla 3 se aprecia uno de los defectos más comúnmente señalados por los críticos de la metodología estructurada: la separación clara e insalvable entre el análisis y el diseño, es decir, entre lo que se quiere que haga el sistema y cómo lo hace. En la OO, la frontera entre el análisis y el diseño es más difusa (véase la nota de la página 30), y los objetos se introducen desde el principio. Cuanto más detallado es el análisis, más inmediato es el paso a la implementación, lo que reduce el tiempo dedicado al diseño. Como he mencionado hace dos páginas, la OO hace más natural pasar del análisis a la implementación que la metodología estructurada: el salto conceptual es menor. En las metodologías OO, todas o casi todas las clases obtenidas en el análisis se usan para el diseño y la implementación.

Metodología estructurada	Metodología orientada a objetos
Etapas de análisis: se determina qué debe hacer el sistema que se desea construir.	Etapas de análisis: se determina qué debe hacer el sistema que resolverá el problema y se extraen las clases y objetos del dominio del problema. Se modela el dominio del problema mediante la notación OO.
Etapas de diseño: se extraen funciones o procedimientos (muy vinculados a los datos con que se va a trabajar). Esta fase no puede omitirse ni simplificarse.	Etapas de diseño: se usa el análisis OO del problema para generar una especificación que implemente el sistema teniendo presente las restricciones impuestas por la implementación. Muchas veces, esta fase puede simplificarse si se parte de un análisis OO detallado.
Etapas de programación: se implementan las funciones en un lenguaje de programación adecuado.	Etapas de programación: se implementan los objetos en un lenguaje de programación orientado a objetos.

Tabla 3. Comparación entre el análisis, el diseño y la programación en ambas metodologías.

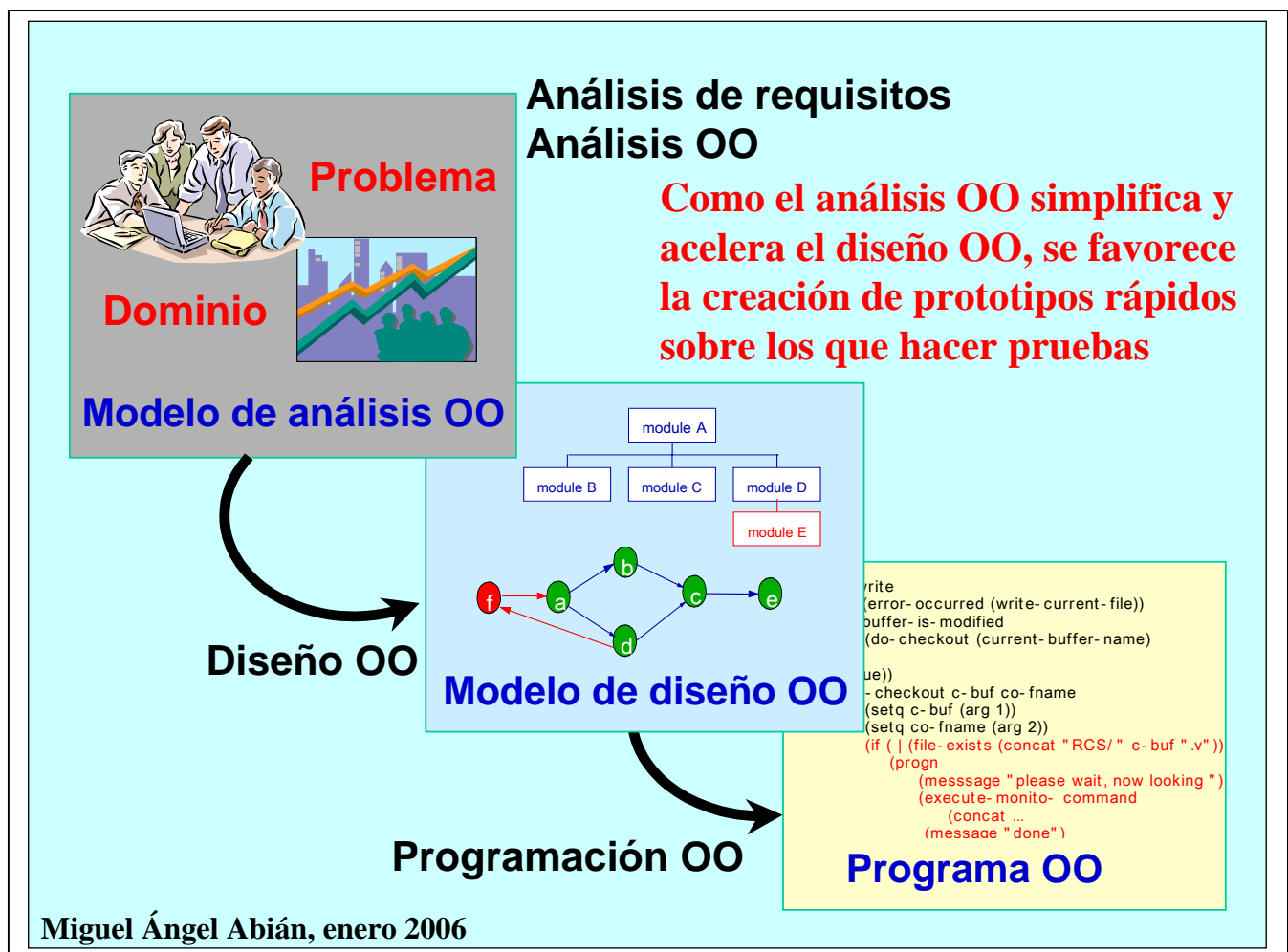


Figura 12. Muchas veces, la OO permite construir prototipos rápidos, pasando rápidamente por la etapa de diseño. En el enfoque estructurado, la etapa de diseño no puede simplificarse.

Algunos autores consideran que la OO ha evolucionado a partir de la metodología estructurada; otros, que es un salto revolucionario, cualitativo más que cuantitativo. Desde luego, los segundos suelen ser firmes partidarios de la OO. Así, en la obra *Object-Oriented Modeling and Design* [**James Rumbaugh et al, 1991**] se considera que el diseño orientado a objetos es una nueva forma de pensar en los problemas, mediante modelos sobre conceptos del mundo real; y en *Software Engineering: A Practitioner's Approach 5th edition* [**Roger S. Pressman, 2001**], Pressman refiere que

a diferencia de otros métodos de diseño, el diseño orientado a objetos da como resultado un diseño que interconecta los objetos de datos y las operaciones, de forma que modulariza la información y el procesamiento en vez de sólo la información. La naturaleza del diseño orientado a objetos está ligada a tres conceptos básicos: abstracción, modularidad y ocultación de la información.

Mi opinión, dado que no hay un acuerdo unánime, se resume en que la OO utiliza abstracciones ausentes en la metodología estructurada, que no admiten equivalencia en ésta. Nótese que hablo en términos conceptuales, no de lenguajes de programación. Teóricamente, todo lo que puede hacer un lenguaje OO podría hacerlo un lenguaje estructurado. De hecho, las primeras versiones de C++ utilizaban un preprocesador que, antes de la compilación, traducía el código fuente a C. Incluso un lenguaje OO “puro” como Eiffel permite la traducción del código Eiffel a código C.

4. Fundamentos de la orientación a objetos

La orientación a objetos se fundamenta en los siguientes principios:

- Abstracción
- Modularidad
- Encapsulación
- Jerarquía

4.1 Abstracción

La abstracción es una aproximación que hace hincapié en los aspectos más importantes de algo, sin preocuparse por los detalles menos relevantes. De acuerdo con el *Dictionary of Object Technology: The Definitive Desk Reference* [Donald Firesmith & Edward Eykholt, 1995], la abstracción es “*cualquier modelo que incluye los aspectos más importantes, esenciales o distinguibles de algo mientras suprime o ignora los detalles menos importantes, inmateriales o que pueden distraer*”. La abstracción es específica del dominio. Por ejemplo, la abstracción *Persona* es distinta en un dominio de censos electorales que en un dominio hospitalario.

Acaso el ejemplo más gráfico de lo que es una abstracción lo dé el cuadro *La Trahison des Images* (La traición de las imágenes), del pintor surrealista belga René Magritte (1898-1967). En el cuadro aparece dibujada una pipa, bajo la cual se lee la frase *Ceci n'est pas une pipe* (Esto no es una pipa), escrita con caligrafía escolar.

Efectivamente, el cuadro es una abstracción o modelo de una pipa, no una pipa (cuán sutiles eran los surrealistas). En el cuadro se han perdido muchas cosas: la tridimensionalidad de la pipa real, sus pequeñas imperfecciones, el humo, etc. Si se desea explicar a alguien qué forma tiene una pipa, el cuadro de Magritte es una buena abstracción o modelo de una pipa real. No sería una buena abstracción si, verbigracia, se quisiera fabricarla en serie. El cuadro no aporta información sobre las dimensiones o sobre la madera de la que está hecha. En la ingeniería del software y en la OO es fundamental partir de buenas abstracciones del problema que vaya a abordarse.

El código fuente de un programa es, al igual que el cuadro de Magritte, una abstracción o modelo: en lugar de representar una pipa, representa un programa. Un programa es una serie de señales eléctricas variables que se propagan en un determinado hardware, no unas líneas de código escritas en un papel o en un editor de texto. Si Magritte hubiera usado ordenadores, quizás habría pintado un cuadro en que aparecieran unas cuantas líneas de código encabezadas por el siguiente comentario: ***Esto no es un programa.***

En la orientación a objetos, las clases (de objetos) son la abstracción fundamental. Mediante la abstracción, se parte de unos pocos casos concretos (objetos) de entidades reales o conceptuales y se generan clases. Como hemos visto en 3.2.2, el uso de la abstracción durante el análisis OO consiste en estudiar los conceptos fundamentales del dominio bajo estudio, obviando temporalmente las decisiones de diseño o implementación. En una clase de software, la abstracción se muestra en que los nombres de la clase y de los atributos se almacenan una vez en cada clase, en lugar de una vez por instancia. Si se desea que todas las instancias de una clase tengan un nuevo atributo o método, basta añadirlo a la clase.



Figura 13. Cuando la abstracción se confunde con la realidad: *La traición de las imágenes* (1926), de René Magritte. ¿Es esto una pipa? Si fuera de verdad una pipa, ¿no debería poder encenderse y echar humo?

4.2 Modularidad

Es la propiedad que tienen ciertos productos (o procesos) de descomponerse en subproductos (o subprocesos) más sencillos y manejables. Un producto modular se fabrica a partir de componentes estandarizados intercambiables (módulos). La modularidad se emplea para un gran número de productos: programas de computadora, automóviles, ordenadores, bicicletas, aviones, centrales eléctricas... Un módulo puede ir desde lo más simple (un recambio de tinta para un bolígrafo, p. ej.) a lo más complejo (un circuito de seguridad en una central nuclear, p. ej.).

Según el *Dictionary of OT*, la modularidad es “*la descomposición lógica de las cosas (por ejemplo, responsabilidades y software) en agrupaciones simples, pequeñas (por ejemplo, requisitos y clases, respectivamente), que aumentan las posibilidades de lograr las metas de la ingeniería de software*”.

En sentido general, un módulo es cualquier parte identificable de un sistema y que tiene sentido por sí mismo. Tanto en la ingeniería del software como en otras ingenierías, los módulos que interesan son aquellos **cuyos elementos estructurales están fuertemente vinculados entre sí y débilmente vinculados con elementos de otras unidades**.

Cuando nos restringimos a software, un módulo es un conjunto de sentencias bajo el *paraguas* de un nombre por el cual puede ser llamado o invocado. Los módulos son la

unidad de programación. En Java, una clase constituye un módulo, pero eso no quiere decir que toda clase de Java sea un “buen” módulo. Las características deseables de un módulo, las que aumentan su modularidad, son las siguientes:

- **Alta cohesión.** La cohesión mide el grado de relación funcional interna de los componentes de un módulo. En la ingeniería del software, la cohesión se traduce en una medida del grado en que las líneas de código dentro del módulo colaboran para ofrecer una función concreta. Los módulos con cohesión alta son deseables porque la alta cohesión se relaciona con programas robustos (no fallan ante entradas inesperadas o erróneas), reutilizables (partes del programa se pueden emplear en otros programas), legibles (fáciles de entender) y compactos (no más largos de lo necesario).
- **Bajo acoplamiento.** El acoplamiento mide la interconexión entre módulos. En la ingeniería del software, el acoplamiento se traduce en una medida de la relación entre líneas de código pertenecientes a módulos diferentes. Lo dicho con respecto las ventajas de la alta cohesión se aplica también al bajo acoplamiento, pues el bajo acoplamiento suele implicar alta cohesión, y viceversa.

La idea de construir aplicaciones juntando módulos de software estandarizados e intercambiables proviene de la producción industrial en serie. La modularidad, además de ser relevante en muchos sectores industriales, desempeñó un importante papel en la derrota alemana en la II Guerra Mundial. Alemania fabricaba muchos modelos distintos de tanques, aviones y barcos, cada uno con piezas exclusivas y que, por tanto, no podían intercambiarse con las de otros modelos.

En el caso de los tanques, la falta de modularidad era extrema: el uso de distintas piezas para cada modelo (Pzkw IV, Tiger, King Tiger, Panther, Hetzer, Ferdinand, Elephant, etc.), a menudo con acabados casi artesanales, se veía como una prueba indiscutible de la superioridad técnica alemana. Consecuentemente, en los campos de batalla se juntaban tanques completamente distintos, lo que dificultaba sobremanera el abastecimiento de piezas y las reparaciones. Al final, la falta de componentes estandarizados hacía que muchos tanques alemanes se abandonaran cuando se averiaban.

En el caso de los submarinos, Otto Merker, director general del Comité Principal para la Construcción de Navíos, investigó cómo podía mejorarse la fabricación de submarinos y recomendó que se fabricaran de forma modular; pero el destino de la Alemania nazi estaba ya sellado cuando se aprobó la producción en serie de los nuevos diseños modulares (agosto de 1943).

Estados Unidos y la Unión Soviética, por el contrario, apostaron desde el principio de la guerra por la producción en serie y la modularidad forzosa: fabricaban uno o dos modelos de cada tipo de transporte de tropas, de manera que cada componente o pieza era, por definición, estándar o casi estándar.

En el caso estadounidense, la influencia de Henry Ford resulta evidente: él mismo preparó unas instalaciones industriales para la producción en serie de los bombarderos *Flying Fortress*. La velocidad con la que se fabricaron los barcos de carga *Liberty* fue posible porque se adoptaron técnicas de ensambladura modular.

En el caso soviético, el uso de sólo dos tipos de tanques (el famoso T-34 y el KV-2) se reveló decisivo para reducir el número de tipos de piezas que había que fabricar y para facilitar, en condiciones muy difíciles, el abastecimiento y las reparaciones. (Es

cierto que el T-34 tuvo tres grandes submodelos –T34/76a, T34/76b y T34/76c–, pero sólo diferían en el cañón o en la torreta; el resto de las piezas eran completamente intercambiables entre modelos.) El T-34 fue crucial en la batalla de Kursk, que hizo que Alemania perdiera la iniciativa en el frente ruso: a partir de entonces, la Unión Soviética fue la que marcó el paso.



Figura 14. Modularidad en acción: un tanque T-34. La apuesta soviética por fabricar en serie sólo dos modelos de tanques evitó la proliferación de distintas piezas, no intercambiables, para los mismos cometidos.

4.3 Encapsulación

Es el ocultamiento de la información. Según el *Dictionary of OT*, es “*la localización física de las propiedades dentro de una sola abstracción de caja negra que oculta su implementación (y las decisiones asociadas de diseño) tras una interfaz pública*”.

La abstracción de “caja negra” se utiliza ampliamente en física, electrónica e informática; y consiste en esconder todos los detalles internos del sistema que se estudia bajo una caja negra imaginaria, cuando sea más importante comprender **qué** hace el sistema que **cómo** lo hace.

La encapsulación es como introducir el sistema dentro de una caja negra con dos ranuras llamadas “Entrada” y “Salida”. Por ejemplo, en metrología, cuando se quiere calibrar un dinamómetro, no se estudia la incertidumbre que proporciona cada circuito, cada muelle, cada parte móvil (entre otras cosas, porque el problema se tornaría casi inabordable) y luego se combinan para obtener la incertidumbre del aparato. En la práctica, lo que se hace es considerar el dinamómetro como una caja negra, el cual se

calibra viendo las lecturas ("Salida") que da frente a fuerzas patrón ("Entrada"), olvidando sus componentes internos.

Así, por poner otro ejemplo lejano al campo de la informática, un complejo circuito electrónico que forme parte de un sistema mayor puede sustituirse imaginariamente, para facilitar el estudio del sistema completo, por una caja negra que reciba una señal eléctrica y devuelva otra. Es más, el circuito podría caracterizarse por su función de transferencia ("lo que hace") sin que fuera necesario conocer la forma en que modifica la señal de entrada ("cómo lo hace") ni sus componentes.

Del mismo modo, cuando en Java se utiliza la clase *Vector*, no es necesario saber qué operaciones internas realiza para añadir o eliminar elementos; es más, si se necesitara conocerlas se incumpliría el principio de encapsulación.

En los lenguajes OO, la encapsulación garantiza que los usuarios de un objeto no puedan modificar el estado del objeto sin usar su **interfaz** (conjunto de métodos que son accesibles a otros objetos). Es decir, no pueden cambiar su estado de maneras no fijadas de antemano. En un objeto bien diseñado, los otros objetos sólo pueden interaccionar con él mediante su interfaz. Una buena encapsulación separa siempre las vistas que del objeto tienen el constructor y el usuario.

La gran ventaja de la encapsulación consiste en que, si un módulo cambia internamente sin modificar su interfaz, el cambio no supondrá ninguna otra modificación del sistema. Por tanto, la encapsulación permite evitar que los programas sean tan interdependientes que un pequeño cambio provoque muchos efectos secundarios.

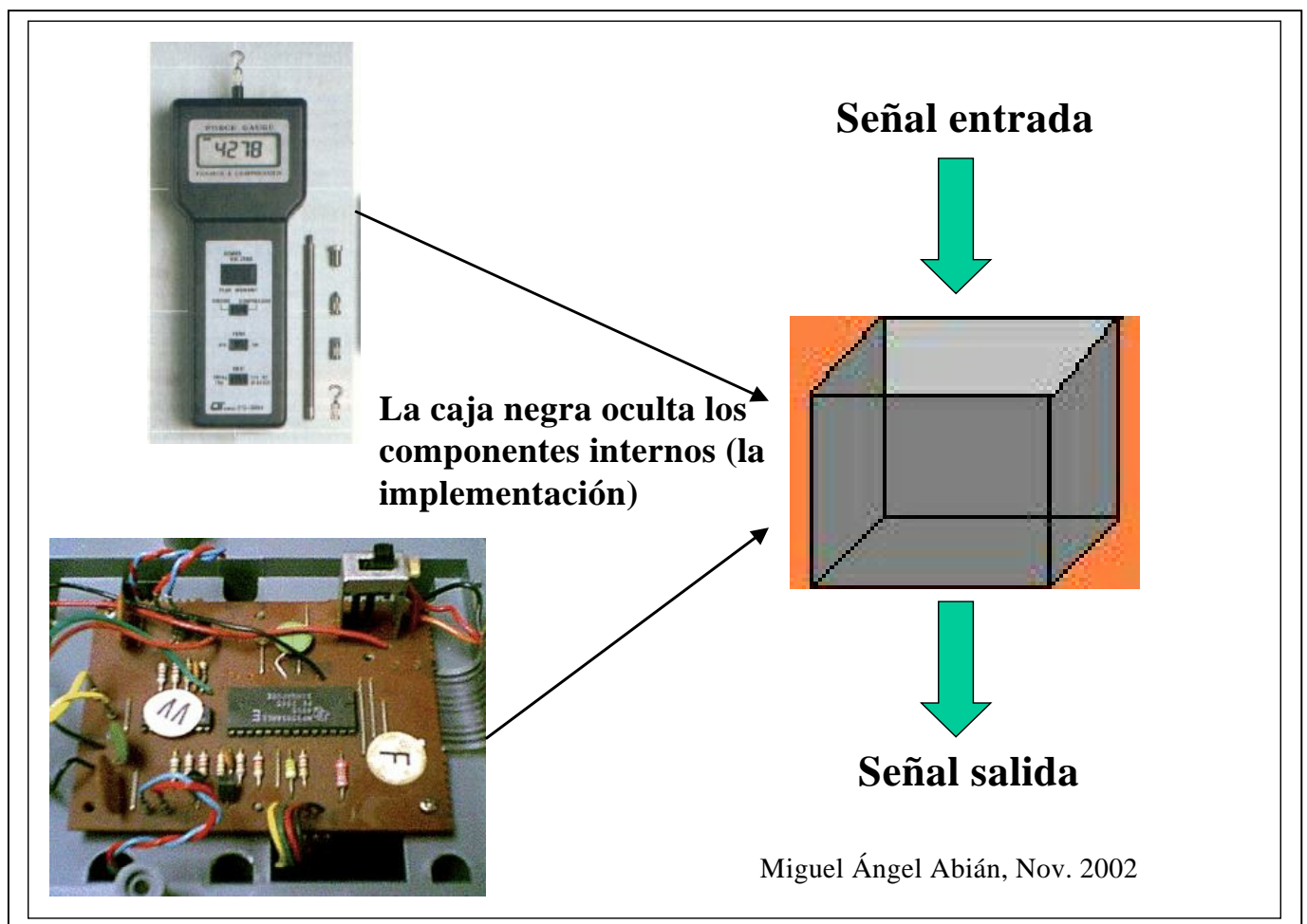


Figura 15. La encapsulación es una herramienta poderosa en muchos campos

Resulta llamativo que el término *encapsulación* provenga, precisamente, de la ingeniería electrónica. Esta coincidencia terminológica, al igual que el circuito electrónico de la figura 5, no es ni mucho menos casual. La ingeniería electrónica siempre ha trabajado, una vez pasada la época de los tubos de vacío, con módulos o componentes (“cápsulas”) en los que se integran otros muchos subcomponentes. La idea tras los componentes siempre ha sido la reutilización.

Resultaría inaceptable que se estropeará un componente electrónico y que hubiera que construir otro idéntico desde cero; o que los nuevos componentes tuvieran siempre que desarrollarse combinando constituyentes elementales –resistencias, condensadores y bobinas–, sin poder aprovechar nada de todos los componentes ya fabricados; o que fuera imprescindible conocer todos los elementos de un componente antes de poder saber para qué podría servir.

Antes de que la modularidad y la encapsulación tuvieran la importancia capital que tienen hoy en la ingeniería del software (hasta mediados de la década de 1960 no se empezó a hacer un uso intensivo de la modularidad), en la programación sucedía lo que era inaceptable en la electrónica. Empezar un nuevo programa era, a menudo, comenzar desde cero y la reutilización de “componentes de programación” era mínima. En los pocos casos en que podían reutilizarse esos componentes, se necesitaba conocer cómo funcionaban “por dentro”, lo que tornaba imposible la encapsulación. Había, en fin, más “artesanía del software” que ingeniería del software.

4.4 Jerarquía

Según el *Dictionary of OT*, la jerarquía es “*cualquier clasificación u ordenación de abstracciones en una estructura de árbol. Tipos: Jerarquía de agregación, jerarquía de clases, jerarquía de herencia, jerarquía de partición, jerarquía de especialización, jerarquía de tipo*”.

Las jerarquías se han utilizado desde hace mucho tiempo en las ciencias naturales para clasificar a los seres vivos. Con ello no quiero decir que los primeros botánicos o zoólogos pudieran ser, de haber vivido en esta época, unos extraordinarios programadores; sino que entendieron desde el principio la importancia de dividir los problemas en una jerarquía de ideas. Quizás Aristóteles fuera un poco desencaminado en cuanto a los detalles (véase el apdo. 3.1), pero andaba bastante acertado en cuanto a las ideas generales. Los dos tipos de jerarquía más comunes son la jerarquía de generalización/especialización (“Una nevera es un electrodoméstico”) y la de todo/parte (“Una mesa se compone de cuatro patas y una superficie”).

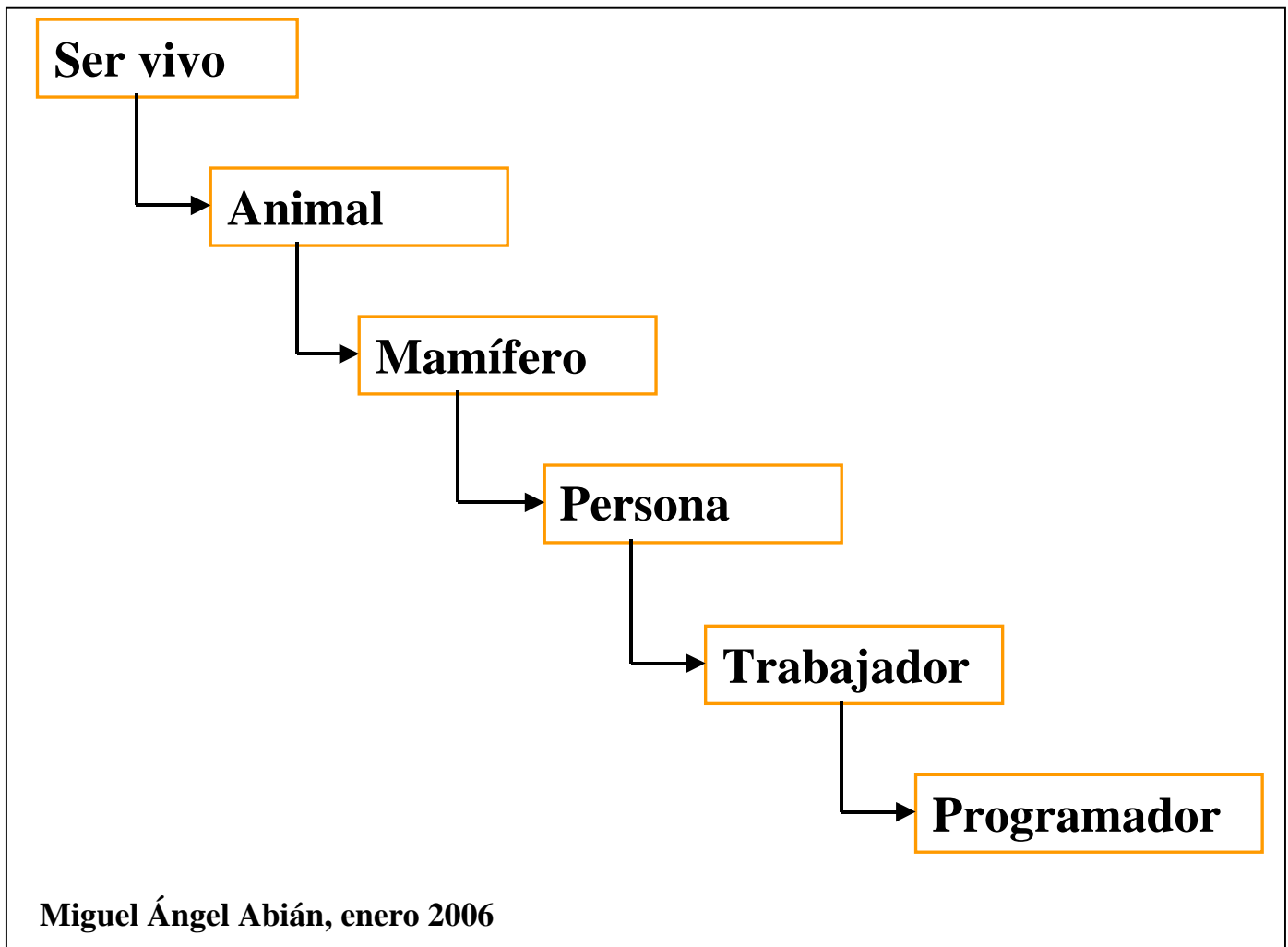


Figura 16. Un ejemplo de jerarquía del tipo generalización/especialización. Por ejemplo, toda persona es un mamífero, pero no todo mamífero es una persona.

La jerarquía de generalización/especialización se basa en que las propiedades de una categoría general se transmiten a todas las categorías que la especializan. Se suele aludir a esta jerarquía con la regla “es un” o “es un tipo de”, pues una categoría especializa a otra si es un tipo de la segunda. En la OO, el término **jerarquía de clases** significa un conjunto de clases relacionadas por la jerarquía de generalización/especialización.

En una jerarquía de clases, una clase es una especialización (subclase o clase “hija”) de otra (superclase o clase “madre”) cuando representa, con respecto a la superclase, un elemento más específico en el dominio modelado. Las subclases heredan los atributos de la superclase.

En los sistemas de software, la jerarquía de generalización/especialización suele implementarse mediante la **herencia**. Este término designa la facultad que tienen algunos lenguajes OO –no todos: véase la segunda parte del tutorial– de formar nuevas clases usando clases ya existentes. Las clases derivadas o subclases heredan los atributos y métodos de su superclase, sin que tengan que volver a implementar los métodos ni declarar los atributos.

Muy relacionado con la herencia se encuentra el **polimorfismo**. Aunque se trata con detalle en la 2ª parte del tutorial, adelanto aquí que el polimorfismo permite sobrescribir en las subclases los métodos de la superclase. Un método es la implementación de una operación para una clase. El polimorfismo permite que una misma operación esté asociada a métodos distintos en clases diferentes. A diferencia de lo que ocurre con los lenguajes estructurados, en los lenguajes OO sucede que el método con que se responde a un mensaje depende del objeto concreto al que se envía.

Por ejemplo, consideremos una subclase *CuentaMancomunada* que reescribe el método *sacarDinero()* de una superclase *CuentaBancaria*. Cuando un objeto *CuentaBancaria* reciba un mensaje *sacarDinero* transferirá el dinero; cuando un objeto *CuentaMancomunada* reciba ese mismo mensaje, antes de transferirlo comprobará que la operación está solicitada por todos los titulares de la cuenta, y si no es así anulará la operación y enviará un mensaje de aviso. En ambos casos se envía el mismo mensaje (la petición es la misma: sacar dinero), pero el comportamiento final depende del objeto que lo recibe. En un programa, una misma variable puede representar, conforme se va ejecutando el programa, distintos tipos de objetos. Por tanto, también puede llamarse a distintos métodos de la variable durante el tiempo de ejecución.

Como se vio en 3.3, la metodología estructurada se basa en la abstracción por descomposición. La OO incluye también la descomposición como un mecanismo de abstracción y la complementa con la jerarquía de generalización/especialización, que se basa en la abstracción por clasificación. En mi opinión, la abstracción por clasificación no tiene equivalente en la metodología estructurada y constituye, por tanto, una característica novedosa de la OO. A fecha de hoy, el debate de la evolución-revolución de la OO frente a la metodología estructurada continúa abierto.

Junto con la modularidad y la encapsulación, la herencia es responsable del éxito de la programación orientada a objetos, por cuanto permite la reutilización del código OO. La correcta utilización de la herencia, la modularidad y la encapsulación permite concebir los programas OO como una unión de “cápsulas” intercambiables con otros programas, las cuales pueden transmitir sus propiedades (atributos y métodos) a nuevas “capsulas”, evitándose así tener que desarrollarlas desde cero. En los sistemas de software, el mantenimiento ha supuesto siempre un coste muy grande. Esas tres propiedades permiten reducir el coste de las modificaciones que sufren los programas según se van mellando por el contacto con la realidad.

5. Lenguajes de programación orientados a objetos. ¿Puede utilizarse la OO en lenguajes no orientados a objetos?

Por lo general, la OO se relaciona con el uso de lenguajes de programación orientados a objetos. Como metodología general de desarrollo de sistemas, no depende de ningún lenguaje, sea orientado a objetos o no. El análisis y diseño OO puede aplicarse a cualquier sistema, independientemente de que se implemente al final en un lenguaje orientado a objetos o no. En cuanto a la POO, es posible incluso en lenguajes no orientados a objetos.

En la bibliografía no existe un acuerdo común e inapelable sobre los requisitos que debe cumplir un lenguaje para considerarse orientado a objetos, pero generalmente se admite que un lenguaje OO debe tener como mínimo:

- a) Encapsulación
- b) Herencia
- c) Polimorfismo

Adicionalmente, en un lenguaje OO puede suceder que

- d) todos los tipos de datos predefinidos en el lenguaje sean objetos;
- e) todos los tipos de datos que puedan ser definidos por los programadores sean objetos;
- f) todas las posibles operaciones se realicen enviando mensajes entre objetos.

Se consideran lenguajes OO “puros” aquellos que tienen las 6 características a)-f) y se consideran “híbridos” aquellos que carecen de alguna de las 3 últimas: d), e) ó f). En la segunda parte de este trabajo (<http://www.javahispano.org/tutorials.item.action?id=33>) se tratan los lenguajes OO más conocidos y se explican en detalle la herencia y el polimorfismo.

Característica	Fortran 77	C++	C#	Java	Smalltalk	Eiffel
a)	NO	SI	SI	SI	SI	SI
b)	NO	SI	SI	SI	SI	SI
c)	NO	SI	SI	SI	SI	SI
d)	NO	NO	NO	NO	SI	SI
e)	NO	NO	NO	SI	SI	SI
f)	NO	NO	NO	NO	SI	SI

Tabla 4. Ejemplos de lenguajes no OO, OO “híbridos” y OO “puros”.

Como ya he mencionado, pueden escribirse programas OO en lenguajes no OO. Para conseguirlo, el programador debería implementar las principales características de la OO en el lenguaje seleccionado (convertir en objetos las estructuras de datos propias del lenguaje no OO, almacenarlos en memoria, elegir cuidadosamente los nombres de los métodos para asociarlos inmediatamente a los objetos correspondientes, implementar manualmente la herencia y el polimorfismo...); proceso innecesario en un lenguaje OO, pues ya posee de partida esas implementaciones.

En la práctica, siempre resulta más rápido y eficaz trabajar directamente con lenguajes OO que implementar las características de la OO en lenguajes no orientados a objetos.

Como ejemplo de las dificultades de usar los conceptos de la OO en lenguajes no OO, implemento aquí, en Fortran 77 y C, una clase *Rectangulo* y la herencia para dos clases *Circulo* y *Elipse* que derivan de una clase *Punto*. (Omito las tildes para los nombres de las clases: estamos en la fase de implementación.) En la segunda parte del tutorial explico cómo se implementan los conceptos de la OO en los lenguajes orientados a objetos más conocidos.

Implementación de la clase *Rectangulo* en Fortran 77

implicit none

```
common /rectangulo/ x1, x2, y1, y2, num_rectangulo
real x1(100), x2(100), y1(100), y2(100)
integer num_rectangulo
```

Este código define una clase *Rectangulo*, en la que cada rectángulo se caracteriza por las coordenadas de sus vértices. Guarda en memoria simultáneamente 100 objetos de tipo *Rectangulo* (en Fortran 77 no existe la asignación dinámica de memoria) y el índice entero *num_rectangulo* identifica en todo momento cualquier objeto *Rectangulo*. El bloque *common* permite que ciertas variables se comportan entre varias subrutinas.

En Fortran 77, un objeto sólo puede representarse por una colección de cadenas de variables, en la que cada variable es un atributo del objeto. No existen tipos de datos definidos por el usuario.

Implementación de la clase *Rectangulo* en C

```
struct rectangulo{

    float x1;
    float x2;
    float y1;
    float y2;

}
```

Para iniciar un objeto rectángulo se usaría algo así:

```
struct rectangulo miRectangulo={0, 2, 3, 5}
```

Implementación de la herencia en Fortran 77 con las clases *Elipse* y *Circulo*, que derivan de la clase *Punto*

Implementación de las clases *Elipse* y *Circulo*.

implicit none

```
common /figura/ x, y, semiejea, semiejeb, radio, num_figura, clase_figura
real x(100), y(100), semiejea(100), semiejeb(100), radio(100)
integer num_figura
integer clase_figura(100)
```

```
common /clases/ ELIPSE, CIRCULO
integer ELIPSE /1/, CIRCULO /2/
```

Este modo de implementar la herencia resulta muy ineficaz, pues reserva espacio en memoria tanto para los atributos de la clase base *Punto* (coordenadas x e y) como para los atributos adicionales de las clases derivadas *Circulo* (radio) y *Elipse* (semiejes).

Para crear objetos derivados o “hijos”, el código sería similar a éste:

```
function crear_elipse(x0, y0, semiejea0, semiejeb0)
```

```
common /figura/ x, y, semiejea, semiejeb, radio, num_figura, clase_figura
real x(100), y(100), semiejea(100), semiejeb(100), radio(100)
integer num_figura
integer clase_figura(100)
```

```
common /clases/ ELIPSE, CIRCULO
integer ELIPSE /1/, CIRCULO /2/
```

```
integer crear_elipse
num_figura=num_figura + 1
x(num_figura)=x0
y(num_figura)=y0
semiejea(num_figura)=semiejea0
semiejeb(num_figura)=semiejeb0
radio(num_figura)=0
clase_figura(num_figura)=ELIPSE
crear_elipse=num_figura
```

```
end
```

```
function crear_circulo(x0, y0, radio0)
```

```
common /figura/ x, y, semiejea, semiejeb, radio, num_figura, clase_figura
real x(100), y(100), semiejea(100), semiejeb(100), radio(100)
integer num_figura
integer clase_figura(100)
```

```

common /clases/ ELIPSE, CIRCULO
integer ELIPSE /1/, CIRCULO /2/

integer crear_circulo
num_figura=num_figura + 1
x(num_figura)=x0
y(num_figura)=y0
semiejea(num_figura)=radio0
semiejeb(num_figura)=radio0
radio(num_figura)=radio0
clase_figura(num_figura)=CIRCULO
crear_circulo=num_figura

```

end

Para implementar el polimorfismo, bastaría comprobar, dado un objeto (caracterizado por su *num_figura*), cuál es el valor de *clase_figura(num_figura)* y, dependiendo de si es *CIRCULO* o *ELIPSE*, asignarle un método u otro.

Implementación de la herencia en C con las clases *Elipse* y *Circulo*, que derivan de la clase *Punto*

```

struct punto{
    float x;
    float y;
};

struct circulo{
    struct punto origen_figura;
    float radio;
};

struct elipse{
    struct punto origen_figura;
    float semiejea;
    float semiejeb;
};

```

He escogido a propósito Fortran 77 para los ejemplos porque es un lenguaje *primitivo* –pero extremadamente potente para aplicaciones que requieran cálculos numéricos intensivos– si lo comparamos con los lenguajes de programación actuales. Fortran es el equivalente informático de un mamut lanudo que hubiera escapado de una glaciación y perviviera en nuestros días. Para ser justo, añado que la versión anterior (Fortran 66) se utilizaba con tarjetas perforadas y que las versiones actuales (Fortran 90 y 95) incorporan asignación dinámica de la memoria y punteros, tipos de datos definidos por el usuario, módulos –en el sentido de la OO– y otras características. Por ser un lenguaje que tiene un único tipo de estructuras de datos (matrices) resulta posible, pero muy complicado e ineficaz, implementar los principios de la OO.

En C resulta mucho más fácil implementar la herencia, porque permite al programador definir sus propias estructuras de datos, y el uso de la asignación dinámica de la memoria hace que ésta se aproveche mucho más eficazmente que en Fortran 77.

En resumen, debemos extraer dos lecciones de este apartado. Por un lado, que el análisis y diseño OO se puede emplear siempre, aunque la implementación final se haga en un lenguaje no OO. Por otro, que es posible escribir programas orientados a objetos con lenguajes no OO, si bien uno debe encontrarse verdaderamente desesperado para desaprovechar las ventajas que brindan los lenguajes OO.

6. El paradigma orientado a objetos: éxitos y fronteras

6.1 El éxito del paradigma orientado a objetos, ¿se basa únicamente en criterios objetivos?

Una vez expuestos los fundamentos de la OO, y antes de definir detenidamente sus conceptos, es interesante reflexionar sobre el éxito de la OO desde una perspectiva poco ortodoxa y un tanto herética.

En su célebre libro *La estructura de la revoluciones científicas* (publicado en 1962, la primera edición en español data de 1971), Thomas S. Kuhn definió **paradigma** en el sentido de matriz disciplinaria: elementos ordenados de diversas maneras, cuyo orden hay que especificar, que son posesión común de los profesionales de una disciplina. Cuando se dice que unos científicos comparten un mismo paradigma, se está diciendo que comparten “una manera de ver el mundo y practicar la ciencia en él”.

Kuhn proponía que cuando los científicos observan determinados fenómenos no los ven objetivamente, sino a través del filtro de un conjunto de ideas y conceptos (procedentes de la tradición cultural, social y científica en la que han sido educados) que han formado/deformado su percepción. En otras palabras, los ven a través de un paradigma firmemente establecido. Solo así podía entenderse –pensaba Kuhn– que ideas como que la velocidad de caída de los cuerpos era proporcional a su masa hubieran pervivido desde la Grecia antigua hasta Galileo Galilei, cuando unos sencillos experimentos hubieran demostrado, más allá de toda duda razonable, su falsedad. Si creemos a Kuhn, los científicos de todos esos siglos intermedios aceptaron la idea de que los cuerpos caían con una velocidad proporcional a su masa como parte de su cultura (y de su paradigma), y por ello no se plantearon hacer experimentos ni “vieron” los hechos que iban contra esa idea. Del mismo modo, los primeros que miraban a través del telescopio de Galileo no veían más que manchas y sombras, porque aún no habían asimilado el nuevo paradigma astronómico. Donde Galileo veía cuerpos celestes, los otros veían manchas sin sentido o ilusiones ópticas, que atribuían al propio aparato.

Muchos siglos después, todavía entristecen las frases con las que Galileo tuvo que cerrar su nuevo paradigma astronómico, que sobrevivió a los inquisidores y sigue vigente:

Yo, Galileo, hijo del difunto Vincenzo Galilei, florentino, de setenta años de edad, emplazado en persona ante este tribunal y arrodillado ante vos, eminentísimo y reverendísimo Cardenal Inquisidor General contra la depravación herética en el orbe cristiano, teniendo ante mis ojos y tocando con mis manos los Santos Evangelios, juro que siempre he creído, creo y creeré, con la ayuda de Dios, todo lo que es sostenido, predicado y enseñado por la Santa Iglesia Católica y Apostólica. Tras la prohibición que me ha sido impuesta judicialmente por este Santo Oficio al efecto de que yo abandone completamente la falsa opinión de que el Sol es el centro del mundo, y está inmóvil, y que la Tierra no es el centro del mundo, y se mueve, y de que no debo sostener, defender o enseñar dicha falsa doctrina de ninguna de las maneras, ni verbalmente ni por escrito [...] (**Fórmula de abjuración de Galileo Galilei, 22 de junio de 1633**).

Según Kuhn, cuando un científico –o un grupo de científicos– se coloca las “gafas” de un nuevo paradigma, el resto de la comunidad científica lo ataca o lo rechaza hasta que el paradigma emergente demuestra su validez (mediante hechos empíricos o nuevas líneas de investigación); o hasta que fallecen los científicos educados en el paradigma anterior. Al cabo de una o dos generaciones, todos los científicos se educan ya en el nuevo paradigma; y los conflictos conceptuales entre el paradigma antiguo y el

nuevo son recordados solamente por filósofos e historiadores de la ciencia que mantienen encarnizadas batallas dialécticas sobre la respectiva superioridad o inferioridad de cada paradigma. Y así sucede una y otra vez.

En resumen, Kuhn venía a decir que hay un componente sociológico y cultural, ajeno a la pretendida ciega objetividad científica, en el modo en que los científicos de una época observan el mundo y que los paradigmas son consensos tácitos dentro de una comunidad (científica, técnica, etc.).

La orientación a objetos es el paradigma aceptado actualmente para construir sistemas de software y ha relevado al antiguo paradigma estructurado (dicho con la jerga de Kuhn, es el paradigma dominante). Denominar “paradigma” a la OO (o la metodología estructurada) no es una exageración o una libertad que me tomo: **la OO es un paradigma porque ha cambiado el modo en que los ingenieros de software y los desarrolladores piensan sobre el software**. La OO es el paradigma dominante: los estudiantes de ingeniería del software se forman ya en la OO, que está incluida en los planes académicos y cuyos términos son de uso común. Más aún: los lenguajes OO (Java, sobre todo) han desplazado a los lenguajes estructurados tradicionales (Pascal, C) que se utilizaban en las asignaturas de introducción a la programación.

¿Por qué se ha impuesto como paradigma la orientación a objetos? En gran parte, por su efectividad: el desarrollo de gigantescos sistemas de software volvió inservible el paradigma estructurado, y la OO permite abordar viejos problemas de nuevas maneras y con costes menores. Sin el nuevo paradigma, muchos proyectos modernos no se habrían ejecutado. También se ha impuesto por su sencillez conceptual (¿quién no sabe lo que es un objeto?, ¿cuántos de nosotros pensamos con algoritmos?): la OO está mucho más cercana al pensamiento humano que el paradigma estructurado.

Con todo, no cabe cegarse ante la evidencia: hay un cierto elemento de profecía autocumplida en el éxito de la OO. Supóngase que los investigadores de prestigiosas universidades comenzasen a desarrollar lenguajes MM, crearan revistas y publicaciones con las siglas MM –engordando de paso sus currículos y sus carteras– y abrieran nuevas parcelas de investigación con promesas de sencillez, productividad, ahorro de costes, etc. Algunas empresas se harían eco de esas promesas (aunque solo fuera por miedo a que otras usasen las nuevas técnicas antes que ellas) y la industria apoyaría el nuevo paradigma; aparecerían libros, técnicos y del tipo “Aprenda MM en 21 días”, comités de normalización, becas para investigar la MM. Con el tiempo, las empresas anunciarían orgullosas sus inversiones en MM, y se presentarían como caminantes por el “filo de la tecnología”. ¿Acaso no sería difícil que el nuevo paradigma, avalado y apoyado por la industria y la comunidad investigadora, fracasara? Es probable que a base de esfuerzos e inversiones se consiguieran algunos éxitos que reafirmarían las promesas iniciales de sencillez, productividad, etc. ¿Cuesta mucho sustituir MM por OO?

El éxito de la OO, como el éxito de muchas teorías científicas, tiene un pequeño componente sociológico y comunitario. Difícilmente se embarcaría hoy nadie en un gran proyecto de software sin seguir la metodología OO, porque es el paradigma dominante y resulta muy difícil conseguir financiación para ideas o proyectos fuera del paradigma dominante. Y lo que es peor: **nadie quiere fracasar a contracorriente; se prefiere fracasar siguiendo las ideas aceptadas que triunfar con ideas novedosas**. Ya sabe, es mejor morir según las reglas que salvarse a última hora en contra de ellas. Que se lo digan a los Rolling Stones.

6.2 Los límites de la POO

Aunque constituye el paradigma dominante en el campo de la programación, la programación orientada a objetos no aporta capacidades o métodos de resolución de problemas irresolubles –en sentido teórico– por otros medios o técnicas. Una máquina de Turing podría ser simulada por cualquier lenguaje de programación que permitiera sentencias condicionales y saltos. De acuerdo con la conjetura de Church, cualquier computación para la que exista un algoritmo funcional (ya sea el tiempo de cómputo finito o no) podría ser realizada por una máquina de Turing. Por lo tanto, cualquier computación con un algoritmo funcional puede realizarse en un lenguaje de programación que permita sentencias condicionales y saltos, sea o no orientado a objetos. Uso la palabra *realizarse* en sentido teórico; en la práctica hay limitaciones de tiempo, memoria, coste...

La OO suministra unos nuevos “anteojos” conceptuales a través de los cuales ver el mundo y solucionar los problemas de un modo más rápido, eficaz, económico y natural que con el paradigma estructurado. Ahora bien, su aplicación a la programación no resuelve problemas insolubles con otros medios o técnicas.

Nota biográfica del autor: Miguel Ángel Abián nació en Soria, se licenció en Ciencias Físicas a pesar de la Universidad de Valencia y obtuvo la suficiencia investigadora 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 colaborado en diversos programas de investigación TIC relacionados con fibras ópticas y cristales fotónicos, ha obtenido becas de investigación del IMPIVA (Instituto de la Mediana y Pequeña Industria Valenciana) y de la Universidad Politécnica de Valencia (para innovación tecnológica en las empresas), y ha publicado artículos en el *IEEE Transactions on Microwave Theory and Techniques* y en el *European Congress on Computational Methods in Applied Sciences and Engineering*, relacionados con el análisis de guías de ondas.

En el ámbito laboral ha trabajado como gestor de carteras y asesor fiscal para una agencia de bolsa y actualmente trabaja simultáneamente en los departamentos de **I+D** y de **Tecnologías de la Información** de AIDIMA (Instituto Tecnológico del Mueble y Afines), en proyectos nacionales e internacionales de **I+D** e **I+D+i**. En dicho instituto (www.aidima.es) se desarrollan proyectos europeos de comercio electrónico B2B para la industria del mueble, basados en Java y XML. Ha impartido formación en calidad, normalización y programación para ELKEDE (Grecia), CETEBA (Brasil) y CETIBA (Túnez), entre otros.

En los últimos años, aparte de asesorar técnica y financieramente a diversas empresas, es investigador en las **Redes de Excelencia INTEROP** (www.interop-noe.org) y **ATHENA** (www.athena-ip.org) del **Sexto Programa Marco de la Comisión Europea**, las cuales pretenden marcar las pautas para tecnologías de la información en la próxima década y asegurar la posición de Europa en la sociedad del conocimiento. Ambas redes abordan la interoperabilidad (estudian tecnologías como J2EE, .Net y CORBA, servicios web, la Web semántica, ontologías, modelado empresarial, etc.), y en ellas participan empresas como IBM U.K., TROUX, SIEMENS, FIAT, TXT, SAP y EADS, además de numerosas universidades europeas y centros de investigación en ingeniería del software.

Sus intereses actuales son la evolución de la programación orientada a objetos, Java, la Web semántica y sus tecnologías, el intercambio electrónico de datos, el surrealismo y París, siempre París.