

Desarrollo de software basado en reutilización

Macario Polo Usaola

PID_00184467



Los textos e imágenes publicados en esta obra están sujetos –excepto que se indique lo contrario– a una licencia de Reconocimiento-NoComercial-SinObraDerivada (BY-NC-ND) v.3.0 España de Creative Commons. Podéis copiarlos, distribuirlos y transmitirlos públicamente siempre que citéis el autor y la fuente (FUOC. Fundación para la Universitat Oberta de Catalunya), no hagáis de ellos un uso comercial y ni obra derivada. La licencia completa se puede consultar en <http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.es>

Índice

Introducción.....	5
Objetivos.....	6
1. Introducción a la reutilización.....	7
1.1. Un poco de historia	9
1.2. Reutilización de conocimiento: patrones de diseño	12
1.3. Beneficios y costes de la reutilización	13
1.3.1. El impacto positivo de la reutilización en el proceso de mantenimiento	14
1.3.2. Los costes de reutilizar	15
2. Reutilización en diseño de software orientado a objetos.....	17
2.1. Introducción	18
2.2. Abstracción, encapsulación y ocultamiento	21
2.3. Herencia y polimorfismo	23
3. Reutilización a pequeña escala: soluciones técnicas de reutilización.....	26
3.1. Introducción	26
3.2. Una historia ilustrativa	26
3.3. Librerías	28
3.3.1. Librerías de enlace estático	29
3.3.2. Librerías de enlace dinámico	29
3.3.3. Otras librerías	31
3.4. Clases	31
3.5. Patrones arquitectónicos	32
3.5.1. Arquitectura multicapa	33
3.5.2. Arquitectura <i>de pipes and filters</i> (tuberías y filtros)	34
3.5.3. Arquitecturas cliente-servidor	35
3.5.4. Arquitecturas P2P (<i>peer-to-peer</i> , de igual a igual)	36
3.6. Patrones de diseño	37
3.6.1. Patrones de Gamma	38
3.6.2. Patrones de Larman	47
3.6.3. El patrón modelo-vista-controlador	52
3.7. Componentes	52
3.7.1. Ingeniería del software basada en componentes	53
3.7.2. Componentes y clases	58
3.7.3. Desarrollo de componentes	59
3.8. Frameworks.....	59

3.8.1. <i>Framework</i> para la implementación de interfaces de usuario	60
3.8.2. <i>Frameworks</i> de persistencia para “mapeo” objeto-relacional	61
3.8.3. <i>Framework</i> para el desarrollo de aplicaciones web	63
3.9. Servicios	65
3.9.1. Introducción a los servicios web	65
3.9.2. Desarrollo de servicios web	66
3.9.3. Desarrollo de clientes	68
3.9.4. Tipos de datos	69
4. Reutilización a gran escala / Soluciones metodológicas de reutilización.....	70
4.1. Introducción	70
4.2. Ingeniería del software dirigida por modelos	70
4.2.1. Diferentes aproximaciones: metamodelos propios y metamodelos estándares	72
4.2.2. UML como lenguaje de modelado en MDE	76
4.2.3. Soporte automático	80
4.3. Lenguajes de dominio específico (DSL: <i>domain-specific languages</i>)	82
4.3.1. Definición de DSL	83
4.4. Línea de producto de software	84
4.4.1. Ingeniería de dominio	86
4.4.2. Ingeniería del software para líneas de producto basada en UML	92
4.5. Programación generativa	93
4.5.1. Ingeniería de dominio	95
4.5.2. Tecnologías	96
4.6. Fábricas de software	102
Resumen.....	104
Actividades.....	105
Ejercicios de autoevaluación.....	107
Solucionario.....	109
Glosario.....	128
Bibliografía.....	130

Introducción

Todo el que haya programado alguna vez habrá sentido, en algún momento, la necesidad de recuperar alguna función o algún bloque de código que tenía escrito de algún proyecto previo para incorporarlo en el que le esté ocupando en ese momento. Las posibilidades que nos dan el Copy & Paste nos permiten aprovechar de esta manera el esfuerzo que dedicamos en momentos anteriores.

Sin embargo, esta utilización de elementos predesarrollados no casa completamente con el concepto de reutilización en el sentido de la ingeniería del software, donde reutilizar tiene un significado más relacionado con:

- 1) La encapsulación de funcionalidades (previamente desarrolladas y probadas) en elementos que, posteriormente, podrán ser directamente integrados en otros sistemas.
- 2) El desarrollo de estas funcionalidades mediante métodos de ingeniería de software.
- 3) El aprovechamiento del conocimiento y la experiencia producidos durante décadas de práctica en la construcción de software.

Este módulo se estructura en cuatro partes: en la primera se presenta una visión general de la reutilización, poniéndose algunos ejemplos y haciendo un breve repaso por su historia. Las aportaciones a la reutilización de la orientación a objetos, que ha desempeñado un papel importantísimo en este campo, se describen e ilustran en la segunda parte. Algunas soluciones técnicas, muchas de ellas evoluciones casi naturales de la orientación a objetos, se presentan en la tercera parte. La cuarta y última parte se enfoca más a soluciones metodológicas para desarrollo de software, que toman la reutilización como un valor esencial.

Objetivos

Al finalizar este curso, el alumno debe:

- 1.** Ser consciente de la conveniencia de reutilizar software y de desarrollar software reutilizable.
- 2.** Conocer las principales técnicas para desarrollo de software reutilizable.
- 3.** Conocer técnicas para integrar software reutilizable en los proyectos nuevos.
- 4.** Conocer metodologías que permitan el desarrollo de proyectos que hagan un uso intensivo de la reutilización.

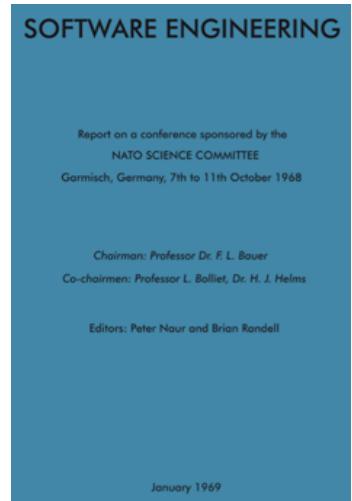
1. Introducción a la reutilización

La primera vez que se introdujo de forma pública y relevante la idea de la reutilización del software fue en 1968, en un célebre congreso sobre ingeniería del software que organizó el Comité de Ciencia de la OTAN. En este encuentro, además de acuñarse también el término *ingeniería del software*, el ingeniero de Bell Laboratories, M.D. McIlroy, afirmó que “La industria del software se asienta sobre una base débil, y un aspecto importante de esa debilidad es la ausencia de una subindustria de componentes”. En un contexto tecnológico mucho más primitivo que el actual, McIlroy asimilaba un componente a una rutina o conjunto de rutinas reutilizable: explicaba que en su empresa utilizaban más de un centenar de distintas máquinas de procesamiento de información, pero que casi todas ellas necesitaban una serie de funcionalidades en común (rutinas para convertir cadenas a números, por ejemplo) que tenían que implementar una y otra vez.

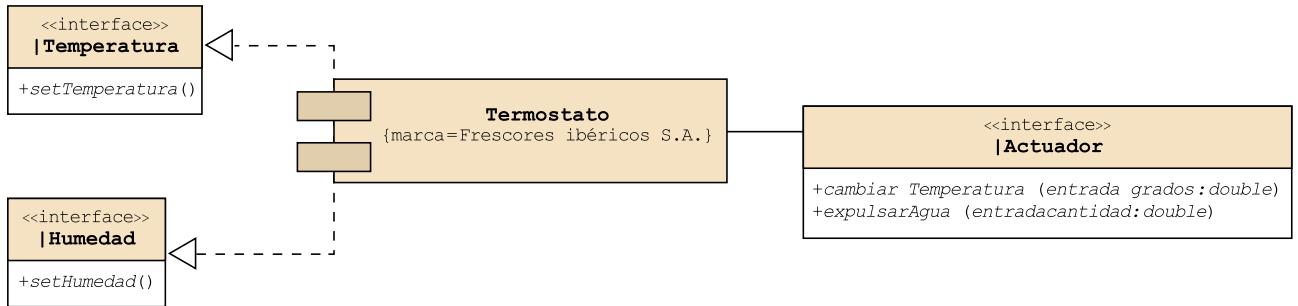
En el contexto actual de la ingeniería del software, todo el mundo entiende que un componente se corresponde con un fragmento reemplazable de un sistema, dentro del cual se encuentran implementadas un conjunto de funcionalidades. Un componente se puede desarrollar aislado del resto del mundo.

Un botón de un editor visual de aplicaciones es un pequeño componente que permite, por ejemplo, que el usuario lo pulse, y que ofrece al desarrollador, por un lado, una interfaz pública para que este le adapte su tamaño a la ventana, le cambie el color, el texto que muestra, etcétera; y por otro, la posibilidad de colocar diferentes instancias de ese mismo botón en la misma o en otras aplicaciones, evitando la tediosa tarea de programar su código cada vez que quiera hacer uso de él.

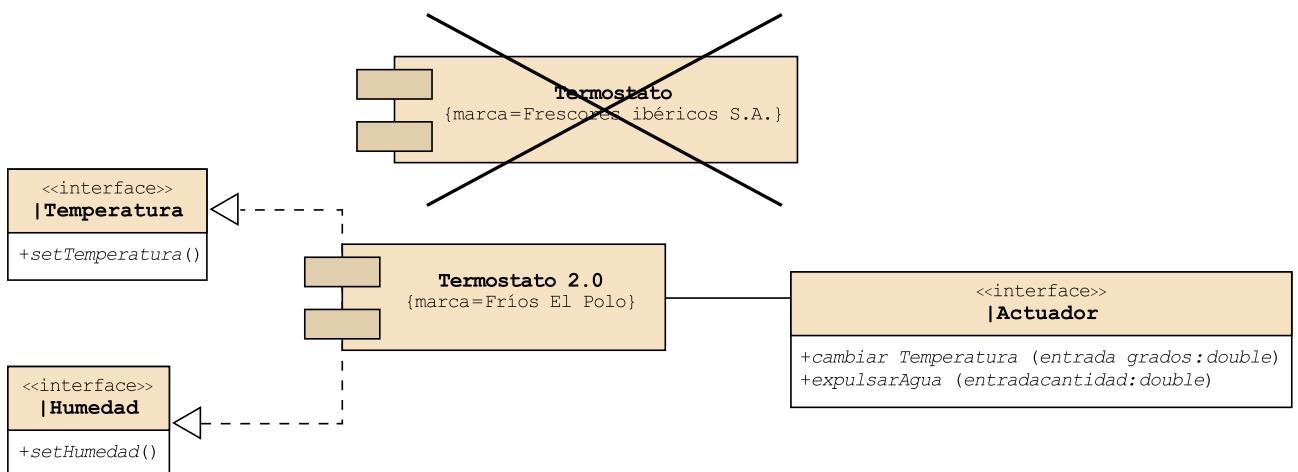
Para que un componente sea, en efecto, reutilizable, debe encapsular su funcionalidad, y debe ofrecerla al exterior a través de una o más interfaces que el propio componente debe implementar. Así, cuando el sistema en el que incluimos el componente desea utilizar una de las funcionalidades que este ofrece, se la pide a través de la interfaz. Pero, del mismo modo, el componente puede también requerir otras interfaces para, por ejemplo, comunicar los resultados de su cómputo. En el ejemplo de la figura siguiente se muestra un componente *Termostato* que recibe datos de dos sensores (uno de temperatura y otro de humedad): estos le comunican periódicamente su información a través de las dos interfaces que les ofrece el componente (*ITemperatura* e *IHumedad*). Este, en función de los parámetros recibidos, ordena a través de algún elemento que implementa la *interfaz IActuador* que se suba o se baje la temperatura (operación *cambiarTemperatura*) o que se expulse agua a la estancia para incrementar la humedad.



La conferencia se organizó en la ciudad de Garmisch (Alemania), del 7 al 11 de octubre de 1968. Sus actas se han escaneado y procesado mediante OCR y se encuentran publicadas en Internet: <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF>



Supongamos que el componente de la figura de arriba no respondiese con la precisión requerida ante ciertos cambios atmosféricos: por ejemplo, incrementa o decremente la temperatura solamente cuando la diferencia con la medida anterior es superior a 2 grados. En este caso, es posible que encontremos otro componente del mismo o de otro fabricante para que podamos construir una versión nueva del sistema sin demasiado esfuerzo: en la siguiente ilustración, el viejo termostato es sustituido por un *Termostato 2.0* adquirido a un distinto proveedor. Para que esta nueva versión del componente pueda integrarse perfectamente en nuestro sistema, debe ofrecer y requerir exactamente las mismas interfaces.



La ingeniería del software basada en componentes (CBSE: *component-based software engineering*) se ocupa del desarrollo de sistemas software a partir de componentes reutilizables, como en el sencillo ejemplo que acabamos de presentar. Pero la CBSE, sin embargo, es solo una de las líneas de trabajo de la comunidad científica y de la industria del software para favorecer la reutilización y, de este modo, evitar la reinvenCIÓN continua de la rueda o, en nuestro entorno, impedir el análisis, el diseño, la implementación y la prueba de la misma solución, desarrollándola una y otra vez en mil y un proyectos distintos.

Ved también

Ved la asignatura *Ingeniería del software de componentes y sistemas distribuidos* del grado de Ingeniería Informática.

De este modo, la reutilización¹ abarata los costes del desarrollo: en primer lugar, porque no se necesita implementar una solución de la que ya se dispone; en segundo lugar, porque aumenta la productividad, al poder dedicar los recursos a otras actividades más en línea con el negocio²; en tercer lugar, porque probablemente el elemento que reutilizamos ha sido suficientemente probado.

⁽¹⁾Krueger la define como “el proceso de crear sistemas software a partir de software preexistente, en lugar de crearlos empezando de cero”.

do por su desarrollador, con lo que los *testers* podrán dedicarse a probar otras partes más críticas del sistema³, obteniendo entonces unos niveles de calidad mucho más altos que si el sistema se desarrolla desde cero.

⁽²⁾Imaginemos que, para cada aplicación visual que fuese necesario que construir, se debiese escribir el código necesario para implementar un botón corriente.

1.1. Un poco de historia

La introducción en los años ochenta del paradigma orientado a objetos supuso un avance importantísimo en cuanto a reutilización de software: en efecto, una clase bien construida encapsula en sí misma una estructura y un comportamiento que pueden aprovecharse en otros proyectos; además, un conjunto de clases que colaboran entre sí para servir algún propósito pueden agruparse en librerías, que luego, si se desea, se importan para utilizarlas más de una vez.

⁽³⁾Aunque también tendrán que probar la integración del elemento reutilizado con el resto del sistema.

Consulta recomendada

C. W. Krueger (1992). "Software Reuse". *ACM Computing Surveys* (núm. 24, págs. 131-183).

En los años noventa se desarrollaron librerías que permitían la programación dirigida por eventos, librerías de uso compartido y librerías de enlace dinámico. Las *Microsoft foundation classes* (MFC, clases fundamentales de Microsoft) de los años noventa agrupaban gran cantidad de clases de uso común en una serie de librerías de enlace dinámico, que el programador podía utilizar para construir tipos muy diversos de aplicaciones.

También en los años noventa se comienzan a desarrollar los primeros componentes, aptos para ser integrados en aplicaciones sin demasiado esfuerzo. Algunas compañías construyen y venden lo que se dio en llamar componentes COTS (*commercial off-the shelf*), que no eran sino componentes cerrados y puestos a la venta junto a la especificación de sus interfaces y sus manuales de integración y utilización.

Ved también

Ved los componentes COTS en la asignatura *Ingeniería de requisitos* del grado de Ingeniería Informática.

Se construyen también librerías que encapsulan funcionalidades de comunicaciones, que permiten, por ejemplo, la compartición de una misma instancia por otros objetos situados en máquinas remotas. En este sentido, cada gran compañía de software desarrolló su propia tecnología de comunicaciones.

Así, por ejemplo, Sun desarrolla RMI (*remote method invocation*) y Microsoft hace lo propio con DCOM (*distributed component object model*), dos sistemas incompatibles pero que persiguen el mismo propósito. Ambos modelos de objetos distribuidos comparten una sencilla idea: la elaboración de un protocolo para compartir objetos y transmitir mensajes entre objetos remotos.

Estos modelos no son, al fin y al cabo, nada muy distinto de un *socket* que envía por su canal ristras de bytes con la información codificada de alguna manera. Lamentablemente, los formatos de estas ristras de bytes eran distintos en RMI y en DCOM, por lo que un objeto Java no podía comunicarse (al menos directamente) con un objeto Microsoft.

Ved también

Ved RMI y DCOM en la asignatura *Ingeniería del software de componentes y sistemas distribuidos* del grado de Ingeniería Informática.

Supongamos que dos compañías distintas desarrollan una sencilla calculadora remota empezando desde cero. Ambas calculadoras implementan y ofrecen vía de acceso remoto a las operaciones aritméticas básicas, como *suma* ($x : \text{int}$, $y : \text{int}$) : int . Cada compañía ofrecerá acceso a su calculadora imponiendo un formato diferente: la primera puede decirle al cliente que, cuando desee invocar una operación, envíe por un *socket* una ristra de bytes con el formato *operación#parametro1#parametro2*, mientras que la segunda quizás utilice el símbolo @ como carácter de separación y, además, una almohadilla al final. Obviamente, una ristra de bytes para invocar al primer servicio no sirve para llamar al segundo, que devolverá un error. Los desarrolladores de RMI y DCOM no imponían evidentemente un formato tan simple, sino otros más complejos, que, no obstante, permanecen transparentes para el desarrollador, que los utiliza a través de las correspondientes librerías, que son las que codifican los mensajes.

Este problema de incompatibilidades se soluciona parcialmente en los primeros años de este siglo con la **introducción de los servicios web**.

Un servicio web no es más que una funcionalidad que reside en un sistema remoto y que se hace accesible a otros sistemas a través de una interfaz que, normalmente, se ofrece a través de protocolo http.

Las invocaciones a los servicios ofrecidos por la máquina remota (que actúa de servidor) viajan desde el cliente en un formato estandarizado de paso de mensajes llamado SOAP (*simple object access protocol*, protocolo simple de acceso a objetos) para cuya descripción, por fortuna, se pusieron de acuerdo Microsoft, IBM y otras grandes empresas. SOAP utiliza notación XML y, aunque una invocación a la operación remota *suma* ($x : \text{int}$, $y : \text{int}$) : int no sea exactamente como se muestra en la figura siguiente (en la que se solicita la suma de los números 5 y 8), la idea no es muy diferente.

```
<SOAP_Invocation>
  <operation name='suma'>
    <parameter type='int' value='5' />
    <parameter type='int' value='8' />
  </operation>
</SOAP_Invocation>
```

Gracias a esta estandarización, podemos utilizar una funcionalidad ofrecida en una máquina remota sin importarnos en qué lenguaje ni en qué sistema operativo se esté ejecutando. Desde cierto punto de vista, un servicio web es un componente remoto que nos ofrece una interfaz de acceso en un formato estandarizado.

Ved también

Ved los servicios web en la asignatura *Ingeniería del software de componentes y sistemas distribuidos* del grado de Ingeniería Informática.

La reutilización que permiten los servicios web es muy evidente, pues se integran en la aplicación, sin necesidad de hacer instalación alguna, las funcionalidades que un tercero ha desarrollado. Lo único que el desarrollador necesita será una clase que constituirá el punto de acceso al servidor (un *proxy*) que:

1) codifique adecuadamente las llamadas al servicio remoto⁴;

(4) Es decir, que las traduzca a ristras de bytes en formato SOAP.

2) se las envíe;

- 3) reciba los resultados (también en formato SOAP), y
 4) los descodifique a un formato entendible por la aplicación.

(5) Es decir, el desarrollador que integra un servicio web en su aplicación.

Además, el usuario del servicio web⁵ tampoco debe preocuparse de implementar el citado *proxy*, ya que cualquier entorno de desarrollo moderno lo construye a partir de la especificación de la interfaz del servicio, que también se ofrece en un formato estandarizado llamado WSDL (*web service description language*, lenguaje de descripción de servicios web).

A partir de la utilización más o menos masiva de servicios web se empieza a hablar de arquitectura orientada a servicios (SOA: *service oriented architecture*) que, básicamente, consiste en el desarrollo de aplicaciones utilizando un número importante de funcionalidades expuestas como servicios que, no obstante, no tienen por qué ser obligatoriamente servicios web.

Véase también

Véase SOA en la asignatura *Ingeniería del software de componentes y sistemas distribuidos* del grado de Ingeniería Informática.

De aquí se ha avanzado rápidamente a la computación en la nube (*cloud computing*), mediante la que se ofrecen multitud de servicios (ejecución de aplicaciones, servicios de almacenamiento, herramientas colaborativas, etcétera) a los usuarios, que los utilizan de manera totalmente independiente de su ubicación y, por lo general, sin demasiados requisitos de hardware.

Web recomendada

En Wikipedia hay una disertación muy completa sobre *cloud computing*: http://en.wikipedia.org/wiki/Cloud_computing

Otra línea interesante respecto de la reutilización se encuentra en la **fabricación industrial del software**, tanto en los llamados mercados de masas⁶ como en los mercados de cliente⁷. En algunos entornos se ha dado un paso más y se aplican técnicas de líneas de producto a la fabricación de software: el desarrollo de software mediante líneas de producto de software (en lo sucesivo, LPS) surge hace en torno a una década (aunque es ahora cuando está cobrando más interés) con el objetivo de flexibilizar y abaratar el desarrollo de productos de software que comparten un conjunto amplio de características (*common features* o características comunes).

(6) El mismo producto se puede vender varias veces –lo que se llaman economías de escala-, copiando los prototipos mecánicamente.

(7) Cada producto es único y el beneficio se consigue a través de la reutilización sistemática.

Nota

En inglés, a las LPS se las conoce como *software product lines*, *software product architectures* y *software product families* (SPL, SPA y SPF respectivamente).

Véase también

Veremos las LPS en detalle en el apartado “Ingeniería del software para líneas de producto basada en UML 87” de este módulo.

Las líneas de producto llevan años aplicándose en otros entornos industriales, como la fabricación de vehículos en cadenas de montaje. Las diferentes versiones de un mismo modelo de coche difieren en ciertas características de su equipamiento (presencia o ausencia de aire acondicionado, elevalunas eléctricos, etc.), lo cual no impide que todos ellos comparten las mismas cadenas de montaje y producción, imitando así a las industrias de productos manufacturados que se pusieron en marcha a partir de la Revolución Industrial, en el siglo XIX, y que Charles Chaplin parodió en el filme *Tiempos modernos*.

Más recientemente, las líneas de producto se utilizan también para el desarrollo y construcción de teléfonos móviles, que, además del conjunto básico de funcionalidades, ofrecen diferentes tamaños de pantalla, presencia o ausencia de bluetooth, varias resoluciones de la cámara, etcétera. Por lo general, el desarrollo de productos de software más o menos similares mediante LPS per-

mite aprovechar casi al máximo los esfuerzos dedicados a la construcción de las *common features* (características comunes) reutilizando estas y dedicando los recursos a la implementación de las *variable features* (características propias o variables) de cada producto y a su ensamblado.



1.2. Reutilización de conocimiento: patrones de diseño

Cuando en ingeniería de software se habla de reutilización, no debemos creer que esta consiste solamente en volver a utilizar funcionalidades predesarrolladas y empaquetadas en clases, componentes o servicios. También puede reutilizarse el conocimiento y la experiencia de otros desarrolladores. Efectivamente, existen multitud de problemas que se presentan una y otra vez cuando se desarrolla un producto software.

La gestión de la persistencia de objetos, por ejemplo, se presenta cada vez que se construye una aplicación que vaya a manipular información conectándose a una base de datos.

Como ingenieros de software, debemos ser capaces de identificar estas situaciones que se presentan frecuentemente y que han sido ya, con toda probabilidad, resueltas con antelación por otras personas de manera eficiente. Este tipo de conocimiento, en el que se describen problemas que aparecen de manera más o menos frecuente, y que incluyen en su descripción una o más buenas maneras de resolverlo, conforma lo que se llama un patrón. Un **patrón**, entonces, describe una solución buena a un problema frecuente, como ese que hemos mencionado de la gestión de la persistencia.

Ved también

Ved la asignatura *Análisis y diseño con patrones* del grado de Ingeniería Informática.

Revisitando el ejemplo del componente *Termostato* que se ha reemplazado por otro, a veces ocurre que el nuevo componente no puede integrarse directamente en el sistema: en la vida real, a veces es preciso colocar un poco de estopa o algún adhesivo entre dos tuberías que se deben conectar y que no llegan a encajar perfectamente. En el caso de la sustitución de componentes o servicios, el problema es muy parecido: disponemos de un componente o servicio que nos ofrece las funcionalidades que requerimos, pero las interfaces ofrecidas y esperadas no coinciden con el contexto en el que queremos integrarlo. Este es un problema recurrente que puede solventarse aplicando *glue code*⁸ quizás mediante un adaptador, que es la solución que propone el patrón *Wrapper*. El *proxy* que se citaba en el epígrafe anterior, y que decíamos que se utiliza para conectar el sistema a un servicio web externo, es también la solución que se propone en el patrón de diseño *proxy* para conectar un sistema a otro sistema remoto.

⁽⁸⁾Literalmente ‘código-pegamento’, que correspondería a esa estopa o a ese adhesivo.

Todo este conocimiento producido durante años de investigación y desarrollo en ingeniería del software puede reutilizarse y, de hecho se recopila y se hace público, en catálogos de patrones. El más famoso es el de Gamma, Helm, Johnson y Vlissides, pero todos los años se celebran, en diversos lugares del mundo, conferencias sobre patrones: la PLoP (Conference on Pattern Languages of Programs) a nivel internacional, la EuroPLoP en Europa o la KoalaPLoP en Oceanía. Para publicar un patrón en estas conferencias, el autor debe demostrar que el problema es importante y que se presenta con frecuencia, y debe presentar al menos tres casos distintos en los que se haya aplicado la misma solución: de esta manera, se demuestra que la solución ha sido utilizada satisfactoriamente más de una vez, y ofrece a la comunidad la posibilidad de reutilizar ese conocimiento.

1.3. Beneficios y costes de la reutilización

Existen dos enfoques bien distintos en cuanto a reutilización: el **enfoque oportunista** aprovecha elementos software construidos en proyectos anteriores, pero que no fueron especialmente desarrollados para ser reutilizados; en un **enfoque más planificado o proactivo**, los elementos se construyen pensando en que serán reutilizados, lo que puede significar, en el momento de su desarrollo, mayor inversión de recursos, puesto que se debe dotar al elemento de suficiente generalidad.

Nuestro enfoque en este material se centra en la **reutilización proactiva**.

De forma general, la reutilización permite:

- Disminuir los plazos de ejecución de proyectos, porque se evita construir nuevamente funciones, funcionalidades o componentes.
- Disminuir los costes de mantenimiento, que se discute más extensamente a continuación.
- Aumentar la fiabilidad, siempre que se utilicen elementos reutilizables procedentes de suministradores seguros.
- Aumentar la eficiencia, sobre todo si se reutiliza software desarrollado bajo el enfoque planificado o proactivo, porque los desarrolladores implementarán versiones optimizadas de algoritmos y de estructuras de datos. En un desarrollo nuevo, en el que la reutilización surgirá más adelante quizás por un enfoque oportunista, la presión de los plazos de entrega impide, en muchos casos, mejorar u optimizar los algoritmos.

Consulta recomendada

La primera edición del libro *Design patterns* se publicó en 1994. Ha sido traducido a multitud de idiomas, entre ellos el castellano: E. Gamma; R. Helm; R. Johnson; J. Vlissides (2002). *Patrones de diseño: elementos de software orientado a objetos reutilizables*. Addison Wesley.



- Aumentar la consistencia de los diseños y mejorar la arquitectura del sistema, especialmente cuando se utilizan *frameworks* o bibliotecas bien estructuradas que, en cierta manera, imponen al ingeniero de software un buen estilo de construcción del sistema.
- Invertir, porque el gasto ocasionado en dotar a un elemento de posibilidades de reutilización se verá recompensado con ahorros importantes en el futuro.

1.3.1. El impacto positivo de la reutilización en el proceso de mantenimiento

Un trabajo de Kung y Hsu de 1998 asimila la tasa de llegadas de peticiones de mantenimiento al modelo de ecología de poblaciones de May. En un ecosistema cerrado en el que solo hay predadores y presas, el aumento de predadores supone una disminución del número de presas, lo que implica que, al haber menos alimento, disminuya el número de predadores y aumente nuevamente el de presas. Con el avance del tiempo, estos altibajos van tendiendo hacia un punto de equilibrio.

Lectura adicional

H.-J. Kung; Ch. Hsu (1998). *Software Maintenance Life Cycle Model*. International Conference on Software Maintenance (págs. 113-121). IEEE Computer Society.

Modelo de Kung y Hsu

En el modelo de Kung y Hsu, los errores en el software son presas que son “devoradas” por las acciones de mantenimiento correctivo, que son los predadores. La introducción de un sistema supone la introducción masiva de errores, que se consumen en la etapa de crecimiento, al ser corregidos por los reportes de los usuarios. En la madurez apenas dan presas de la primera generación; sin embargo, las peticiones de perfectivo (adición de nuevas funcionalidades) introduce nuevos errores (nuevas presas, por tanto), que son consumidas por las correcciones que realizan los programadores.

Lectura adicional

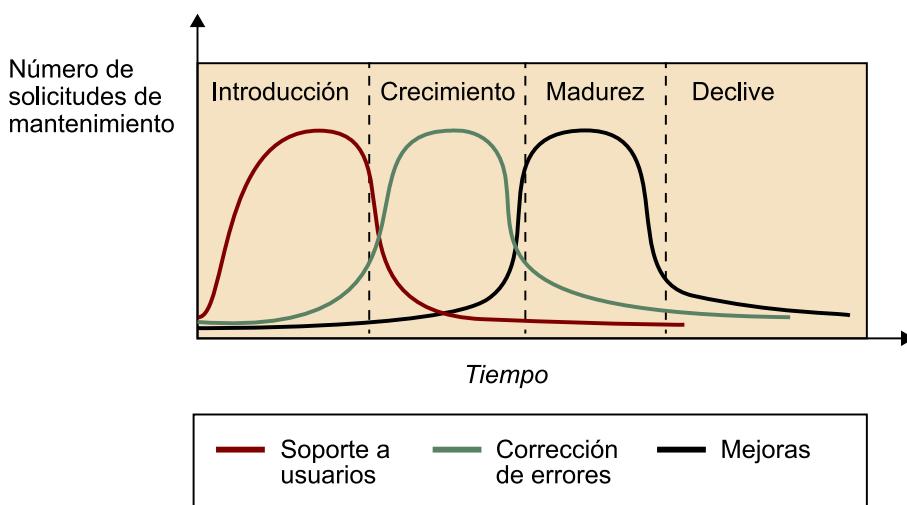
R. M. May (1974). *Biological Populations with Nonoverlapping Generations: Stable Points, Stable Cycles, and Chaos*. Science (vol. 4164, núm. 186, págs. 645-647).

El proceso de mantenimiento es uno de los procesos principales del ciclo de vida del software. Comienza una vez que el sistema ha sido instalado y puesto en explotación. En su caracterización, Kung y Hsu identifican cuatro etapas en la vida de un sistema:

- **Introducción**, que tiene una duración relativamente breve. Durante este periodo, los usuarios aprenden a manejar el sistema y envían al equipo de mantenimiento peticiones de soporte técnico, con objeto de aprender a utilizarlo por completo.
- **Crecimiento**. En esta etapa disminuye el número de peticiones de soporte porque los usuarios ya conocen bien el sistema y no necesitan ayuda técnica. Sin embargo, los usuarios comienzan a encontrar errores en el sistema y envían peticiones de mantenimiento correctivo.
- **Madurez**. Las correcciones que se han ido realizando durante la etapa de crecimiento han disminuido el número de peticiones de mantenimiento correctivo, y el sistema tiene ahora muy pocos errores. Los usuarios, sin

embargo, que ya conocen el sistema, demandan nuevas funcionalidades que se traducen a peticiones de mantenimiento perfectivo.

- **Declive.** Con el paso del tiempo, nuevos sistemas operativos y nuevos entornos favorecen la sustitución del sistema por uno nuevo, o su migración hacia plataformas más modernas, lo que puede dar lugar bien a la retirada del sistema o bien a su migración, que implicará la presentación de peticiones de mantenimiento adaptativo.



Al servir una petición de mantenimiento, el ingeniero de software debe, en primer lugar, ubicar el área del código que debe ser modificada, entender la lógica de la aplicación y sus conexiones con sistemas externos (tareas que son, por lo general, las más costosas), implementar el cambio y realizar las pruebas que sean necesarias, lo que incluirá pruebas de regresión para comprobar que la modificación no ha introducido errores que antes no existían.

La reutilización de software supone una disminución del mantenimiento correctivo, ya que la probabilidad de encontrar errores en una función utilizada en varios proyectos anteriores es muy pequeña. También disminuye los costes de mantenimiento perfectivo (adición de funcionalidades), porque resulta más sencillo entender un sistema en el que muchos de sus elementos son “cajas negras” que ofrecen correctamente una funcionalidad determinada y que, por tanto, no es necesario examinar.

1.3.2. Los costes de reutilizar

La reutilización de software tiene siempre unos costes asociados. Meyer indica que, al menos, estos son los siguientes:

- **Coste de aprendizaje** (al menos la primera vez).
- **Coste de integración en el sistema.** Por ejemplo, en el caso de la reutilización de componentes, muchas veces es preciso escribir adaptadores para

Consulta recomendada

B. Meyer (1997). *Object-oriented software construction* (2.^a ed.). Prentice Hall Professional Technical Reference.

conectar adecuadamente el componente en el sistema que se está desarrollando.

Obviamente, el enfoque proactivo en el desarrollo de software para que este sea reutilizable tiene también unos costes asociados. Una visión cortoplacista desaconsejaría dedicar esfuerzos especiales a la construcción de componentes ensamblables o de elementos de software genéricos (hablaremos de todo esto más adelante). Griss (1993) aconseja el desarrollo de modelos de cálculo del ROI (*return of investment*, retorno de la inversión) para convencer a la dirección de la conveniencia de tratar la reutilización como un activo importante.

Ved también

Ved la reutilización de componentes en el apartado “Componentes”.

Lectura adicional

Martin L. Griss (1993).
“Software reuse: from library to factory”. *IBM Software Journal* (vol. 4, núm. 32, págs. 548-566).

2. Reutilización en diseño de software orientado a objetos

Bertrand Meyer define la construcción de software orientado a objetos como la:

"Construcción de sistemas software en forma de colecciones estructuradas de, posiblemente, implementaciones de tipos abstractos de datos".

Los tipos abstractos de datos (TAD) describen matemáticamente el conjunto de operaciones que actúan sobre un determinado tipo de elemento. Cada operación se describe algebraicamente en términos de:

- 1) las transformaciones que realiza sobre las ocurrencias del tipo de dato;
- 2) los axiomas de la operación, y
- 3) las precondiciones necesarias para que la operación pueda ejecutarse.

Puesto que son abstractos y siguen el principio de abstracción, los TAD exhiben una vista de alto nivel de las posibilidades de manipulación de información que permite el tipo de dato, mostrando al exterior la parte que interesa y escondiendo los detalles que no interesa mostrar. Así, se hace también gala de la encapsulación y del ocultamiento de la información.

Ved también

Repasad los TAD y la orientación a objetos en las asignaturas *Diseño de estructuras de datos* y *Diseño y programación orientada a objetos* del grado de Ingeniería Informática.

Como en otras áreas del desarrollo de software, los TAD describen lo que sucede o debe suceder, pero no cómo sucede.

Ocurre lo mismo, por ejemplo, en el análisis funcional: el ingeniero de software identifica los requisitos del sistema y puede representarlos en un diagrama de casos de uso. En este caso, se indican las funcionalidades que debe servir el sistema, pero sin entrar en los detalles de cómo servirlas.

Junto a otros principios fundamentales del desarrollo de software (como el mantenimiento de la alta cohesión y del bajo acoplamiento), la abstracción, la encapsulación y el ocultamiento de la información son tres herramientas esenciales del diseño de software en general, no solo del orientado a objetos. Pero la orientación a objetos aporta también las posibilidades que brindan la herencia y el polimorfismo. En este apartado se presentan las principales características que ofrece la orientación a objetos en aras de permitir y fomentar la reutilización del software.

2.1. Introducción

En la mayoría de los libros de introducción a las estructuras de datos, se utilizan estructuras de almacenamiento de información (listas, pilas, colas, árboles, grafos) como ejemplos habituales.

Una pila, por ejemplo, se corresponde con una estructura de datos en la que los elementos se van colocando uno encima de otro, y de la que podemos sacar elementos en orden inverso al que han sido colocados: de este modo, la estructura de datos *Pila* abstrae el comportamiento de una pila real, como la pila que se crea cuando se friegan los platos, y luego cuando se van tomando en orden inverso para poner la mesa: se sigue una política LIFO (*last-in, first-out*: último en entrar, primero en salir) para gestionar sus elementos.

Cuando se programa el código de la pila en el ordenador, se puede representar la estructura de datos mediante un *array* estático, un vector dinámico, una lista enlazada, etcétera; sin embargo, a los diseñadores y a los usuarios del TAD pila les interesa tan solo la descripción de su comportamiento, sin que interese ofrecer los detalles de su implementación. De esta manera, el TAD pila es una abstracción de la estructura real pila que, además, encapsula sus detalles de estructura y funcionamiento.

TAD Pila(E)	Tres implementaciones de la Pila
Funciones <pre> apilar: Pila(E) × E → Pila(E) desapilar: Pila(E) → Pila(E) cima: Pila(E) → E esVacia: Pila(E) → booleano crear : Pila(E) </pre>	<p>The diagram illustrates three ways to implement a stack (Pila) using different data structures:</p> <ul style="list-style-type: none"> p(array) = An array of 6 slots (0 to 5). Slots 0, 1, and 2 contain elements A, B, and C respectively. Slots 3, 4, and 5 are empty. A vertical arrow labeled "cima" points down to slot 3, which contains a checkmark. A dashed arrow labeled "D" points from the end of the array to the right. p(Vector) = A vector of 3 slots (0 to 2). Slots 0, 1, and 2 contain elements A, B, and C respectively. A vertical arrow labeled "cima" points down to slot 2, which contains a checkmark. A dashed arrow labeled "D" points from the end of the vector to the right. p(lista enlazada) = A linked list starting at node A. Node A has value A and a pointer to node B. Node B has value B and a pointer to node C. Node C has value C and a pointer to node D. Node D has value D and a pointer to null. A vertical arrow labeled "cima" points down to node C, which contains a checkmark. A dashed arrow labeled "D" points from the end of the list to the right.
Axiomas (sean: $x \in E$; p una <i>Pila(E)</i>): <pre> esVacia(crear) = cierto esVacia(apilar(p, x)) = falso cima(apilar(p, x)) = x desapilar(apilar(p, x)) = p </pre>	
Precondiciones <pre> desapilar(p) requiere [esVacia(p) = falso] cima(p) requiere [esVacia(p) = falso] </pre>	

En la especificación del TAD pila que se da en la figura anterior, la pila trabaja con elementos genéricos de tipo *E* que, en la práctica, pueden ser números enteros o reales, letras, cadenas de caracteres, o estructuras más complejas, como personas, cuentas corrientes, abstracciones del objeto real plato o, incluso, otras pilas. Todas las pilas (almacenén el tipo de elemento que almacenén) se

comportan exactamente igual: si se coloca un elemento en la cima, ese mismo elemento será el primero en salir, independientemente de que apilemos números o letras.

El comportamiento que describimos en el TAD pila es entonces un comportamiento genérico: el lenguaje C++ es uno de los pocos que permiten la implementación de TAD genéricos, como la pila que se muestra en la figura siguiente, en la que se define, mediante una *template* (plantilla) de C++, una pila que actúa sobre un tipo de dato genérico T. Si en el código instanciamos una pila de int y otra del tipo persona, el compilador genera una versión en código objeto de la pila de enteros y otra de la pila de personas.

Pila genérica en C++	Pila genérica en UML
<pre>template <class T> class Pila { public: Pila() { cima = -1; } void apilar(T e) { elementos [++cima] = e; } T desapilar() { return elementos [cima--]; } T cima() { return elementos [cima]; } int esVacia() { return cima== -1; } private: int cima; T elementos[100]; };</pre>	<pre> classDiagram class Pila { +cima : int +elementos : T[] +Pila() +apilar(e : Apilable) +desapilar() : Apilable +cima() : Apilable +esVacia() : boolean } T "1" -- "1" Pila </pre>

Otros lenguajes de programación orientados a objetos no permiten la definición de plantillas, aunque sí dejan la descripción de pilas de cualquier tipo de elemento mediante la herencia. Es el caso de C# y de las primeras versiones de Java, cuyo tipo Object representa a cualquier tipo de objeto. En estos dos lenguajes, todos los objetos son también instancias del tipo genérico Object, por lo que se puede implementar una pila de objetos en la que será posible almacenar cualquier cosa, como cadenas, enteros o reales, pero también *Termómetros* o *IActuadores* como los que mostrábamos en el apartado anterior.

Nota

A partir de la versión 5 de Java se pueden utilizar los *generics*, que permiten crear plantillas similares a C++:

```
public class Pila <T>
```

Vé tambiéñ

Veremos la programación genérica en el apartado “Fábricas de software”.

Pila de objetos en Java	Pila de objetos en UML
<pre>public class Pila { private int cima; private Object[] elementos; public Pila() { this.cima=-1; this.elementos=new Object[100]; } public void apilar(Object e) { elementos[++cima] = e; } public Object desapilar() { return elementos[cima--]; } public Object cima() { return elementos [cima]; } public boolean esVacia() { return this.cima== -1; } }</pre>	<pre> classDiagram class Pila { -cima : int -elementos +Pila() +apilar(e : Object) +desapilar() : Object +cima() : Object +esVacia() : boolean } class Object Pila "0..100" *-- "elements" Object </pre> <p>The diagram illustrates the UML representation of the stack class. It consists of two classes: Pila and Object. The Pila class has attributes <code>-cima</code> (int) and <code>-elementos</code> (Object[]), and operations <code>+Pila()</code>, <code>+apilar(e : Object)</code>, <code>+desapilar() : Object</code>, <code>+cima() : Object</code>, and <code>+esVacia() : boolean</code>. It has a directed association named <code>elements</code> pointing to the Object class, with multiplicity <code>0..100</code> at the Pila end and no multiplicity at the Object end.</p>

Del mismo modo, haciendo uso de la herencia se puede definir un tipo de dato *Apilable*, que represente los objetos que, en el dominio del problema que se esté resolviendo, puedan colocarse en la *Pila*:

Pila de objetos apilables implementada en Java	Pila de objetos apilables en UML
<pre>public class Pila { private int cima; private Apilable[] elementos; public Pila() { this.cima=-1; this.elementos=new Apilable[100]; } public void apilar(Apilable e) { elementos[++cima] = e; } public Apilable desapilar() { return elementos[cima--]; } public Apilable cima() { return elementos [cima]; } public boolean esVacia() { return this.cima== -1; } }</pre>	<pre> classDiagram class Pila { -cima : int -elementos : Apilable[] +Pila() +apilar(e : Apilable) +desapilar() : Apilable +cima() : Apilable +esVacia() : boolean } Pila "0..100" -- "elementos" Apilable </pre>

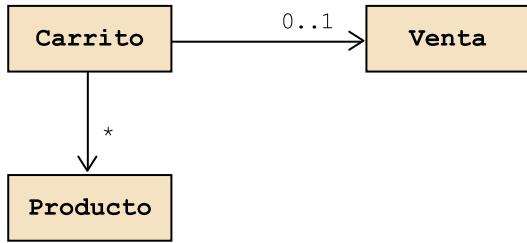
2.2. Abstracción, encapsulación y ocultamiento

En nuestro contexto, la abstracción es el mecanismo que utiliza el ingeniero de software para representar conceptos del mundo real utilizando alguna notación que, en algún momento y de alguna manera (tal vez mediante varios pasos intermedios), pueda ser procesada y traducida a algún formato entendible por un ordenador.

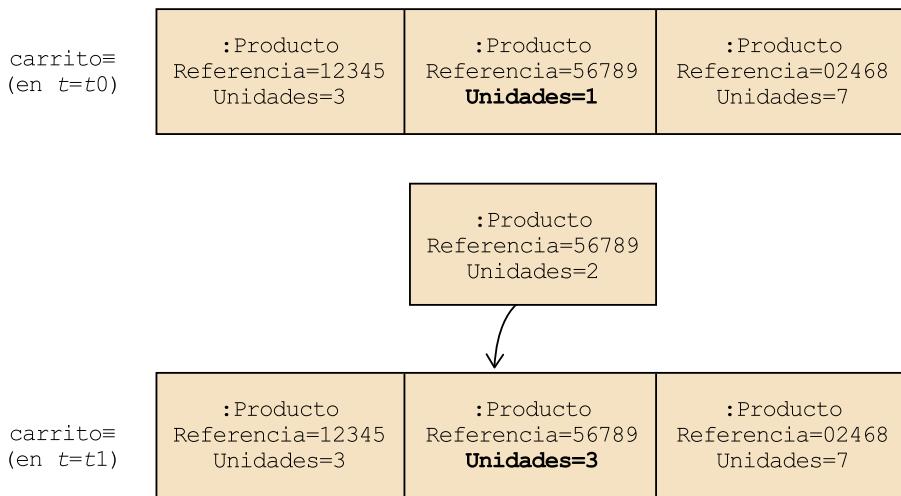
Como se ha sugerido en párrafos anteriores, la orientación a objetos surge como una evolución de los tipos abstractos de datos: de acuerdo con Jacobson (uno de los padres del UML) y colaboradores, tanto un TAD como una clase son abstracciones definidas en términos de lo que son capaces de hacer, no de cómo lo hacen: son, entonces, generalizaciones de algo específico, en las que la encapsulación juega un papel fundamental. En el diseño de una tienda virtual, por ejemplo, se utilizan conceptos (abstracciones) como *Carrito*, *Producto* o *Venta*, que pueden representarse utilizando una notación abstracta (como UML), incluyendo las relaciones entre dichas abstracciones.

Consulta recomendada

I. Jacobson; M. Christerson; P. Jonsson; G. Övergaard (1992). *Object-Oriented Software Engineering. A use case approach*. Addison-Wesley.



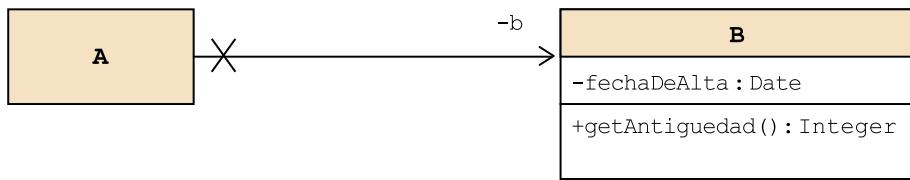
A la hora de implementar el *Carrito*, el *Producto* y la *Venta*, el programador deberá considerar, por ejemplo, si la colección de productos contenidos en el carrito la va a representar con un vector, con una tabla de dispersión (*hash*) o con una estructura de datos de otro tipo. Independientemente de la estructura de datos seleccionada, el programador debe conseguir por ejemplo que, si se añaden dos unidades de un producto que ya estaba contenido en el carrito (el producto con referencia 56789, por ejemplo), tan solo se incremente el número de unidades. La forma con la que el programador consiga ese comportamiento no forma parte del principio de abstracción (porque es un detalle demasiado concreto sobre el funcionamiento de los carritos en la tienda virtual). Por el contrario, este comportamiento que el programador ha implementado (y que desde el punto de vista de la abstracción no nos interesa) sí que se corresponde con el principio de encapsulación: digamos que no sabemos cómo funciona la operación, pero sí sabemos que funciona.



En orientación a objetos, una clase es una plantilla de estructura y comportamiento que se utiliza para crear objetos. La parte de estructura se utiliza para representar el estado de los objetos de esa clase (es decir, de sus instancias), mientras que la de comportamiento describe la forma con la que puede alterarse o consultarse el estado de dichos objetos, o solicitarles que ejecuten algún servicio sobre otros. A nivel práctico, la estructura se compone de un conjunto de campos o atributos, mientras que el comportamiento se describe mediante una serie de operaciones o métodos. Por lo general, la estructura de una clase (es decir, su conjunto de campos) permanece oculta al resto de ob-

jetos, de manera que solo el propio objeto es capaz de conocer directamente su estado: cada objeto muestra al exterior solo aquello que el propio objeto está interesado en mostrar.

En la siguiente figura, cada objeto de clase A conoce una instancia B; las instancias de B poseen un estado que viene determinado por un campo privado (es decir, oculto al exterior) llamado `fechaDeAlta`; sin embargo, B exhibe a A una porción de su estado (la antigüedad en años), que hace accesible mediante la operación pública `getAntiguedad()`, y que probablemente se calculará en función de la fecha del sistema y de la fecha de alta.



Del mismo modo, cuando un objeto A quiere hacer algo sobre otro objeto B, A solo podrá actuar utilizando los elementos de B que el propio B le permita: es decir, B ofrece a A un conjunto de servicios (operaciones o métodos públicos) que son un subconjunto de las operaciones incluidas e implementadas en B. B, por tanto, es el objeto que tiene realmente la responsabilidad de controlar sus cambios de estado: A puede intentar desapilar un elemento de una pila vacía, pero la propia instancia de *Pila* ha de ser responsable de decir, de alguna manera (quizás mediante el lanzamiento de alguna excepción), que de ella no puede desapilarse porque no tiene elementos, de manera que su estado permanezca inalterable.

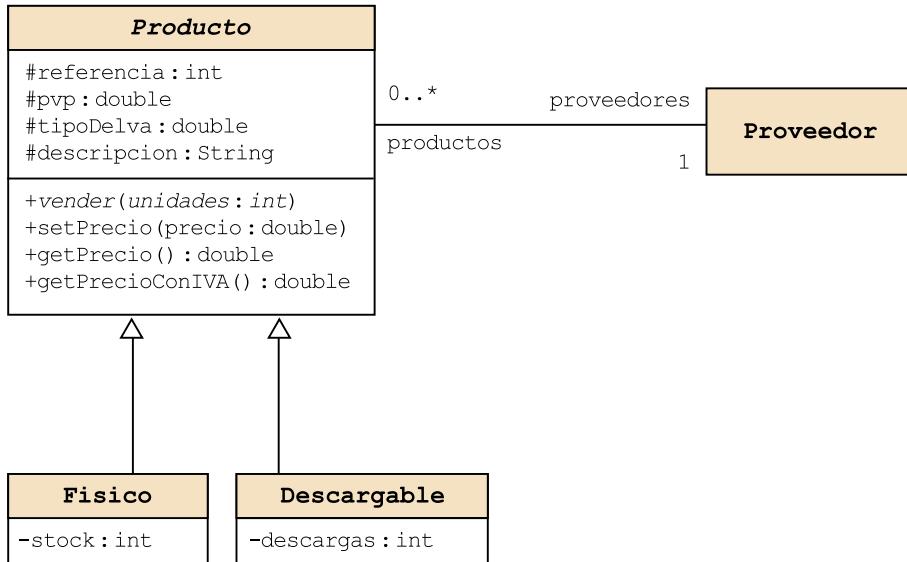
2.3. Herencia y polimorfismo

Supongamos que la tienda virtual oferta dos tipos de productos: **productos convencionales**, que requieren un almacén y de los que hay un número determinado de unidades en stock, y **productos electrónicos**, que se venden por descargas y que no tienen necesidad de almacenamiento físico. Es muy probable que ambos tipos de productos comparten un conjunto amplio de características⁹ y parte del comportamiento¹⁰. El producto físico, sin embargo, aporta a su estructura (es decir, a su conjunto de campos) el número de unidades que se tienen en el almacén, mientras que el producto descargable aporta, por ejemplo, el número de descargas que ha tenido.

⁽⁹⁾Por ejemplo: referencia, descripción, proveedor, precio y tipo de IVA.

⁽¹⁰⁾Los dos se dan de alta y de baja, los dos se venden, se puede modificar el precio de ambos, etcétera.

Así pues, la tienda manipulará *Productos* que podrán ser *Físicos* o *Descargables*. Puesto que ambos tienen en común parte de la estructura y del comportamiento, es posible definir, en un diseño orientado a objetos, una superclase *Producto* en la que ubicaremos todos aquellos elementos comunes a las dos subclases que, en este caso, serán *Físico* y *Descargable*.



En la figura anterior se representa la estructura de la clase *Producto*, en la que se han ubicado todos los elementos comunes a los dos tipos de productos que vende la tienda. El hecho de que aparezca en cursiva el nombre de la superclase y en redonda los de las subclases denota que *Producto* es una clase abstracta, mientras que *Físico* y *Descargable* son clases concretas: es decir, en la tienda existen *Productos*, pero han de ser o bien *Físicos* o bien *Descargables*, no pudiendo existir *Productos* como tales.

La figura ilustra la capacidad de reutilización que ofrece la herencia: la definición genérica (de estructura y de comportamiento) que se da en la superclase se reutiliza completamente en las subclases que, en este ejemplo, aportan algo de estructura a lo heredado.

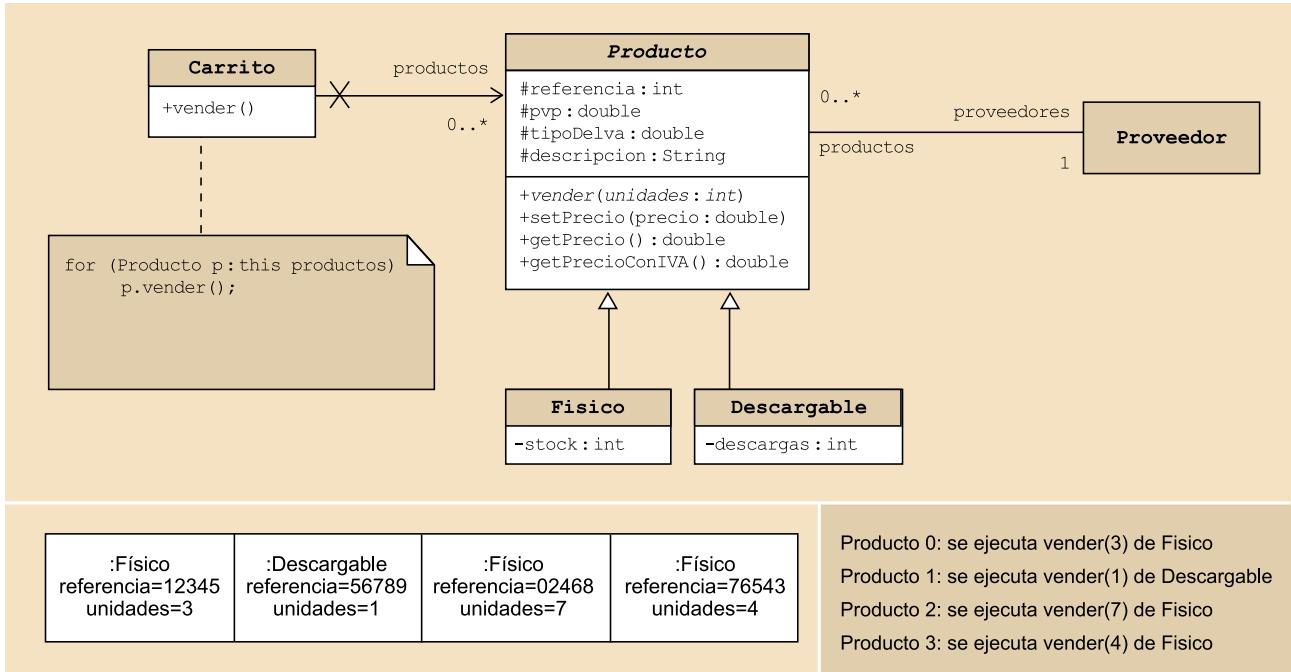
Obsérvese, por otro lado, que la operación *vender()* de *Producto* aparece (al igual que el nombre de la clase contenedora) también en cursiva, lo que denota que se trata de una operación abstracta: puesto que es diferente vender un producto físico de un producto descargable (en el primero se decrementa el stock; en el segundo se incrementan las descargas), es preciso dar una implementación diferente en las subclases a la operación abstracta que se está heredando.

Producto, además, tiene una serie de operaciones concretas (*setPrecio()*, *getPrecio()* y *getPrecioConIVA()*), que se comportan exactamente igual con todos los productos, sean físicos o descargables: en este caso, la herencia nos permite reutilizar ya no solo parte de la estructura, sino también parte del comportamiento, a través de las operaciones heredadas.

Además, la inclusión de la operación abstracta *vender()* en *Producto* nos permite manejar, por ejemplo, colecciones de productos genéricos¹¹ y ejecutar sobre ellos la operación *vender()*. En tiempo de ejecución, y en función del tipo concreto de cada *Producto* contenido en la colección, se decide cuál de las dos

(11)En tiempo de ejecución, estarán obviamente instanciados a productos concretos de los subtipos físico y descargable, ya que no pueden existir instancias reales de la clase abstracta producto.

versiones de la operación *vender()* debe ejecutarse: si la instancia es del subtipo *Físico*, la operación *vender()* será de *Físico*; si es del subtipo *Descargable*, *Vender()* será de *Descargable*.



El carrito contiene una lista de productos “genéricos”. En tiempo de ejecución, cada producto está instanciado a uno de los dos subtipos concretos. Al ejecutar *vender* en el carrito, se llama a la operación abstracta *vender(unidades:int)* de cada producto. Se ejecuta una u otra versión de *vender* en función del tipo concreto de cada producto.

La operación *vender(unidades:int)* es una operación polimórfica, pues “tiene muchas formas¹²”. Como se ilustra en el caso de la figura anterior, el polimorfismo permite hacer referencia a distintos comportamientos de distintos objetos, pero que se encuentran designados por el mismo mensaje (*vender*, en este caso).

⁽¹²⁾En este ejemplo realmente solo dos, pero podría haber más versiones diferentes de la operación si hubiera más subtipos de *Producto*.

Polimorfismo

Cuando una instancia envía un estímulo a otra instancia, pero sin que aquella esté segura de a qué clase pertenece la instancia receptora, se dice que tenemos polimorfismo.

Consulta recomendada

Jacobson; M. Christerson; P. Jonsson; G. Övergaard (1992). *Object-Oriented Software Engineering. A use case approach*. Addison-Wesley.

3. Reutilización a pequeña escala: soluciones técnicas de reutilización

3.1. Introducción

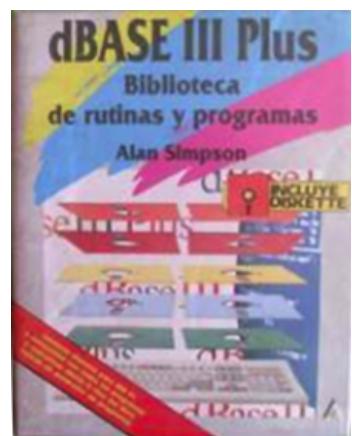
Como se comentó en las primeras líneas del primer apartado, la necesidad de producir software reutilizable se puso de manifiesto en el mismo congreso en el que se empleó por primera vez, hace más de cuarenta años, el término *ingeniería del software*. Esta coincidencia permite tomar conciencia de la importancia que, desde el nacimiento mismo de esta disciplina, tiene la reutilización de software y los esfuerzos llevados a cabo para poder establecer métodos que permitan su producción industrial.

Escribir una y otra vez las mismas rutinas o funciones es una práctica que debería ser evitada a toda costa: la investigación y el desarrollo en ingeniería del software ha llevado a la implementación de diferentes tipos de soluciones para favorecer la reutilización. Tras presentar un fragmento muy ilustrativo de la reutilización de software de hace solo 20 años, este apartado revisa las soluciones de reutilización más próximas a la implementación de código: librerías, clases, patrones arquitectónicos, patrones de diseño, componentes, *frameworks* y servicios.

3.2. Una historia ilustrativa

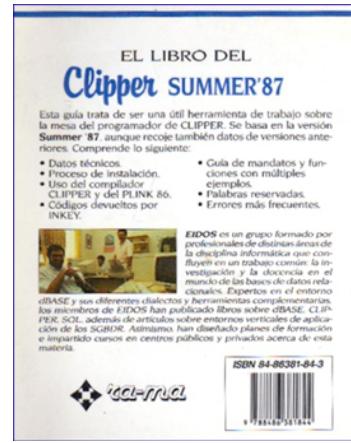
dBase fue un sistema gestor de bases de datos muy popular a finales de los años ochenta y principios de los noventa y que fue desarrollado y comercializado por la empresa Ashton-Tate. Además de permitir la creación de tablas e índices, dBase incorporaba un lenguaje de programación con el que se podían implementar de manera relativamente sencilla aplicaciones para gestionar sus propias bases de datos, lo que incluía la posibilidad de crear pantallas (a la sazón, basadas en texto y no en gráficos) con menús, formularios de mantenimiento de registros, obtención de listados, impresión, etcétera.

Uno de los problemas de dBase es que sus programas eran interpretados y no compilados, por lo que los desarrolladores debían entregar a sus clientes el código fuente de los programas que escribían, con el consiguiente perjuicio en cuanto a la conservación de sus derechos de autor y propiedad intelectual. Sabedores de esta desventaja, la empresa Nantucket Corporation (posteriormente adquirida por Computer Associates) creó en 1985 el sistema Clipper, que era, en sus principios, un compilador del lenguaje de dBase que generaba ejecutables en formato binario, lo que impedía el acceso al código fuente de los programas.



dBase III Plus es uno de los libros sobre bibliotecas y rutinas para reutilización de código más vendido en aquellos años. Puede encontrarse más información en Wikipedia: <http://es.wikipedia.org/wiki/DBase>.

De la mano de sus fabricantes, Clipper evolucionó de manera paralela a dBase, incorporando, con el tiempo, nuevas funciones y mejoras respecto de las ofrecidas por el sistema interpretado de Ashton-Tate. De esta manera, se llegó a que cualquier programa escrito en dBase podía ser compilado por Clipper; además, los programas que estaban escritos directamente para Clipper permitían la inclusión en su código de llamadas a funciones adicionales que, gracias a que Clipper estaba implementado en Ensamblador y en C, podían residir en ficheros separados y que no eran, entonces, ejecutables desde el entorno de dBase. Clipper traducía los programas dBase a ficheros objeto con extensión .obj.



Contraportada de *El libro del Clipper Summer'87*, publicado en 1989 por la editorial Ra-Ma. Explicaba, por ejemplo, los métodos de compilación y enlace.

Para construir el fichero que finalmente pudiera ser ejecutado por el sistema operativo de que se tratase, era preciso enlazar o *linkar* el programa objeto con los ficheros externos en los que residían las funciones auxiliares utilizadas por el programador. A estos ficheros externos (que se almacenaban en disco con extensión .lib) se los llamaba librerías o bibliotecas (*libraries*) y ofrecían al programador funciones de uso común. Las librerías estándares distribuidas con Clipper eran CLIPPER.LIB, EXTEND.LIB y OVERLAY.LIB, y ofrecían funciones como abs¹³, aCopy¹⁴ u Overlay¹⁵.

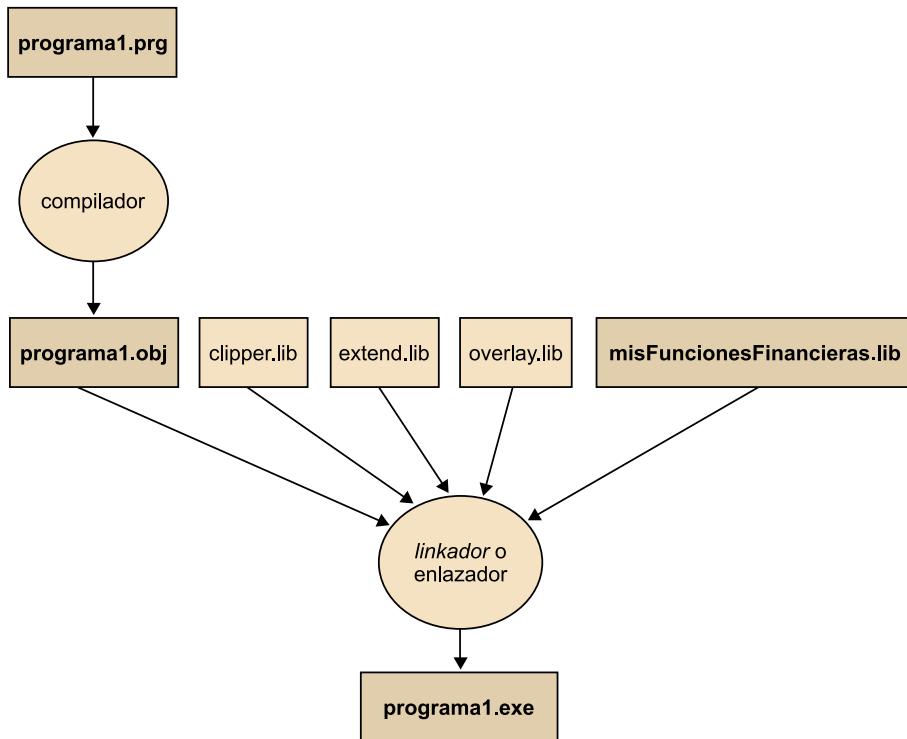
Sin embargo, cualquier programador de C podía implementar funciones adicionales en otras librerías que, previo enlace, podían ser utilizadas en cualquier programa¹⁶. Una de estas librerías no estándares que gozó de cierto éxito era FiveWin, del fabricante español FiveSoftware, y que se utilizó para migrar las primitivas aplicaciones Clipper (desarrolladas para terminales basados en texto) hacia el entonces emergente sistema de Microsoft Windows, que ya ofrecía a los usuarios un entorno gráfico de ventanas.

(¹³)Para calcular el valor absoluto, y que residía en CLIPPER.LIB.

(¹⁴)Utilizada para copiar arrays y que se encontraba en EXTEND.LIB.

(¹⁵)Permitía "puentejar" al sistema operativo para gestionar más eficazmente la memoria del ordenador, entonces muy limitada, y que se encontraba en OVERLAY.LIB

(¹⁶)En la figura siguiente se representa, con trazo más grueso, una librería no estándar que incluye funciones financieras que se utilizan en el programa1.prg.



El enlace estático producía programas ejecutables de gran tamaño, lo que podía suponer problemas de gestión de memoria con los sistemas operativos más comunes en aquellos años, como MS-DOS.

3.3. Librerías

El concepto de librería se ajusta bastante bien a la idea que, respecto de ellas, se ha presentado en el epígrafe anterior dedicado a dBase y Clipper.

En efecto, una librería es un conjunto reutilizable de funciones que se agrupan normalmente en un solo fichero.

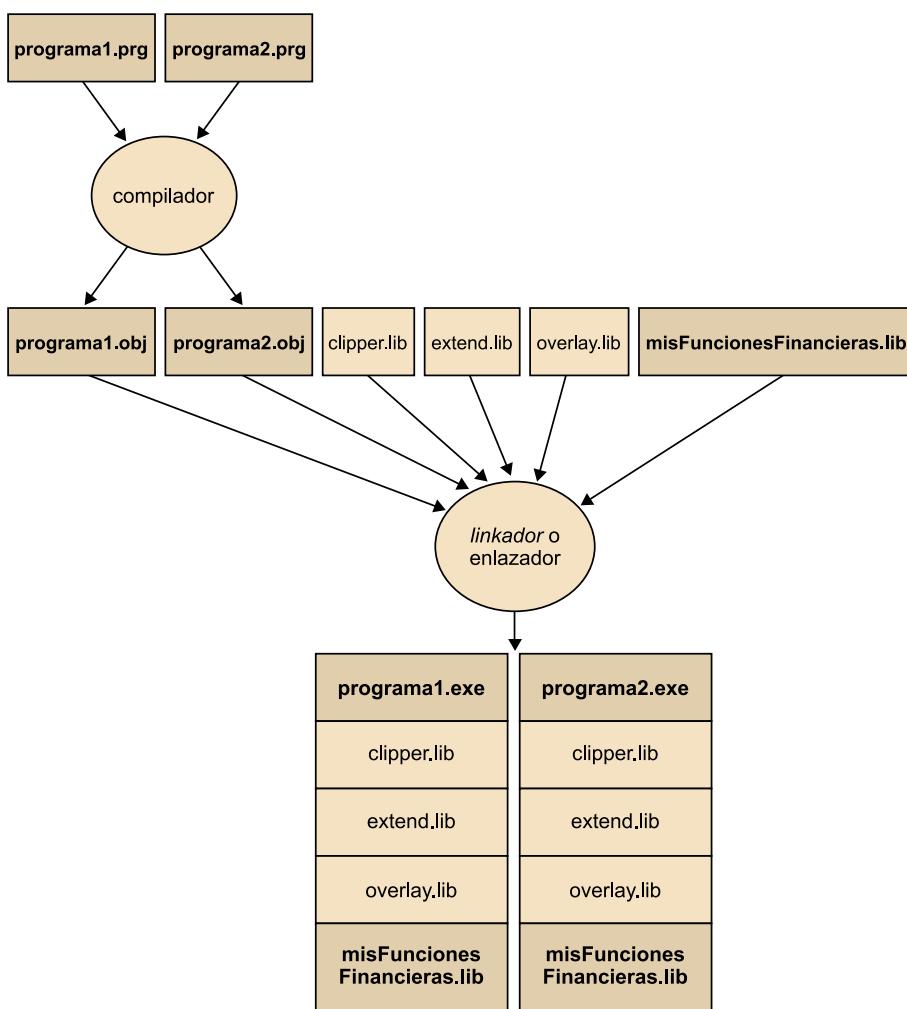
Si el formato de la librería y el entorno operativo lo permiten, los desarrolladores pueden incluir en sus programas llamadas a las funciones ofrecidas en la librería.

En general, cada librería agrupa un conjunto cohesionado de funciones, como por ejemplo: librerías de funciones matemáticas y trigonométricas, librerías para la creación de gráficos, librerías para funciones de búsqueda en textos mediante expresiones regulares, etcétera.

Existen dos enfoques importantes a la hora de hacer uso de las funciones contenidas en una librería, y que vienen motivados por el hecho de que las librerías residen en ficheros externos al sistema que se está desarrollando: el **enlace estático** y el **enlace dinámico**.

3.3.1. Librerías de enlace estático

En los primitivos sistemas de librerías, como los que se han ilustrado con el ejemplo de Clipper, todos los programas que se desarrollaban y que hicieran uso de, por ejemplo, la función *abs* debían ser enlazados (para generar el ejecutable) junto a la librería clipper.lib. Este hecho causaba desventajas importantes en cuanto al uso del disco y de la memoria, pues todo el código objeto de la librería era combinado, durante el proceso de enlazado, con el código objeto del programa, lo que producía programas de tamaño muy grande: si los programas *programa1* y *programa2* usaban la función *abs*, sus dos ejecutables correspondientes incluían la librería completa.

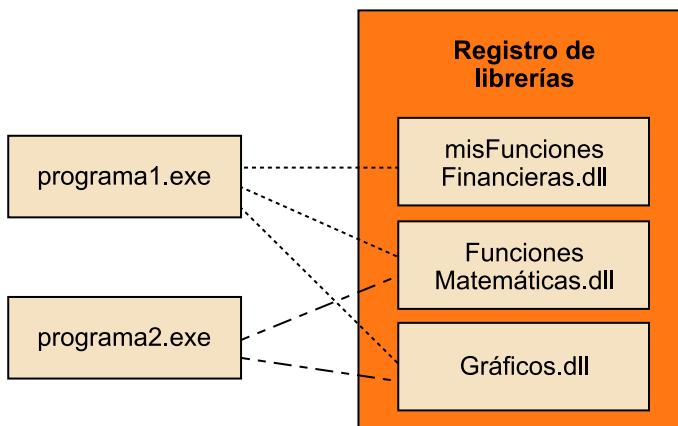


3.3.2. Librerías de enlace dinámico

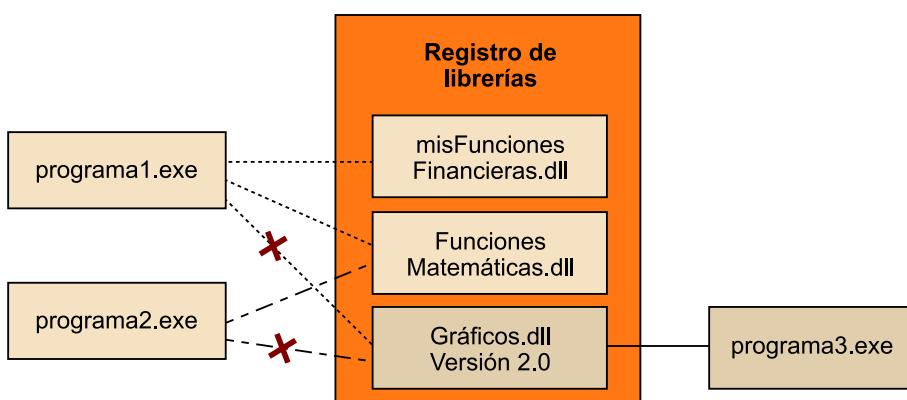
La solución al problema anterior consiste en la implementación de librerías de enlace dinámico (*dynamic-link libraries* o DLL), que no requieren el enlazado del código escrito por el programador con el código incluido en la librería. Ahora, cuando un ejecutable necesita llamar a una función de librería, se la

pide al sistema operativo, que guarda un registro de todas las DLL con sus funciones. De este modo, dos programas que requieran usar la misma función no tienen código duplicado, sino que comparten un único ejemplar de la librería.

En la siguiente figura se ilustra este concepto: ambos programas utilizan (comparten) las librerías de funciones matemáticas y de manejo de gráficos, y el programa1 es el único que utiliza las funciones financieras. Como se observa, las librerías residen en ficheros separados que no se enlazan para generar los ejecutables.

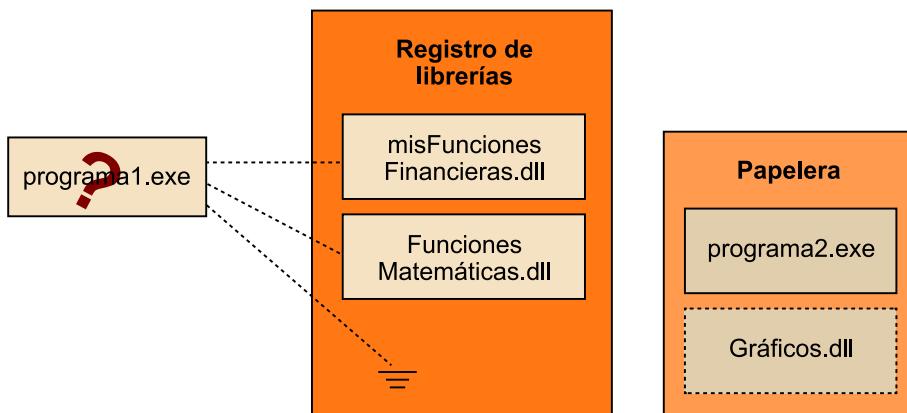


Sin embargo, la introducción de librerías de enlace dinámico supuso también la aparición de problemas nuevos que, con el enlace estático, no se presentaban: así, la instalación de una nueva aplicación en el sistema podía reemplazar una versión antigua de una DLL por una más moderna, sucediendo en ocasiones que esta nueva versión era incompatible con otros programas previamente instalados. Si en la figura anterior se instala un nuevo programa3 en el sistema que instala también una versión nueva de la librería Gráficos.dll y esta es incompatible con los dos programas preexistentes, estos podrían dejar de funcionar.



Análogamente, la desinstalación de una aplicación puede suponer la eliminación de sus librerías de enlace dinámico asociadas. Si el sistema operativo no lleva un control adecuado de las aplicaciones que hacen uso de las librerías registradas, o si ha habido instalaciones o desinstalaciones manuales, es posible

que se eliminen librerías requeridas por otras aplicaciones. La figura siguiente ilustra esta situación: por un error en el registro de librerías, la eliminación del programa2 conlleva también la eliminación de Gráficos.dll, por lo que es previsible que el programa1 deje de funcionar o tenga comportamientos inesperados.



3.3.3. Otras librerías

El término *librería* o *biblioteca* se utiliza normalmente para hacer referencia a librerías o bibliotecas de funciones, pero pueden referirse a cualquier tipo de almacén estático de recursos reutilizables, como archivos de imágenes o de sonidos.

3.4. Clases

La creciente implantación de la orientación a objetos, con las capacidades para la abstracción que permiten la herencia, el polimorfismo y la encapsulación de comportamiento y datos en una misma unidad, favoreció la aparición de diversos tipos de elementos reutilizables: para el desarrollo de un nuevo proyecto, una compañía puede reutilizar los diseños hechos para un proyecto desarrollado con anterioridad; puede también reutilizar clases que ofrezcan un conjunto de servicios que, nuevamente, puedan ser útiles; y puede, en tercer lugar, reutilizar el código fuente para, mediante algunas modificaciones, adaptarlo al nuevo desarrollo.

Puesto que la documentación técnica del proyecto, en muchos casos, no evoluciona acorde con la implementación a la que dio lugar, la reutilización pura y simple del diseño de un sistema encierra el riesgo grave de reutilizar un diseño que contenga defectos que se detectaron con el producto ya implementado, y que fueron corregidos solo sobre el código fuente: como indica Meyer, la sola reutilización del diseño puede llevar a reutilizar elementos incorrectos u obsoletos. Para este autor, la reutilización de diseños debe llevarse a cabo con un enfoque que elimine la distancia entre diseño e implementación, de tal manera que el diseño de, por ejemplo, una clase, pueda ser considerado

como una clase a la que le falta algo de implementación; y viceversa: gracias a los mecanismos de abstracción, una clase implementada puede considerarse un elemento de diseño más.

La reutilización del código fuente, por otro lado, tiene la ventaja de que permite al reutilizador modificarlo para adaptarlo a sus necesidades reales; sin embargo, al entregar todo el detalle de la lógica del sistema se está violando el principio de ocultamiento de información, y el desarrollador puede estar tentado de, al no llegar a entender exactamente la política de privacidad o publicidad de las operaciones, alterar el código, introduciendo efectos colaterales no deseados.

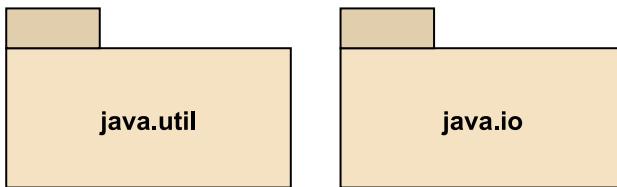
Nota

No siempre se desea poner a disposición de terceros el código de las aplicaciones: recuérdese que una de las razones que llevaron a Nantucket Corporation a crear Clipper fue que los programadores de dBase se veían obligados a entregar su código.

El elemento básico de reutilización en orientación a objetos es la clase. Normalmente, se agrupa en un paquete (*package*, en UML) un conjunto cohesionado de clases reutilizables.

Un paquete de UML es un mecanismo utilizado para agrupar elementos de cualquier tipo (casos de uso, clases, diagramas de secuencia de un cierto caso de uso, una mezcla de casos de uso y clases, otros paquetes, etcétera).

Si bien los paquetes de clases son los más interesantes desde el punto de vista de la reutilización: el paquete `java.util` ofrece un conjunto muy amplio de clases e interfaces para manejar colecciones, eventos, fechas y horas, internacionalización y otras pocas más; `java.io` contiene clases para manejar la entrada y salida.

**Nota**

Dentro del JDK, Java ofrece una amplia API con paquetes formados por clases e interfaces relacionadas para una enorme variedad de usos utilitarios, que permite a los desarrolladores hacer de este un uso para construir aplicaciones Java. El API para la versión 7 de Java SE se encuentra en docs.oracle.com/javase/7/docs/api/.

3.5. Patrones arquitectónicos

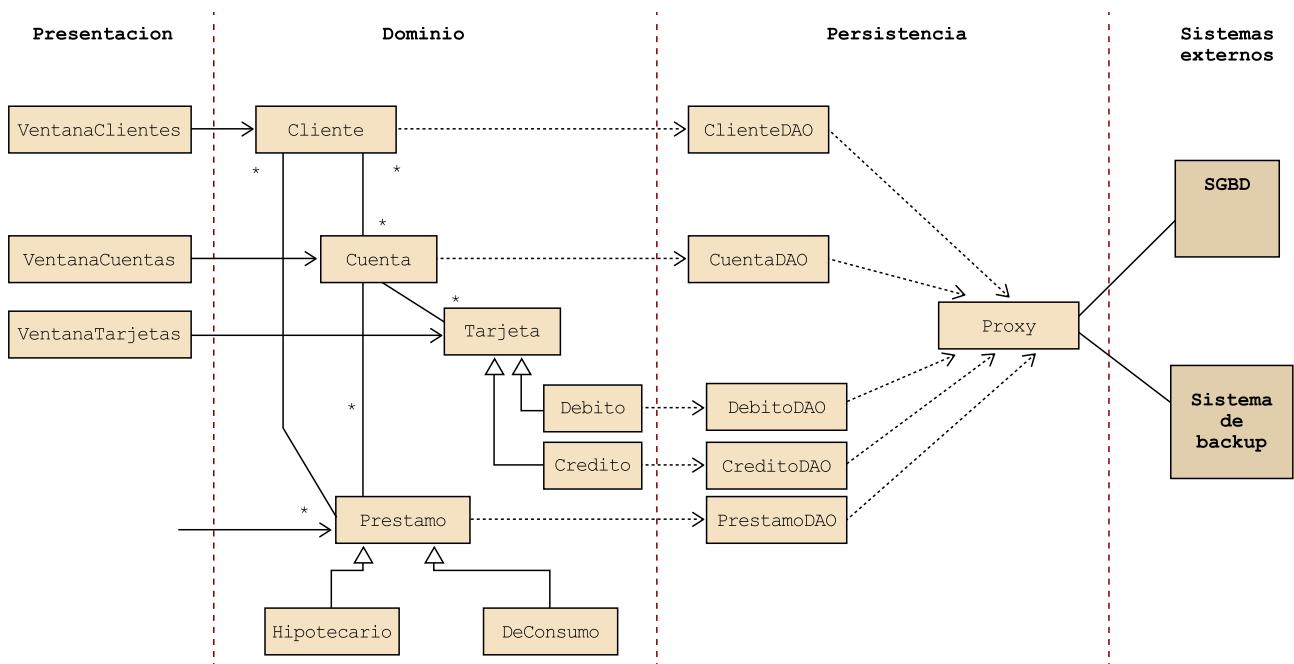
La **arquitectura** (o diseño arquitectónico) de un sistema se corresponde con su diseño al más alto nivel. Al plantear la arquitectura de un sistema se prescinde de los detalles, enfocando la atención en aquellos elementos generales que, merced al principio de abstracción, llevan a cabo una o más tareas de computación. Así como un diseño arquitectónico equivocado llevará probablemente al fracaso del proyecto, uno bueno permite:

- Reconocer relaciones de alto nivel entre sistemas o subsistemas.

- Favorecer la reutilización y la ampliación de sistemas o subsistemas ya existentes.
- Hacer un aporte fundamental para entender y describir, a alto nivel, un sistema complejo.

3.5.1. Arquitectura multicapa

En un diseño arquitectónico multicapa, el sistema se descompone en capas (*tiers o layers*), en cada una de las cuales se ubica un conjunto más o menos relacionado de responsabilidades. En rigor, una capa debe conocer solo a sus adyacentes. La figura siguiente muestra el diseño de un sistema con tres capas: una primera capa de presentación, en la que se ubican las clases que utilizan el usuario para interactuar con el sistema, y que estarán constituidas por ventanas; una segunda capa de dominio, en la que residen las clases que se encuentran en el enunciado del problema, y que son las que tienen la responsabilidad real de resolverlo; una tercera capa de persistencia, en la que se ubican las clases que se encargan de gestionar la persistencia de las instancias de clases de dominio. Hay, además, una cuarta capa en la que residen otros sistemas externos con los que este se comunica pero que, en rigor, está fuera del alcance del sistema objeto de estudio.

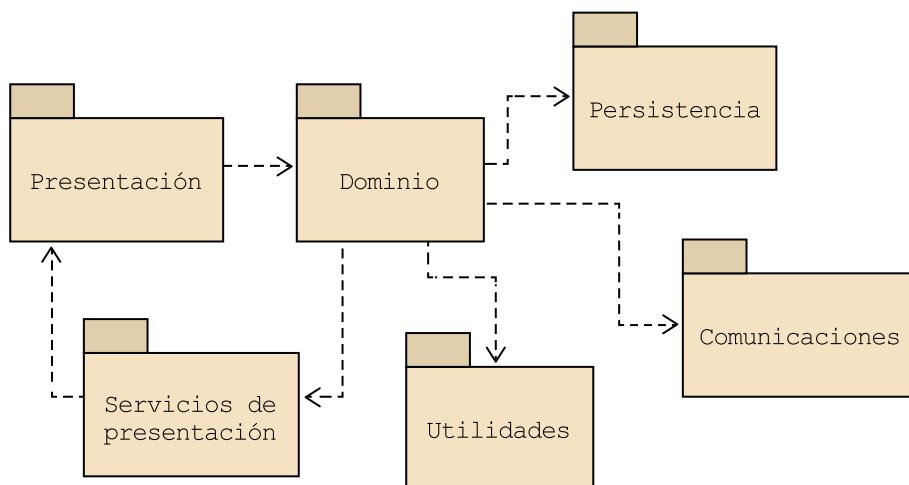


Arquitectónicamente, las capas pueden representarse como paquetes de UML. Las relaciones de dependencia representan habitualmente relaciones de uso: en la figura siguiente, la capa de presentación conoce a dominio; ya que dominio es la capa que contiene las clases que verdaderamente resuelven el problema (y que tienen la mayor parte de la complejidad y, probablemente, el mayor interés en ser reutilizadas) interesa que el dominio esté poco acoplado al resto de capas: por ello, se desacopla el dominio de la presentación mediante

Ved también

Ved también los patrones en el apartado 3.6. En particular en el apartado 3.6.2. encontraréis el patrón *Fabricación pura* que explica las clases DAO de la figura.

un subsistema de servicios (que incluirá observadores, patrón que se verá más adelante), y se desacopla (mediante indirecciones, fachadas, fabricaciones púras, *proxies*, etcétera) de las capas de persistencia y comunicaciones (patrones todos ellos que también se explican más adelante).



3.5.2. Arquitectura de *pipes and filters* (tuberías y filtros)

Los diseños arquitectónicos de *pipes and filters* se pueden representar como grafos en los que los nodos se corresponden con filtros¹⁷ y los arcos se corresponden con tuberías (*pipes*)¹⁸.

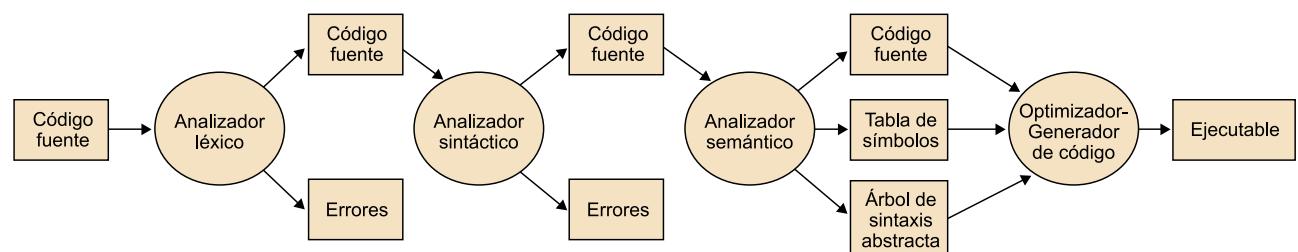
(17) Elementos o componentes que toman entradas y producen salidas.

(18) Mecanismos de comunicación entre los nodos.

Cada filtro lee cadenas de datos de sus entradas, las transforma (las “filtrá”) y produce salidas, que deja en las tuberías para que sean leídas por otros filtros. Los filtros son independientes (no comparten su estado con otros filtros) y no conocen ni a sus filtros predecesores ni sucesores.

Un filtro es un elemento que simplemente recibe información, la transforma y la deja en algún lugar, desentendiéndose en ese momento.

La arquitectura de *pipes and filters* se utiliza, por ejemplo, en el desarrollo de compiladores:



Algunas características de estas arquitecturas son:

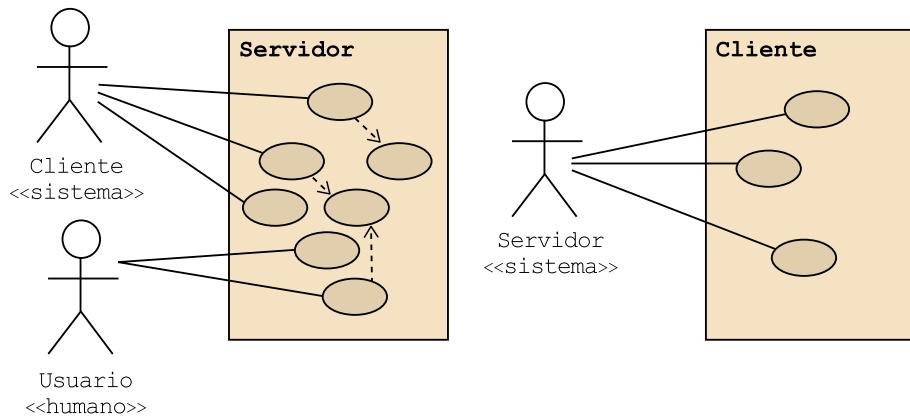
- Es fácil comprender el funcionamiento general del sistema.
- Es fácil reutilizar filtros.
- El mantenimiento es relativamente sencillo.
- También es relativamente sencillo realizar algunos tipos de análisis especializados, como la identificación de *deadlocks* y cuellos de botella.
- Es fácil hacer ejecuciones concurrentes, poniendo cada filtro a ejecutarse en un proceso separado.
- Sin embargo, son difícilmente aplicables a sistemas interactivos, sirviendo casi exclusivamente para sistemas en lot.

Nota

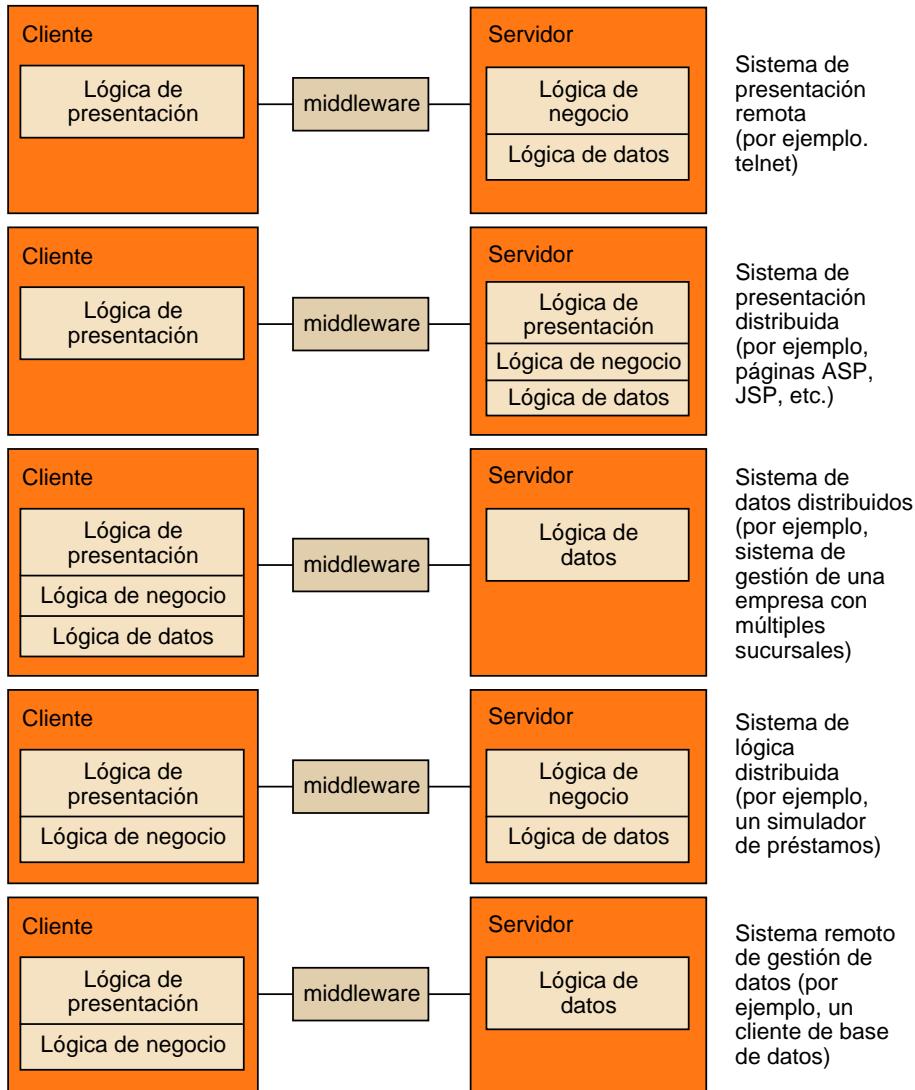
Los sistemas en lot también se conocen como *batch*.

3.5.3. Arquitecturas cliente-servidor

Los **sistemas cliente-servidor** consisten en aplicaciones cuya lógica se encuentra distribuida entre dos máquinas remotas: el cliente y el servidor, que se comunican a través de algún tipo de *middleware*. Al construir el sistema, es conveniente entenderlo como dos sistemas: el cliente se interpreta como un actor que actúa sobre el servidor, y el servidor como un actor que actúa sobre el cliente:

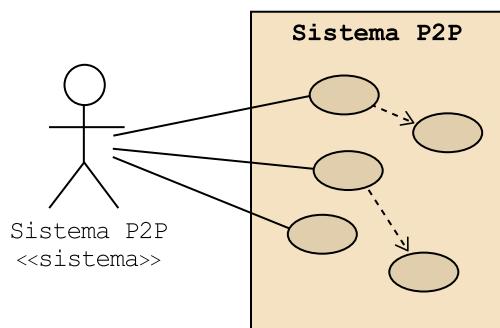


Con esta consideración, cada uno de los dos subsistemas puede tener un diseño arquitectónico diferente, si bien será necesario que los desarrolladores de ambos determinen las responsabilidades que residirán en el cliente y cuáles residirán en el servidor. En función de esta distribución, pueden darse diferentes situaciones, como por ejemplo las cinco que se muestran a continuación:

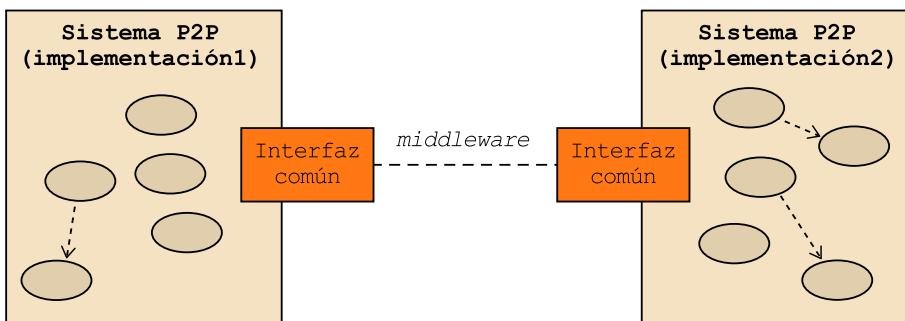


3.5.4. Arquitecturas P2P (*peer-to-peer*, de igual a igual)

En este caso, dos sistemas iguales se relacionan entre sí. A muy alto nivel, un sistema P2P puede entenderse como un sistema que se relaciona con otros sistemas que son como él: obsérvese, en la figura, que el nombre del sistema (Sistema P2P) coincide exactamente con el nombre del actor con el que se comunica:



En realidad, los dos sistemas pueden ser diferentes, pero deben comunicarse entre sí ofreciendo e implementando las mismas interfaces. Estas, junto con su implementación, pueden incluirse en un componente reutilizable que se integre en diferentes tipos de clientes P2P. Con una notación algo libre, aunque tipo UML, podemos representar esto de la siguiente forma:



Para implementar las interfaces y permitir la comunicación entre los pares se puede reutilizar Gnutella, un protocolo para compartir archivos entre pares que es P2P “puro”: no hay servidor central y todos los nodos tienen exactamente las mismas funciones dentro de la red. Gnutella define la sintaxis de los mensajes para que un nodo se enlace a una red ya existente, busque archivos dentro de la red y descargue archivos.

3.6. Patrones de diseño

Los patrones de diseño son soluciones buenas a problemas de diseño de software que se presentan frecuentemente. De hecho, para que una solución se pueda considerar un patrón, debe haber sido probada con éxito en más de una ocasión. La solución, además, se debe describir de forma suficientemente general como para que sea reutilizable en contextos diferentes.

Como se verá, muchos patrones requieren la aplicación de niveles sucesivos de indirección (es decir, en lugar de enviar el mensaje directamente al objeto interesado, se le envía a un intermediario) que, en muchos casos, pueden aparentar ser demasiado artificiales. Por lo general, todos los patrones persiguen el diseño de sistemas con alta cohesión y bajo acoplamiento, lo que produce elementos software más reutilizables. A veces, sin embargo, la utilización exagerada de patrones puede dar lugar a diseños muy complejos, difíciles de seguir y entender, lo que puede dificultar su mantenimiento.

El conjunto de patrones más conocido es el de Gamma y colaboradores, que en 1995 publicaron un célebre libro titulado *Design Patterns: Elements of Reusable Object-Oriented Software*, del que se han imprimido multitud de ediciones y ha sido traducido a numerosos idiomas. Otros autores, como Craig Larman, han publicado también libros muy exitosos de ingeniería del software en los

Nota

En la asignatura *Análisis y diseño con patrones* del grado de Ingeniería Informática ya se describen muchos de estos patrones. En este texto intentamos presentarlos con un punto de vista más orientado a la reutilización.

que incluyen sus propios catálogos de patrones: en el caso de Larman, a estos patrones se los conoce como patrones GRASP (*general responsibility assignment software patterns*, patrones generales de asignación de responsabilidades).

3.6.1. Patrones de Gamma

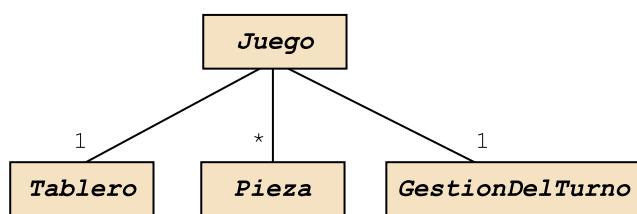
En su libro Gamma, y otros presentan 23 patrones agrupados en tres categorías:

- **Patrones de creación**, que pretenden resolver el problema de cómo crear objetos complejos.
- **Patrones estructurales**, que describen soluciones acerca de cómo unir objetos para que formen estructuras complejas.
- **Patrones de comportamiento**, que presentan algoritmos y mecanismos para comunicar objetos.

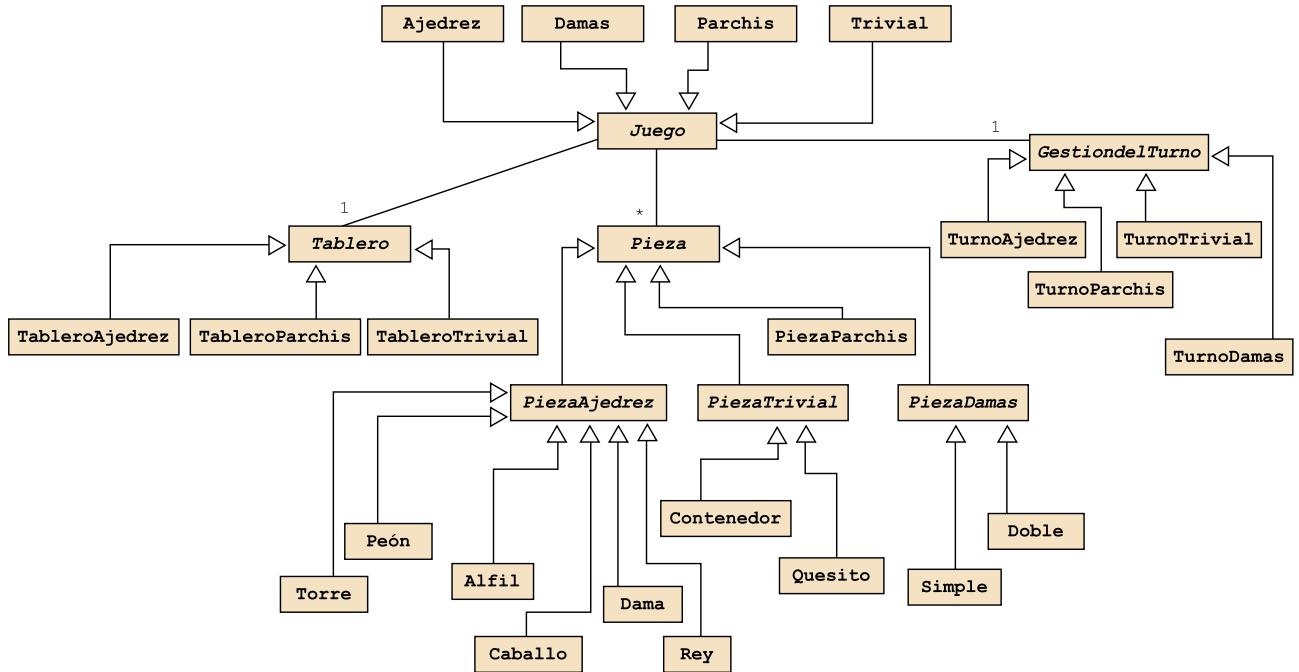
A continuación se describen algunos de ellos.

Patrón de creación *abstract factory* (fábrica abstracta)

Supongamos que debemos crear un sistema de software para jugar a diferentes tipos de juegos de mesa: damas, ajedrez, parchís y trivial. Además de sus propias reglas y políticas de gestión del turno, cada juego tiene un tipo diferente de tablero y unas piezas distintas. Aprovechando esta estructura común de los juegos, podemos crear una estructura de clases abstractas para representar los tres juegos en general:



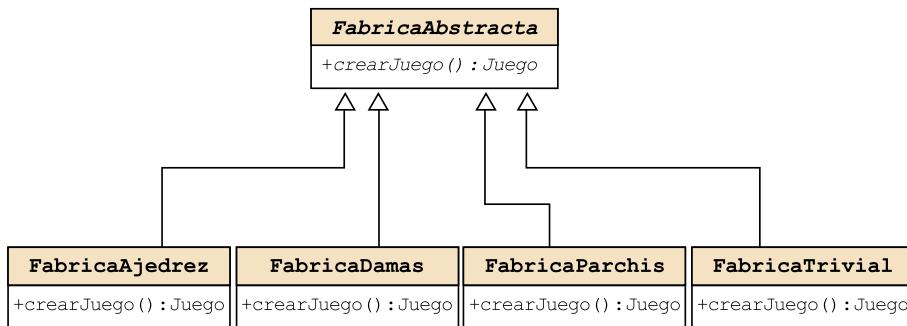
El número de especializaciones de estas clases abstractas dependerá del número de juegos y de las características específicas de cada juego:



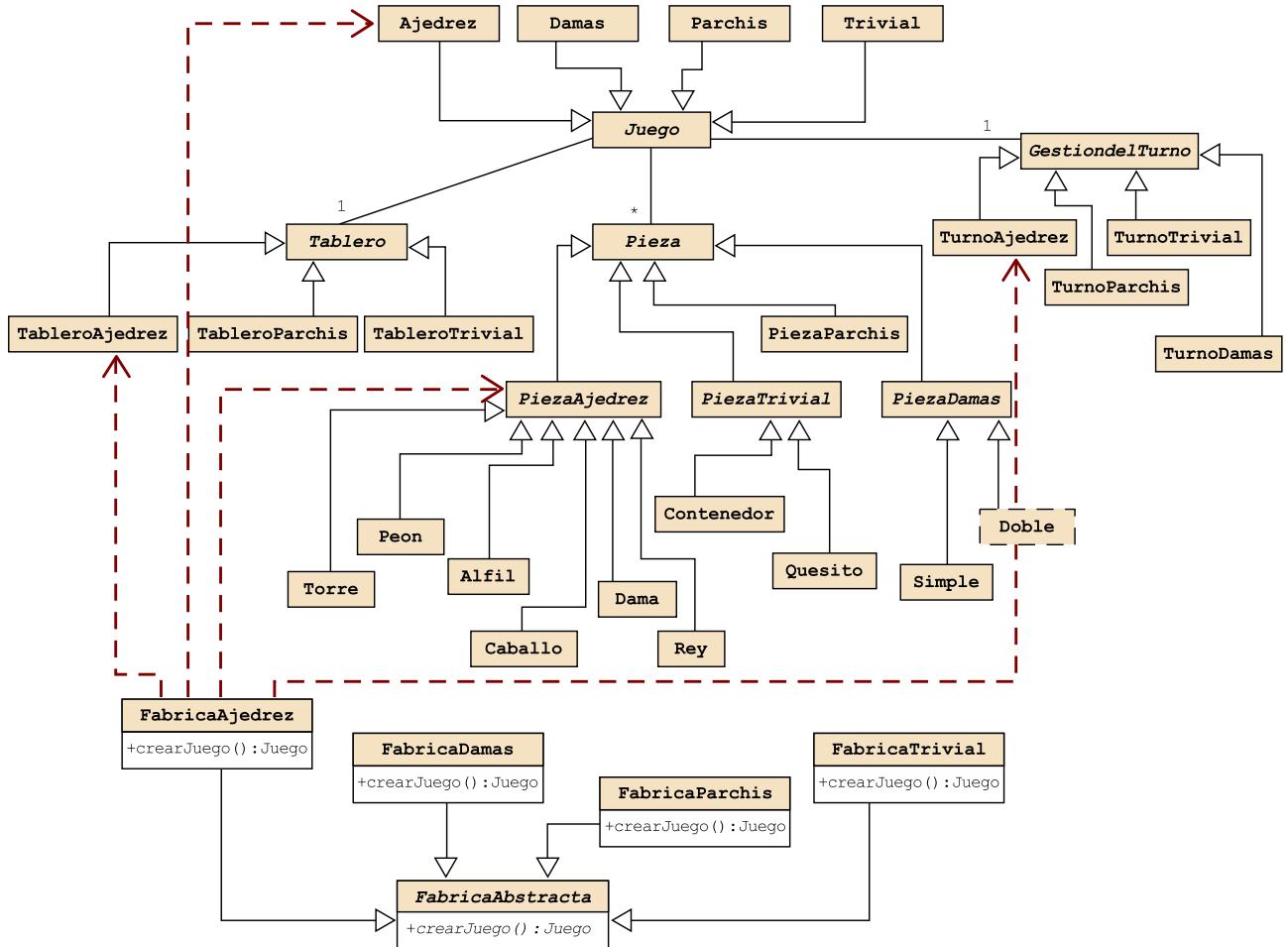
A la hora de crear un objeto de, por ejemplo, clase *Ajedrez*, será preciso instanciar un tablero de ajedrez, varias piezas (peones, alfiles, etcétera) de colores diversos y su política concreta de gestión del paso del turno entre los jugadores. El patrón *abstract factory* nos ayuda a resolver la siguiente pregunta:

¿A quién asignar la responsabilidad de crear una familia de objetos complejos?

La respuesta es asignársela a una clase externa que fabrique los objetos. Habrá una fábrica abstracta y tantas especializaciones concretas como familias de objetos haya:



Cada fábrica concreta es responsable de crear todos los objetos correspondientes a la familia de objetos a la que está asociada. La *FabricaAjedrez*, por ejemplo, brega solamente con los subtipos de *Tablero*, *Pieza* y *GestiónDelTurno* que le corresponden:



La implementación del método *crearJuego()* en, por ejemplo, *FabricaAjedrez* será de este estilo:

```
public Juego crearJuego() {
    Juego resultado=new Ajedrez();
    resultado.setTablero(new TableroAjedrez());
    resultado.setPiezas(buildPiezas());
    resultado.setTurno(new TurnoAjedrez());
    return resultado;
}
```

Patrón de creación **builder** (constructor)

El *builder* proporciona una solución diferente al mismo problema planteado en el *abstract factory*. En esencia, la diferencia radica en que ahora uno de los objetos de la familia es el responsable de crear el resto de objetos. Si en el ejemplo asignamos a la clase *juego* las responsabilidades de construir los objetos, el código de la clase puede tomar esta forma:

```

public abstract class Juego {
    private Tablero tablero;
    private Vector<Pieza> piezas;
    private GestionDelTurno turno;

    public Juego() {
        this.tablero=crearTablero();
        this.piezas=crearPiezas();
        this.turno=crearTurno();
    }

    protected abstract Tablero crearTablero();
    protected abstract Vector<Pieza> crearPiezas();
    protected abstract GestionDelTurno crearTurno();
}

```

El código anterior muestra también un ejemplo del patrón de comportamiento *template method* (método-plantilla): obsérvese que, desde una operación concreta (el constructor *Juego()*) de una clase abstracta (*Juego*), se está llamando a tres operaciones declaradas como abstractas en la propia clase (*crearTablero()*, *crearPiezas()* y *crearTurno()*). El patrón método-plantilla describe el comportamiento genérico de una jerarquía de objetos, pero dejando a las subclases la responsabilidad concreta de ejecutar las operaciones: en el constructor, se dice que lo primero que hay que hacer es crear el tablero, luego las piezas y luego el sistema de gestión del turno; sin embargo, deja a las clases concretas (ajedrez, damas, parchís y trivial) la responsabilidad concreta de cómo hacerlo.

Patrón de creación *singleton* (creación de una única instancia)

En ocasiones es necesario garantizar que, para alguna clase, puede crearse un máximo de una instancia: puede interesar, por ejemplo, dotar al sistema de un único punto de acceso a una base de datos externa, de manera que este punto de acceso gestione las múltiples conexiones que puedan establecerse.

Para garantizar que se crea una sola instancia de este punto de acceso (que, por otra parte, puede entenderse como un patrón *proxy* o un patrón *broker*, – agente– de base de datos, de Larman), la clase que lo implementa puede ser un singleton. Por lo general, un singleton tiene:

- Un campo estático del mismo tipo que la clase singleton que, por ejemplo, se llama *yo*.
- Un constructor de visibilidad reducida (privado o protegido).
- Un método público estático para recuperar la única instancia existente de la clase que, por ejemplo, puede llamarse *getInstancia*. Al llamar a este método, se pregunta si existe *yo* (es decir, si es distinto de *null*): si *yo* sí que es distinto de *null*, el método devuelve la instancia; si no, se llama al constructor (que sí que es visible para *getInstancia*), con lo que se construye *yo* y después se devuelve la instancia recién creada.

- El conjunto de métodos de negocio que corresponda, que no tienen por qué ser estáticos.

El siguiente código muestra un ejemplo de un agente de base de datos singleton que, a pesar de ser único, es capaz de gestionar hasta 1.000 conexiones simultáneas a la base de datos: cuando un cliente necesita ejecutar una operación sobre la base de datos, en lugar de crear él mismo su propia conexión, recupera el agente (llamando a *getInstancia*); una vez que tiene la referencia al agente, el cliente puede pedirle a este que le devuelva la conexión.

```

package persistencia;

import ...;

public class Broker {
    protected static Broker yo;
    private Vector<Conexion> libres, ocupadas;
    private static int CONEXIONES = 1000;

    protected Broker () throws ClassNotFoundException, SQLException {
        this.libres=new Vector<Conexion>();
        this.ocupadas=new Vector<Conexion>();
        for (int i=0; i<CONEXIONES; i++) {
            this.libres.add(new Conexion(i+1));
        }
    }

    public static Broker getInstance() throws
        NoHayConexionesLibresException, SQLException {
        if (yo==null) {
            try {
                synchronized(Broker.class) {
                    yo=new Broker ();
                }
            } catch (ClassNotFoundException e) {
                throw new NoHayConexionesLibresException();
            }
        }
        return yo;
    }

    public Conexion dameConexion() throws
        NoHayConexionesLibresException {
        ...
    }
    ...
}

```

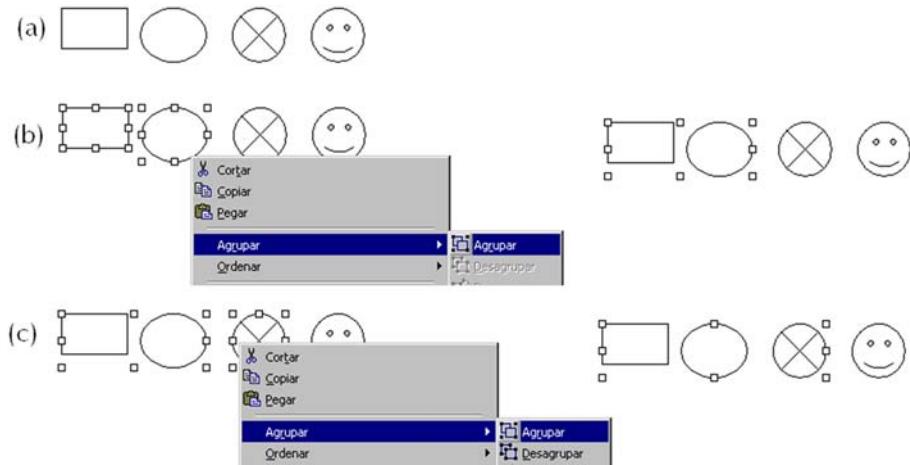
Patrón estructural *adapter* (adaptador)

Este patrón se aplica cuando una clase cliente desea utilizar otra clase servidora que, sin embargo, le ofrece una interfaz diferente de la que espera el cliente. Lo veremos con cierto detalle en la sección de componentes.

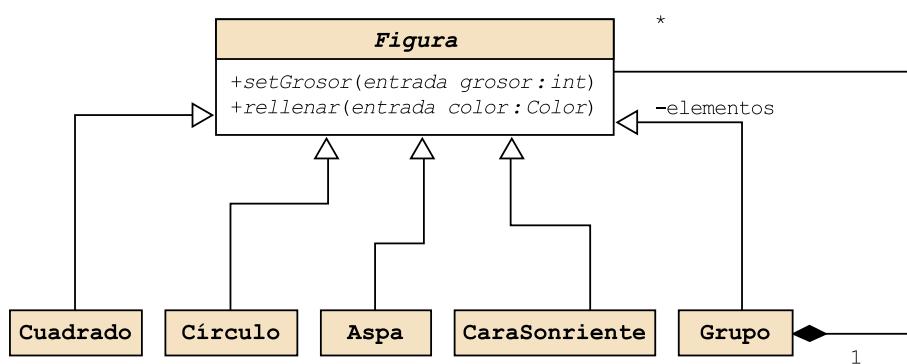
Patrón estructural *composite* (compuesto)

En ocasiones, un objeto está formado por objetos que, a su vez, poseen en su interior objetos que son de la clase del objeto original. En el procesador de textos Microsoft Word, por ejemplo, es posible agrupar varios objetos de

dibujo en un “grupo”; este grupo se maneja como un único objeto, y puede ser también agrupado junto a otros objetos, de forma que un grupo puede contener, mezclados, objetos simples y grupos. La siguiente figura muestra cuatro objetos sencillos dibujados con Word (a); en el lado izquierdo de (b) creamos un grupo a partir de dos objetos, que ya se muestran agrupados en el lado derecho; en el lado izquierdo de (c), creamos un nuevo grupo formado por el grupo creado anteriormente y un nuevo objeto simple, cuyo resultado se muestra a la derecha.



Todos estos objetos de dibujo pueden responder, por ejemplo, a una operación `setGrosor(pixels:int)`, que asigna el *grosor* que se pasa como parámetro a las líneas que conforman el dibujo, o a `rellenar(color:Color)` para llenar la figura con el *color* que se establezca. Para manejar de manera uniforme todo el grupo de figuras se puede utilizar el patrón Composite, en el que las figuras simples y compuestas se representan de esta manera:



Se está diciendo que las figuras pueden ser figuras simples (cuadrados, círculos, aspas o caras sonrientes) o figuras compuestas (grupos); estas, a su vez, están formados por figuras que también pueden ser simples o compuestas. Como se observa, todas las subclases son concretas, lo que significa que implementan las dos operaciones abstractas que se muestran en la superclase. Las figuras “simples” darán a las operaciones la implementación que corresponda; la im-

plementación de estas operaciones en grupo, sin embargo, consistirá en llamar a la misma operación en todas las figuras a las que conoce, que son accesibles a través de la relación de agregación:

```
public void setGrosor(int grosor) {  
    for (Figura f : elementos)  
        f.setGrosor(grosor);  
}
```

En tiempo de ejecución se decidirá, en función del tipo de los objetos contenidos en el grupo, qué versión de la operación debe ejecutarse.

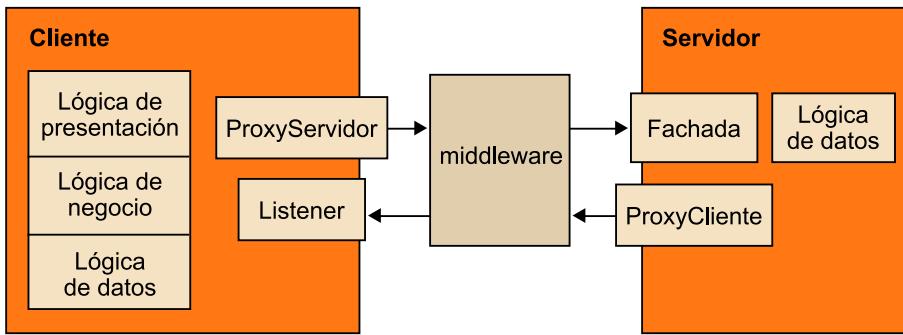
Patrón estructural *façade* (fachada)

Una fachada representa un punto de acceso único a un subsistema. Si en una aplicación multicapa disponemos de varias ventanas en la capa de presentación y de varias clases en la capa de negocio, puede ser conveniente crear una fachada que represente el punto de acceso único desde la capa de presentación a la de dominio. Cuando un objeto de presentación le quiere decir algo a uno de dominio, no se lo dirá directamente, sino que le hará la petición a la fachada, que la encaminará al objeto de dominio que corresponda.

Patrón estructural *proxy*

Un *proxy* representa un sustituto de “algo” que, por lo general, será un sistema o dispositivo externo. En lugar de permitir que cualquier parte del sistema acceda directamente al sistema externo, colocamos un *proxy* intermedio que será su único punto de acceso. Podemos considerar que una fachada es un *proxy* de acceso a un subsistema, y es cierto; cuando utilizamos una clase intermedia para acceder a un sistema o dispositivo externo (un servidor o un escáner, por ejemplo), lo llamaremos *proxy*. Los *proxies*, en muchas ocasiones, son además clases singleton.

La siguiente figura completa uno de los diseños arquitectónicos que mostrábamos al hablar de los estilos arquitectónicos cliente-servidor con dos *proxies*: uno en el sistema cliente para acceder al servidor; otro en el servidor para comunicar con los clientes. El servidor, además, ofrece sus servicios a través de una fachada (que, por otro lado, ha de implementar algún tipo que permita la comunicación remota); el cliente escucha al servidor a través de un *listener*, que puede entenderse como una fachada más simplificada.



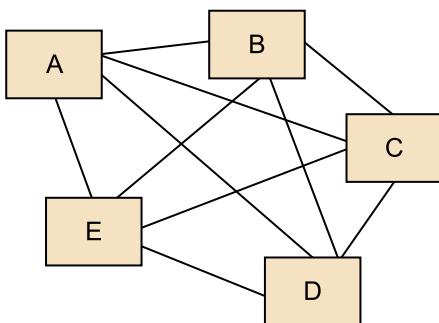
Patrón de comportamiento *chain of responsibility* (cadena de responsabilidad)

Este patrón se utiliza para desacoplar al objeto emisor de una petición del objeto que debe recibirla. La idea pasa por tener una cadena de objetos que pasen la petición hasta que aparezca un objeto que sea capaz de atenderla.

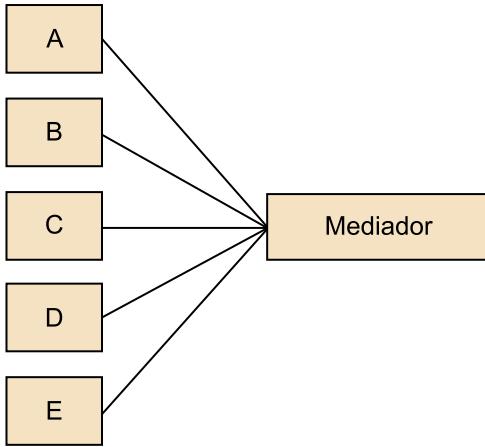
La solución se modela como una superclase abstracta con una operación abstracta que se corresponde con la petición; esta clase, además, conoce a un objeto de su mismo tipo, al que se llama sucesor. Todas las subclases concretas implementan la operación: las que pueden atender la petición, la atienden; las que no, la derivan al sucesor, que están heredando de la superclase.

Patrón de comportamiento *mediator* (mediador)

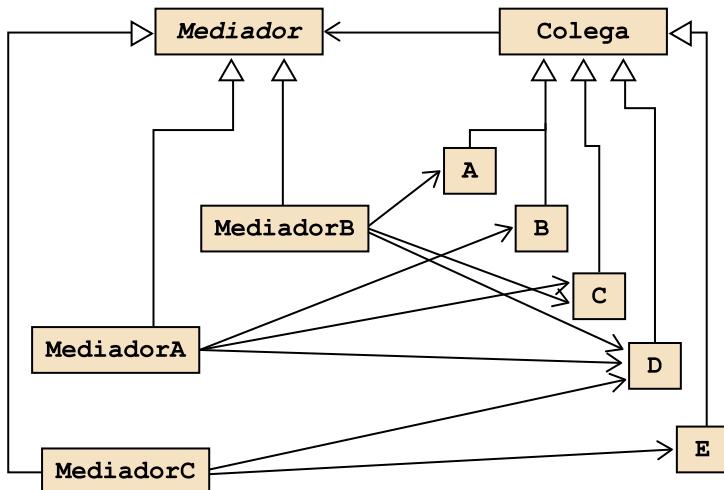
A veces, un cambio en un objeto cualquiera de una familia de objetos debe ser notificado a todo el resto de objetos. Supongamos que tenemos cinco clases cuyas instancias deben estar al tanto de los cambios de estado que se producen en los objetos de las otras cuatro. Una solución posible es estructurar el sistema de tal manera que todas las clases se conozcan entre ellas, como se muestra en esta figura:



Este diseño, sin embargo, es difícil de comprender, de mantener y de probar, y tiene un altísimo acoplamiento que, probablemente, lo haga muy propenso a fallos. El patrón mediador aconseja la creación de una clase que reciba las notificaciones de los cambios de estado y que las comunique a las interesadas:



El patrón admite variantes que aprovechan la herencia: si, por ejemplo, los cambios de estado de A solo interesan a B, C y D, los de B solo a A, C y D, los de C solo a D y E, y los demás no interesan, se puede replantear el diseño de esta manera:



Patrón de comportamiento *observer* (observador)

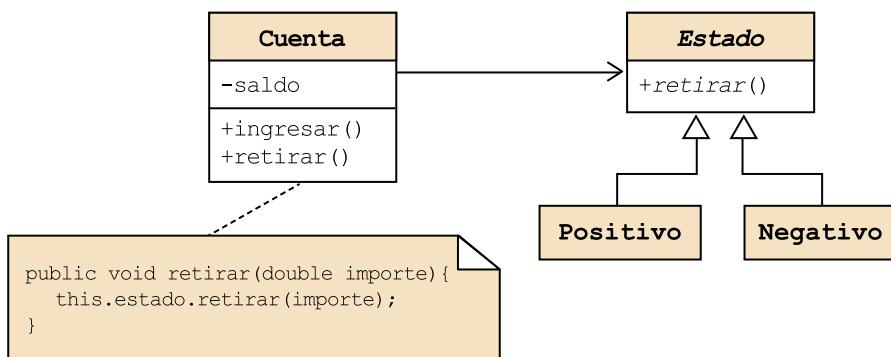
Si en el patrón anterior se resolvía el problema de cómo notificar cambios en múltiples objetos a otros múltiples objetos, el patrón observador da una solución a cómo notificar los cambios de estado de un único objeto a un conjunto de objetos que lo están observando.

La solución consiste en la creación de una clase intermedia (el observador), a la que suscriben los objetos interesados en observar al objeto observable. Cuando este experimenta un cambio de estado, lo comunica al observador, que lo notifica a los objetos que mantiene suscritos.

El patrón alta cohesión (uno de los patrones de Larman), que se presenta más adelante, incluye un ejemplo del patrón observador.

Patrón de comportamiento state (estado)

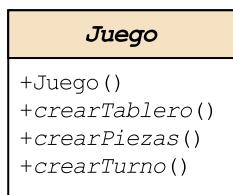
Este patrón delega en una clase asociada aquellas operaciones cuyo comportamiento depende de su estado. Supongamos que la operación *retirar()* de una cuenta corriente depende de si su estado es positivo o negativo: la operación puede implementarse en la propia clase cuenta y, antes de efectuar la retirada, preguntar si el saldo es suficiente para afrontar el importe que desea retirarse. Mediante el patrón estado, se crea una clase abstracta Estado con dos especializaciones: positivo y negativo. Cuando a la clase cuenta le llega una llamada a *retirar()*, esta invoca a la operación retirar de su estado asociado, que estará instanciado a una de las dos subclases.



Patrón de comportamiento template-method (método plantilla)

Este patrón se utiliza para representar de manera general el comportamiento de una determinada operación: en una clase abstracta se crea una operación concreta que llama a operaciones abstractas y concretas.

Mostrábamos un ejemplo de código al presentar el patrón builder, que corresponde a la siguiente clase en UML:



3.6.2. Patrones de Larman

Larman presenta nueve patrones que ayudan al ingeniero de software a determinar a qué objeto se le debe asignar cada responsabilidad. El autor explica que las responsabilidades de los objetos son de dos tipos: de conocer¹⁹ y de hacer²⁰.

⁽¹⁹⁾Sus propios datos, los de objetos relacionados, y otras cosas que puede derivar o calcular.

⁽²⁰⁾Puede hacer algo el propio objeto, puede iniciar una acción en otro objeto o puede coordinar y controlar actividades entre objetos.

Patrón experto

Este patrón nos dice que la responsabilidad debe asignarse a la clase “experta en la información”: es decir, a aquella clase que tiene la información necesaria para realizar la responsabilidad.

Patrón creador

Este patrón dice que se debe asignar a la clase A la responsabilidad de crear instancias de B cuando:

- A agrega o contiene objetos de B
- A registra objetos de B
- A utiliza “más estrechamente” objetos de B
- A tiene la información de inicialización necesaria para crear instancias de B

Patrón bajo acoplamiento

El acoplamiento indica el grado en que una clase está relacionada con otras: cuanto mayor sea el acoplamiento, mayor dependencia de la clase respecto de cambios en las otras. Además, en diversos estudios experimentales se ha comprobado que el alto acoplamiento es el mejor indicador de la propensión a fallos de una clase.

Más que un patrón, el mantenimiento del bajo acoplamiento es un principio general de diseño de software, incluyendo diseño de software orientado a objetos.

La clase A está acoplada a B cuando:

- A tiene un atributo de tipo B.
- A invoca a algún método de B.
- En algún método de A se declara una variable de tipo B.
- En algún método de A se utiliza algún parámetro de tipo B.
- A es una subclase, directa o indirecta, de B.
- B es una interfaz, y A una implementación de B.

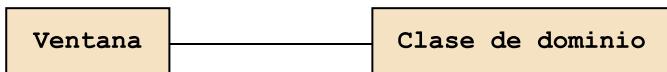
Para mantener el bajo acoplamiento, se deben asignar las responsabilidades de manera que se eviten, en la medida de lo posible, relaciones como las enumeradas arriba.

Patrón alta cohesión

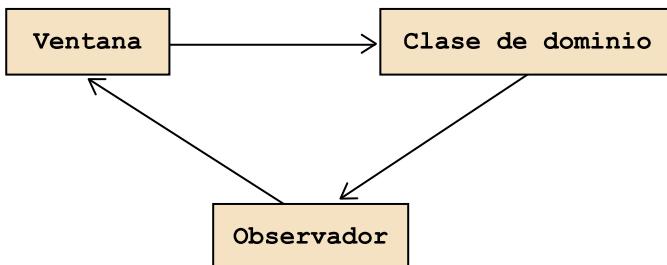
La cohesión da una medida del grado en que están relacionados los elementos de una clase. Una clase con baja cohesión hace varias cosas diferentes que no tienen relación entre sí, lo que lleva a clases más difíciles de entender, de reutilizar y de mantener.

Al igual que el bajo acoplamiento, el mantenimiento de la alta cohesión es también un principio general en el diseño de software.

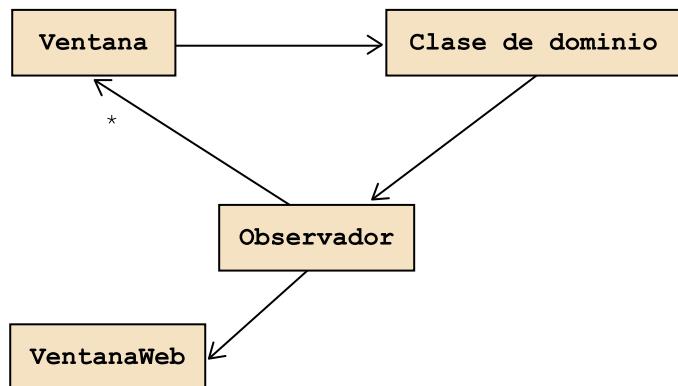
Muchas veces, conseguir una cohesión alta supone subir también el acoplamiento, y disminuir este suele suponer una bajada de aquella. En la siguiente figura se muestra un pequeño diagrama con dos clases: una **ventana**, ubicada en la capa de presentación y una **clase de dominio**, que encierra parte de la lógica realmente compleja del sistema. Los cambios de estado en el objeto de dominio se deben notificar a la ventana, con objeto de que el usuario esté al tanto de la evolución del sistema. En la solución que se está representando, la ventana conoce al dominio, y el dominio conoce al objeto. Desde un punto de vista de mantenimiento de la alta cohesión, el objeto de dominio no debería ser responsable de notificar a la ventana sus cambios de estado.



Podemos introducir un observador intermedio que se responsabilice únicamente de notificar a la ventana los cambios de estado en el objeto de dominio:



De esta manera, el objeto de dominio queda más cohesionado, pues delega a otra clase (el observador) la responsabilidad de notificar sus cambios a quien esté interesado. El acoplamiento general del sistema, sin embargo, aumenta, pues pasamos a tener tres relaciones en lugar de dos (de ventana a dominio y de dominio a ventana). No obstante, este acoplamiento nuevo es “menos malo” que el anterior, pues la clase de dominio deja de estar acoplada con una ventana: si en un futuro hubiese más tipos de ventanas interesadas en el objeto de dominio, o si fuese necesario notificar a más instancias de la ventana actual, no será preciso reconstruir el objeto de dominio, ya que esta responsabilidad se le ha asignado al observador:



Patrón controlador

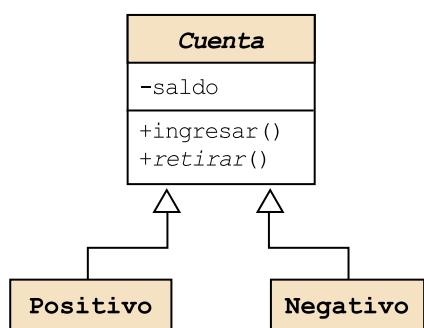
Este patrón recomienda asignar los eventos del sistema (los provocados por actores: usuarios u otros sistemas externos) a una clase controlador que:

- Represente el sistema global (lo que se corresponde con un “controlador de fachada”).
- Represente un escenario del caso de uso en el que se recibe el evento (“controlador de caso de uso”).

Patrón polimorfismo

El uso de múltiples instrucciones condicionales en un método dificulta la reutilización de la clase en la que está contenido. Por ello, cuando el comportamiento de una operación varíe según el tipo de la clase, debe asignarse la responsabilidad en función de los subtipos para los que varíe el comportamiento.

Cuando presentábamos el patrón estado, delegábamos a una clase asociada la ejecución de la operación *retirar*; con el patrón polimorfismo, dejamos la operación *ingresar* (cuyo comportamiento siempre es igual) como concreta en la superclase *Cuenta*, que es abstracta porque *retirar* sí que se comporta de manera diferente en función del subtipo:



Desde luego, este patrón nos prohíbe la utilización de operaciones de comprobación del tipo como la *instance of* que permite Java.

Patrón fabricación pura

Supongamos que la *Cuenta* que hemos utilizado en el ejemplo anterior es una clase persistente (es decir, que sus instancias deben almacenarse como registros en una base de datos). Las operaciones de gestión de su persistencia²¹ pueden ubicarse en la propia clase e implementarlas ahí, incluyendo la construcción de instrucciones SQL para acceder a la base de datos.

⁽²¹⁾Llamadas habitualmente CRUD, siglas de *create, read, update* y *delete*; respectivamente, crear, que correspondería a una instrucción SQL tipo *insert into*; leer, que corresponde a un *select from*; actualizar, que es un *update*, y borrar, correspondiente a *delete from*.

Desde el punto de vista del patrón experto, esta es la solución que debería seguirse, pues qué mejor clase que la propia cuenta que conoce sus propios datos y tiene mejor capacidad para construir las instrucciones SQL que correspondan. Sin embargo, incluir SQL en una clase de dominio no es muy conveniente, pues la acopla directamente al gestor de base de datos que se esté utilizando. En situaciones como esta, es más conveniente crear una clase artificial, que no existe en el enunciado del problema que pretende resolverse, y delegarle a ella las operaciones de persistencia. Este tipo de clases artificiales son las fabricaciones puras: clases en las que se delegan una serie de operaciones que no corresponden realmente a otra. La cohesión aumenta, porque la clase de dominio no hace cosas que no le corresponden, y también la fabricación pura es una clase altamente cohesiva. De este modo, cuando una instancia de la clase cuenta quiere insertarse, se lo pide a su fabricación pura asociada.

Las clases *clienteDAO*, *cuentaDAO*, etcétera, que se mostraban en el diagrama con el que ilustrábamos la arquitectura multicapa, son fabricaciones puras que tienen delegadas las responsabilidades de persistencia de las clases de dominio.

Patrón indirección

Mediante este patrón, se asigna la responsabilidad a un objeto intermedio. Se utiliza para minimizar el acoplamiento directo entre dos objetos importantes y para aumentar el potencial de reutilización.

Obsérvese que una fabricación pura como la que comentábamos arriba, por la que se asignan a una clase artificial las responsabilidades de persistencia de otra clase de dominio, es esencialmente una indirección que desacopla el dominio de la base de datos, y que permite que la clase cuenta del ejemplo sea más reutilizable. Los patrones observador y mediador son también formas específicas del patrón indirección.

Patrón variaciones protegidas

Para evitar que las clases o subsistemas inestables tengan un impacto negativo en el resto del sistema, este patrón recomienda crear una interfaz estable a su alrededor.

3.6.3. El patrón modelo-vista-controlador

El patrón modelo-vista-controlador (MVC: *model-view-controller*) es un patrón de diseño muy utilizado para mantener la separación entre capas (en particular, para separar las capas de presentación y dominio).

El modelo se corresponde con el conjunto de objetos que representan el enunciado del problema, y que están normalmente ubicados en la capa de dominio; la vista se corresponde con la interfaz de usuario (la GUI: *graphical user interface*), que este utiliza para manipular y conocer el estado de los objetos de dominio. Entre medias se ubica un controlador. Si bien hay implementaciones diversas del patrón MVC, en la más habitual el controlador recibe eventos desde la GUI y los pasa al modelo; los cambios de estado en los objetos del modelo se pasan a la GUI también a través del controlador, que actúa entonces como una especie de fachada bidireccional.

3.7. Componentes

A diferencia de una clase, un componente es, según Sommerville (2007), una unidad de implementación ejecutable, construida mediante un conjunto de clases, que ofrece un conjunto de servicios a través de una interfaz. Por lo general, el usuario entiende el componente como una caja negra (de hecho, lo habitual es no disponer de su código fuente), de la que conoce tan solo los servicios ofrecidos a través de su interfaz, así como los resultados que la ejecución de estos servicios le permitirá obtener.

Con objeto de que el usuario pueda utilizar con garantías la funcionalidad ofrecida, Szyperski indica que la interfaz del componente debe estar especificada contractualmente. Para una determinada operación, el contrato debe especificar qué resultados podrán obtenerse a partir de las entradas suministradas a la operación, en términos de precondiciones y poscondiciones.

Szyperski también apunta que el componente debe ser “componible” (o, en otras palabras, susceptible de ser compuesto con otros componentes), lo cual indica que debe ser posible integrar un componente con otro para construir una aplicación basada en componentes. Para esto, puede ser necesario que el componente disponga también de una interfaz requerida: es decir, de un medio de comunicación para solicitar la ejecución de servicios a otros dispositivos, como en el ejemplo del *Termostato* que utilizábamos en el apartado 1 y que, como buenos reutilizadores, revisitamos nuevamente la figura siguiente con la idea de poder conectarlo a otros dos componentes:

- 1) El componente *Termostato* ofrece dos interfaces (*ITemperatura* e *IHumedad*), que utiliza para recoger datos de dos sensores externos. Este componente, además, requiere de un dispositivo que implemente la interfaz *IActuador*.

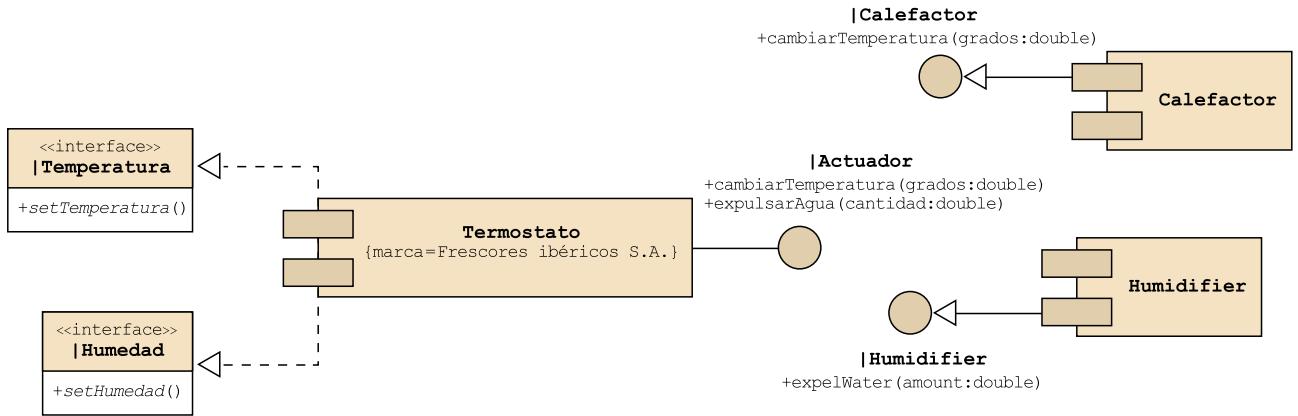
Consulta recomendada

I. Sommerville (1992). *Software Engineering* (8.^a ed.). Addison Wesley.

Consulta recomendada

C. Szyperski (2002). *Component Software: Beyond Object-Oriented Programming* (2.^a ed.). Boston: MA: Addison-Wesley.

2) *Calefactor* y *Humidifier* son dos componentes que ofrecen dos interfaces diferentes. La operación *cambiarTemperatura* ofrecida por el *Calefactor* es un subconjunto de la interfaz requerida por *IActuador*. *Humidifier*, sin embargo, que podría ser un componente importado del Reino Unido, ofrece una operación (*expelWater*) parecida a la operación *expulsarAgua* requerida por el *Termostato*.



Como se observa, no podemos conectar directamente el *Termostato* a los otros dos componentes: en primer lugar, porque las interfaces ofrecidas *ICalefactor* e *IHumidifier* son parecidas, pero no iguales, a la interfaz *IActuador*, que es la que requiere el *Termostato*; en segundo lugar porque, aunque ambas interfaces fuesen absolutamente compatibles, el *Termostato* conoce a un único *IActuador*, pero hay dos componentes a los que conectarlo, ya que uno actúa sobre la temperatura y otro sobre la humedad. ¿Cómo solucionarlo? A continuación exploraremos algunas formas de solucionar este problema.

3.7.1. Ingeniería del software basada en componentes

La ingeniería del software basada en componentes (CBSE, por sus siglas en inglés: *component-based software engineering*) se basa en que los sistemas software tienen una base común suficiente que justifica, en muchos casos, la utilización de componentes reutilizables. Frente a esto, el enfoque tradicional de desarrollo de sistemas involucra generalmente la construcción de cada una de las piezas desde cero, cada una de ellas implementada a propósito para el proyecto en curso. Algunas de estas piezas o elementos provienen de proyectos anteriores y son reutilizados con algunos ajustes adicionales.

Ved también

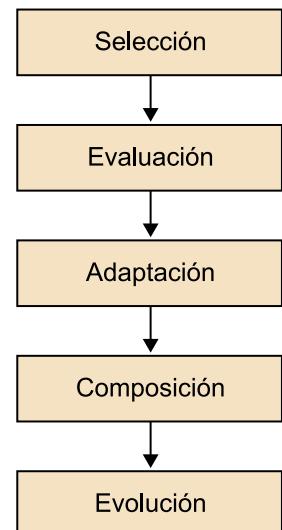
Ved la asignatura *Ingeniería de software de componentes y sistemas distribuidos*.

La tendencia hacia la reutilización y hacia la disminución de los esfuerzos de desarrollo ha conducido a la CBSE, que tiende a concentrar el esfuerzo en la composición de piezas ya desarrolladas (los componentes), que por lo general poseen una funcionalidad autocontenido. Algunos componentes, no obstante, pueden requerir una vinculación a otros componentes para proveer la funcionalidad completa. Tanto la interconexión de componentes como la integración de componentes con los elementos del sistema que se van desarrollando se realiza entendiendo los componentes como cajas negras, pues se uti-

lizan tal y como están disponibles y sin considerar ninguna modificación interna. De hecho, la CBSE trata del desarrollo de sistemas a partir de la mera composición e integración de componentes reutilizables.

Así, el proceso de desarrollo tradicional se modifica en CBSE para incorporar tareas de:

- **Selección y evaluación de componentes:** es decir, identificación de componentes candidatos a servir una funcionalidad o parte de ella que debe ser provista por el sistema.
- **Adaptación:** es posible que el componente no pueda ser directamente utilizable y sea preciso realizar algún tipo de adaptación en el sistema para poder usarlo.
- **Composición:** para enlazar varios componentes a través de sus interfaces.
- **Evolución:** se lleva a cabo durante el mantenimiento del sistema. En ocasiones requiere la sustitución de algún componente por una versión más moderna, o por un componente distinto que ofrezca la misma interfaz.



Selección

Una vez que se han determinado los requisitos del sistema y se ha establecido su diseño arquitectónico, el equipo de desarrollo identifica aquellas funcionalidades que, en lugar de ser implementadas de nuevo, puedan ser servidas por un componente preexistente. En particular, se preguntará:

- 1) ¿Disponemos en la empresa de componentes reutilizables que hayamos desarrollado nosotros mismos (*in-house components*), y que sirvan la funcionalidad requerida?
- 2) ¿Existen en el mercado componentes (*comercial off-the-self components*, CO-TS) que sirvan la funcionalidad requerida y que podamos utilizar?
- 3) En caso de que podamos utilizar un componente ya existente, ¿son sus interfaces compatibles con la arquitectura de nuestro sistema?

En función de las respuestas a las preguntas anteriores, es posible que se redifinen algunos requisitos (funcionales o no funcionales) para facilitar la integración de los posibles componentes reutilizables identificados. Para aquellos requisitos que, aparentemente, sí que puedan servirse con componentes, se siguen cuatro actividades importantes: evaluación de la adecuación del componente para el sistema; si es necesario, adaptación para integrarlo correctamente en el entorno operativo; integración del componente; y actualización del componente.

Evaluación

Un componente suele describirse en términos de un conjunto de interfaces que proveen el acceso a la funcionalidad del componente. A menudo, estas interfaces son la principal fuente de información acerca del componente, dado que la documentación adjunta ofrece, por lo general, tan solo una guía de las operaciones disponibles en su interfaz.

La integración de componentes en un sistema hace que el equipo de desarrollo juegue el doble rol de desarrolladores (porque integra y utiliza el componente dentro de la aplicación que está construyendo) y de usuarios (utiliza la funcionalidad suministrada por el componente), por lo cual es conveniente analizar doblemente cada componente candidato:

- Como usuarios, se evalúa la cualificación del componente para cumplir su cometido: es decir, se evalúa su capacidad para servir los requisitos.
- Como desarrolladores, se evalúan características como su interfaz, herramientas de desarrollo e integración necesarias, requisitos hardware (memoria, etc.), requisitos software (necesidad de otros componentes, sistema operativo, etc.), seguridad y forma en que maneja las excepciones.

Adaptación

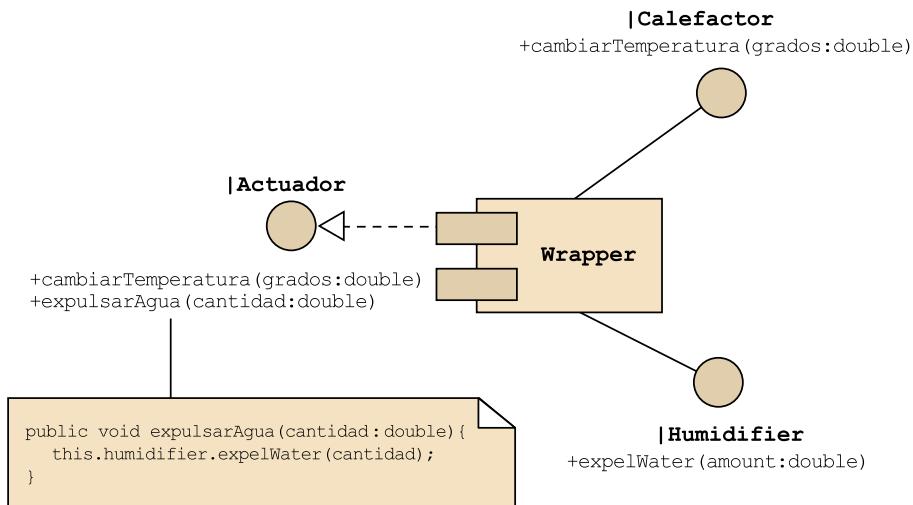
Una vez que se han identificado uno o más componentes para ser integrados en el sistema, el siguiente paso es reparar cualquier incompatibilidad que se haya detectado. En el ejemplo del termostato observábamos que este componente era incompatible con los dos componentes con los que se deseaba integrar. Para llevar a cabo las adaptaciones suelen utilizarse adaptadores (*wrappers* en inglés) que, por ejemplo, alteran la forma en que se llama a los servicios ofrecidos. Los tres tipos de incompatibilidades más comunes son:

- **Incompatibilidad de parámetros:** las operaciones tienen el mismo número de parámetros pero están en distinto orden o son de tipos diferentes. En este caso, el *wrapper* deberá cambiar el orden de los parámetros y/o hacer las conversiones de tipos que correspondan.
- **Incompatibilidad de operaciones:** los nombres de las operaciones en las interfaces esperada y ofrecida son distintos. En este caso, el *wrapper* recibirá una llamada a la operación esperada, y la reconducirá a la interfaz ofrecida.
- **Incompletitud de operaciones:** las operaciones ofrecidas en la interfaz son solo un subconjunto de las esperadas. Este caso no es siempre solucionable, pues el componente cliente espera más del componente servidor.

Nota

Algunos textos llaman “envoltorios” a los *wrappers*.

Para el ejemplo que venimos utilizando, un posible *wrapper* debería ofrecer la interfaz esperada por el *Termostato*, y requerir las dos ofrecidas por el *Calefactor* y el *Humidifier*. En este ejemplo, las dos operaciones de *IActuador* estarían implementadas en el *wrapper*, entre ellas *expulsarAgua(cantidad:double)*, que debería adaptar sus llamadas para que fuesen compatibles con la operación *expelWater(amount:double)*.



El diseño de *wrappers* conlleva la aplicación de algún patrón de diseño. Los tipos de *wrappers* más comunes y su relación con respecto a patrones son:

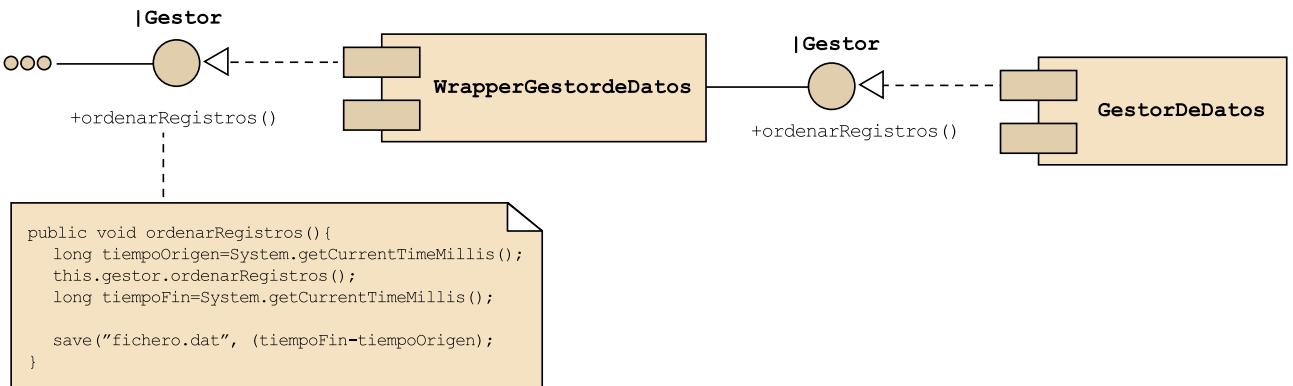
1) Wrappers de aislamiento, que proveen un único punto de acceso mediante una sola API a múltiples interfaces, y se relaciona con el patrón fachada.

Un ejemplo de esto es la API ODBC, que se utiliza en los sistemas Windows para proveer un punto único de acceso a diferentes tipos de bases de datos.

2) Envoltorios, que ocultan o adaptan algún aspecto del componente, como en el ejemplo de la figura anterior, y que se relaciona con el patrón adaptador (*adapter*).

3) Wrappers de instrumentación, que se utilizan para instrumentar, depurar, agregar aserciones, etcétera.

Puede interesar, por ejemplo, conocer cuánto tiempo demora un componente en realizar una determinada operación de las que ofrece a través de su interfaz: con un *wrapper* de instrumentación se captura la llamada a la operación y, antes de ejecutarla, inicia un cronómetro; cuando el resultado se devuelve, el *wrapper* lo captura y detiene el tiempo. Este tipo de *wrappers* se relacionan, entre otros, con los patrones decorador (*decorator*) y proxy.

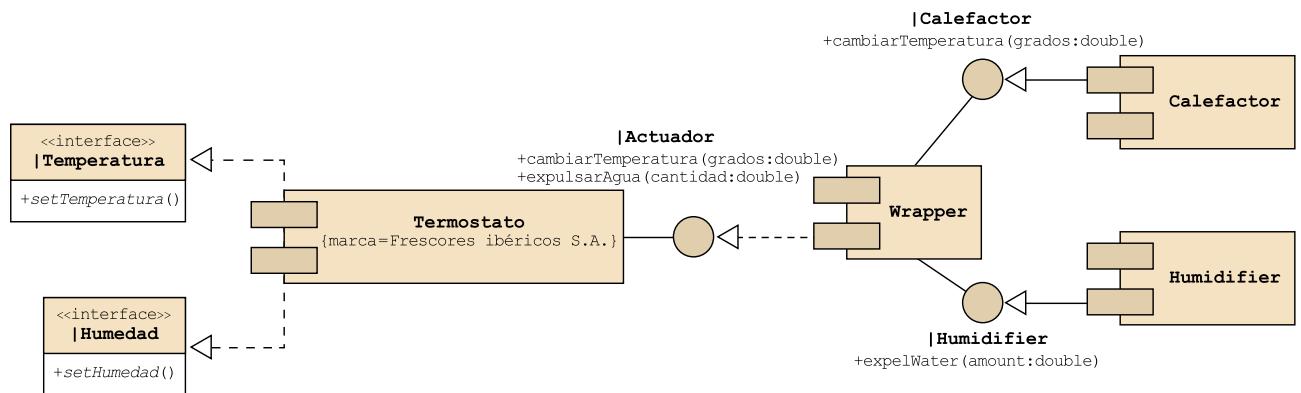


4) Wrappers de mediación, que coordinan diferentes interpretaciones de una misma propiedad del sistema.

Por ejemplo, puede usarse un *wrapper* de este tipo para gestionar la seguridad del acceso a varios componentes. Se relacionan con los patrones mediador (*mediator*) y *proxy*, entre otros.

Composición

Una vez que se han construido los *wrappers* necesarios para enchufar unos componentes con otros, los componentes se componen e integran para servir la funcionalidad. La figura siguiente ilustra cómo, finalmente, el termostato puede conectarse con los dos actuadores a través del *wrapper* que se diseñó e implementó en el punto anterior.



Existen varios enfoques para llevar a cabo la composición de componentes:

1) Composición secuencial: los componentes se van conectando uno a otro secuencialmente, creando para cada conexión los *wrappers* necesarios.

2) Composición jerárquica: los componentes se conectan uno con otro, de manera jerárquicamente ascendente o descendente.

3) Composición aditiva: las interfaces de dos componentes se unen para crear un “supercomponente” que ofrezca las funcionalidades de los dos, y que requiera también las funcionalidades de los dos.

Actualización

Como todo sistema de software, los sistemas basados en componente también llegan, en algún momento de su ciclo de vida, a la fase de mantenimiento, que puede ser correctivo (corrección de errores), perfectivo (adición de nuevas funcionalidades), preventivo (mejora de propiedades sin alterar la funcionalidad) o adaptativo (adaptación del sistema a cambios de entorno). La modificación del sistema puede suponer la necesidad de sustituir un componente por otro; pero además, los fabricantes producen también versiones nuevas de sus componentes en función de sus propias correcciones, mejoras o de necesidades competitivas del mercado. Ante nuevas entregas (*releases*, en inglés), el desarrollador debe decidir si se sustituye o no el nuevo componente por la nueva versión.

Ved también

Ved la asignatura Proyecto de desarrollo de software para más información sobre mantenimiento de software.

La sustitución de un componente debería ser tan sencilla como el reemplazo de una pieza en un puzzle por otra de forma exactamente igual. A veces, sin embargo, hay que analizar la posible necesidad de reescribir *wrappers*, así como realizar pruebas unitarias del componente candidato. El hecho de que el componente sea accesible únicamente a través de sus interfaces permite abaratar los costes de las pruebas: si se superan las pruebas de unidad, probablemente no sea necesario hacer pruebas de integración ni pruebas de sistema.

3.7.2. Componentes y clases

Es posible que una clase ofrezca, a través de una interfaz, una funcionalidad bien determinada y que pueda necesitar una clase tercera para ejecutar algún servicio. Visto así, se podría considerar que esta clase es un componente. Sin embargo, no todas las clases son componentes:

- Un componente es una entidad directamente instalable (*deployable*, en inglés).
- Un componente no define un tipo.
- La implementación de un componente es una caja negra.
- Un componente es una entidad estandarizada.

Si se siguen los principios de alta cohesión y bajo acoplamiento, las clases son módulos de tamaño más o menos pequeño y tendrán unas pocas dependencias con otras clases. Por el contrario, los componentes son más grandes, porque incluyen en su interior la implementación de varias clases; son autocontenidos e independientes; en general, utilizan como parámetros de sus servicios solamente tipos básicos (con objeto de que sean, precisamente, lo más independientes posible del contexto y del entorno); se pueden conectar (enchufar) a otros componentes para construir una aplicación; implementan interfaces bien definidas; son reemplazables.

3.7.3. Desarrollo de componentes

La probabilidad de reutilizar un componente es mayor cuanto más relacionado esté con un dominio de abstracción estable: esto es, cuanto más represente un objeto de negocio (*business object*). En un enfoque “oportunista”, se detecta la posibilidad de reutilizar un componente después de haberlo implementado para un propósito específico. Para hacerlo efectivamente reutilizable en otros contextos, el componente tiene que ser modificado y generalizado:

- Eliminando las llamadas a métodos específicos de la aplicación para la que fue construido.
- Cambiando los nombres de los servicios ofrecidos para que sean más generales.
- Añadiendo nuevos servicios para que cubra un espectro tanto si es más general como más particular.
- Haciendo que cada servicio lance un conjunto bien claro de excepciones (es decir, evitando el lanzamiento de una *exception* genérica, que impide al usuario del componente conocer con precisión la situación anómala que se ha producido).
- Añadiendo una interfaz de configuración que permita adaptar el componente a su contexto de ejecución.
- Integrando en el componente otros componentes, de manera que se reduzca su dependencia y pueda ser integrado de manera más fácil en entornos, contextos y aplicaciones diferentes.

3.8. Frameworks

Así como un componente nos brinda, por su propia construcción, una o más funcionalidades empaquetadas listas para ser utilizadas, un *framework* ofrece un conjunto de clases (algunas abstractas y otras concretas) e interfaces que ayuda a la resolución de problemas frecuentes y que, además, impondrá (o en todo caso guiará) al desarrollador cierto planteamiento del diseño arquitectónico del sistema. Larman lo explica diciendo que:

“Aun a riesgo de simplificar en exceso, un *framework* es un conjunto extensible de objetos para funciones relacionadas [...]. La señal de calidad de un *framework* es que proporciona una implementación para las funciones básicas e invariables, e incluye un mecanismo que permite al desarrollador conectar las funciones que varían, o extender las funciones”.

Ved también

Hablaremos con mayor profundidad del tema de los dominios al tratar las líneas de producto software, los DSL y la programación generativa en el apartado 4 de este material.

Consulta recomendada

C. Larman (2002). *UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Prentice-Hall.

Los *frameworks*, en efecto, están normalmente formados por un conjunto interrelacionado de clases fácilmente extensibles (de hecho, sus clases están pensadas para ser extendidas: esto es, especializadas). Algunas de las características de los *frameworks* que cita Larman son:

- Disponen de un conjunto cohesivo de interfaces y clases que colaboran para proporcionar los servicios de la parte central e invariable de un subsistema lógico.
- Contienen clases concretas y abstractas que definen las interfaces a las que ajustarse, interacciones de objetos en las que participar, y otras invariantes.
- Normalmente, requieren que el usuario del *framework* defina subclases de las clases que se incluyen en él para utilizar, adaptar y extender los servicios del *framework*. Estas subclases recibirán mensajes desde las clases predefinidas del *framework* que, normalmente, se manejan implementando métodos abstractos heredados de las clases abstractas del *framework*, y que el usuario habrá tenido que redefinir.

3.8.1. **Framework para la implementación de interfaces de usuario**

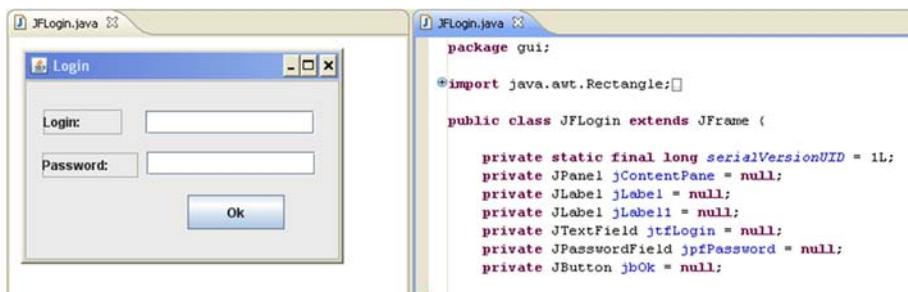
Swing es un *framework* de Java que incluye un conjunto muy amplio de componentes visuales (botones, cajas de texto, etcétera) para crear interfaces gráficas de usuario en aplicaciones Java, tales como JMenu, JButton, JTextarea, etc.

La figura siguiente muestra un momento del diseño de una ventana para identificarse con un nombre de usuario y una contraseña: mientras el usuario va diseñándola con el asistente visual de su entorno de desarrollo, este va añadiendo el código correspondiente al aspecto de la ventana, que luego el programador tendrá que completar. Obsérvese cómo, en el código fuente, la clase que se ha creado (JFLogin) es una especialización de la clase JFrame, que es una de las proporcionadas por Swing. Este es un ejemplo claro de la tercera característica de los *frameworks* que indicaba Larman y que hemos citado más arriba:

“Normalmente requieren que el usuario del framework defina subclases de las clases que se incluyen en él para utilizar, adaptar y extender los servicios del framework”.

Web recomendada

En <http://www.javaswing.org/> se encuentra una lista completa de los componentes visuales de este *framework*.



Swing permite, además, adecuarse al patrón modelo-vista-controlador (MVC), ya que cada componente visual está empaquetado en una clase y puede, además, asociarse a un tipo de modelo específico. En Swing se incluyen también modelos adecuados para cada tipo de componente. A las tablas (JTable), por ejemplo, se les puede asociar un objeto de tipo TableModel que contenga los datos mostrados en la tabla. Si otro objeto actualiza el TableModel asociado, el correspondiente objeto visual (la JTable) cambia automáticamente.

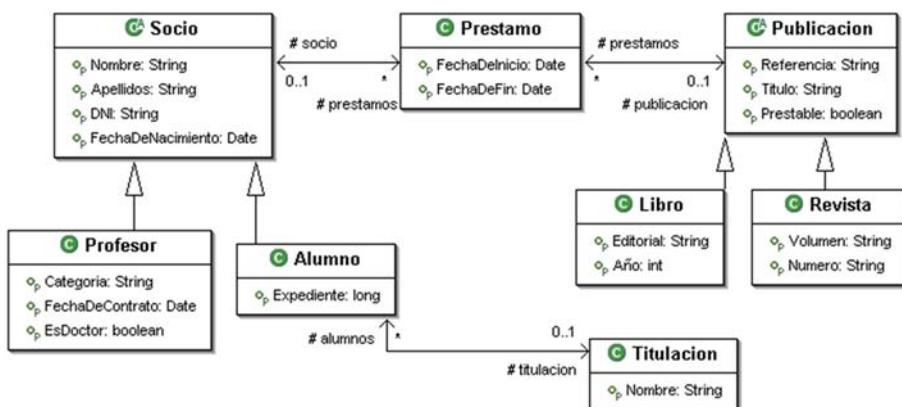
Véase también

Hemos visto el patrón MVC en el apartado “Patrones de diseño”.

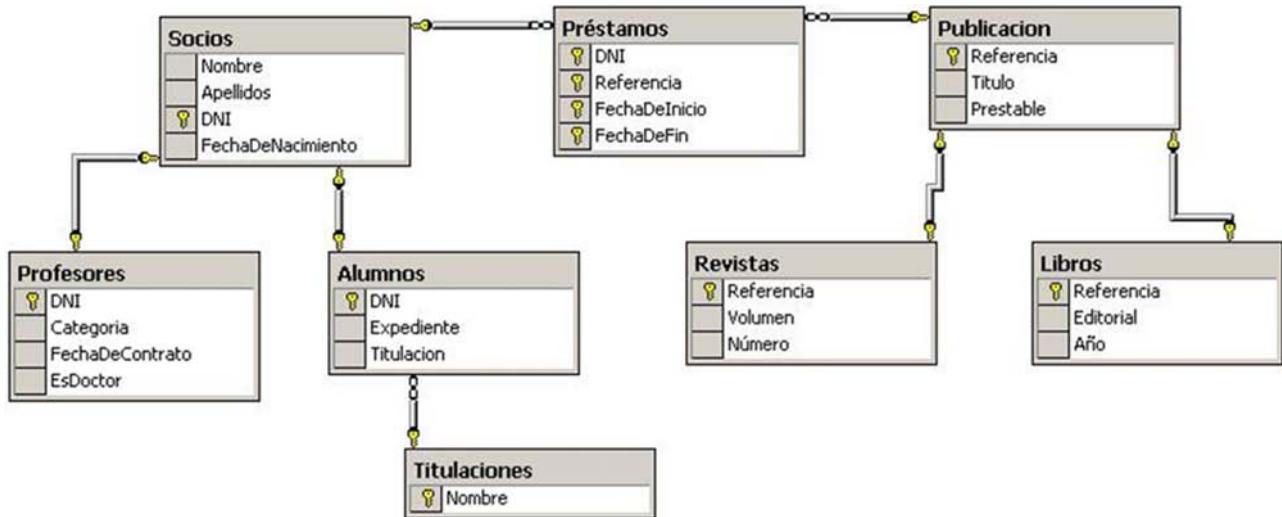
3.8.2. Frameworks de persistencia para “mapeo” objeto-relacional

La gestión de la persistencia de objetos es un problema muy recurrente en el desarrollo de aplicaciones. El “mundo de objetos” y el “mundo de tablas” son muy diferentes, por lo que se necesitan buenas políticas de mapeo entre uno y otro.

En primer lugar, es conveniente mantener diseños equivalentes entre la estructura de clases y la estructura de tablas, que puede lograrse mediante la aplicación de patrones de transformación, como el patrón “Una clase, una tabla”: de cada clase persistente se construye una tabla; asociaciones y agregaciones se transforman a relaciones de clave externa; la herencia se transforma a una relación entre dos tablas relacionadas por sus claves primarias.



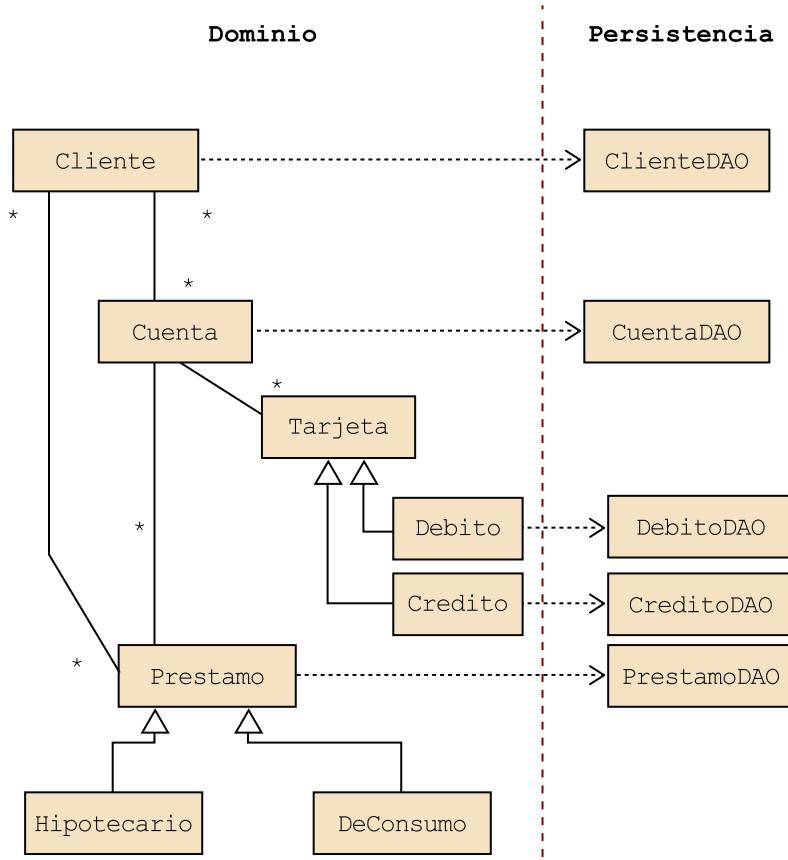
La aplicación del patrón “Una clase una tabla” al diagrama de clases anterior, que representa una biblioteca, puede dar lugar al siguiente diagrama relacional:



En segundo lugar, se deben asignar las responsabilidades de persistencia de forma que cumplan los principios básicos de diseño de software (especialmente el mantenimiento de la alta cohesión y el bajo acoplamiento): cuando se presentó el patrón “fabricación pura”, se explicó que se utiliza con frecuencia para delegar a otras clases la persistencia de clases de dominio: como se muestra en la figura siguiente, determinadas clases de dominio tienen asociada una fabricación pura que se encarga de gestionar su persistencia. La implementación de las operaciones de persistencia en las clases DAO (*data access objects*, objetos de acceso a datos) debe ser coherente con el diseño de la base de datos.

Véase también

Véase las clases DAO en la asignatura *Ingeniería de software de componentes y sistemas distribuidos* del grado de Ingeniería Informática.



Existen varios *frameworks* que ayudan a resolver el problema de la gestión de la persistencia (se conocen como *frameworks ORM: object-relational mapping*) que proponen soluciones parecidas a la mostrada en la figura anterior. Por lo general, estos *frameworks* proporcionan:

- Una API para gestionar las operaciones CRUD.
- Un lenguaje para especificar las consultas que se refieren a las clases y a sus propiedades.
- Una forma de definir y especificar los metadatos del mapeo.
- Una técnica para la implementación del mapeo que permita la interacción con objetos transaccionales soportando diversas funciones.

Algunos de estos *frameworks* son: Hibernate y NHibernate (para Java y .NET, respectivamente), Torque y Object-Relational Bridge.

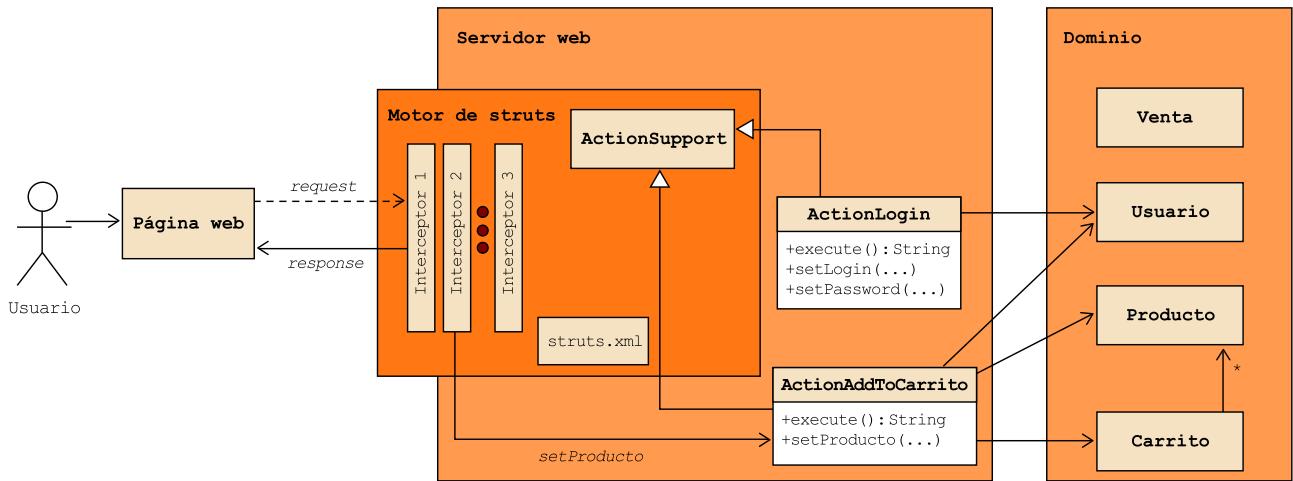
3.8.3. **Framework para el desarrollo de aplicaciones web**

En desarrollo web tradicional, toda la información que envían los usuarios desde sus navegadores se recoge en el servidor en forma de cadenas de caracteres (*strings*). Los programadores web recogen estos parámetros, los convierten al tipo de dato que corresponda²², comprueban su validez según las características del sistema y luego, llaman a la lógica de negocio.

(22)Puede recibirse en el servidor una cadena como "03-12-2011" que debe ser traducida a un objeto de tipo Date.

Struts2 es un *framework* de Java que incluye un conjunto muy amplio de clases para desarrollar aplicaciones web, que evita a los desarrolladores la escritura y reescritura de código muy parecido y que, además, impone una arquitectura muy sólida para el diseño de las aplicaciones.

Este *framework* requiere la extensión del servidor web con un motor auxiliar de ejecución que recoge las peticiones que llegan desde los clientes. Los parámetros se reciben, como siempre, en forma de cadenas de caracteres, pero se capturan por una serie de interceptores que pueden procesarlos de diferentes formas. La figura siguiente ilustra un fragmento del diseño de una tienda virtual: el usuario interactúa con una serie de páginas web que envían parámetros al servidor; estos son capturados por una pila de interceptores que pueden, por ejemplo, validar su formato.



Si los parámetros enviados no son correctos, se interrumpe la ejecución de la pila de interceptores y se devuelve el error (o la respuesta que corresponda) al usuario. Algunos de los interceptores pueden llamar a acciones concretas (especializaciones de la clase *ActionSupport* del framework Struts2, que es la clase fundamental): el Interceptor 2 de la figura, por ejemplo, ejecuta el método *setProducto* sobre la acción *ActionAddToCarrito* que, quizás, añada el producto pasado como parámetro al carrito de la compra.

Cada acción se corresponde, aproximadamente, con un requisito funcional o caso de uso del sistema, por lo que este *framework* resulta muy conveniente para migrar aplicaciones de escritorio hacia entornos web, ya que se separa completamente la lógica de negocio de la interfaz de usuario, favoreciendo la reutilización. Cuando se ha ejecutado el último interceptor, el motor de ejecución llama al método *execute():String* de la acción, que, normalmente, consta de un bloque *try* seguido de uno o más *catch*: el bloque *try* se corresponde con el escenario normal del caso de uso representado por la acción, mientras que cada *catch* se corresponde con un escenario alternativo.

La cadena devuelta por *execute* la interpreta el motor de ejecución a partir de los valores declarados en el fichero *struts.xml*. Aquí se encuentra una lista de todas las acciones junto a los posibles valores de tipo *string* que sus métodos *execute* pueden devolver: cada valor posible²³ tiene asociada una página web distinta, que se corresponde con la respuesta que se le devuelve al usuario.

Web recomendada

El *framework* y la documentación de Struts2 están disponibles en <http://struts.apache.org/>

⁽²³⁾Algunos predefinidos, como *success*, *error* o *input*, mientras que otros pueden estar definidos en la propia aplicación, como *NoExisteElProducto*.

Además de reutilizar la clase *ActionSupport*, los desarrolladores pueden crear nuevos interceptores, alterar el orden de las pilas de interceptores para modificar el tratamiento por defecto o crear pilas nuevas. Además, cada acción dispone de un contexto de ejecución (*ActionContext*) a partir del cual se pueden recuperar los objetos habituales (la *session*, el *request*, el *response*, etcétera) que, en otro caso, se manejan de forma transparente. Por último, cada acción tiene, en su *ActionContext*, una *ValueStack* (pila de valores) en la que se colocan los objetos manipulados por la acción (en la figura, en la *ValueStack* de la *ActionLogin* habrá un objeto de tipo *usuario*; en la de *ActionAddToCarrito* hay un

usuario, un *Producto* y un *Carrito*). Las acciones pueden compartir los objetos que almacenan en las *value stacks* de sus *action contexts* por medio de un mapa de objetos.

El hecho de utilizar el framework Struts2 ya supone la reutilización de las clases que él mismo incluye. Además, como hemos dicho, aporta una serie de ventajas muy importantes respecto de la reutilización de la lógica de dominio de aplicaciones desarrolladas inicialmente para escritorio: si una aplicación de escritorio está bien diseñada (en cuanto a buena separación de la lógica de dominio respecto de la de presentación), puede crearse una acción por cada caso de uso. La acción, en este caso, se correspondería aproximadamente con un patrón controlador (de caso de uso) de los propuestos por Larman.

3.9. Servicios

Un servicio es una funcionalidad que un proveedor de servicios ofrece a sus clientes de forma remota. Esencialmente, el servicio puede ser visto como un componente, pues el cliente o usuario del servicio (en nuestro caso, un ingeniero de software que va a hacer uso de él en su sistema) solo conoce las operaciones que ofrece a través de su interfaz.

Los servicios en ingeniería del software pueden entenderse desde dos puntos de vista: como desarrolladores del servicio y como usuarios. En esta sección presentamos la tecnología de servicios mostrando el caso particular de los servicios web, y presentándola desde estos dos puntos de vista.

3.9.1. Introducción a los servicios web

Los servicios web se ejecutan sobre un tipo de *middleware* mediante el que pueden comunicarse aplicaciones remotas. En esencia, funciona como cualquier otro tipo de *middleware* (RMI, CORBA...), pero con la diferencia importante de que los mensajes que se envían y se reciben se adhieren a un protocolo estandarizado llamado SOAP (*simple object access protocol*). Tanto la llamada al servicio remoto como la respuesta se codifican en SOAP y se transportan, normalmente, mediante http.

Todos los protocolos de mensajería pueden verse como en la siguiente figura, que muestra dos máquinas cualesquiera que se comunican: cuando A desea enviarle un mensaje a B, prepara el mensaje en un formato equivalente al que espera B, y lo envía. En la figura, CA representa el elemento de A que codifica el mensaje, mientras que EB representa el elemento de B por el que esta escucha. Este modelo es válido para RMI, CORBA, servicios web o incluso para un protocolo de mensajería totalmente propietario y que podría implementarse mediante *sockets*: un requisito necesario para que las dos máquinas se entien-

Ved también

Ved el patrón controlador en el apartado “Patrones de diseño”.

Ved también

Ved los servicios web en la asignatura *Ingeniería de software de componentes y sistemas distribuidos* del grado de Ingeniería Informática.

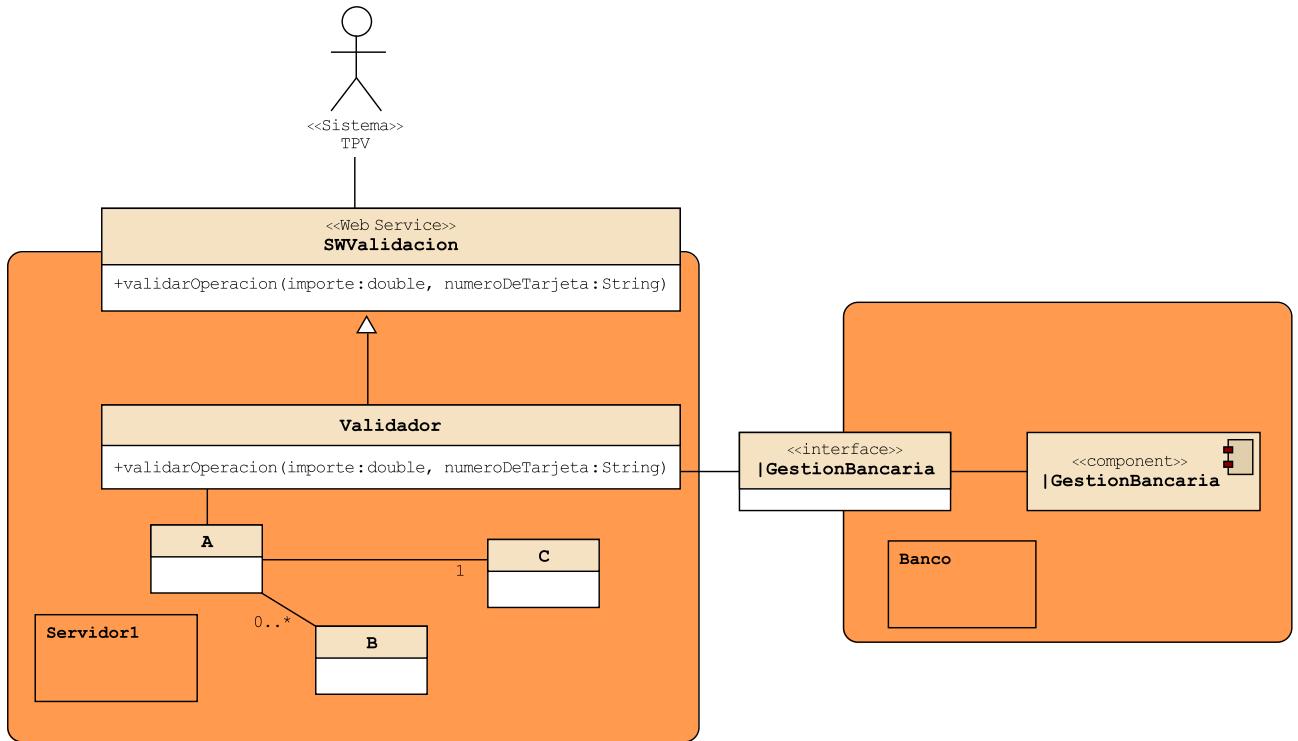
dan es la definición del formato de los mensajes que quieren enviarse desde A hasta B y desde B hasta A, y luego implementar el mecanismo de codificación y descodificación.



Con objeto de estandarizar la comunicación entre máquinas remotas y, a la vez, dar un paso importante en cuanto a la oferta de servicios remotos y a las arquitecturas orientadas a servicios, se ha propuesto SOAP, un protocolo de mensajería basado en XML: así, la llamada a una operación ofrecida por un servidor consiste realmente en la transmisión de un mensaje SOAP, el resultado devuelto es otro mensaje SOAP, etc. De este modo, el cliente puede estar construido en Java y el servidor en .NET, pero ambos conseguirán comunicarse gracias a la estructura de los mensajes que intercambian.

3.9.2. Desarrollo de servicios web

Los servicios web se ofrecen mediante interfaces remotas que se publican en algún servidor, y que se encuentran implementadas en este. En la siguiente figura se ilustra, a muy alto nivel, un servidor 1 que ofrece un servicio de validación de operaciones con tarjetas de crédito a terminales de punto de venta (TPV). Como se ve, la lógica del validador se encuentra oculta al usuario del servicio, que solo conoce la operación ofrecida a través del *web service*. La implementación de la operación, en el servidor 1 utiliza varias clases y, en el ejemplo, un componente externo (tal vez un EJB o un DCOM), que reside en un sistema remoto.



Por lo general, el desarrollo de un servicio web sigue estos pasos:

- 1) Escritura de una clase que implemente las operaciones que se van a ofrecer.
En la figura anterior, sería el caso de validador, que actúa de fachada.
- 2) Generación del servicio web. Todos los entornos de desarrollo actuales incluyen la posibilidad de ofrecer las operaciones contenidas en una clase mediante un servicio web.
- 3) Despliegue y publicación del servicio web en un servidor.
- 4) A la vez que se despliega el servicio, se genera automáticamente un documento WSDL (*web services description language*) que los usuarios del servicio (es decir, los desarrolladores que implementan los sistemas clientes) necesitan para conocer las operaciones ofrecidas. La figura siguiente muestra un fragmento del documento WSDL generado para el servicio WSValidador: obsérvese que se incluye una etiqueta (tag) de tipo *element* para *validarOperacion* que toma dos parámetros (*importe* y *numeroDeTarjeta*).

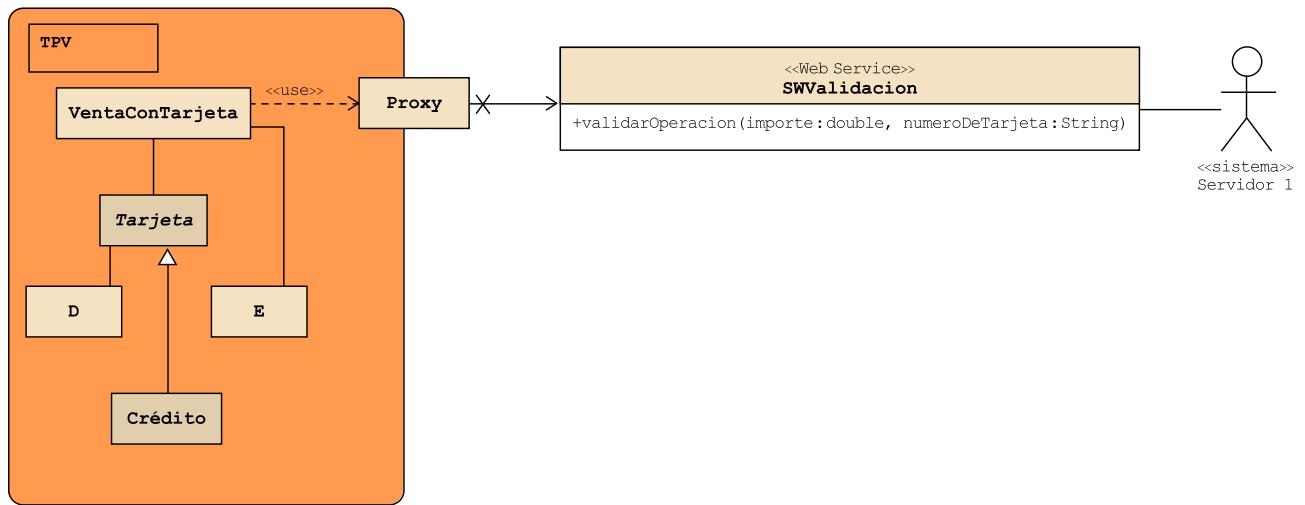
```

<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:ns1="http://org.apache..>
<wsdl:types>
  <xsd:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://dominio">
    <wsdl:documentation>WSValidador</wsdl:documentation>
    <xsd:element name="validarOperacion">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element minOccurs="0" name="importe" type="xsd:double"/>
          <xsd:element minOccurs="0" name="numeroDeTarjeta" nillable="true" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="validarOperacionResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element minOccurs="0" name="return" type="xsd:boolean"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</wsdl:types>

```

3.9.3. Desarrollo de clientes

Una vez que el documento WSDL se encuentra publicado, el usuario del servicio necesita recuperarlo para crear, de alguna manera, un mecanismo de acceso a él que le permita utilizarlo. La figura siguiente muestra el mismo sistema, pero ahora desde el punto de vista del desarrollador del sistema TPV: además de sus clases de dominio, ventanas, etcétera, el desarrollador añadirá a su sistema un *proxy* que represente, para el sistema cliente, el punto de acceso al servicio remoto.



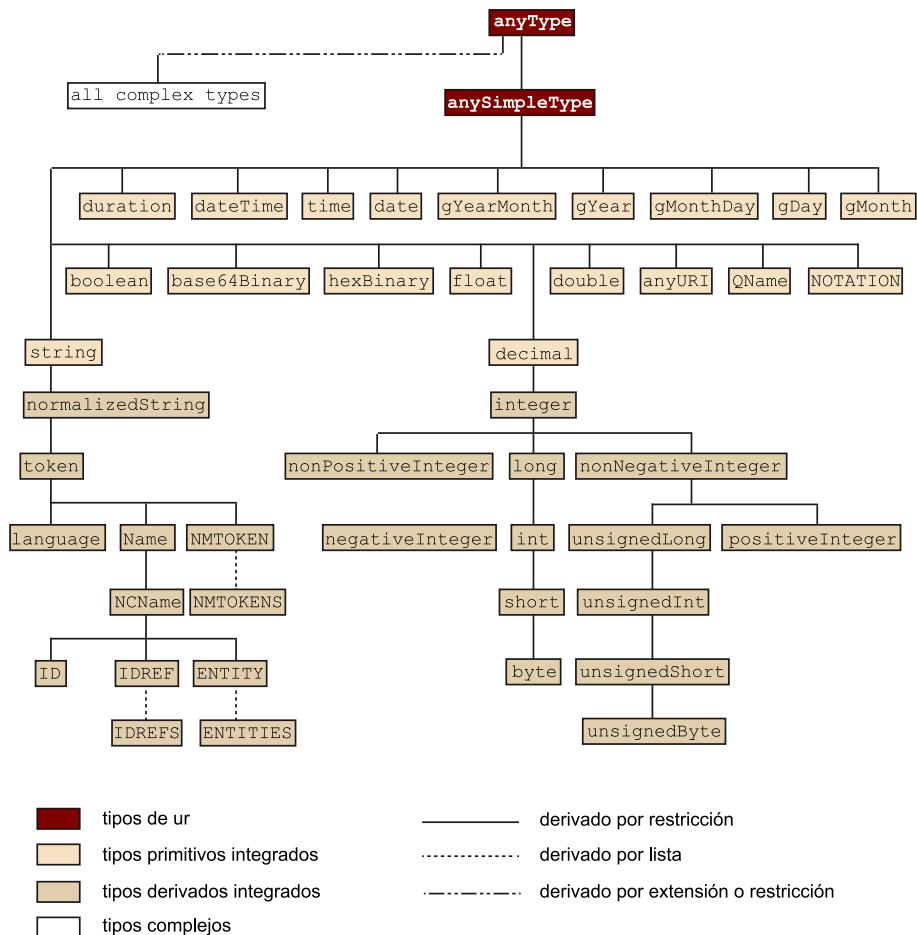
Igual que los entornos de desarrollo actuales, como se ha dicho, incorporan asistentes para publicar las operaciones contenidas en clases en forma de servicios web, también incluyen asistentes que analizan el código de los documentos WSDL y generan *proxies* de conexión de forma automática.

Una de las ventajas que se ha comentado de los servicios web es la interoperabilidad entre plataformas: podemos acceder al servicio web WSValidador, que está implementado en Java, mediante un cliente escrito en el C# de .NET.

3.9.4. Tipos de datos

Los intercambios de información entre el cliente y el servidor se llevan a cabo mediante mensajes en formato SOAP. Cada parámetro enviado o cada resultado devuelto está definido según un tipo de dato. Los tipos de datos que SOAP soporta son los recogidos en el documento XML Schema Part 2: Datatypes Second Edition que se muestran en la figura siguiente, tomada del citado documento: como se ve, se contemplan los tipos primitivos existentes en la mayoría de los lenguajes de programación (enteros, reales, cadenas de caracteres, booleanos, etcétera).

Los tipos complejos (como *Tarjeta*, *ClienteBancario*, etcétera) pueden serializarse para ser codificados y transmitidos, y aparecerían recogidos, en la figura, en la categoría *all complex types*. Los asistentes de generación e inclusión de servicios web bregan con los tipos de datos no estándares, con lo que pueden utilizarse sin problemas en la gran mayoría de las ocasiones.



4. Reutilización a gran escala / Soluciones metodológicas de reutilización

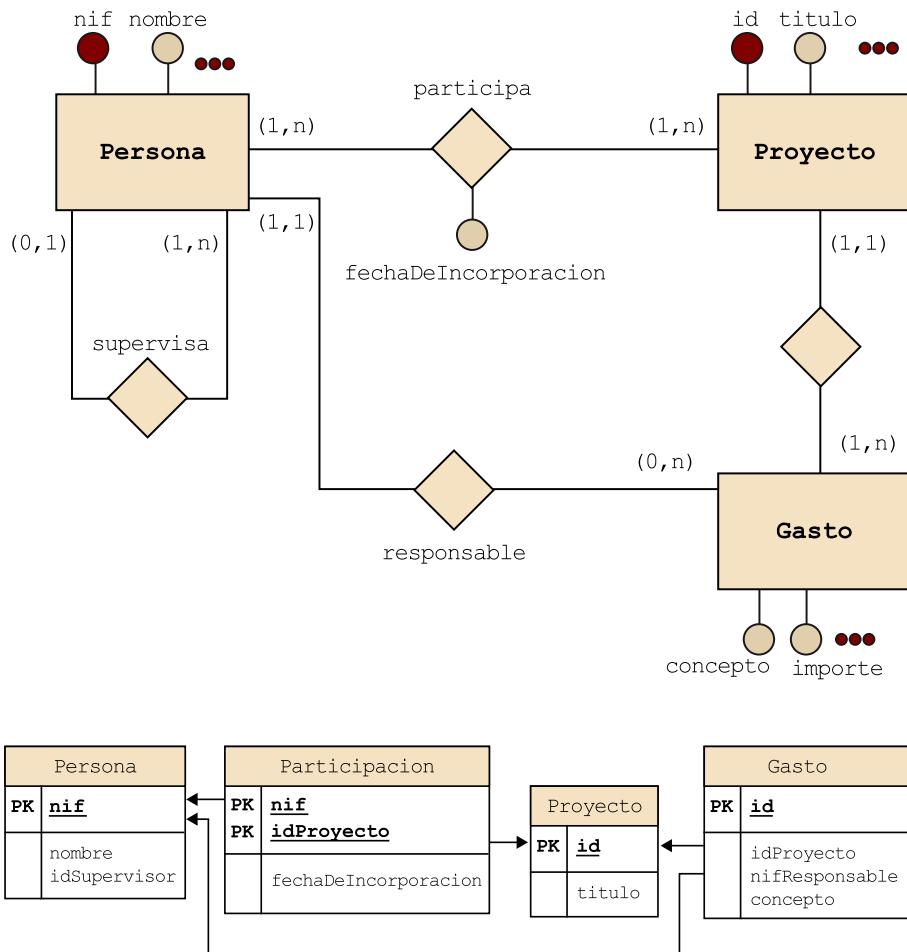
4.1. Introducción

En el apartado anterior se han presentado algunas soluciones técnicas para favorecer la reutilización de software, poniendo la atención en algunos artefactos software concretos, como componentes, librerías o servicios web. Pero la fabricación de software y su reutilización van más allá del mero almacenamiento de elementos reutilizables en repositorios a los que uno pueda acceder para tomar lo que necesite: para mejorar y fomentar el reuso, para no hacer tan solo una reutilización oportunista de componentes predesarrollados, es necesario desarrollar el software utilizando metodologías específicas que la consideren desde el principio.

En esta sección se presentan algunas metodologías de desarrollo en las que la reutilización juega un papel esencial. Hablaremos de ingeniería del software dirigida por modelos, líneas de producto de software, lenguajes específicos de dominio, programación generativa y, más brevemente, de fábricas de software.

4.2. Ingeniería del software dirigida por modelos

En desarrollo tradicional de bases de datos, los requisitos de almacenamiento que se van capturando se representan utilizando un modelo conceptual que, en muchos casos, tiene la forma de un diagrama entidad-interrelación. En una etapa posterior, el diagrama conceptual obtenido se transforma de acuerdo al modelo lógico que corresponda y, en general, se obtiene un diagrama relacional. Este, finalmente, se enriquece con las anotaciones que correspondan para el sistema gestor de base de datos en el que vayan a gestionarse los datos y se obtiene un modelo físico:



Hay muchas herramientas que hacen este tipo de traducciones sucesivas de un tipo de diagrama a otro y que, al final, implementan la base de datos sobre el gestor que se haya seleccionado. Este último supone, en primer lugar, la generación del código SQL que corresponda a la definición de los datos; y en segundo lugar, su ejecución sobre el gestor.

El esquema conceptual es un artefacto software tan genérico que en la práctica es útil para construir un esquema válido para cualquier modelo lógico de base de datos: relacional, orientado a objetos, objeto-relacional o, incluso, para modelos lógicos más antiguos, como el jerárquico. Para el modelado conceptual, por tanto, no se toma en consideración la plataforma final de ejecución de la base de datos; igualmente, el modelo lógico sí que depende del soporte tecnológico final, pero no de los detalles concretos del gestor de base de datos.

Ejemplo

En SQL Server, por ejemplo, los valores lógicos pueden representarse con columnas de tipo bit, mientras que en Oracle deben ser de tipo numérico.

La idea de la ingeniería del software dirigida por modelos (MDE: *model-driven engineering*) es muy parecida: el ideal de este paradigma podría resumirse en “Dibujar un sistema y pasar directamente a ejecutarlo”. Para ello se deben representar tanto la estructura del sistema como su comportamiento. Además, y con objeto de facilitar la portabilidad de los sistemas desarrollados, se comienza diseñando modelos independientes de la plataforma final de ejecución que, en un paso posterior, se transforman a modelos dependientes:

Ved también

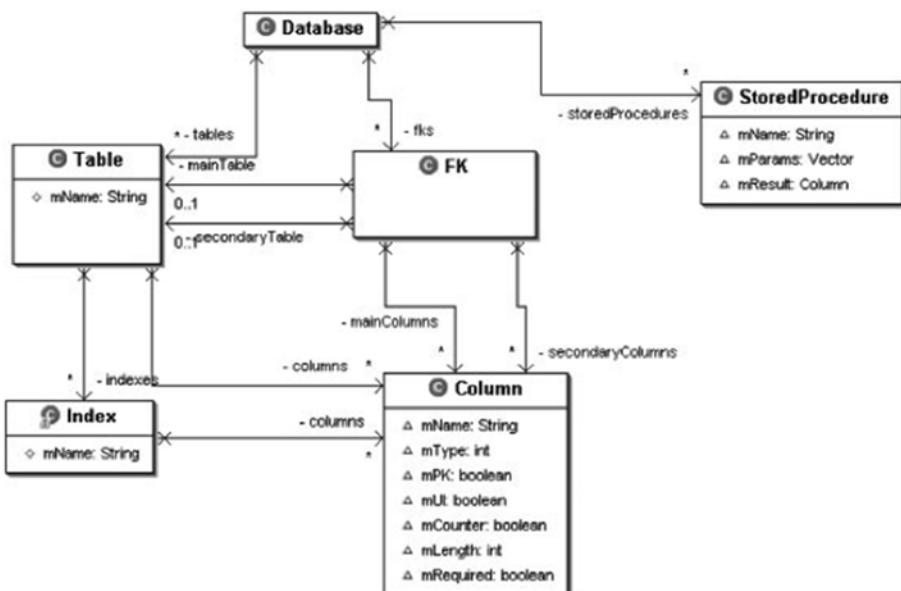
Ved el módulo sobre desarrollo de software dirigido por modelos de esta misma asignatura.

- El ingeniero de software construye un PIM (*platform-independent model*) que describe el sistema sin considerar los detalles finales de implementación.
- El modelo independiente se traduce, mediante una transformación, a un PSM (*platform-specific model*), en el que sí que se tienen en cuenta las características concretas de la plataforma final de ejecución.
- Finalmente, una última etapa de generación de código transforma el PSM a código compilable y ejecutable.

4.2.1. Diferentes aproximaciones: metamodelos propios y metamodelos estándares

Igual que el lenguaje traducido por un compilador requiere adherirse a una sintaxis específica para poder ser entendido, la automatización de la transformación de un modelo desde uno a otro tipo necesita también que el modelo esté representado con alguna sintaxis específica. Esta es la función principal de los metamodelos, que no son sino modelos que se utilizan para representar otros modelos.

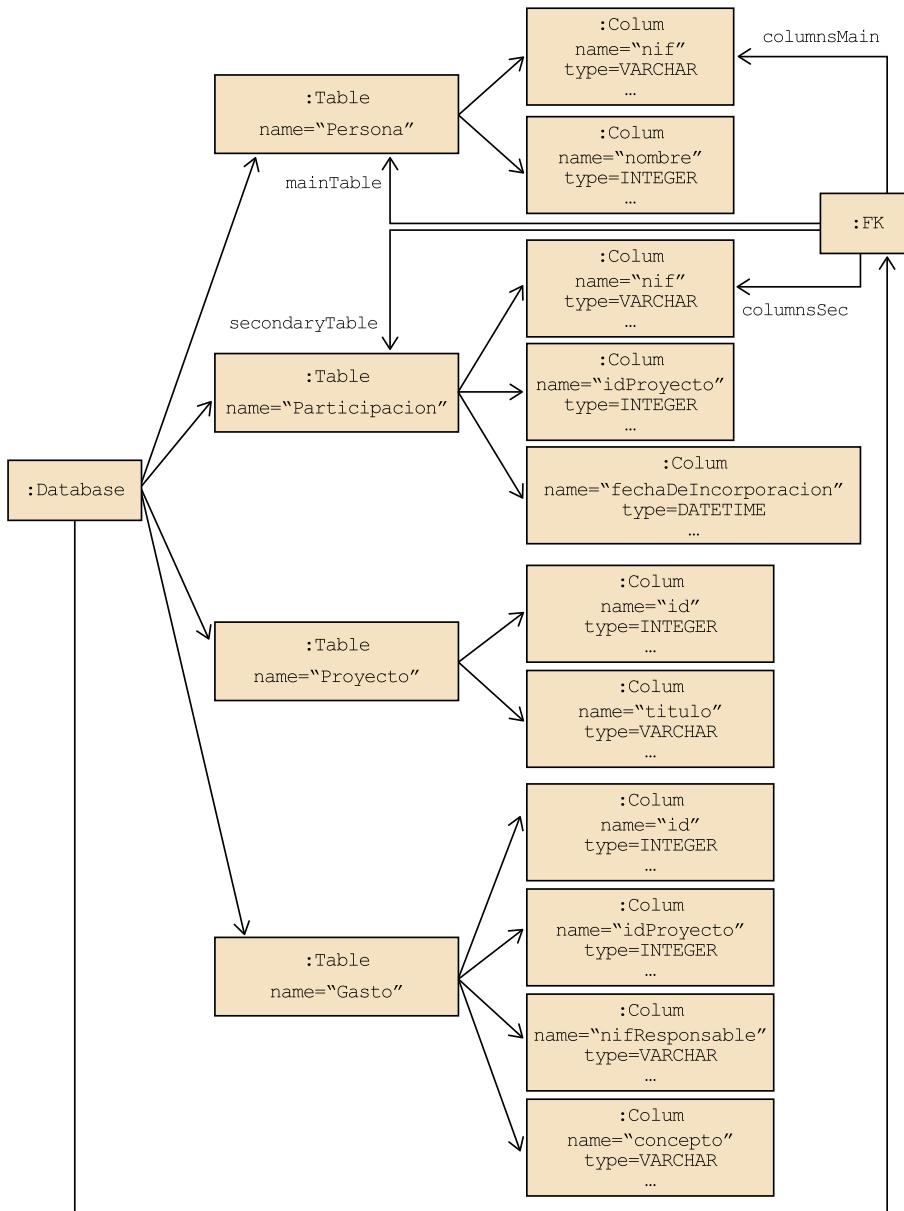
La figura siguiente (tomada de Polo y otros autores, 2007) muestra uno de los metamodelos utilizados por una herramienta que, a partir de una base de datos relacional implementada en Access, Oracle o SQL Server, es capaz de generar diferentes tipos de aplicaciones para gestionar la información almacenada en dicha base de datos. La herramienta lee la información de tablas, columnas y procedimientos almacenados y construye una instancia del metamodelo mostrado en la figura:



Consulta recomendada

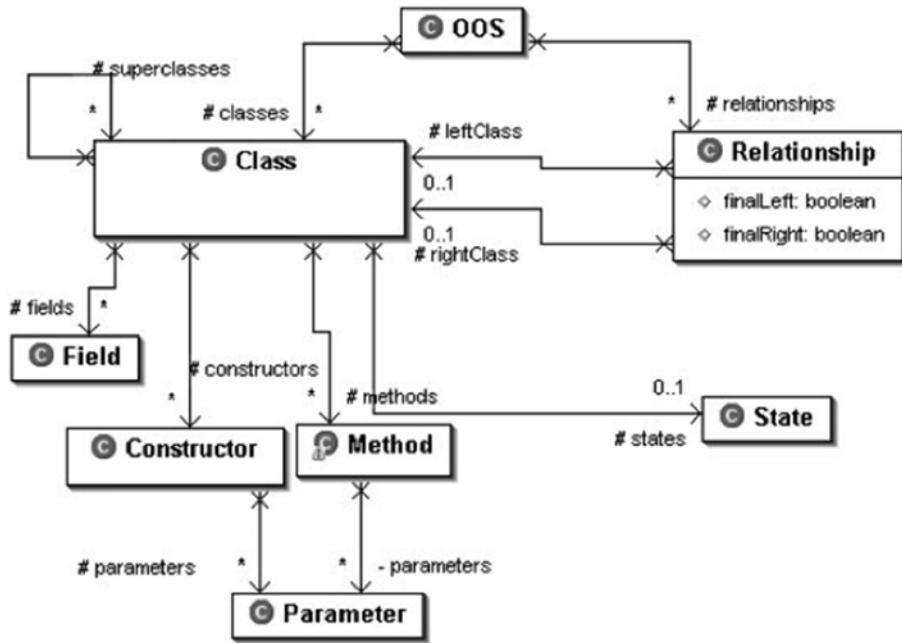
M. Polo; I. García-Rodríguez; M. Piattini (2007). “An MDA-based approach for database reengineering”. *Journal of Software Maintenance & Evolution: Research and Practice* (vol. 6, núm. 19, págs. 383-417).

Así, a partir de la base de datos de personas, proyectos y presupuestos que se mostraba arriba, se obtendría un conjunto de instancias de *Database*, *Table*, *Index*, *Column*, *FK* y *StoredProcedure*. En la figura siguiente se muestran parte de estos objetos: la instancia de *Database* conoce a cuatro instancias de *Table* y a un objeto de tipo *FK* (conoce realmente a más, pero no se han representado para no sobrecargar el diagrama); cada objeto de tipo *Table* conoce a varias instancias de tipo *Column*; la *foreign key* (*objeto FK*) relaciona las instancias de tipo *Table* cuyos campos *name* son "Persona" y "Participación", mediante las columnas apuntadas por *columnsMain* y *columnsSec*.

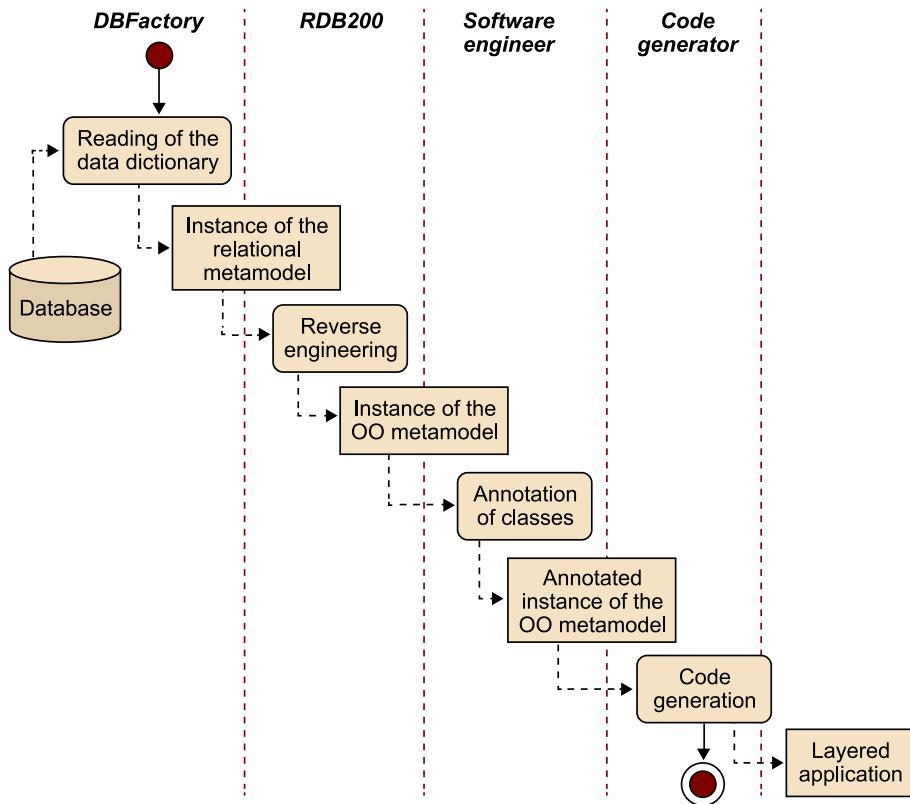


A los modelos que se corresponden con instancias de metamodelos bien definidos se les pueden aplicar algoritmos de transformación para realizar “transformaciones entre modelos”. La herramienta mencionada transforma, me-

diante un proceso de ingeniería inversa, modelos conformes al metamodelo de base de datos relacional a modelos conformes a un metamodelo que representa la estructura de clases de un sistema orientado a objetos:



A los modelos orientados a objetos se les pueden aplicar diferentes algoritmos de generación de código. El proceso completo se ilustra en la figura siguiente tomada también de Polo y otros autores (2007): un proceso lee la información de diseño del gestor de base de datos a partir del diccionario de datos, obteniendo una instancia del metamodelo relacional; a este se le aplica un proceso de ingeniería inversa para transformar el modelo en una instancia del metamodelo de objetos; el usuario puede entonces anotar las clases con máquinas de estado para describir el comportamiento de sus instancias; finalmente, uno o más algoritmos de generación de código producen una aplicación multicapa capaz de gestionar la base de datos con un conjunto básico de operaciones (las operaciones CRUD) más aquellas que el ingeniero de software haya anotado en las máquinas de estado.



Una desventaja importante del uso de metamodelos propios (como los dos mostrados anteriormente) es la dificultad para portar los modelos entre herramientas. En los últimos años, el OMG (Object Management Group) ha publicado y mejorado diversos documentos que se han convertido en estándares *de facto* para favorecer el desarrollo de software siguiendo un enfoque MDE:

- UML, para representar modelos de sistemas. Sus mecanismos de extensión basados en perfiles (mediante estereotipos, restricciones y valores etiquetados) permiten, por ejemplo, aplicar ciertos mecanismos de transformación solo a los objetos que tienen un estereotipo específico: a una clase estereotipada como «ORM Persistable» se le puede aplicar una transformación para, por un lado, generar una tabla y, por otro, generar una clase de dominio y una clase fabricación pura auxiliar que le gestione la persistencia.
 - MOF (*meta object facility*), que es un lenguaje de metamodelo estructurado en cuatro niveles: en el nivel más bajo (M0) se encuentran los datos reales (las personas almacenadas en la base de datos con su nombre, apellidos, etcétera); en el siguiente (M1), la estructura de la base de datos que, por ejemplo, puede ser relacional; en M2 el metamodelo para representar bases de datos relacionales; en M3, finalmente, se encuentra MOF, que representa la estructura genérica que debe tener cualquier metamodelo.
- La siguiente figura ilustra esta relación entre niveles: en M2 se encuentran los metamodelos para representar procesos de negocio (SPEM), sistemas software orientados a objeto (UML) y almacenes de datos (CWM); todos estos metamodelos son conformes a MOF (nivel M3), que es además con-

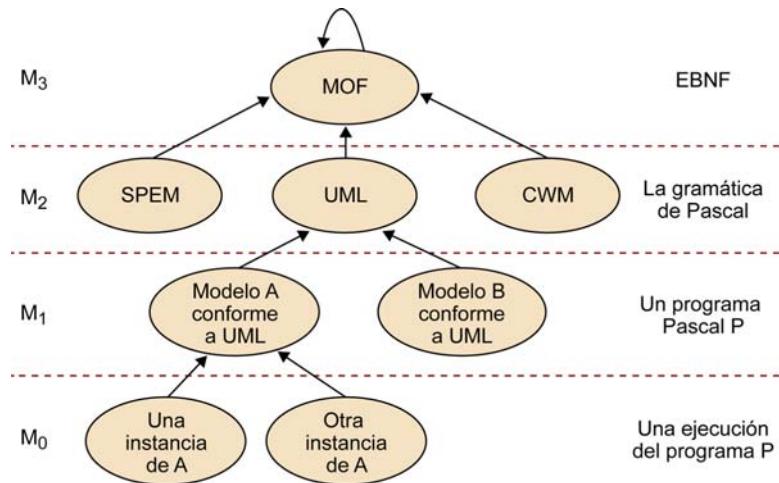
Nota

Las ideas del metamodelado y de la transformación de modelos tienen realmente muchos años. Ya las primitivas CASE disponían de sus propios metamodelos y de herramientas de importación y exportación. Véase por ejemplo el libro: Piattini et al. (1996). *Análisis y diseño detallado de Aplicaciones informáticas de gestión*. Madrid: Editorial Ra-Ma.

forme consigo mismo; los modelos de sistemas software orientados a objetos específicos o el modelo de negocio de cierta empresa se encuentran en M1 y son conformes a su correspondiente metamodelo de M2; por último, los datos, los objetos y los procesos reales de cada ejecución del sistema o del proceso de negocio se encuentran en M0.

Nota

EBNF: Extended Backus-Naur Form, define una manera formal de describir la gramática de un lenguaje de programación.



- OCL (*object constraint language*) es un lenguaje formal que, inicialmente, fue concebido para anotar con restricciones los modelos diseñados con UML (lo que permite eliminar la ambigüedad inherente a los diagramas). En MDE, OCL permite lanzar consultas sobre modelos y escribir pre y pos-condiciones para las transformaciones.
- QVT (*query-view-transformation*) es un lenguaje que permite programar transformaciones entre modelos. Esencialmente, una transformación QVT se puede ver como una función cuyo dominio es el metamodelo origen (por ejemplo, UML) y cuyo recorrido es el metamodelo destino (por ejemplo, el metamodelo para representar bases de datos relacionales): así, a partir de un modelo concreto de un sistema (es decir, una instancia del metamodelo origen) se obtiene un modelo concreto (es decir, una instancia del metamodelo destino).
- XMI (*XML metadata interchange*) es el estándar de OMG para permitir el intercambio de modelos. Un modelo se almacena como un documento XMI y puede ser procesado por cualquier otra herramienta.
- MOF2Text es el estándar para generar texto (código fuente) a partir de modelos. El lenguaje QVT permite la transformación entre modelos, pero hace difícil la obtención de código a partir de modelos.

Nota

Han sido el desarrollo de UML, el impulso de la investigación en universidades e institutos y las actividades de OMG los que han llevado a los avances actuales en MDE.

Ved también

En el módulo sobre desarrollo de software dirigido por modelos se tratan todos estos conceptos con mayor profundidad.

Nota

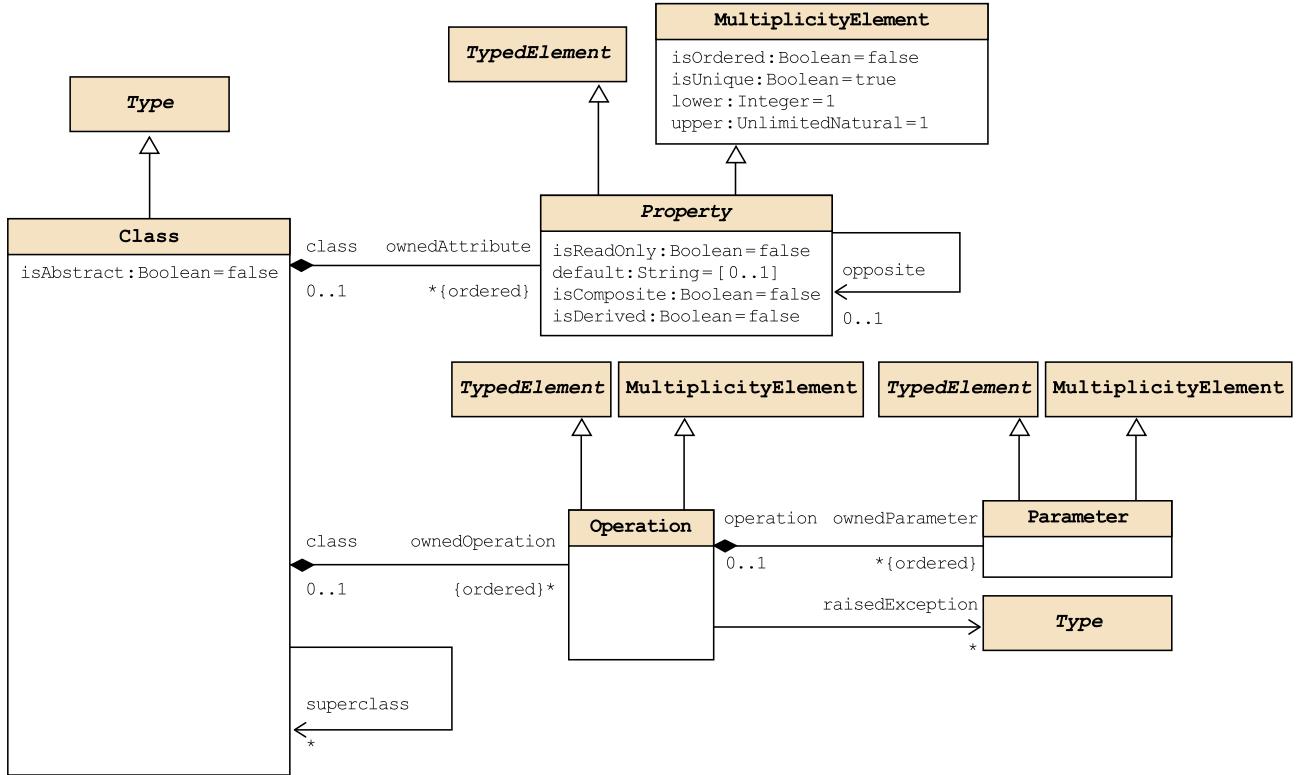
En la edición de 2009 del conocido libro de Pressman aparecen más de 200 siglas utilizadas en ingeniería de software. Sin embargo, no se incluyen algunas de las siglas que hemos visto aquí.

Roger S. Pressman. *Ingeniería del software: un enfoque práctico*. Editorial McGraw-Hill.

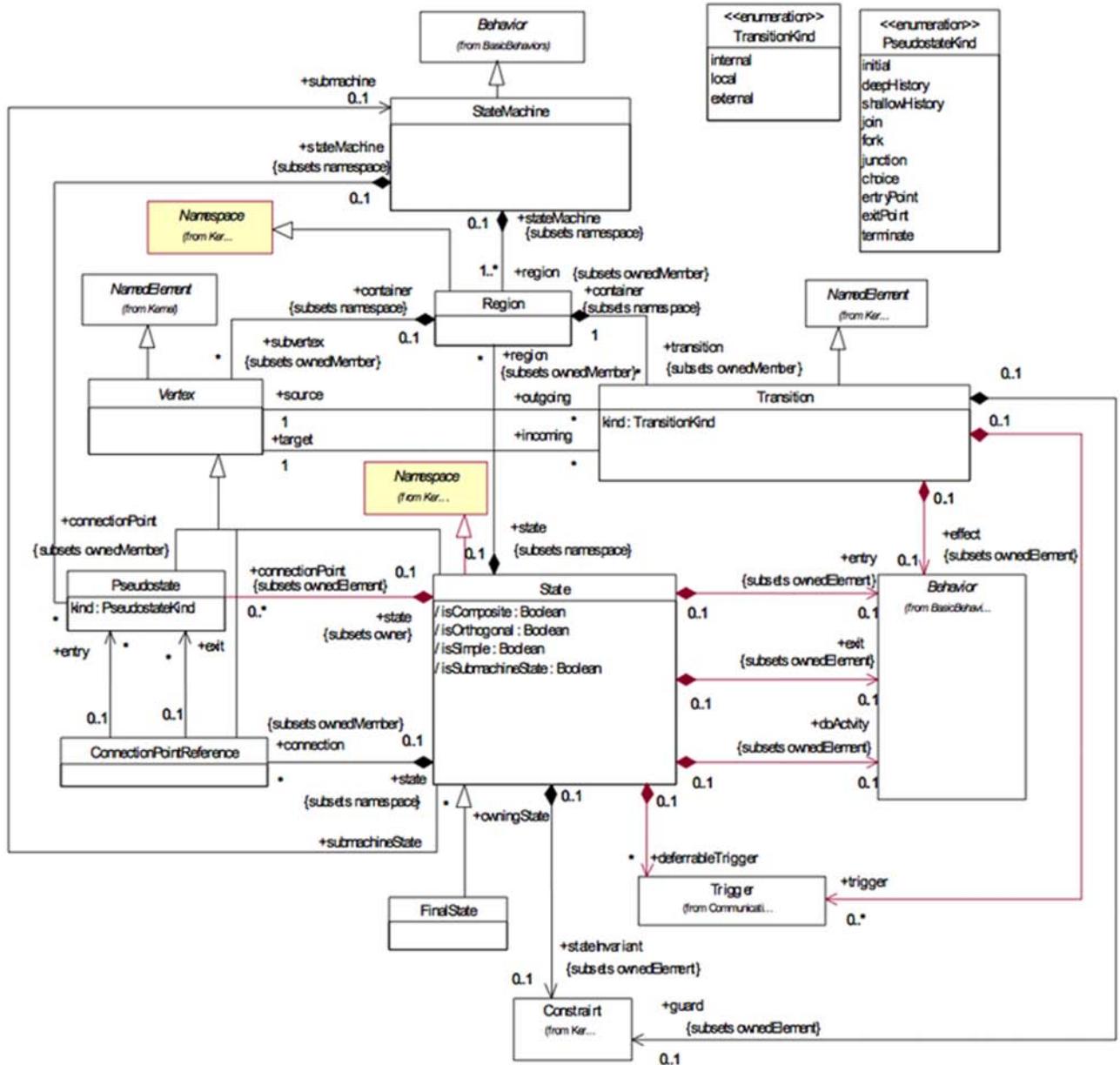
4.2.2. UML como lenguaje de modelado en MDE

La especificación de UML dada por el OMG sigue un enfoque formal de metamodelado: todo modelo UML es una instancia de su metamodelo. Estas relaciones de instanciación permiten aplicar mecanismos estándares de transfor-

mación de modelos. Así, por ejemplo, todo diagrama de clases que sea conforme debe ser una instancia del siguiente metamodelo, pues así se indica en la especificación oficial de UML (documento *Unified modeling language: infrastructure*, versión 2.0, de marzo del 2006, página 105).



Por otro lado, también todos los aspectos de comportamiento están formalmente definidos en UML. Las máquinas de estado, por ejemplo, deben ser conformes al siguiente metamodelo:

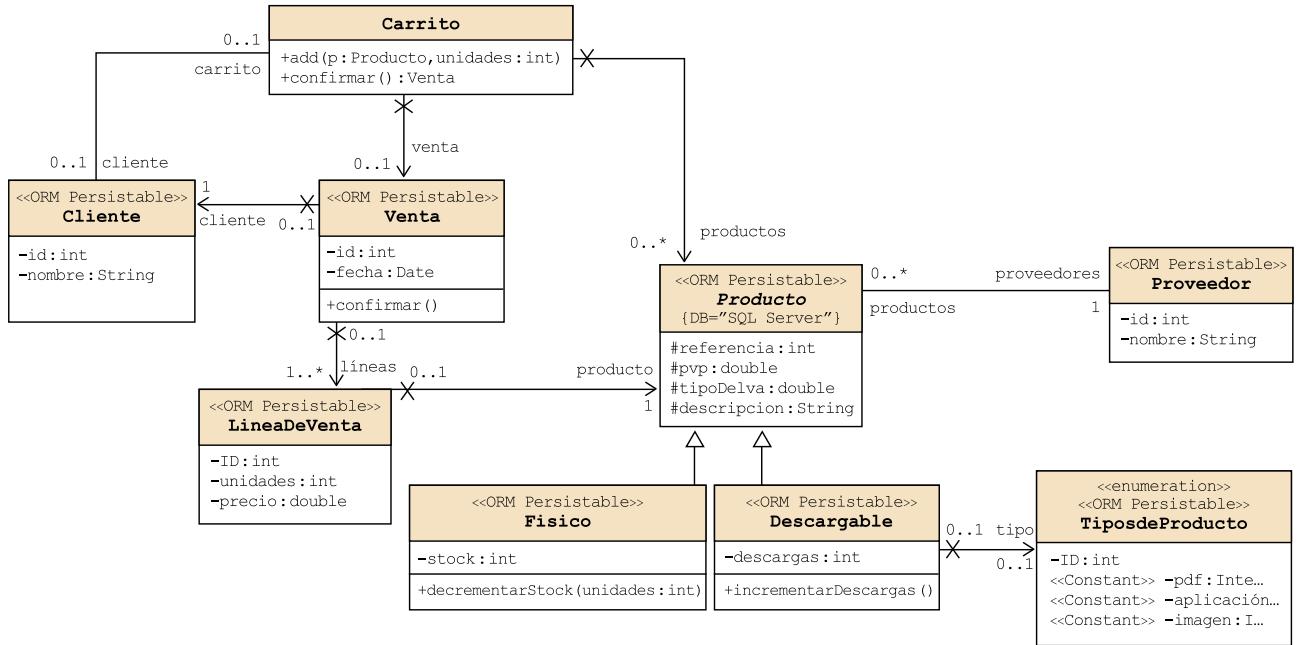


Un estado concreto de una máquina de estados representada conforme al modelo anterior es, realmente, una instancia de la clase *State* que aparece en la figura. Por ello, a las clases incluidas en los metamodelos de UML se las llama metaclasses. Para terminar la ilustración, una clase de un diagrama de clases es una instancia de la metaclass *Class*.

Profiles de UML

Asumiendo ya que conocemos lo que es una metaclass, concepto que acaba de introducirse, los perfiles van a permitir extender metaclasses de los metamodelos para que sean útiles para propósitos específicos. Un perfil se compone de estereotipos y de valores etiquetados (*tagged values*). El diagrama de clases de la siguiente figura representa parte de la capa de dominio de un sistema de gestión de una tienda virtual: todas las clases, salvo *Carrito* (que es una clase volátil), tienen el estereotipo *ORM persistable*, lo cual denota que son clases persistentes que deberán ser transformadas a tablas mediante algún mecanis-

mo de mapeo objeto-relacional. La clase producto, además, tiene el *tagged value* {DB="SQL Server"}, que indica que debe ser transformada a una tabla para el gestor de bases de datos relacionales SQL Server.



Entonces, los perfiles permiten dotar a los elementos de los modelos UML de nuevas características, como por ejemplo:

- Utilizar una terminología adaptada a una plataforma o dominio particular. Dos posibles valores de un *tagged value* de una clase *ORM persistable* podrían ser {Database=SQL Server} y {Database=Oracle}, lo que puede suponer una variación en el tratamiento que hay que dar a la transformación de esas clases.
- Utilizar una notación para los elementos que no tienen notación (como las acciones, por ejemplo).
- Utilizar una notación diferente para símbolos ya existentes (utilizar un icono que muestre un formulario para representar una clase de la capa de presentación).
- Dotar de más semántica a algún elemento del metamodelo.
- Crear semántica para elementos que no existen en el metamodelo (estereotipando, por ejemplo, un actor de UML con el estereotipo *timer* para denotar que es un reloj que, periódicamente, envía algún tipo de estímulo al sistema).
- Evitar construcciones que sí que están permitidas en el metamodelo.

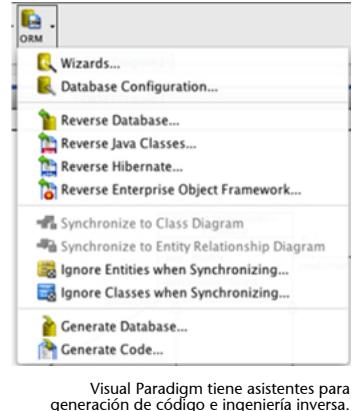
- Añadir información necesaria para hacer transformaciones entre modelos, como el caso del estereotipo ORM *persistable* que se ha comentado.

La especificación de UML incluye diversos ejemplos de conjuntos de estereotipos para varias plataformas de desarrollo de aplicaciones basadas en componentes, como los JEE/EJB, COM, .NET o CCM.

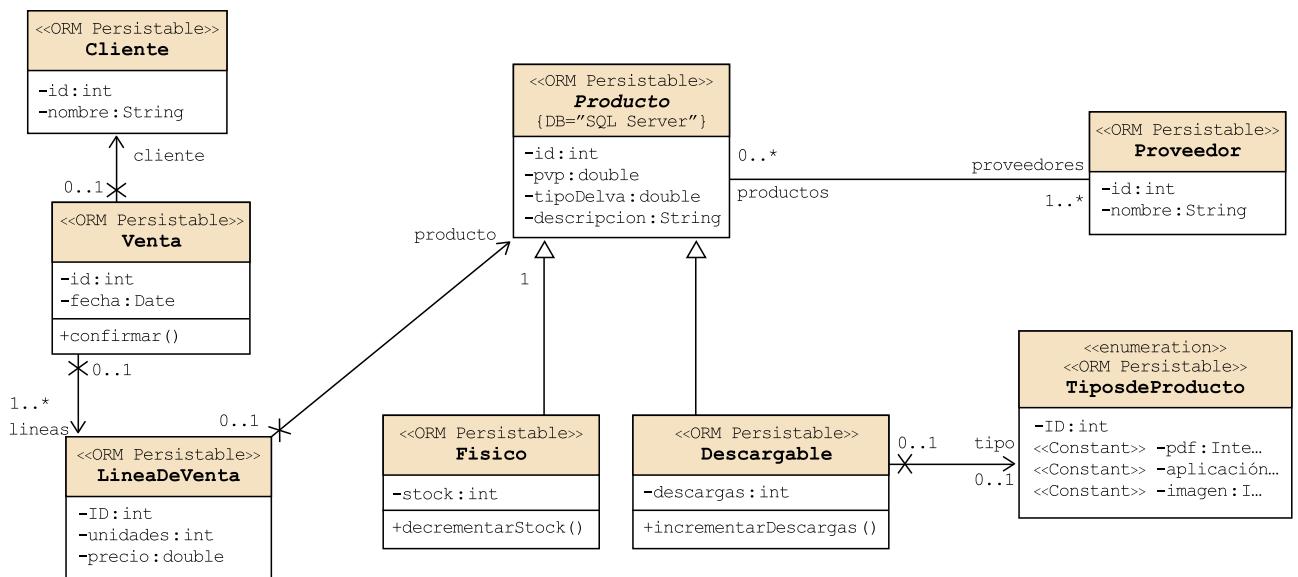
4.2.3. Soporte automático

Cada vez aparecen más herramientas (y *plugins* para herramientas existentes) capaces de bregar con modelos y de hacer transformaciones entre modelos. Una de ellas, compatible con UML 2, es Visual Paradigm, una suite de herramientas comerciales de modelado con diferentes ediciones (estándar, profesional, empresarial), que da, por ejemplo, la posibilidad de generar código en diferentes lenguajes y de obtener diagramas a partir de código.

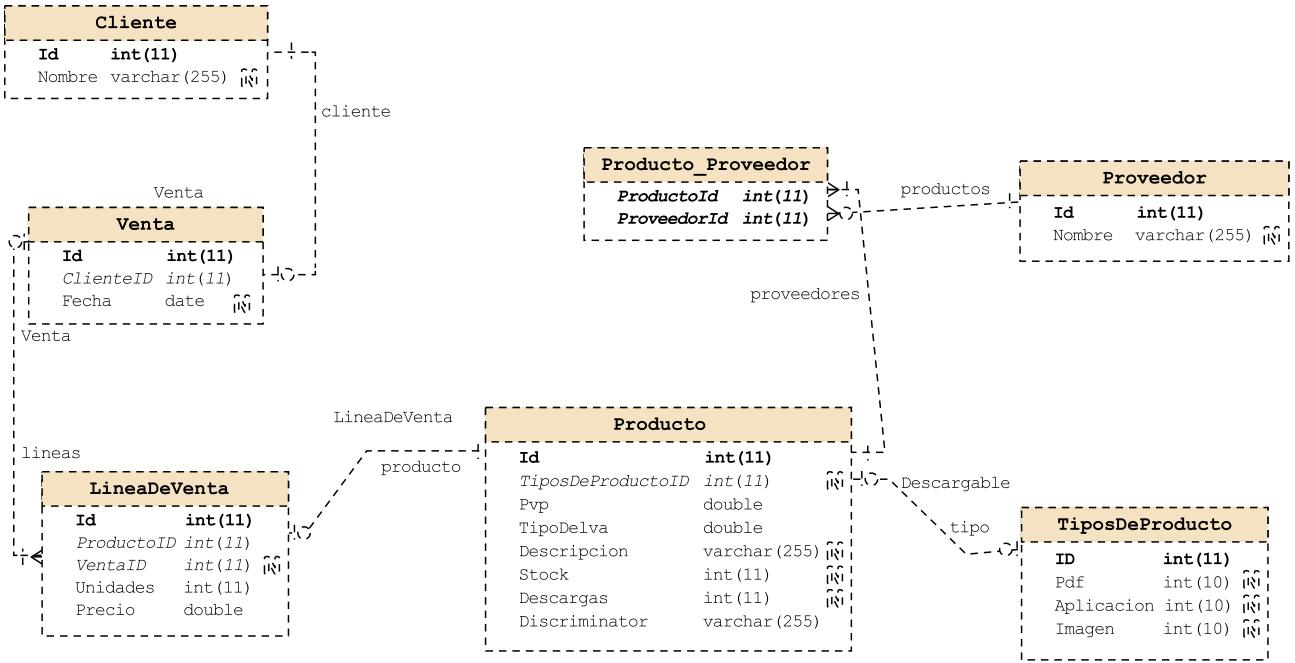
Como se ha comentado anteriormente, una de las transformaciones de modelos más habituales es la transformación de diagramas de clases en bases de datos: en efecto, el diagrama de clases de la capa de dominio se puede utilizar en muchos casos como modelo de datos para construir la base de datos del sistema. Visual Paradigm dispone del estereotipo *ORM persistable*, con el que se anotan las clases que han de transformarse en tablas:



Visual Paradigm tiene asistentes para generación de código e ingeniería inversa.



Mediante un asistente, en el que pueden configurarse muchos aspectos de la transformación, la herramienta genera un diagrama entidad-interrelación, que se muestra en la figura siguiente. Obsérvese, por ejemplo, la creación de la entidad intermedia producto-proveedor, que procede de la asociación de muchos a muchos entre las clases *Producto* y *Proveedor*. Igualmente, el pequeño árbol de herencia del *Producto* y sus dos especializaciones se ha traducido a una sola entidad *Producto*, en la que un atributo *discriminator* se utiliza para diferencias entre los dos subtipos (*Físico* y *Descargable*).



En un último paso, la herramienta conecta vía JDBC con el gestor de base de datos que se desee, se generan las instrucciones SQL de creación de tablas que correspondan y se crea el modelo físico de la base de datos. En este ejemplo, generamos una base de datos para MySQL:

```

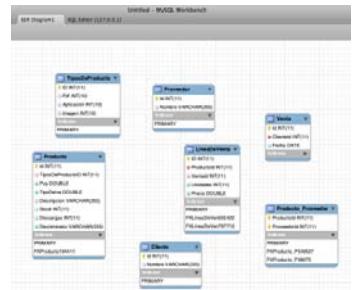
CREATE DATABASE IF NOT EXISTS `tienda`;
USE `tienda`;
-- MySQL dump 10.13 Distrib 5.5.16, for osx10.5 (i386)
--
-- Host: localhost Database: tienda
--
-- Server version      5.1.50

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `Proveedor`
--

DROP TABLE IF EXISTS `Proveedor`;
/*!40101 SET @saved_cs_client      = @@character_set_client */;
/*!40101 SET character_set_client = utf8 */;
CREATE TABLE `Proveedor` (
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `Nombre` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`Id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;

...
  
```



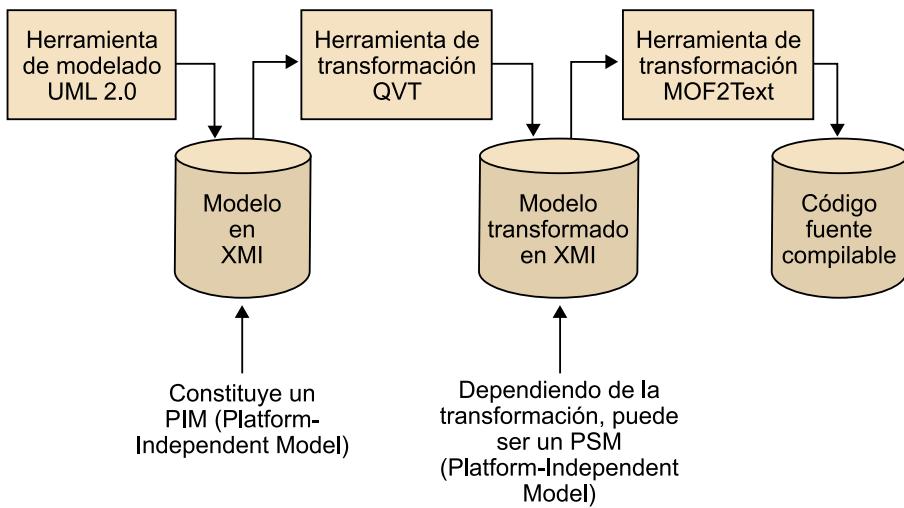
Algunas de las tablas que se crean en la base de datos

Además, los modelos construidos con esta herramienta se pueden exportar en formato XMI y ser importados, por ejemplo, con Medini QVT, un conjunto de herramientas (distribuidas bajo licencia EPL) que implementa el estándar QVT del OMG. De este modo, podemos diseñar un sistema orientado a objetos

con Visual-Paradigm (o con otra herramienta de modelado que exporte en XMI), crear un perfil con estereotipos y valores etiquetados, anotar con estos los elementos del sistema orientado a objetos, y finalmente, implementar las transformaciones que se deseen en Medini.

A los modelos obtenidos con Medini QVT les podremos aplicar una transformación con MOFScript (una implementación de MOF2Text, el estándar de OMG para transformación de modelos a texto) y generar el código que se desee.

Con todo esto, podemos dar soporte a un proceso completo de MDE:



Otra herramienta muy conocida para transformación de modelos es Atlas, desarrollada por AtlanMod, un grupo de investigación del Instituto Nacional de Investigación en Informática y Automática de la Escuela de Minas de Nantes, en Francia. Atlas incluye el lenguaje de transformación ATL y es conforme a MOF (es decir, define su propio metamodelo), aunque no es extensión de UML.

4.3. Lenguajes de dominio específico (DSL: *domain-specific languages*)

Los lenguajes de dominio específico²⁴ (DSL) permiten describir sistemas desde un nivel muy alto de abstracción, utilizando una terminología y una notación muy próxima a la de los expertos del dominio. De hecho, un mismo sistema puede ser descrito con diversos DSL adecuados a los diferentes tipos de expertos. Esta característica permite que sea el propio experto el que modele su sistema; más adelante, esta descripción deberá ser procesable automáticamente

⁽²⁴⁾A menudo se traduce *domain-specific language* por 'lenguaje específico de dominio'. Consideramos más acertado utilizar 'lenguaje de dominio específico', ya que un DSL es un lenguaje especialmente adecuado para representar un dominio de aplicación muy concreto.

por un computador. Para conseguir este procesamiento automático, el modelo construido por el experto se transforma a otro modelo que, ahora sí, podrá ser aplicado en un entorno MDE para generar el código de una aplicación.

Un DSL se compone de:

- Una **sintaxis abstracta**, que contiene los conceptos del lenguaje, el vocabulario, las relaciones entre ellos y las reglas que permiten construir sentencias válidas.
- Una **sintaxis concreta**, que describe la notación permitida para construir modelos, y que puede ser visual o textual.
- Una **semántica**, que asocia significado al modelo y a sus elementos, y que puede ser denotacional⁽²⁵⁾, operacional⁽²⁶⁾ o axiomática⁽²⁷⁾.

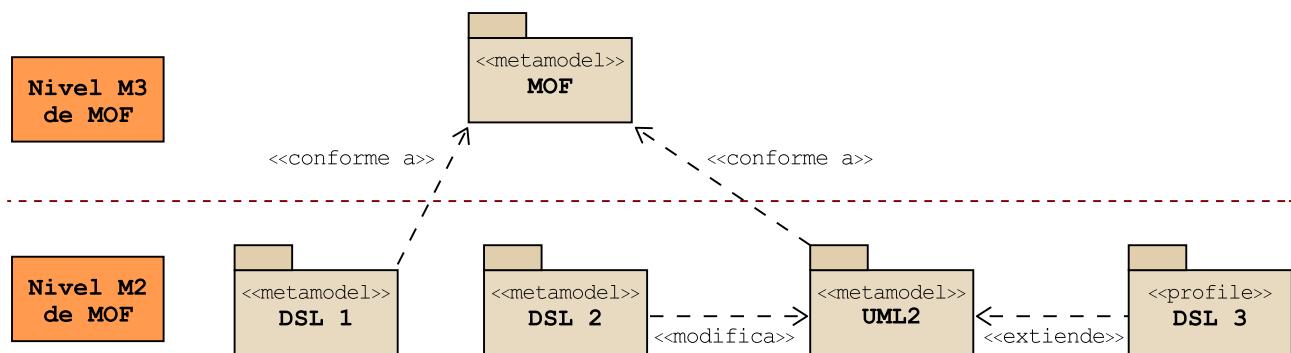
Véase también

En el módulo sobre desarrollo de software dirigido por modelos se tratan los DSL con mayor profundidad.

4.3.1. Definición de DSL

Teniendo en cuenta el requisito de que los dominios modelados con un DSL deben poder ser transformados a modelos procesables en entornos MDE, la construcción de lenguajes de dominio específico puede hacerse de dos maneras:

- Definiendo un metamodelo nuevo que sea conforme a MOF. En la figura siguiente, sería el caso del DSL 1.
- Extendiendo el metamodelo de UML mediante la adición, modificación o eliminación de elementos (lo que se conoce como una “extensión pesada”, *heavy extension*, que en la figura corresponde al DSL 2) o mediante la especialización de algunos de sus conceptos creando un perfil, como el DSL 3.



Nota

La definición de dominios es un concepto esencial tanto en líneas de producto software como en programación generativa. Ambos paradigmas se tratan a continuación. En las

secciones que dedicamos a estos dos temas podrá entenderse la relación de los DSL con la reutilización.

4.4. Línea de producto de software

Clements y Nortrop (2001) definen una línea de producto de software (LPS) como:

“Un conjunto de sistemas software que comparten un conjunto común y gestionado de características (*features*) que satisfacen las necesidades específicas de un dominio o segmento particular de mercado, y que se desarrollan a partir de un sistema común de activos base (*core assets*) de una manera preestablecida”.

Gomaa (2005) apunta que el interés por las LPS surge en el campo de la reutilización del software, cuando los desarrolladores observan que resulta mucho más provechoso reutilizar diseños arquitectónicos completos en lugar de componentes software individuales. De hecho, la idea de las LPS consiste en la construcción de productos software a partir de la reutilización de los mencionados *core assets*.

El enfoque de reutilización en este contexto es más planificado que oportunista (Díaz y Trujillo, 2007): en desarrollo de software tradicional, se aprecia la posibilidad de reutilizar un componente después de haberlo desarrollado; en LPS, la reutilización es planificada, de manera que se reutilizan la mayor parte de los activos base en todos los productos de la línea.

Si se desea, por ejemplo, desarrollar un sistema cliente-servidor que permita a varias personas jugar a juegos de mesa de manera remota (ajedrez, damas, parchís, trivial, etcétera), la empresa de desarrollo puede aplicar un enfoque de LPS e identificar los siguientes *core assets*, que estarán presentes en todos o casi todos los productos que se implementen:

1) Mover ficha. En todos estos juegos se mueve una ficha en el turno del jugador.

2) Gestión del turno. En todos los juegos mencionados se sigue una cierta política de asignación del turno al siguiente jugador, que varía en función del juego²⁸.

3) Lanzamiento de dados. Algunos de los juegos (parchís, trivial y otros) necesitan que el jugador lance los dados para mover.

4) Permanecer en la sala de espera. Para poder comenzar una partida se necesitan exactamente dos jugadores en ajedrez y damas, y dos o más de dos en los restantes.

5) Incorporación a una partida. Para que una partida pueda comenzar, es preciso que los jugadores se incorporen a ella.

Consulta recomendada

P. Clements; L. Northrop (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley.

Consulta recomendada

H. Gomaa (2005). *Designing Software Product Lines with UML. From use cases to pattern-based software architectures*. Addison-Wesley.

Consulta recomendada

Ó. Díaz; S. Trujillo (2010). “Líneas de Producto Software”. En: Piattini y Garzás (eds.). *Fábricas de Software*. Madrid: Ra-Ma.

⁽²⁸⁾Turnos alternativos en ajedrez y damas; dos movimientos seguidos si se sacan dos seises en el parchís o si se acierta una pregunta en el trivial, por ejemplo.

6) Sistema de registro. Dado que se trata de un sistema cliente-servidor, es probable que se exija a todos los jugadores estar previamente registrados en el sistema.

7) Sistema de identificación. Para incorporarse a una partida es necesario estar registrado y haberse identificado mediante, por ejemplo, un sistema basado en un nombre de usuario y una contraseña.

8) Comer. En algunos de los juegos se pueden comer fichas del contrincante: en ajedrez y damas, la ficha comida desaparece; en el parchís, la ficha comida vuelve a la casilla de origen; en el trivial no se comen fichas.

No todos los juegos de mesa tienen todas esas características, y no todos las implementan de la misma forma; además, cada uno tiene otra serie de características propias.

Por ejemplo, el ajedrez y las damas comparten el tablero, pero no el parchís ni el trivial; los movimientos permitidos son diferentes en todos los juegos; en ajedrez se gana cuando se da jaque mate, en las damas vence el jugador que deja al contrario sin fichas, en el trivial, aquel que reúne seis “quesitos”, y el que lleva todas sus fichas a la casilla final en el parchís.

En un desarrollo de software tradicional la empresa comenzaría, probablemente, por la implementación completa de un juego concreto: por ejemplo, el parchís. A partir de este, es posible que abordase la construcción del trivial, para lo que podría reutilizar la lógica responsable del registro e identificación de usuarios, así como la encargada del lanzamiento de datos. Quizá esas funcionalidades estén empaquetadas en componentes. A pesar de esto, la reutilización, en este caso, se produce *a posteriori*.

En LPS se trabaja a dos niveles:

- En el nivel de **Ingeniería de dominio** se analizan, diseñan, implementan y prueban las características comunes, que se incluirán en todos o casi todos los productos de la línea, y que se llevan a cabo mediante la construcción de componentes.
- En el nivel de **ingeniería de producto** se analizan, diseñan, implementan y prueban los productos concretos, incorporando a cada uno de ellos las características que les correspondan y que procedan del nivel de ingeniería de dominio.

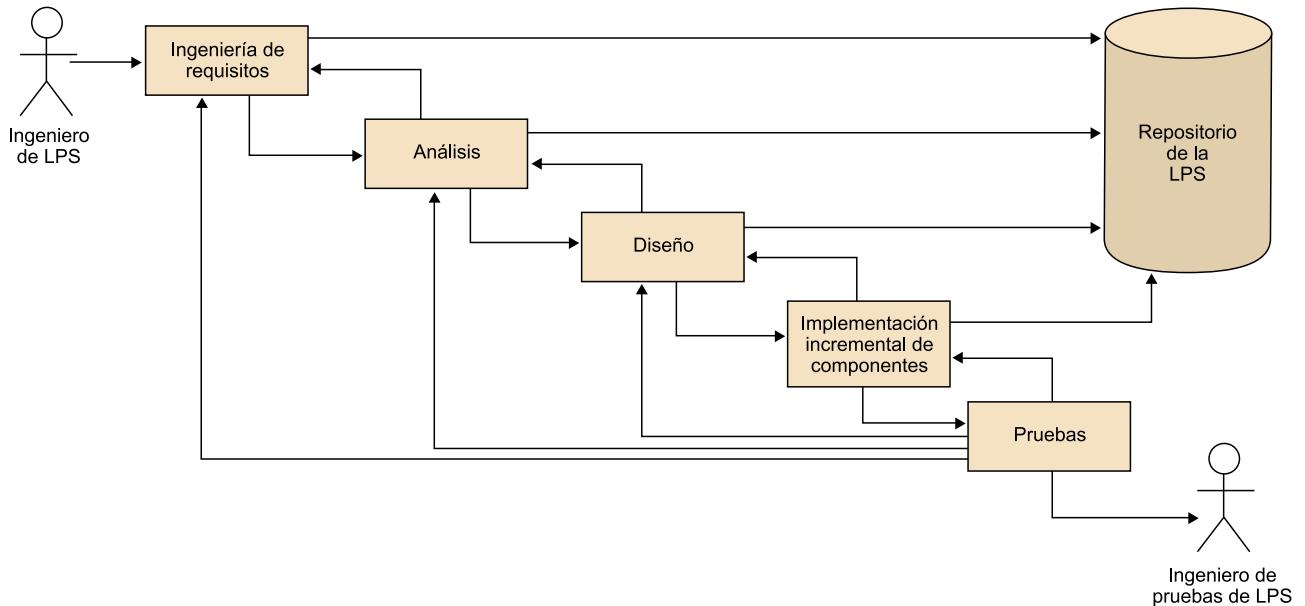
4.4.1. Ingeniería de dominio

La figura siguiente muestra las etapas del proceso de desarrollo a nivel de ingeniería de dominio según la propuesta de Gomaa (2005).

Cada etapa alimenta el repositorio de la LPS. Como ilustran las flechas, la ejecución de las tareas de una etapa puede suponer la revisión de la etapa anterior. La etapa final, de pruebas, puede suponer la introducción de cambios en los productos de cualquiera de las etapas anteriores.

Consulta recomendada

H. Gomaa (2005). *Designing Software Product Lines with UML. From use cases to pattern-based software architectures*. Addison-Wesley.



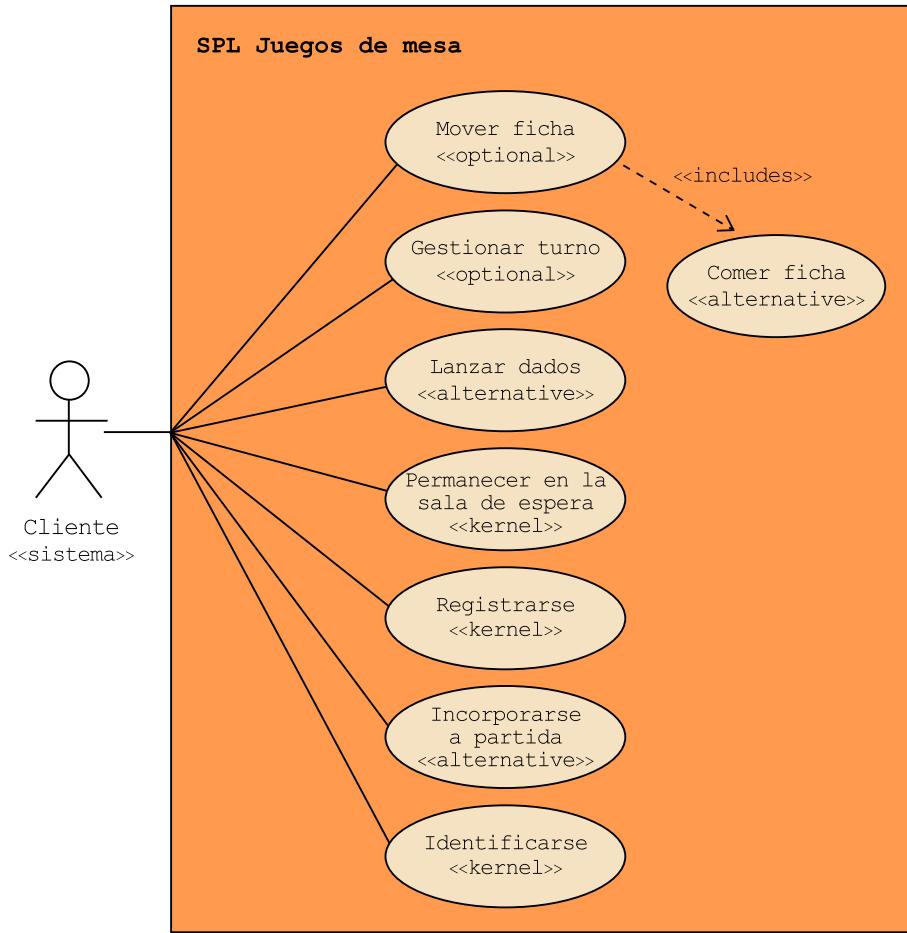
Ingeniería de requisitos

Se desarrolla un modelo de casos de uso y un *feature model* (modelo de características). En LPS, no todos los actores ni todos los casos de uso son necesarios para todos los productos. Por ello, los casos de uso se identifican y estereotipan de acuerdo a si son:

Ved también

Ved la asignatura *Ingeniería de requisitos* del grado de Ingeniería Informática.

- *Kernel*, que son aquellos casos de uso que deberán estar presentes en todos los productos de la línea y cuya implementación es fija.
- *Optional*, que estarán presentes en algunos productos, pero no en todos. Su implementación es también fija: es decir, los productos que incorporan un caso de uso *optional* poseen la misma implementación del caso de uso.
- *Alternative*, que son aquellos casos de uso que están presentes solo en algunos productos y que, además, pueden tener diferentes implementaciones en función del producto. En estos casos de uso se pueden identificar puntos de variación (*variation points*), que son aquellas ubicaciones del caso de uso en las que una situación se gestiona de manera diferente según el producto.



En el *feature model*, por otra parte, se recogen las características (*features*) de la LPS y se representan las dependencias entre ellas.

Una *feature*, por tanto, es un requisito o característica ofrecida por, al menos, uno de los productos de la línea.

Se utilizan seis estereotipos para las *features*:

- *Common feature*, que son aquellas *features* presentes en todos los productos de la línea.
- *Optional feature*, que son características opcionales, que no estarán presentes en todos los productos.
- *Alternative feature*, que son *features* que se presentan en grupos y que son mutuamente excluyentes (es decir, un producto puede ofrecer solo una de las *features* incluidas en el grupo).
- *Parameterized feature*, que es una *feature* que define un parámetro cuyo valor debe definirse en la configuración del sistema. Una característica de este tipo requiere un tipo del parámetro, un rango de valores y, opcional-

mente, un valor por defecto. En el caso de los juegos de mesa, y suponiendo que la interfaz de usuario fuese multiidioma, se podría describir esta característica como en la siguiente figura:

```
<<parameterized feature>>
Idioma
{type=String;
Permitted values=Castellano,
Catalán, Inglés, Suajili;
Default value=Suajili}
```

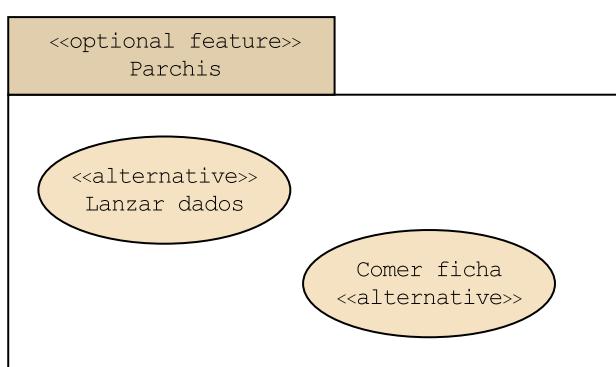
- *Prerequisite features*, que son aquellas *features* necesarias para otras *features*. No es preciso indicar que las *common features* son prerequisitos porque ya lo son por defecto, pero sí deben anotarse las opcionales o alternativas que sean prerequisito de otras. A continuación mostramos la figura anterior, pero extendida con la necesidad de disponer de un diccionario:

```
<<parameterized feature>>
Idioma
{type=String;
Permitted values=Castellano,
Catalán, Inglés, Suajili;
Default value=Suajili;
Prerequisite=Diccionario}
```

- *Mutually inclusive features*, que se utiliza para representar que dos *features* se necesitan mutuamente. En la siguiente figura se representa que el juego del parchís necesita, en efecto, un tablero de parchís.

```
Parchis
<<optional feature>>
{mutually includes=TableroDeParchis}
```

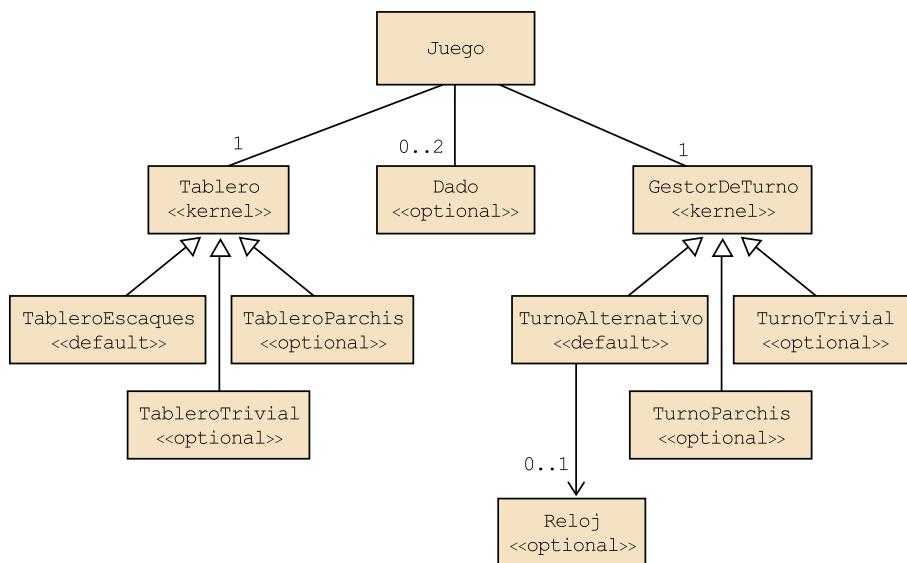
Obviamente, una *feature* puede agrupar varios requisitos funcionales. Para representar los casos de uso incluidos en una *feature* se pueden utilizar paquetes de UML:



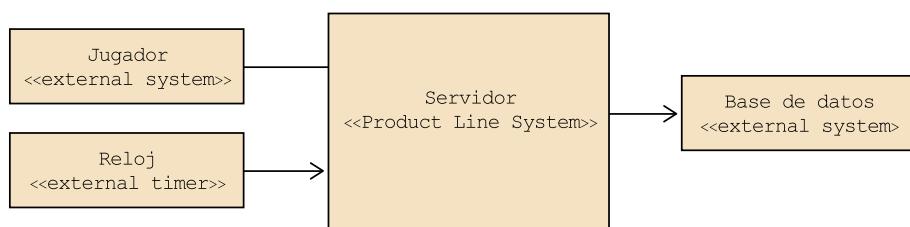
Análisis

De esta etapa se obtienen al menos tres productos:

- Un **modelo de clases de análisis**, en el que cada clase se estereotipa según sea *kernel*, *optional* o *variant*. Puesto que puede haber herencia de clases (incluso en clases *kernel*), alguna de las subclases puede estereotiparse como *default* si va a ser la versión por defecto que se utilice en los productos. En la siguiente figura se representa parte de la estructura de un juego de mesa: un juego tiene un tablero, quizás uno o dos dados y un gestor de turnos. El tipo de tablero por defecto es el formado por escaques, que se utiliza para jugar al ajedrez; el gestor de turno por defecto es el adecuado para estos dos juegos, que pasa el turno alternativamente de un jugador a otro y que puede, como se observa, disponer de un reloj.



Se puede crear, además, un diagrama de clases de contexto, en el que se captura la variabilidad en los límites del sistema, teniendo en cuenta los actores que pueden afectarlo y que serán dispositivos externos de entrada, dispositivos externos de salida o dispositivos externos de entrada/salida. En la práctica, estos sistemas externos (típicamente actores) pueden ser personas, dispositivos, otros sistemas o relojes (*timers*) que, periódicamente, envían una señal al sistema (por ejemplo, hacer una copia de seguridad). Para representar las relaciones de estos actores con el sistema, este puede representarse con un símbolo de clase (un recuadro) estereotipado como «*Product Line System*».



El modelo de clases de contexto proporciona una vista del sistema de muy alto nivel. El desarrollo puede continuar identificando, para el Product Line System, las clases *boundary*, *entity*, *controllers*, etcétera.

- Un **modelo dinámico del sistema**, que estará formado por diagramas de colaboración entre objetos y, para aquellos objetos con un comportamiento dinámico significativo, también por máquinas de estado. En general se seguirá el principio de *kernel first*: es decir, modelar primero las interacciones y el comportamiento de las clases *kernel*, y seguir después con las clases *optional* y *variant*.
- Una **especificación de las clases** que intervienen en cada *feature*.

Diseño

En esta fase se diseña la arquitectura de la línea, teniendo en cuenta la próxima implementación de los componentes. El modelo de análisis, que representa el dominio del problema, se traduce a un modelo de diseño, que se corresponde con el dominio de la solución. Las clases de diseño se asocian a componentes, que se implementarán en la fase siguiente.

Los puntos de variación, que se utilizan en la descripción de los casos de uso para representar en qué lugares puede variar el procesamiento, se trasladan a las clases en esta fase de diseño. Para ello, se utilizan los estereotipos que se muestran en el árbol de herencia que aparece más abajo y que se explican a continuación. En todos los casos, el sufijo *vp* denota que se trata de un punto de variación (*variation point*):

1) Kernel: clase que se incluye sin cambios en todos los productos.

- *Kernel-param-vp* (clase kernel parametrizada): clase kernel con un parámetro cuyo valor se establece en tiempo de configuración.
- *Kernel-abstract-vp* (clase kernel abstracta): clase abstracta incluida en cada producto. Esencialmente, representa una interfaz de comunicación para sus subclases.
- *Kernel-vp* (clase kernel concreta): representa una clase concreta que se redefine mediante especializaciones y que, a diferencia de la anterior, puede ser instanciada.

2) Optional: clase que se incluye en algunos productos de la LPS. Si se usa, se utiliza sin cambios.

- *Optional-param-vp* (clase opcional parametrizada): representa una clase parametrizada que estará presente en algunos productos de la LPS.

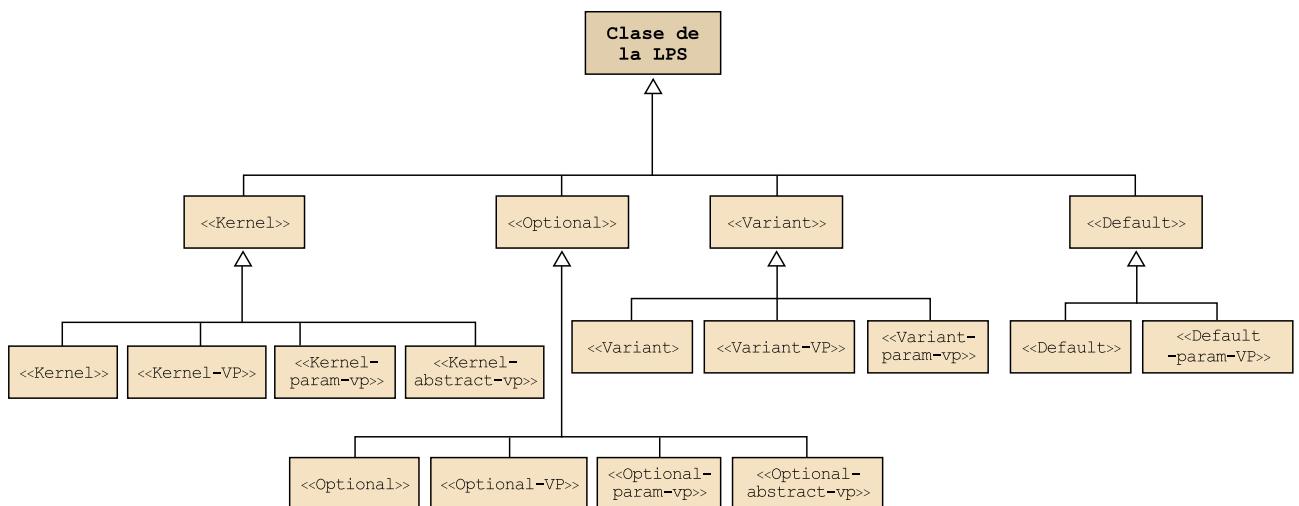
- *Optional-abstract-vp* (clase opcional abstracta): es una clase opcional no instanciable, y que proporciona un medio de comunicación con sus subclases.
- *Optional-vp* (clase opcional concreta): clase opcional instanciable y que, al igual que sucede con las estereotipadas con *kernel-vp*, se redefine mediante especializaciones.

3) Variant: clase contenida en un conjunto de clases similares, algunas de cuyas características son idénticas.

- *Variant-param-vp* (clase variante parametrizada): representa una clase variante, cuyo comportamiento se establece en función de un parámetro de configuración.
- *Variant-vp* (clase variante concreta): se trata de una clase variante instanciable que ofrece una interfaz de comunicación para sus especializaciones.

4) Default: clase variante por defecto, incluida en algunos de los productos de la línea.

- *Default-param-vp* (clase por defecto parametrizable): se trata de la clase por defecto de entre las clases parametrizables por defecto.



Implementación incremental de componentes

En esta fase se elige un conjunto de casos de uso y se implementan los componentes que participan en ellos. Se comienza por la implementación de los componentes incluidos en casos de uso kernel, seguidos por los optional y los variant, según la secuencia que se haya determinado durante el modelado dinámico. Además, se hacen pruebas unitarias de cada componente.

Pruebas

El ingeniero de pruebas hace tanto pruebas de integración como funcionales, poniendo principalmente su atención en los casos de uso kernel. Si es posible, se prueban también los casos de uso optional y variant, aunque estas pruebas deban quizás posponerse hasta que se disponga de productos implementados.

4.4.2. Ingeniería del software para líneas de producto basada en UML

PLUS es una metodología para el desarrollo de LPS propuesta por Gomaa que se apoya en UML y que se basa en el proceso unificado de desarrollo de software. Al igual que este, consta de cuatro fases.

Inicio

De la fase de inicio se obtienen los siguientes artefactos:

- Estudio de viabilidad, alcance, tamaño, funcionalidades, identificación de *commonalities* y *variabilities*, y número estimado de productos.
- Identificación de casos de uso para cada producto potencial.
- Identificación de los casos de uso kernel.
- Diagrama de clases de contexto.
- *Feature model* inicial, indicando cuáles son kernel, cuáles optional y cuáles variant.

Elaboración

Esta fase se lleva a cabo en dos iteraciones:

- En la primera (a la que Gomaa llama *kernel first*: el kernel, primero) se revisa y desarrolla el modelo de casos de uso con más detalle. Se determinan los casos de uso kernel, optional y variant y se hace un análisis más en profundidad de las *features*. El *feature model* obtenido ahora será el principal elemento que guíe en la identificación de los requisitos comunes y variables. Los elementos del kernel se modelan mediante diagramas de análisis y diseño, construyéndose un esbozo inicial de la arquitectura del sistema.
- En la segunda (denominada *product line evolution*, evolución de la línea de productos) se planifica la evolución de la línea considerando los casos de uso optional y variant, las *features* y los puntos de variación. Se hacen diagramas de interacción para los casos de uso optional y variant, y máquinas de estado para las clases optional y variant que tienen una dependencia fuerte del estado. El diseño arquitectónico del sistema, esbozado en la ite-

ración anterior, se extiende ahora para incluir los componentes optional y variant.

Construcción

En las diferentes iteraciones de que puede constar esta fase se construyen los componentes que conforman el kernel. Para ello, se procede realizando el diseño detallado, la codificación y la prueba unitaria de cada componente; también se hacen pruebas de integración de los componentes del kernel.

Transición

Al llegar a esta fase se dispone de una versión mínima ejecutable de la LPS. Se realizan pruebas funcionales y de integración de los componentes del kernel de que se dispone. El sistema, en su estado actual, se puede entregar a los ingenieros de producto.

Iteraciones adicionales

Puede haber iteraciones adicionales en las que se desarrollen componentes optional y variant. El jefe de proyecto debe decidir si estos componentes se implementan durante la ingeniería de dominio o, por el contrario, se dejan para más adelante, para el nivel de ingeniería de producto.

4.5. Programación generativa

La programación generativa aparece en 1998 como una propuesta de Krzysztof Czarnecki para resolver algunos de los problemas que presenta el desarrollo tradicional de software en cuanto a reutilización, adaptabilidad, gestión de la complejidad creciente y rendimiento.

De manera parecida al desarrollo de software en líneas de producto, la programación generativa se dedica a la construcción, con un enfoque de reutilización proactivo, de familias de productos relacionados con algún dominio particular, en lugar de productos independientes.

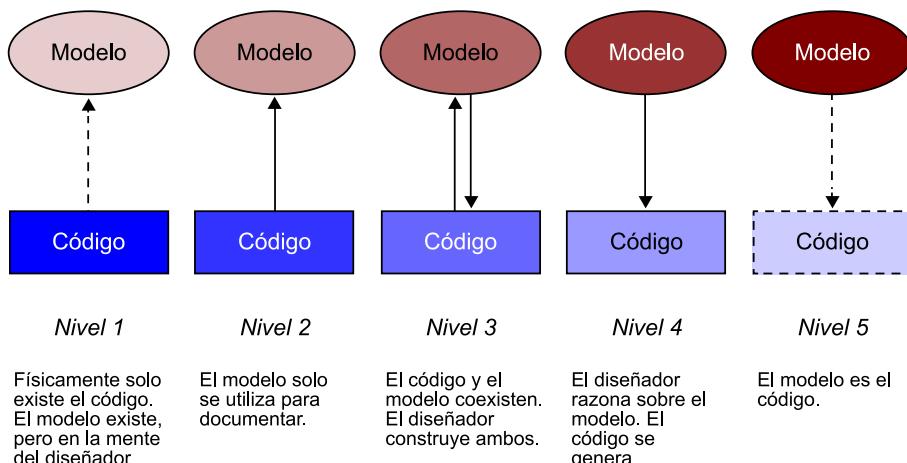
Consultas recomendadas

En Internet se localiza fácilmente el documento PDF con la tesis doctoral de Czarnecki (*Generative Programming. Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*).

Poco después (en 2000), Czarnecki publicó junto a Eissenecker el libro *Generative Programming: Methods, Tools and Applications*, con la editorial Addison-Wesley.

La programación generativa trata del diseño e implementación de software reutilizable para generar familias de sistemas de un dominio, y no sistemas independientes.

Uno de los objetivos de la programación generativa es la disminución del salto conceptual entre los conceptos del dominio y el código que los implementa. Idealmente, y tal y como se muestra en el nivel 5 de la siguiente figura (adaptada de la tesis de máster de César Cuevas), se pretende que “el modelo sea el código”.



Para ello, este paradigma pretende la generación automática de programas a partir de especificaciones dadas a muy alto nivel. Así, se incrementa la eficiencia en la producción de software mediante la escritura y construcción de elementos abstractos que, adecuadamente parametrizados y procesados, provean una implementación real de la solución del problema. Para conseguir que tales especificaciones “a tan alto nivel” sean directamente procesables es para lo que se trabaja a nivel de dominios de aplicación y, por ello, la programación generativa se apoya fuertemente en el uso de lenguajes de dominio específico (DSL).

Con el fin de facilitar la reutilización, otra de las características de la programación generativa es la separación de funcionalidades²⁹ en módulos o elementos independientes, de manera que haya tan poco solapamiento como sea posible. Para ello se utiliza intensivamente el modelado de características (*feature modeling*) y técnicas de programación orientada a aspectos (AOP, *aspect-oriented programming*). Con estas técnicas se pretende obtener un conjunto de elementos software de funcionalidad autocontenido, ensamblables y reutilizables.

Consulta recomendada

La tesis de máster de César Cuevas Cuesta (Herramientas para el despliegue de aplicaciones basadas en componentes), de 2009, dedica un apartado a la programación generativa. Se encuentra disponible (29 de febrero del 2012) en la web de la Universidad de Cantabria.

Ved también

Repasad, si es preciso, el apartado “Lenguajes de dominio específico”.

⁽²⁹⁾ La separación de funcionalidades que citamos es lo que Dijkstra llamaba *separation of concerns* [E. W. Dijkstra (1982). “On the role of scientific thought”. *Selected writings on Computing: A Personal Perspective* (págs. 60-66). Springer-Verlag], término cuya traducción no nos resulta fácil sin perder semántica o sin utilizar una expresión demasiado larga. A veces se traduce como ‘separación de incumbencias’.

Sin embargo, dentro incluso de un mismo dominio, pueden existir diferentes particularidades que dificulten la generación del producto final. Por esto, la programación generativa utiliza también técnicas de programación genérica, la cual se basa en el uso intensivo de mecanismos genéricos de programación, como las plantillas de C++ (*templates*) o Java (*generics*).

Nota

DSL: *domain specific language*
FM: *feature modeling*
AOP: *aspect-oriented programming*

No hay que olvidar que, idealmente, el modelo debe ser transformable directamente al código ejecutable del producto final. La programación generativa aprovecha el estado actual de la tecnología combinando las técnicas que se acaban de indicar (DSL, FM, AOP, programación genérica).

Ved también

Encontraréis más información sobre la programación genérica más adelante en el apartado "Tecnologías".

4.5.1. Ingeniería de dominio

Desde el punto de vista de la programación generativa, un dominio es un conjunto de conceptos y términos relacionados con un área de conocimiento determinada (por ejemplo: hospitales, gasolineras, universidades) y que entienden las personas involucradas en dicha área (médicos, gasolineros, profesores y alumnos). Además, la identificación y estructuración de los conceptos que pertenecen al dominio debe orientarse hacia la construcción de sistemas de software para esa área y deben maximizar la satisfacción de los requisitos de las personas involucradas.

Con el fin de modelar adecuadamente el dominio de una familia de sistema se necesita aplicar de manera rigurosa técnicas de ingeniería de dominio, que es la actividad en la que, de forma similar a lo que se hace en la ingeniería de dominio en LPS, se recoge y organiza y la experiencia obtenida en la construcción de sistemas o partes de sistemas de un dominio particular. Esta experiencia se conserva en forma de activos reutilizables (*reusable assets*). Obviamente, estos activos se deben almacenar de manera que puedan reutilizarse cuando se construyan nuevos sistemas: tal infraestructura de almacenamiento debe ofrecer los medios para recuperar, cualificar, distribuir, adaptar y ensamblar los activos reutilizables guardados.

Nota

En el apartado "Componentes" ya hablábamos de su cuatificación, adaptación y ensamblado. La práctica en este contexto es diferente, pero la idea es exactamente la misma.

La ingeniería de dominio tiene tres fases principales dedicadas al análisis del dominio (en la que se define un conjunto de requisitos reutilizables para los sistemas de ese dominio), al diseño del dominio (donde se determina la arquitectura común de los sistemas de ese dominio) y a la implementación del dominio (en la que se implementan tanto los activos reutilizables en forma de componentes, DSL, generadores, etc., como la infraestructura de almacenamiento).

Nota

Czarnecki explica que añade explícitamente el postfijo "de dominio" a las fases de análisis, diseño e implementación para enfatizar precisamente su orientación a familias de productos y no a productos independientes o aislados.

Análisis del dominio

El “conjunto de requisitos reutilizables” que se ha citado en el párrafo anterior se obtiene a partir del estudio del dominio de que se trate. Así, las fuentes de información pueden ser muy variadas: sistemas ya existentes, entrevistas con expertos, libros, prototipos, experimentación o requisitos ya conocidos de sistemas futuros.

De esta etapa se obtiene un modelo de dominio, en el que se explicitan las propiedades comunes y variables de los sistemas del dominio, así como las dependencias entre las propiedades variables.

Algunos contenidos habituales del modelo de dominio son:

Véase también

En la sección dedicada a las líneas de producto software ya hicimos una discusión extensa sobre las *commonalities*, las *variabilities* y los *feature models*.

- **Definición:** se describe el alcance del dominio, se dan ejemplos y contrarejemplos de sistemas que están dentro y fuera de él, y se indican reglas genéricas de inclusión y exclusión.
- **Vocabulario:** define los términos más importantes del dominio de que se trate.
- **Modelo de conceptos:** se describen los conceptos del dominio utilizando algún formalismo apropiado (diagramas de objetos, de interacción, entidad-interrelación...).
- **Modelo de características (feature model):** describe los requisitos reutilizables y configurables con un nivel de granularidad adecuado, en forma de características (*features*).

Diseño del dominio

Ya que se desarrollan familias de sistemas y no sistemas independientes, la arquitectura que se obtenga de esta etapa de diseño debe ser suficientemente flexible como para soportar diversos sistemas y la evolución de la familia de sistemas.

Implementación del dominio

En esta fase se implementan los componentes, generadores para composición automática de componentes y la infraestructura de reutilización que permita la recuperación, cualificación, etc. de componentes.

4.5.2. Tecnologías

Además de los DSL y los *feature models*, que ya se han tratado anteriormente, en programación generativa se utilizan conceptos de programación orientada a aspectos y de programación genérica.

Programación orientada a aspectos

La programación orientada a aspectos surge como una evolución de la orientación a objetos que pretende incrementar la modularidad y la reutilización: en los sistemas software existen muchas funcionalidades “transversales”, en el sentido de que se utilizan en otras múltiples funcionalidades. Antes de realizar una operación sobre una base de datos, por ejemplo, quizás sea preciso realizar una comprobación sobre el nivel de permisos del usuario que lanza la operación. En un desarrollo orientado a objetos tradicional, el código correspondiente a esta comprobación se insertaría antes de cada llamada a una operación de persistencia. La modificación de la política de seguridad u otro tipo de cambio (quizás incluso pequeño) puede suponer la modificación de este código en todos aquellos lugares en los que aparece.

Este tipo de funcionalidades transversales es lo que se conoce como “aspecto”. Cada aspecto se implementa de forma modular y separada del resto en lo que se llama un *advice* (literalmente: aviso). El lugar del programa en el que se llama a la funcionalidad dada por un *advice* (es decir, a la implementación del aspecto) es lo que se llama un “punto de enlace” (*join point*). Los puntos de enlace se anotan en “puntos de corte” (*pointcuts*), que describen mediante patrones de nombres, expresiones regulares y una serie de palabras reservadas cuándo debe llamarse al *advice*.

El entretejido (*weaving*) es el proceso de combinar los aspectos con la funcionalidad a la que están asociados, de lo que se encarga el *weaver*. En el **entretejido estático**, la combinación del código de la aplicación y de los aspectos se lleva a cabo en tiempo de compilación. En el **dinámico**, la combinación se realiza en tiempo de ejecución.

La siguiente figura resume un ejemplo dado en el libro *Aspect-Oriented Programming with AspectJ*: la función *main*, incluida en la clase A, construye una instancia de A y llama a *a(5)*. *a(5)* devuelve el resultado de *b(5)* que, a su vez, devuelve el valor que recibe como parámetro (5, en este caso). Entretanto, la llamada *a b* ejecuta también una llamada al método *c* de B.

En la parte inferior aparece un aspecto con tres puntos de corte y sus tres correspondientes *advices*:

- La expresión que hay a la derecha del nombre del primero (llamado `int_A_a_int()`) enlaza ese punto de corte con la llamada al método cuya firma es `int A.a(int)` (es decir, el método `a()` de la clase A). La palabra reservada `call` es un “descriptor de punto de corte”. Más abajo se encuentra el *advice* que corresponde a este punto de corte: con la palabra reservada `before()` se denota que el *advice* se debe ejecutar antes de llamar a ese método concreto.

Consulta recomendada

I. Kiselev (2003). *Aspect-Oriented Programming with AspectJ*. Editorial SAMS.

Nota

La mayoría de los lenguajes orientados a aspectos son extensiones de otros “lenguajes base”:

- AspectJ (de Java)
- AspectC (de C)
- AspectC++ (de C++)

- El segundo *advice* (llamado `int_A_all_int()`) se ejecuta después (*after*) de las llamadas (también se utiliza del descriptor de punto de corte `call`) a todos los métodos de `A` que tomen un parámetro de tipo `int` y que devuelvan un `int`, lo cual se está representando por la expresión `int A.*(int)`, que contiene el carácter comodín (*) de las expresiones regulares.
- El tercer punto de corte (`all_all_c_all()`) afecta a todos los métodos que se llamen `c`, independientemente del tipo que devuelvan (primer comodín) y de la clase en la que están contenidos (segundo), pero que tomen un solo parámetro (tercer carácter comodín, situado entre los paréntesis que corresponden a los parámetros). La palabra reservada (*around*, en este caso) sirve para sustituir por completo las llamadas al punto de enlace capturado. No obstante, con la llamada a `proceed` se consigue llamar al punto de enlace original, si se desea.

<pre> public class A { int a(int x) { return b(x); } int b(int x) { c("x"); return x; } String c(String x) { (new B()).c(3.14); return x; } public static void main(String args[]) { A t = new A(); t.a(5); } } </pre>	<pre> public class B { void c(double x) { ... } } </pre>
<pre> public aspect Showcase { pointcut int_A_a_int(): call(int A.a(int)); pointcut int_A_all_int(): call(int A.*(int)); pointcut all_all_c_all(): call(* *.c(*)); before(): int_A_a_int() { System.out.println("Before: " + thisJoinPoint); } after(): int_A_all_int() all_all_c_all() { System.out.println("After: " + thisJoinPoint); } Object around(): all_all_c_all() { System.out.println("Start around: " + thisJoinPoint); Object o = proceed(); System.out.println("End around: " + thisJoinPoint); return o; } } </pre>	

Aspectos, *features* y programación generativa

Con no mucho margen de error, podemos asimilar una *feature* a un aspecto. Por ello, el *feature model* que construimos en la ingeniería de dominio nos sirve también como modelo para la implementación de los aspectos; las relaciones entre las *features* pueden,

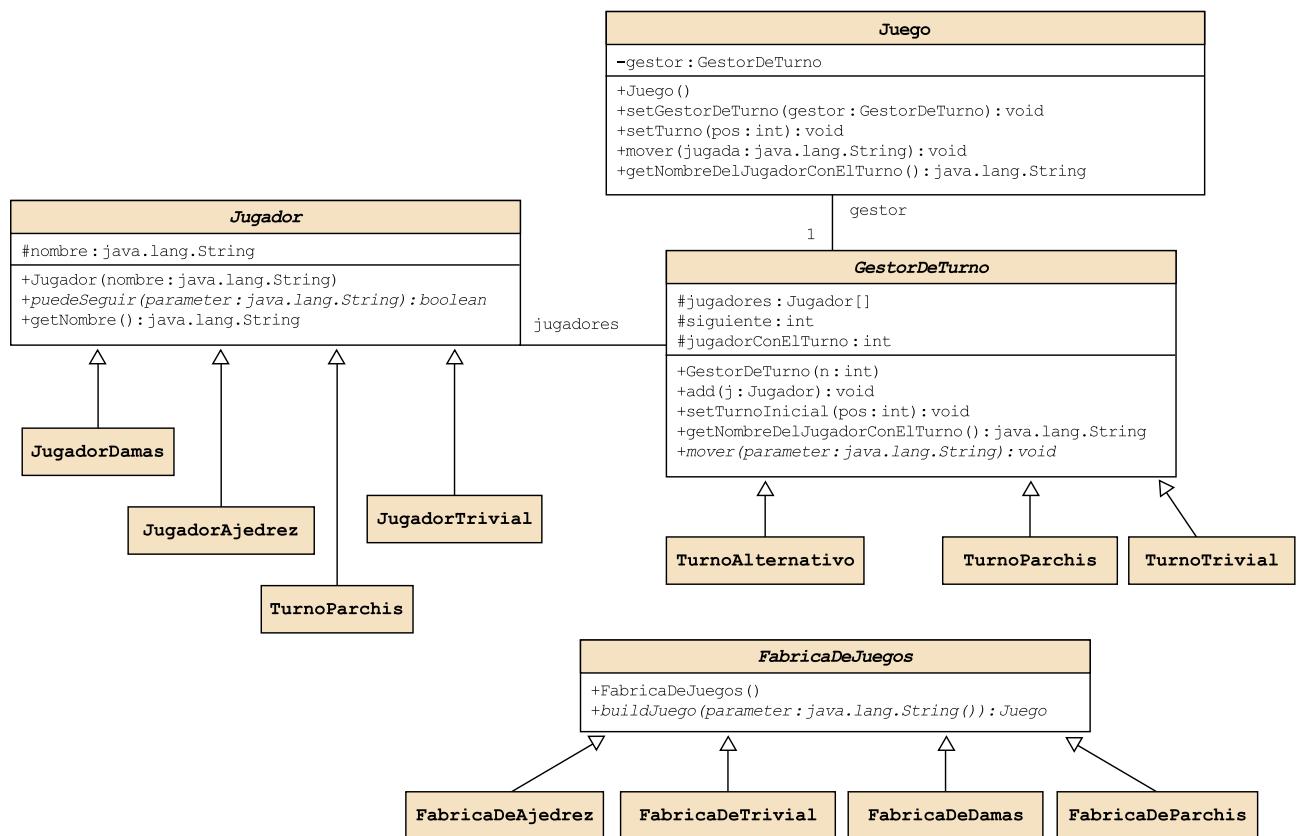
igualmente, asimilarse a los puntos de enlace entre los aspectos y las funcionalidades del sistema.

Al componer los aspectos, debemos intentar que el aspecto sea tan independiente de la funcionalidad como sea posible. Así conseguiremos aspectos reutilizables. Para el efecto recíproco (que las funcionalidades también lo sean), es conveniente utilizar estándares adecuados de codificación en cuanto a nombrado de clases, operaciones, campos, variables, etcétera.

Programación genérica

La programación genérica se dedica más al desarrollo de algoritmos que al diseño de las estructuras de datos sobre las que aquellos operan. Ya en el apartado dedicado a la reutilización en diseño de software orientado a objetos, se recordó la noción de las *templates* de C++ y los *generics* de Java: mediante estas construcciones, podemos crear clases que actúen sobre tipos de datos arbitrarios, que se declaran en tiempo de escritura de código, pero que se instancian en tiempo de compilación.

La siguiente figura muestra parte de un posible diseño del sistema para jugar a juegos de mesa que veíamos en la sección dedicada a líneas de producto software: se dispone de una clase abstracta *Jugador* y de otra *GestorDeTurno*, que tienen varias especializaciones en función del tipo de juego que se quiera construir, lo cual se consigue a partir de la *FabricaDeJuegos* (estructura que se corresponde al patrón fábrica abstracta). Existe además una clase *Juego* que actúa de *wrapper*.



El diseño mostrado arriba aprovecha la herencia, pero no utiliza en absoluto la programación genérica. De hecho, el código de las clases *GestorDeTurno* y *TurnoAlternativo* (la implementación utilizada para jugar al ajedrez y a las damas) nos lo confirma:

```
package dominio.sinGenerics.turnos;

import dominio.sinGenerics.jugadores.Jugador;

public abstract class GestorDeTurno {
    protected Jugador[] jugadores;
    protected int siguiente, jugadorConElTurno;

    public GestorDeTurno(int n) {
        this.jugadores=new Jugador[n];
        this.siguiente=0;
        this.jugadorConElTurno=-1;
    }

    public void add(Jugador j) {
        if (this.siguiente<this.jugadores.length)
            this.jugadores[this.siguiente]=j;
    }

    public void setTurnoInicial(int pos) {
        this.jugadorConElTurno=pos;
    }

    public String getNombreDelJugadorConElTurno() {
        return this.jugadores[this.jugadorConElTurno].getNombre();
    }

    public abstract void mover(String jugada);
}

package dominio.sinGenerics.turnos;

public class TurnoAlternativo extends GestorDeTurno {

    public TurnoAlternativo(int n) {
        super(n);
    }

    @Override
    public void mover(String jugada) {
        this.jugadorConElTurno=
            (this.jugadorConElTurno+1) % 2;
    }
}
```

El siguiente trozo de código, por otro lado, crea una partida de ajedrez mediante la llamada al método *buildJuego* de la fábrica concreta *FabricaDeAjedrez*:

```
FabricaDeJuegos f=new FabricaDeAjedrez();
String[] nombres={"Capablanca", "Bobby Fischer"};
Juego j=f.buildJuego(nombres);
```

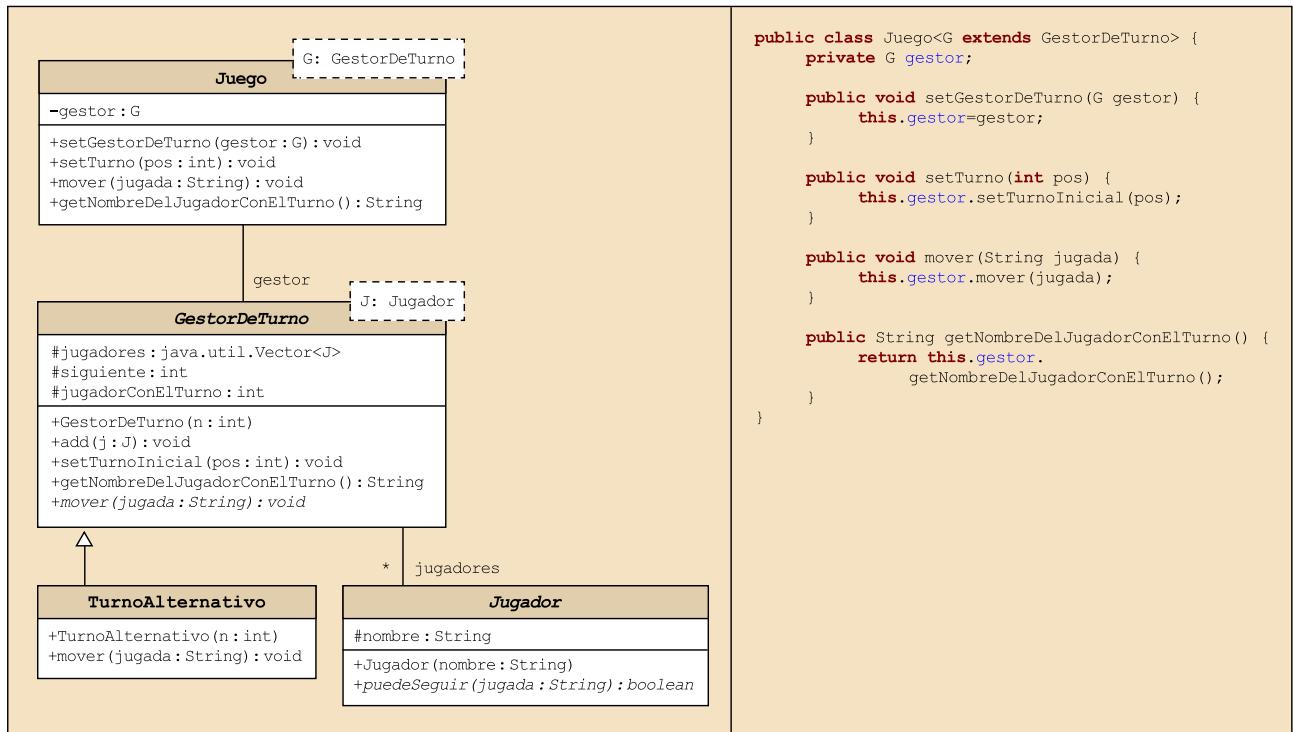
Para darle un enfoque genérico al diseño del mismo sistema, podemos pensar en las diferentes configuraciones de juegos que podemos crear. De este modo, en lugar de tener una clase *Juego* que conoce a un *GestorDeTurno* abstracto que, a su vez, conoce a un *Jugador* también abstracto (estos dos últimos objetos se instanciarán a los subtipos adecuados, de modo que la configuración esté formada por objetos compatibles), y en lugar de delegar la responsabilidad de crear la configuración a cada fábrica concreta, dejaremos el código preparado con tipos genéricos para que la configuración se cree en tiempo de compilación.

Con un enfoque genérico, diremos que el juego contiene un *GestorDeTurno*, y que este tiene, a su vez, un *Jugador*. El *GestorDeTurno* conocido por el *Juego* se pasa como parámetro y se instancia en tiempo de compilación. Igualmente, el *Jugador* al que conoce el *GestorDeTurno* se le pasa a este como parámetro y también se instancia en tiempo de compilación. Así, el resultado de la compilación es una configuración concreta. El siguiente trozo de código, por ejem-

plo, monta una configuración formada por un *TurnoAlternativo* y dos jugadores de ajedrez: obsérvese la instanciación que se hace del *Juego* en la primera línea, indicando que se desea que se monte con un *TurnoAlternativo*:

```
public static void main(String[] args) {
    Juego<TurnoAlternativo> juego=new Juego<TurnoAlternativo>();
    String[] nombresDeJugadores={"Capablanca", "Bobby Fischer"};
    for (String s : nombresDeJugadores) {
        JugadorAjedrez j=new JugadorAjedrez(s);
        juego.gestor.add(j);
    }
    ...
}
```

Visualmente, el diseño del sistema es el que se muestra en el lado izquierdo de la siguiente figura; en el lado derecho aparece el código correspondiente a la clase *juego*: obsérvese que contiene un objeto gestor del tipo genérico G, que se declara como parámetro en la cabecera de la clase.



Como se ha dicho, la configuración concreta se monta en tiempo de compilación. El diagrama de clases de la figura anterior procede de haber aplicado ingeniería inversa al código que se viene utilizando como ejemplo: los tipos parametrizados aparecen en pequeñas cajitas con el borde punteado.

4.6. Fábricas de software

De acuerdo con Piattini y Garzás (2010), una fábrica de software requiere una serie de buenas prácticas que organicen el trabajo de “una forma determinada, con una considerable especialización, así como una formalización y estandarización de los procesos”. Entre estas buenas prácticas, los autores citan:

- La definición de la estrategia de fabricación, tanto en desarrollo tradicional, como en generación automática de código, como en reutilización.
- La implantación de un modelo de procesos y de un modelo de calidad de software. La certificación de las fábricas de software en algún nivel de madurez es, muchas veces, un requisito imprescindible para el desarrollo de proyectos con grandes empresas o con la Administración pública.
- La implantación de una metodología de fabricación.
- Integración continua y gestión de configuración.
- Control de calidad exhaustivo, periódico y automático.
- Diseño basado en el conocimiento, que evite la dependencia excesiva de determinadas personas.
- Construir el software tomando los casos de uso como pieza esencial del desarrollo. El desarrollo basado en casos de uso facilita la trazabilidad desde los requisitos al código y permite estimar costes y esfuerzos, monitorizar el avance del proyecto, planificar y ejecutar las pruebas funcionales, guiar en el desarrollo de la interfaz de usuario y también, organizar y redactar el manual del usuario de acuerdo con ellos.
- Utilizar ciclos de desarrollo evolutivos e iterativos, y no en cascada.
- Estimar los costes utilizando puntos función adaptados, para lo que hay varias propuestas en la literatura que dan buenos resultados.
- Guiar las acciones por su valor, evaluando el retorno de la inversión de cada una.

Como se observa, hay una diferencia de actitud con respecto a la que Griss apuntaba en 1993, y que ya citamos al hablar de los costes de desarrollar software reutilizable. Las fábricas de software han dejado atrás esa consideración de la reutilización como un gasto, pasando a considerarlo una inversión.

La proactividad en la reutilización

Griss cita otros problemas de la proactividad en la reutilización, muchos de los cuales se van erradicando progresivamente. Algunos son los siguientes:

- El síndrome NIH (*not-invented here*, no inventado aquí), que puede parecer un atentado a la creatividad de los desarrolladores.
- La falta de técnicas de gestión adecuadas para proyectos de desarrollo de software reutilizable.
- La evaluación individual del trabajo de los desarrolladores se ve dificultada en proyectos de esta naturaleza, en donde pueden no obtenerse beneficios a corto plazo.

Evidentemente, el desarrollo orientado a objetos, el desarrollo de componentes, el enfoque de líneas de producto software, la programación generativa y el resto de temas que hemos tratado en estas páginas entran en esas estrategias de fabricación, metodologías, etcétera que se citan en los puntos anteriores.

Resumen

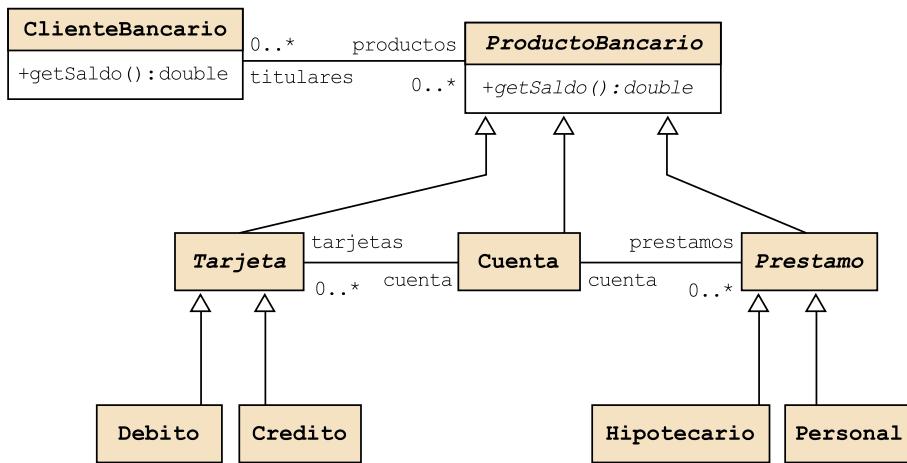
A lo largo de este texto, se han dado unas pinceladas acerca de la evolución histórica en cuanto a reutilización, habiendo enfatizado la importancia que supuso la introducción del paradigma orientado a objetos. Este fue el pilar fundamental sobre el que se desarrollaron otras tecnologías, como las librerías de clases, los *frameworks* y los componentes. A su vez, los componentes han sido la base para la aparición de otros paradigmas como las líneas de producto software (en las que el modelado desempeña un papel fundamental) o la programación generativa (con el uso intensivo de nuevas evoluciones de la orientación a objetos, como la AOP o la genericidad). La reutilización proactiva implica la gestión adecuada de todos los artefactos software que se van construyendo a lo largo del desarrollo, lo que supone la monitorización no solo del código fuente, sino también de los modelos de los cuales procede. En este sentido, también el desarrollo dirigido por modelos ha supuesto (y sigue suponiendo, pues es una materia de un enorme interés en la I+D+i en informática) un avance importantísimo.

Esperamos que este texto haya servido al lector para alcanzar los objetivos que se plantearon en el inicio del módulo: pensamos que la variedad de tecnologías y metodologías presentadas habrán ayudado a la consecución de los tres últimos. Respecto del primero, quizás la lectura de este libro y de las referencias complementarias que se han ido citando, la resolución de los ejercicios planteados y el poso que su estudio habrá ido dejando hayan conseguido finalmente convencer al lector de que, en efecto, debemos tener una mentalidad suficientemente abierta y considerar la reutilización como un activo fundamental de nuestros sistemas software.

Actividades

1. Un determinado tipo de cuenta corriente ofrece operaciones para ingresar dinero (incrementando el saldo), retirar dinero (decrementándolo), siempre que haya saldo suficiente para el importe que se desea retirar) y para transferir un importe a otra cuenta (decrementando el saldo de la cuenta origen en el importe transferido más un 1% de comisión, siempre que en la cuenta origen haya saldo suficiente). Cuando la cuenta se crea, el saldo es cero. Proporcionad una descripción de la cuenta como un tipo abstracto de dato.

2. El siguiente diagrama muestra la estructura de clases de un posible subsistema bancario. El saldo de un cliente se define como la suma de los saldos de todas sus cuentas, menos la suma de los saldos de sus tarjetas de crédito (se excluyen las de débito) y menos la suma de los saldos de sus préstamos. Escribid el código o el pseudocódigo de la operación getSaldo() de ClienteBancario para que devuelva el resultado según la especificación que se ha dado. Observad que quizás sea conveniente efectuar alguna modificación en el diagrama de clases.

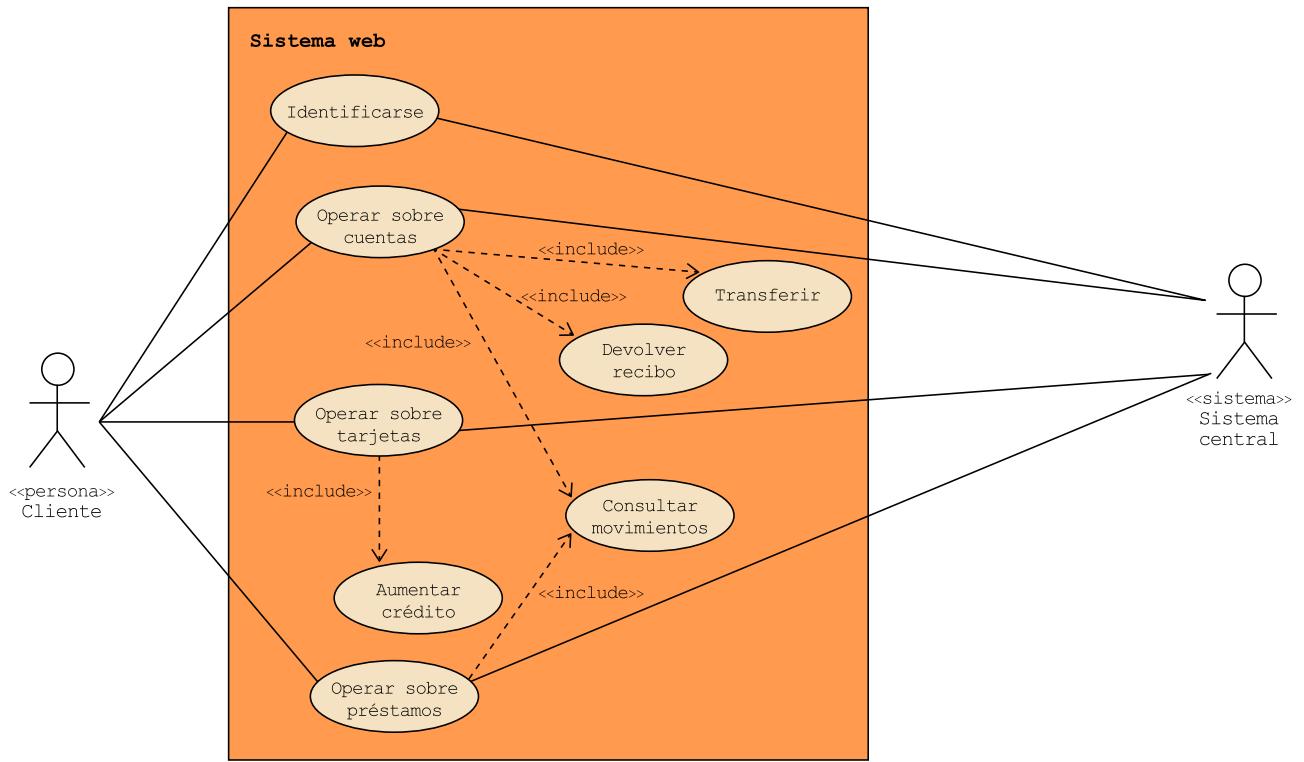


3. El sistema informático de una empresa de ventas realiza todas las noches las siguientes tareas:

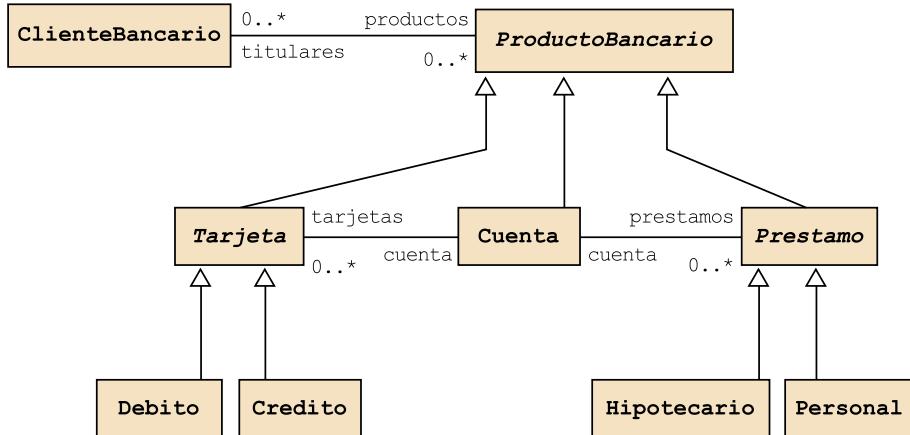
- exporta todos los datos de todas las ventas y nuevos clientes que residen en una base de datos a tres ficheros de texto plano (ventas, líneas de venta y clientes);
- otro proceso importa estos datos y los guarda en otra base de datos diferente;
- otro proceso toma los tres ficheros de la tarea a), los totaliza y los guarda en un fichero de totales.

Se pide que representéis estas tareas con una arquitectura de *pipes and filters*.

4. Un banco desea ofrecer a sus clientes la posibilidad de consultar y manejar la información de sus productos bancarios mediante una aplicación web. Se dispone, por un lado, del diagrama de casos de uso del sistema que se desea desarrollar. Como se observa, todos los casos de uso requieren su interacción con el sistema central del banco, que es el que dispone de toda la información relativa a clientes, cuentas, tarjetas y préstamos:



Por otro lado, se ha construido un diagrama con las clases que realizarán estos casos de uso. Estas clases correrán en el servidor web y, para ejecutar sus operaciones, deberán conectarse al actor sistema central, sistema que también gestiona su persistencia y que, por tanto, no forma parte de este problema.



Se os pide que completéis el diseño de clases anterior para que el sistema tenga una arquitectura multicapa y pueda implementarse utilizando Struts2.

5. A partir del diagrama de casos de uso y del diagrama de clases del ejercicio anterior, se os pide que diseñéis un componente que ofrezca todas las funcionalidades descritas y que sea capaz de conectarse con el sistema central.

6. Modificad los diseños obtenidos en el ejercicio 4 suponiendo que podéis utilizar el componente diseñado en el ejercicio 5.

7. Suponed que el diagrama de clases mostrado en el ejercicio 4 forma parte del sistema de gestión que utilizan los empleados del sistema del banco, de manera que varios empleados pueden estar viendo simultáneamente el estado de una misma instancia. Cuando hay un cambio en alguno de los objetos, se desea que se actualicen las vistas correspondientes a todas las ventanas que están "mirando" a ese objeto. Suponed que se dispone de las siguientes seis ventanas para manipular clientes, cuentas, tarjetas de crédito y débito, y préstamos hipotecarios y personales, y que son estas las ventanas que deben actualizarse cuando hay algún cambio en el objeto de dominio al que están observando.

<<ventana>> VCliente	<<ventana>> VCuenta	<<ventana>> VCredito	<<ventana>> VDebito	<<ventana>> VHipotecario	<<ventana>> VPersonal
---	--	---	--	---	--

Proponed una solución para este problema basada en patrones.

8. La National Oceanic and Atmospheric Administration's de Estados Unidos ofrece un servicio web con varias funciones para conocer la previsión del tiempo en Estados Unidos y Puerto Rico. El documento WSDL con la especificación de la interfaz de acceso se encuentra en: <http://graphical.weather.gov/xml/SOAP>.

Una de las operaciones ofrecidas es NDFDgenByDay(latitud, longitud, diaInicial, numeroDeDías, unidad, formato), que devuelve un documento XML con diversos datos de la previsión del tiempo en la latitud y longitud pasadas como parámetros, a partir del diaInicial y durante numeroDeDías días, en unidades británicas o del sistema internacional, y con las horas en formato de 12 o de 24 horas.

La siguiente llamada, por ejemplo, recupera la información meteorológica de Asheville, en Carolina del Norte (a 35,35°N, 82,33°O), desde el 11 de diciembre de 2011 y durante los 7 días siguientes, en unidades internacionales ("m") y en formato de 24 horas.

```
result = proxy.NDFDgenByDay(35.35M, -82.33M,
    new DateTime(2011, 12, 11), "7", "m", "24 hourly");
```

Se os pide que escribáis el nombre de un cliente que conecte al servicio web y recupere, por ejemplo, la previsión del tiempo en Asheville para los tres próximos días.

9. Se va a desarrollar una herramienta CASE para dibujar y almacenar diagramas de clases. Se desea dotar a esta herramienta de un mecanismo de generación de código para varios lenguajes de programación (Java, C++ y Visual Basic .NET). Se os pide que, utilizando el patrón Abstract Factory, propongáis una solución para generar código a partir de los diagramas de clases.

10. Como se explica en la sección dedicada a MDE, “Otra herramienta muy conocida para transformación de modelos es Atlas, desarrollada por AtlanMod, un grupo de investigación del Instituto Nacional de Investigación en Informática y Automática de la Escuela de Minas de Nantes, en Francia. Atlas incluye el lenguaje de transformación ATL y es conforme a MOF (es decir, define su propio metamodelo), aunque no es extensión de UML”. ¿Podrán los modelos ATL transformarse a modelos UML 2.0 con algún enfoque estándar, como QVT?

11. Se desea desarrollar una herramienta CASE que tendrá, según el precio de venta, diferentes funcionalidades que se ofertarán en tres ediciones: estándar, ejecutiva y empresarial.

La edición estándar permitirá el modelado de diagramas de clase, la generación del esqueleto del código (nombre de la clase o interfaz, campos y cabecera de las operaciones) en lenguaje Java y el modelado de máquinas de estado para las clases definidas en el diagrama, pero no permitirá modelar estados compuestos.

La edición ejecutiva también permitirá el modelado de máquinas de estado con estados compuestos, la generación del código que corresponda a las máquinas de estado y generar código SQL a partir de diagramas de clases. Además de para Java, esta herramienta permite generar código para otros lenguajes.

La empresarial, además de todo lo anterior, incluirá una funcionalidad para generar bases de datos en diversos gestores a partir de diagramas de clases y otra para obtener diagramas de clases a partir de código escrito en los mismos lenguajes para los que es capaz de generar. A vosotros se os pide que modeléis algunos aspectos de esta herramienta siguiendo un enfoque de líneas de producto software.

Ejercicios de autoevaluación

1. Explicad qué significa darle a la reutilización un enfoque “proactivo”.
2. Explicad cómo contribuyen las clases DAO a la reutilización.
3. Indicad en qué se diferencia un caso de uso *optional* de uno *alternative*.

4. Indicad las principales tecnologías utilizadas en la programación generativa.
5. Explicad cómo contribuye el mantenimiento de un diseño arquitectónico multicapa de un sistema si se desea dotar a ese sistema de una interfaz web.

Solucionario

Actividades

1. El enunciado evidencia la existencia explícita de cuatro operaciones sobre este TAD. Además, una operación implícita es la que permite conocer el saldo y que necesitamos para describir los axiomas:

- Ingresar(importe : real)
- Retirar(importe : real)
- Transferir(importe : real, destino : Cuenta)
- Crear una cuenta
- Saldo() : real

Una posible descripción de este TAD es:

a) Funciones

- ingresar: Cuenta x real → Cuenta
- retirar: Cuenta x real → Cuenta
- transferir: Cuenta x real x Cuenta → Cuenta
- crear : Cuenta
- saldo : Cuenta → real

b) Axiomas (sean: x real; c, d Cuenta)

- saldo(crear) = 0
- saldo(crear, ingresar(c, x)) = x
- saldo(retirar(c, x)) = saldo(c) - x
- saldo(transferir(c, x, d)) = saldo(c) - x - 0.01*x

c) Precondiciones

- ingresar(c, x) requiere [x>0]
- retirar(c, x) requiere [x>0 ^ saldo(c)>= x]
- transferir(c, x, d) requiere [x>0 ^ saldo(c)>= 1.01*x]

2. Si el saldo del cliente fuese la suma de los saldos de todos los productos asociados, sin considerar las restricciones indicadas en el enunciado, un posible pseudocódigo de la operación *ClienteBancario::getSaldo():double* podría ser:

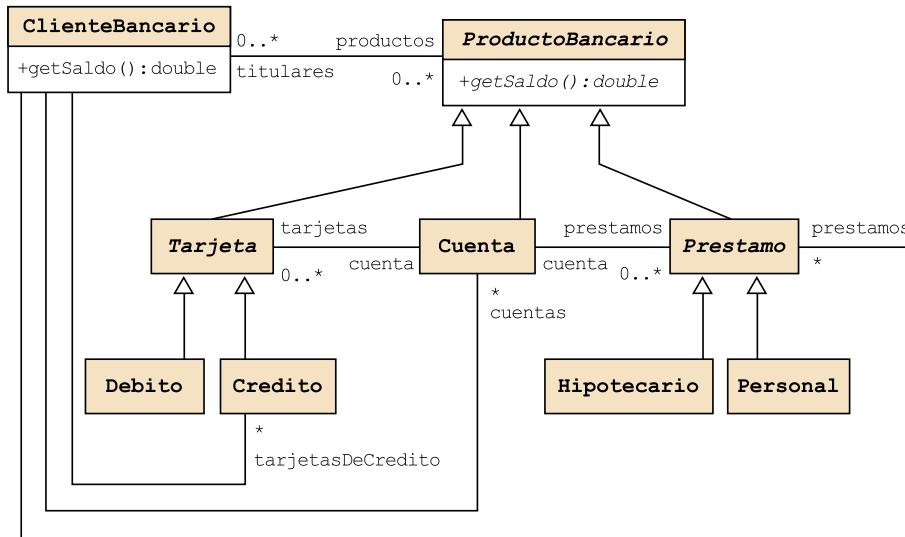
```
public double getSaldo() {
    double result=0.0;
    for (ProductoBancario p : this.productos) {
        result=result+p.getSaldo();
    }
    return result;
}
```

Sin embargo, las restricciones impuestas impiden dar esa implementación. Una solución posible recorrería la colección de productos e interrogaría por el tipo de cada uno de los objetos contenidos en la colección (mediante una instrucción tipo *instanceof* de Java), sumando, restando o no haciendo nada, según el caso:

```
public double getSaldo() {
    double result=0.0;
    for (ProductoBancario p : this.productos) {
        if (p instanceof Cuenta)
            result=result+p.getSaldo();
        else if (p instanceof Credito)
            result=result-p.getSaldo();
        else if (p instanceof Prestamo)
            result=result-p.getSaldo();
    }
    return result;
}
```

Obviamente, el código anterior impide completamente su reutilización, pues hace su implementación totalmente dependiente de la colección de tipos existentes en el sistema. Una solución posible pasa por la modificación del diagrama de clases para hacer que el cliente

conozca directamente a todos sus productos asociados de los tipos *Credito*, *Cuenta* y *Prestamo*, además de conocerlos a través de la asociación con *ProductoBancario*. Es decir:



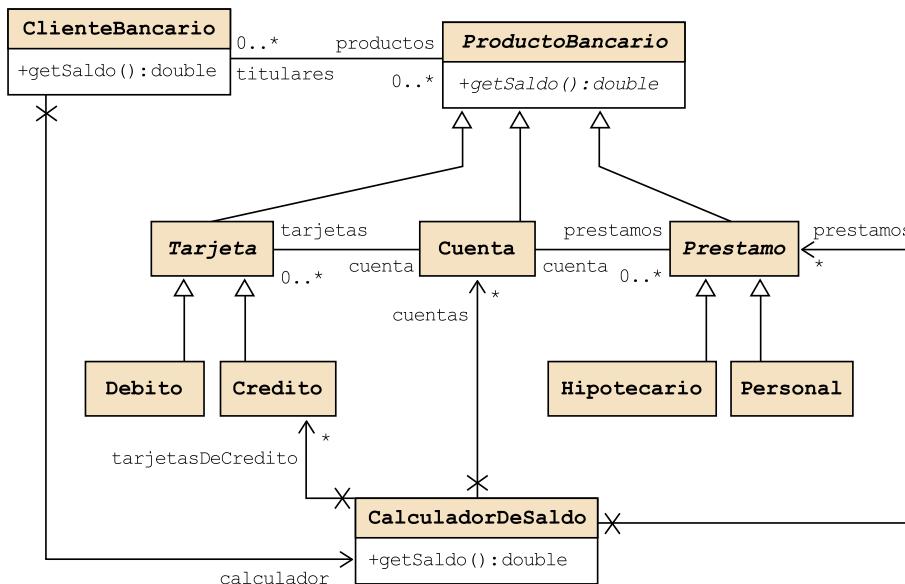
El código de la operación, en este caso, podría ser el siguiente:

```

public double getSaldo() {
    double result=0.0;
    for (Cuenta c : this.cuentas)
        result=result+c.getSaldo();
    for (Credito t : this.tarjetasDeCredito)
        result=result-t.getSaldo();
    for (Prestamo p : this.prestamos)
        result=result-p.getSaldo();
    return result;
}
  
```

La solución dada, sin embargo, aumenta la complejidad del sistema y dificulta también la reutilización.

Otra solución pasa por delegar la responsabilidad de calcular el saldo a una clase especializada que se dedique solo a esto, lo que puede entenderse como una aplicación del patrón *Experto*: asociamos a la clase *ClienteBancario* un *CalculadorDelSaldo* que recorre de la manera adecuada los objetos correspondientes:



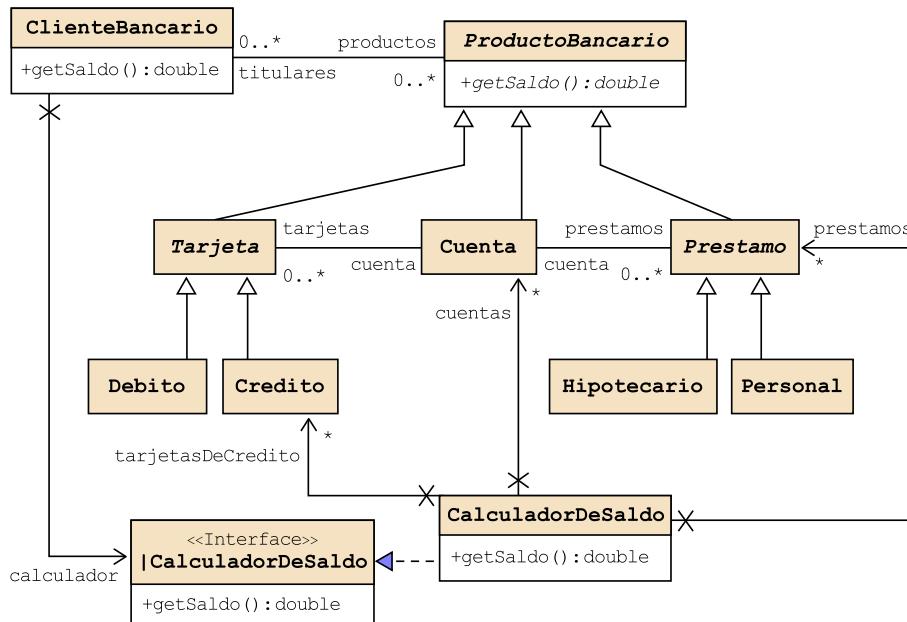
Ahora, el código de `getSaldo()` en `ClienteBancario` consistiría, simplemente, en una llamada a la operación `getSaldo()` de `CalculadorDelSaldo`:

```
public double getSaldo() {
    return this.calculador.getSaldo();
}
```

Y, por otro lado, el código de `getSaldo()` en `CalculadorDelSaldo` sería:

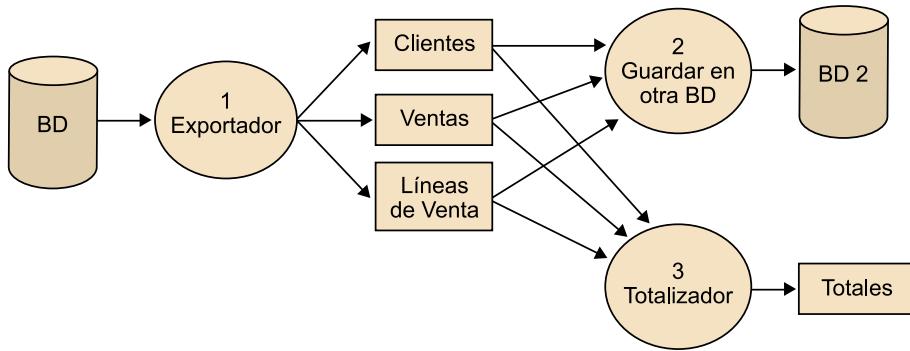
```
public double getSaldo() {
    double result=0.0;
    for (Cuenta c : this.cuentas)
        result=result+c.getSaldo();
    for (Credito t : this.tarjetasDeCredito)
        result=result-t.getSaldo();
    for (Prestamo p : this.prestamos)
        result=result-p.getSaldo();
    return result;
}
```

Esta solución, sin ser perfecta, deja prácticamente sin tocar la clase `ClienteBancario`, que es probablemente importante para el sistema y susceptible de ser reutilizada, aunque obliga a la adición del campo `calculador` a la clase `ClienteBancario`, lo que supone que esta clase queda acoplada a `CalculadorDelSaldo`. Para disminuir el acoplamiento puede crearse una interfaz intermedia, para la cual, no obstante, el último código sigue siendo válido: en la figura siguiente, el `ClienteBancario` conoce a un calculador genérico (la interfaz `ICalculadorDeSaldo`), que está implementado por el `CalculadorDelSaldo` concreto:

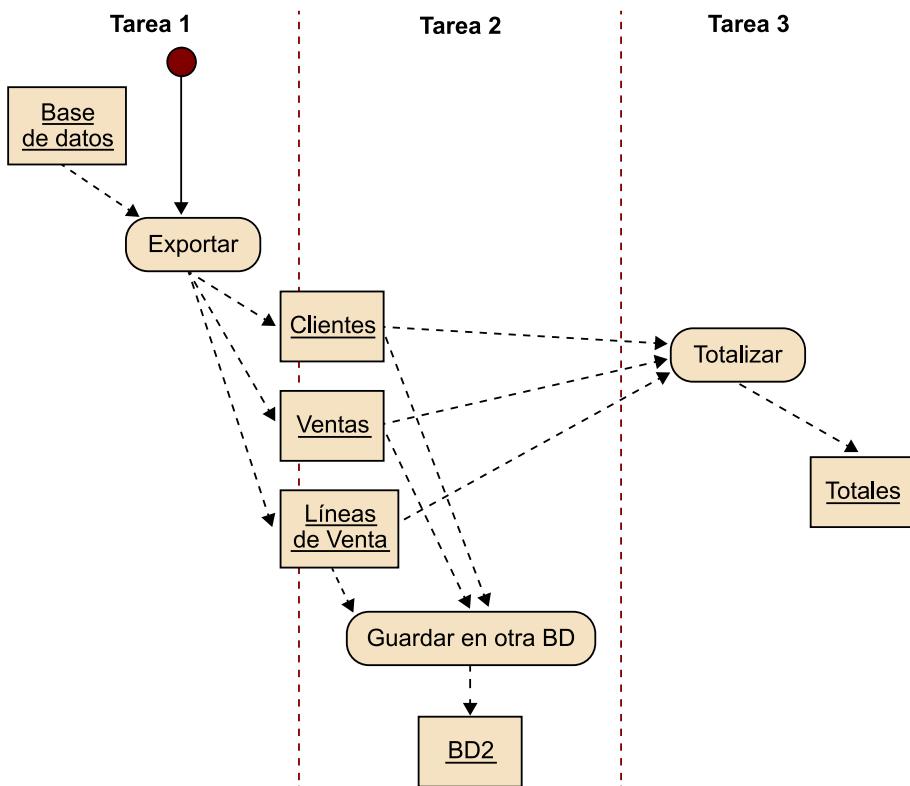


Esta última solución tiene la ventaja de que el `ClienteBancario` no está directamente acoplado al `CalculadorDelSaldo`: si en algún momento el saldo del cliente se desea calcular de otra manera, bastará con sustituir la clase `CalculadorDelSaldo` por otra versión que implemente la interfaz `ICalculadorDeSaldo`.

3. Para representar la arquitectura de *pipes and filters* del sistema solicitado utilizaremos la notación de los diagramas de flujos de datos. Existen tres procesos, varios almacenes de datos y dos entidades externas (la base de datos de origen y la base de datos de destino):

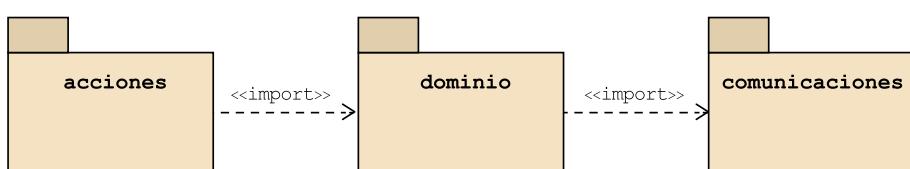


En la figura siguiente se muestra la misma solución utilizando ahora un diagrama de actividad UML, en donde cada *swimlane* representa una de las tres tareas:



4. Como se ha visto en el desarrollo del tema correspondiente, se recomienda crear una acción de *struts* por cada caso de uso. Además, y con objeto de comunicar con el sistema central, crearemos también un subsistema de comunicaciones.

A muy alto nivel, el sistema web constará, al menos, de los siguientes tres subsistemas:

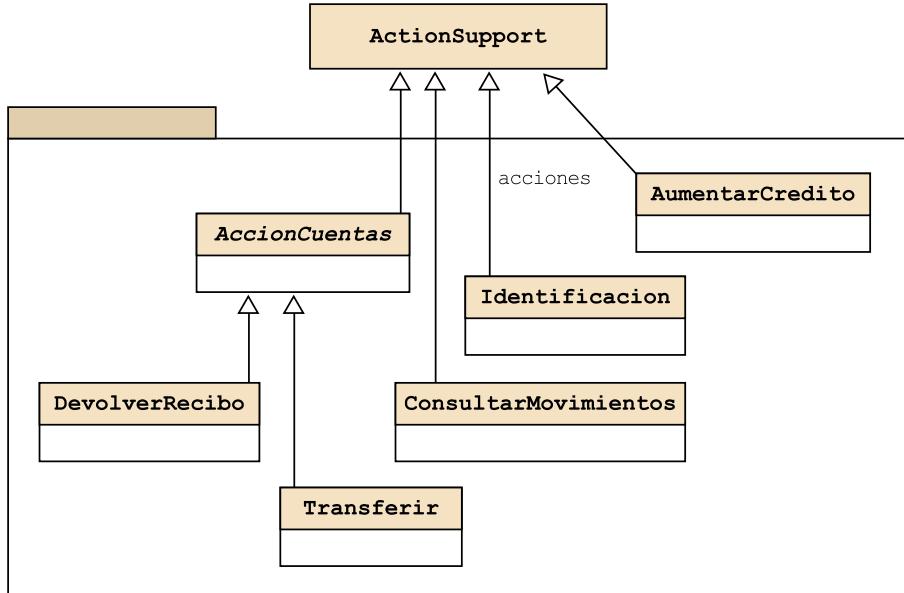


En el paquete de acciones colocaremos las clases que implementan acciones de *struts*:

- Se requiere una acción para el caso de uso identificarse.
- Se requiere una acción para el caso de uso consultar movimientos, que está incluido en operar sobre cuentas, operar sobre tarjetas y operar sobre préstamos.
- Se requiere una acción para transferir y otra para devolver recibo. Estos dos casos de uso están incluidos en operar sobre cuentas, por lo que crearemos una acción abstracta Ac-

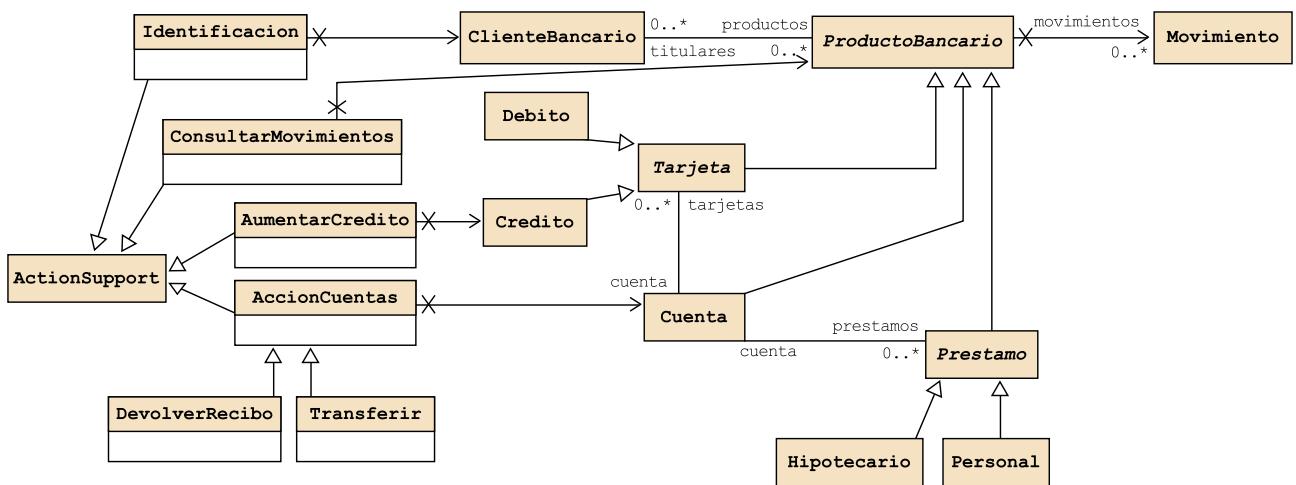
ciónCuentas que tenga la lógica concreta que pueda ser heredada en las acciones Transferir y DevolverRecibo.

El diseño que podemos dar al paquete de acciones es el siguiente:



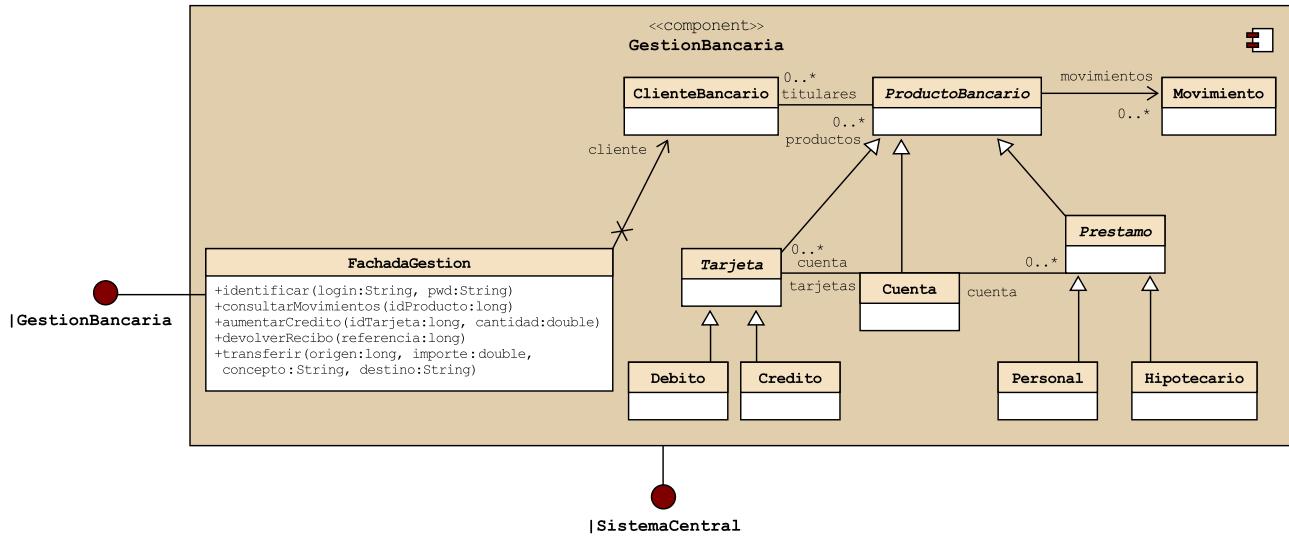
Obviamente, es necesario conectar cada acción a los objetos de dominio adecuados. En la siguiente figura:

- La acción *Identificación* se conecta al *ClienteBancario*, pues se asume que dispondrá de los mecanismos necesarios para identificarse con el sistema central.
- *ConsultarMovimientos* la conectamos a la clase abstracta *ProductoBancario*, pues del diagrama de casos de uso parece deducirse que los movimientos de todos los productos se consultan igual.
- *AumentarCredito* se asocia solamente al subtipo *Crédito de tarjeta*.
- Conectamos *AcciónCuentas* a *Cuenta*. Las dos especializaciones de esta acción reutilizan la estructura y la lógica definidas en la superclase.

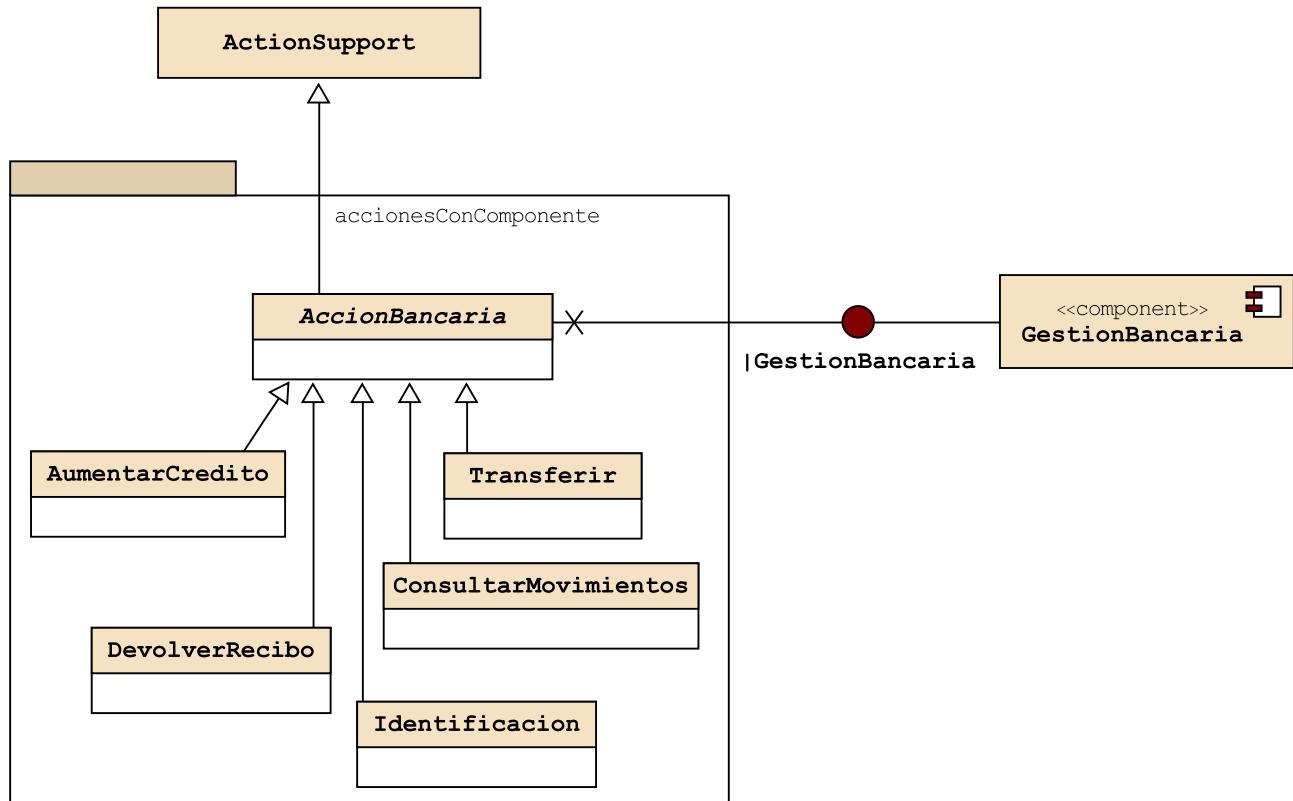


Por último, en el paquete de comunicaciones colocaremos un *proxy* que será utilizado por el sistema web para conectar al sistema central.

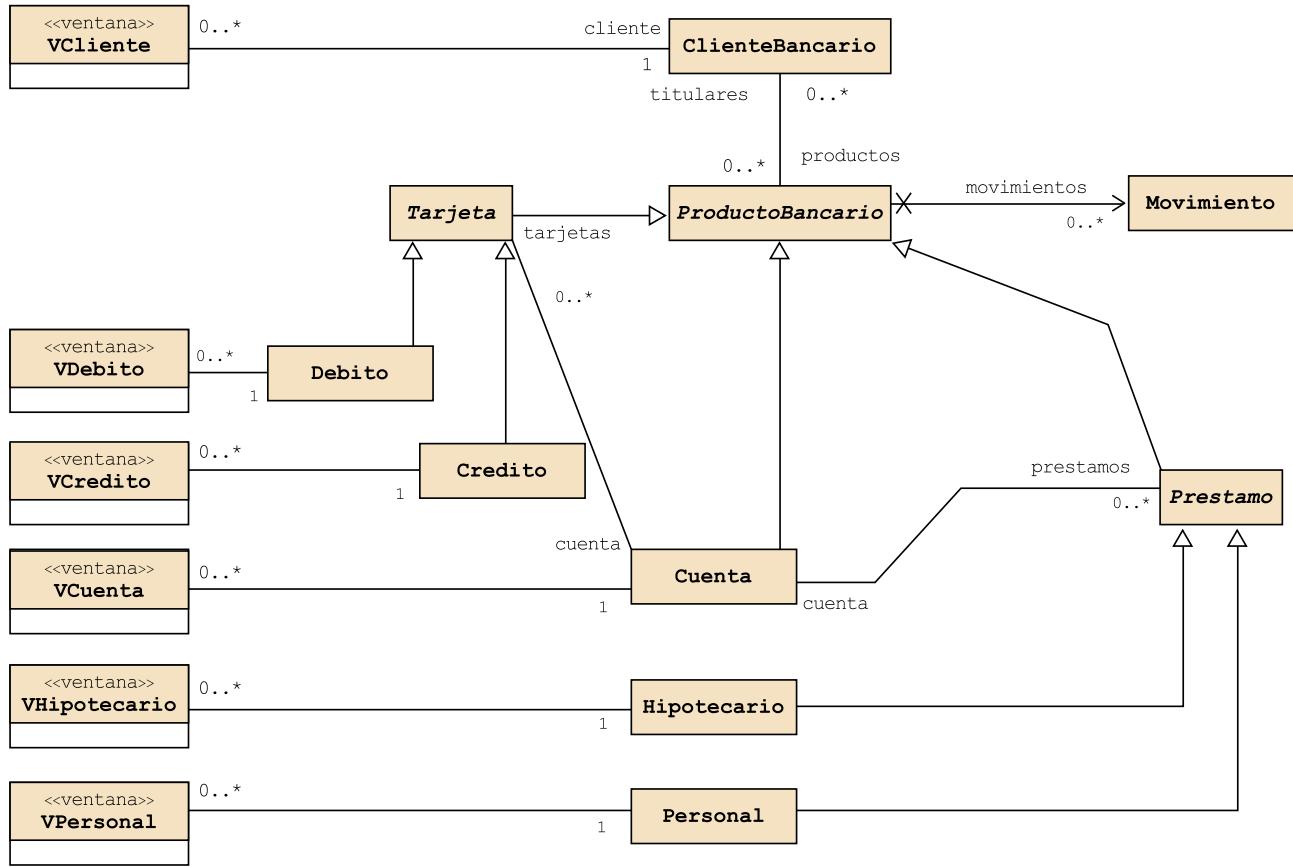
5. Para solucionar este ejercicio, encapsularemos todas las clases incluidas en el diagrama de clases mostrado en el enunciado del ejercicio anterior en un componente. Además, incluiremos en el componente un patrón fachada que implementará todas las operaciones ofrecidas por la interfaz ofrecida (*IGestiónBancaria*). La fachada encamina todas las llamadas a las operaciones a través del *ClienteBancario*, único objeto al que conoce directamente.



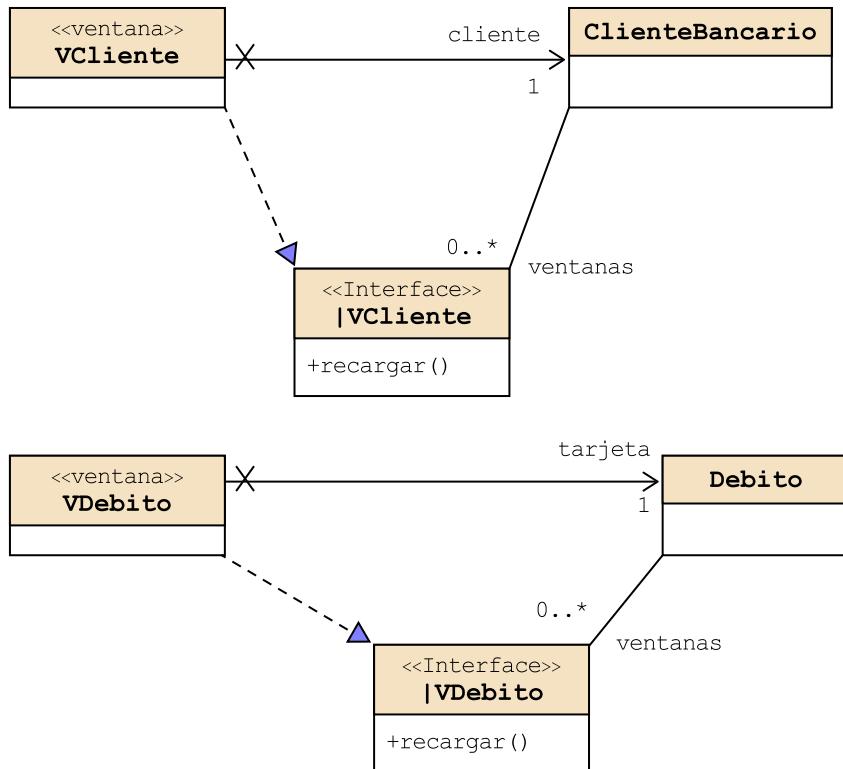
6. En este caso, podemos limitarnos a tener una acción genérica que conozca a una única instancia del componente y que puede ser abstracta. De ella heredan tantas acciones como funcionalidades sea necesario servir. Cada acción da una implementación diferente al método *execute()* en función del objetivo de la funcionalidad.



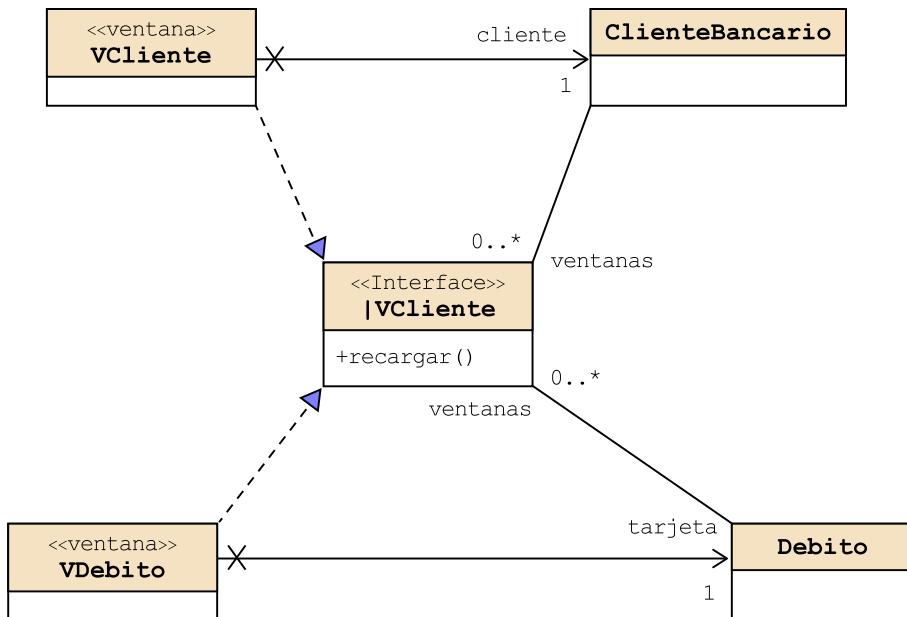
7. Una solución posible pero, desde luego, muy lejos de la ideal y que además no utiliza patrones pasa por acoplar las clases de dominio a las de presentación: cada ventana observa a un objeto de dominio, y cada objeto de dominio conoce a todas las ventanas que “lo están mirando”. Cuando cambia el estado del objeto de dominio, él mismo (lo cual sería conforme con el patrón experto) actualiza todas sus vistas. Esta solución, sin embargo, acopla prohibitivamente el dominio a la presentación y da a las clases de dominio la responsabilidad de actualizar sus vistas, lo que disminuye la cohesión.



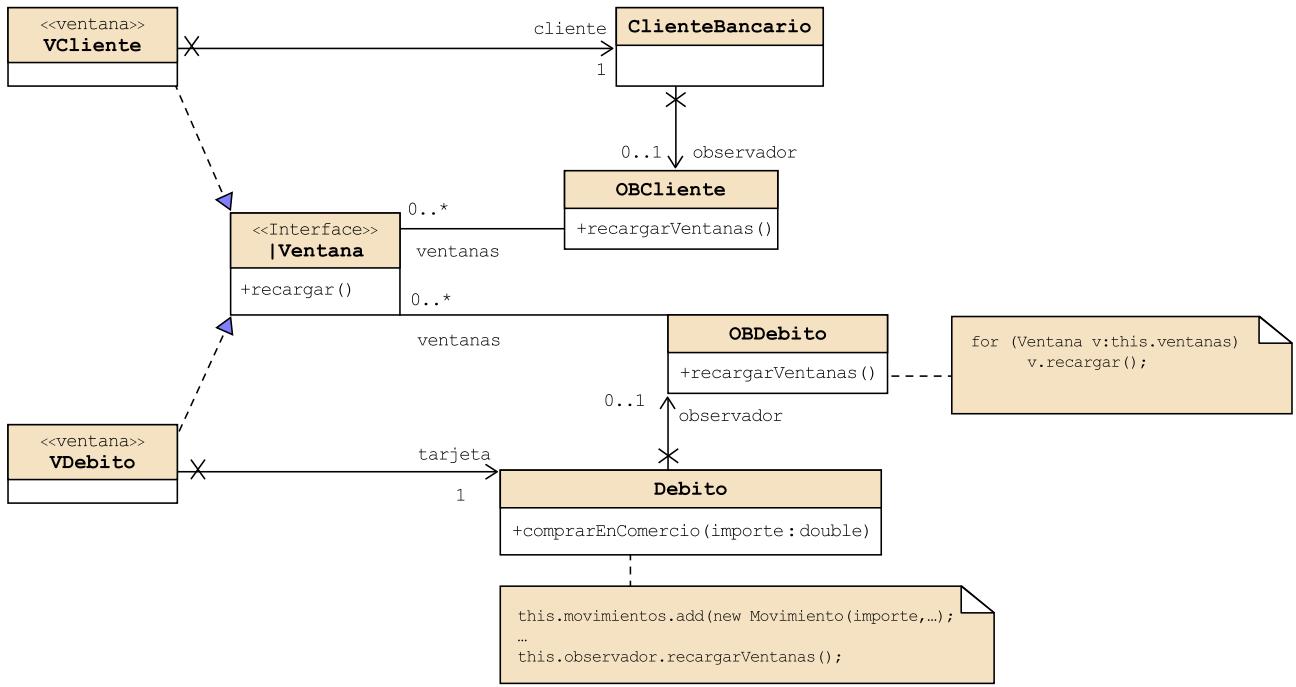
Para mejorar el diseño, pueden crearse interfaces que pueden ubicarse en una capa intermedia de servicios que desacoplen el dominio de la presentación. Para no sobrecargar demasiado el diagrama, la figura siguiente muestra esta solución para el *ClienteBancario*, la tarjeta de *Débito* y sus correspondientes ventanas: cuando el objeto de dominio cambia, ejecuta la operación *recargar()* sobre todos los objetos de tipo *IVCliente* o *IVDebito* a los cuales conoce. Asumimos que la operación *recargar()* en las ventanas recarga y muestra el estado que interese de, en este ejemplo, el cliente o la tarjeta al que la ventana esté observando.



Con esta solución se disminuye el acoplamiento del dominio respecto de la presentación: se tiene ahora un acoplamiento “menos malo”, porque el dominio conoce solo objetos genéricos (interfaces) en lugar de objetos concretos (clases de tipo ventana). Si nos vamos hacia la adopción de esta solución para todo el sistema, podemos crear una única interfaz **IVentana** que declare la operación *recargar()* y que sea implementada por todas las ventanas de la capa de presentación. Continuando con la particularización para *ClienteBancario* y *Débito*:

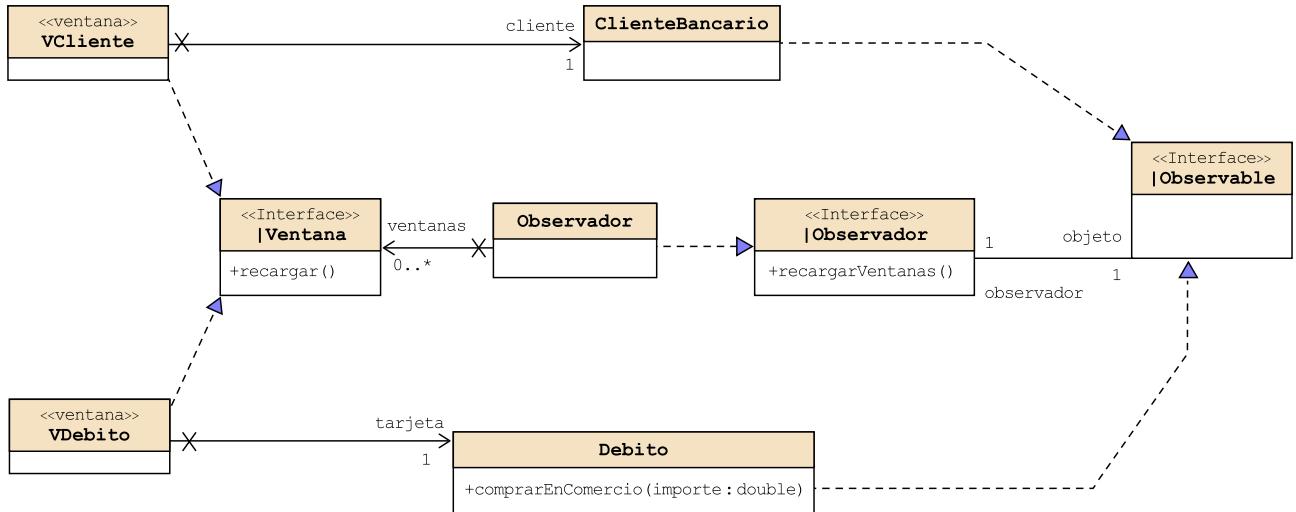


De todos modos, la cohesión del dominio sigue siendo baja con la solución anterior, porque los objetos de dominio siguen teniendo la responsabilidad de indicar a las ventanas que se recarguen. En el siguiente diagrama se delega la responsabilidad de recargar a un observador que asociamos a cada objeto de dominio: cuando cambia el estado de uno de estos (por ejemplo, porque se compre en un comercio con una tarjeta de débito), el objeto ejecuta las operaciones de dominio que correspondan y luego le dice al observador que recargue las ventanas que lo están mirando; el observador, que las conoce todas, ejecuta la operación *recargar* sobre cada una.

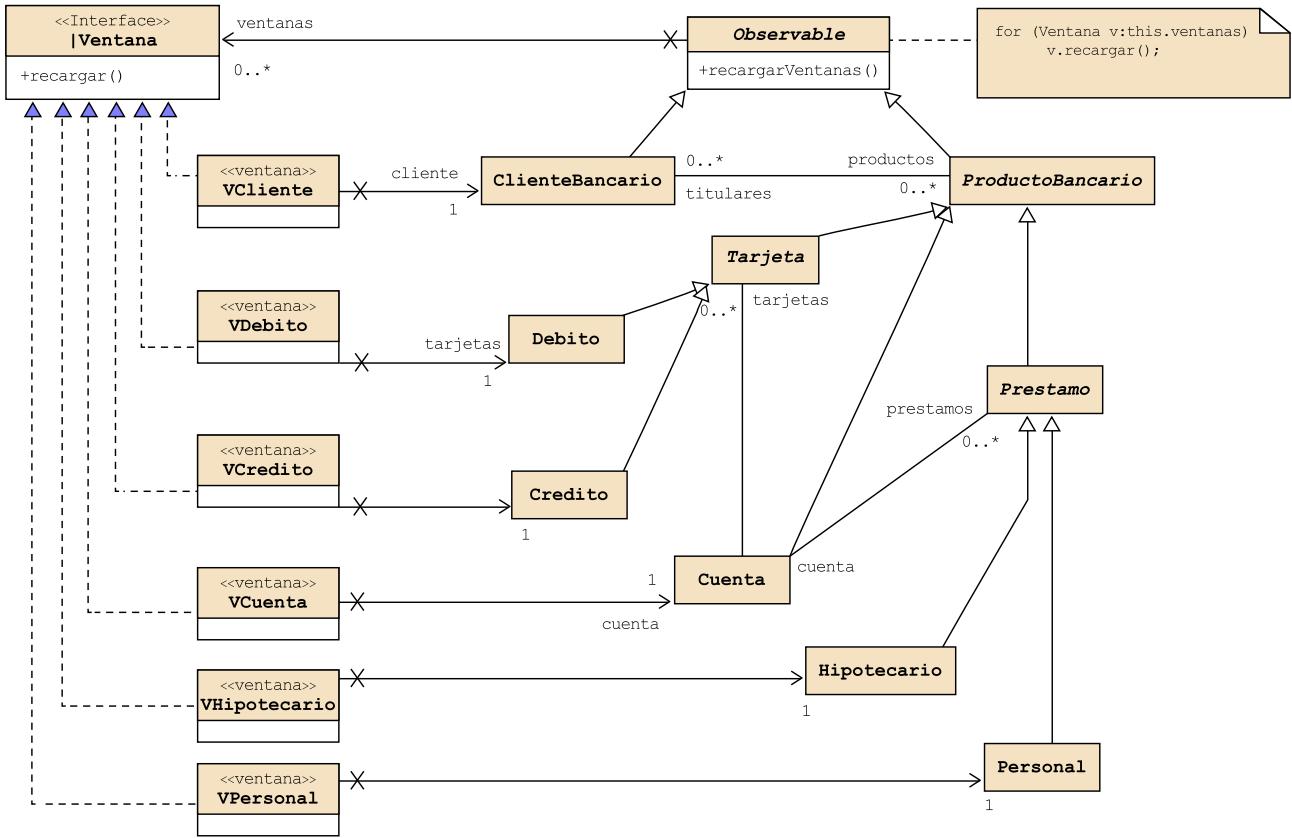


Ya que cada clase de dominio va a tener un observador, bien podemos generalizar todos los objetos de dominio a un supertipo *IObservable*, y crear también un observador genérico que conozca a objetos *IObservables*.

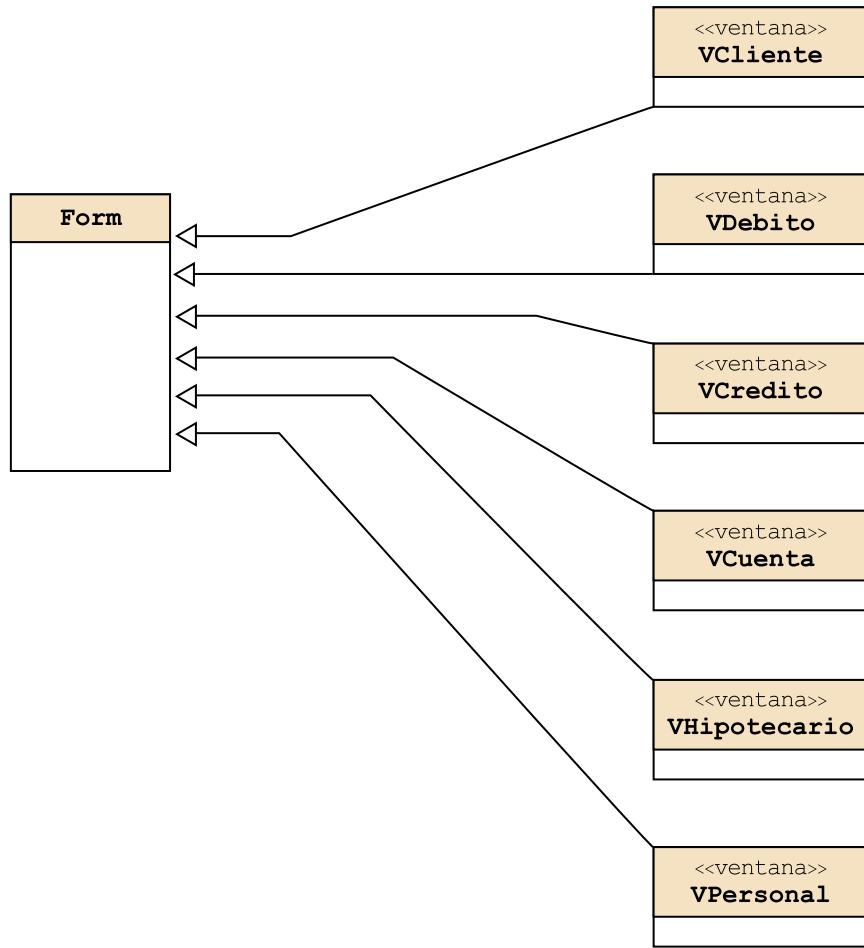
En el siguiente diseño, cada objeto de dominio es un objeto *IObservable*; cada objeto observable conoce a un *IObservador*, y cada *IObservador* observa a un objeto observable:



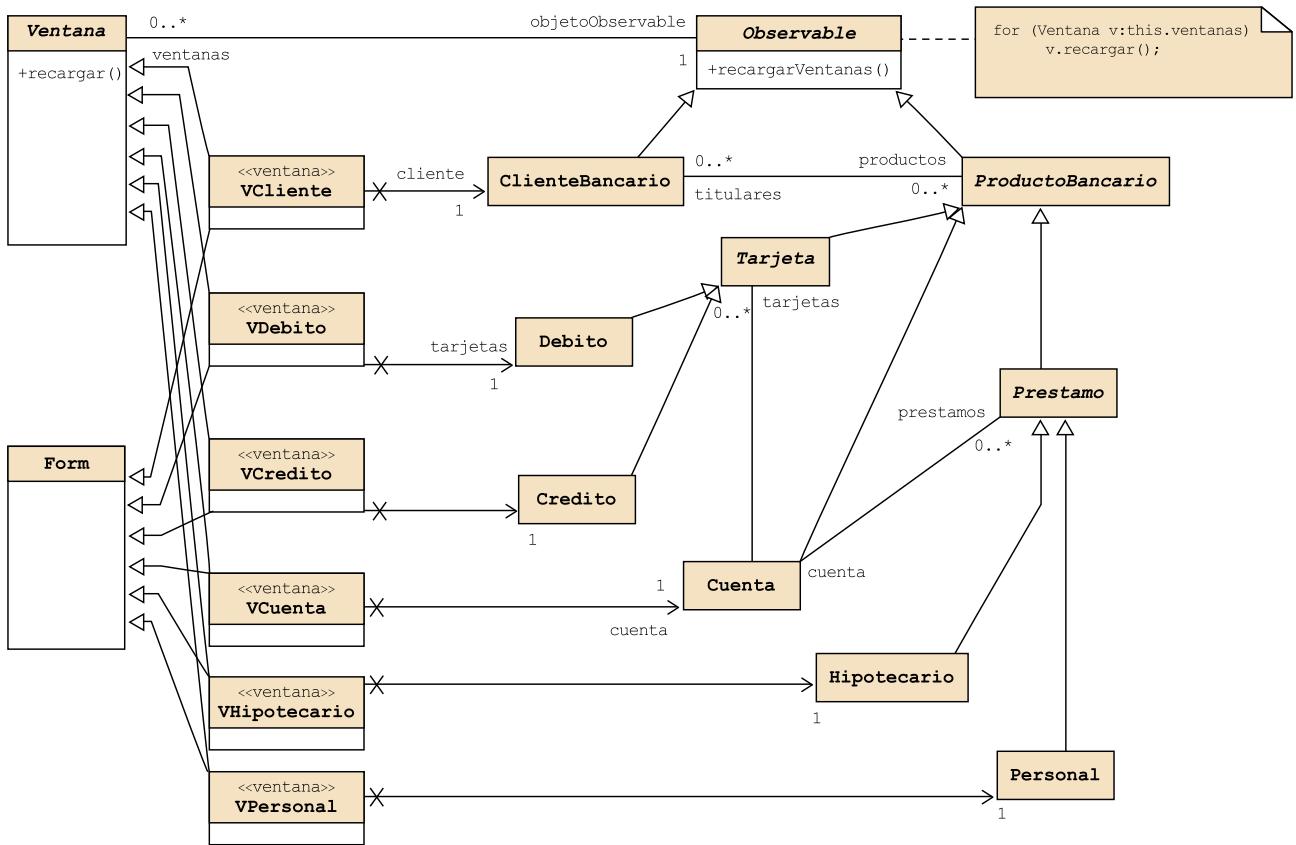
Una solución adicional viene dada por la estructura de herencia que hay en el dominio: podemos hacer que tanto el *ClientBancario* como el *ProductoBancario* sean especializaciones de una clase abstracta *Observable* que, sin embargo, tiene una operación concreta *recargarVentanas()*. Esta clase abstracta conoce a una colección de objetos de tipo *IVentana* (una interfaz), que es implementada por todas las ventanas: cuando cambia el estado de un objeto (se compra con una tarjeta de *Débito*, por ejemplo), se ejecutan las operaciones de dominio que correspondan y luego, se llama a la operación heredada *recargarVentanas()*, cuya implementación se hereda íntegramente de la superclase.



Supongamos, por otro lado, que las ventanas son especializaciones de un supertipo *Form* incluido en un *framework* de creación de interfaces de usuario:

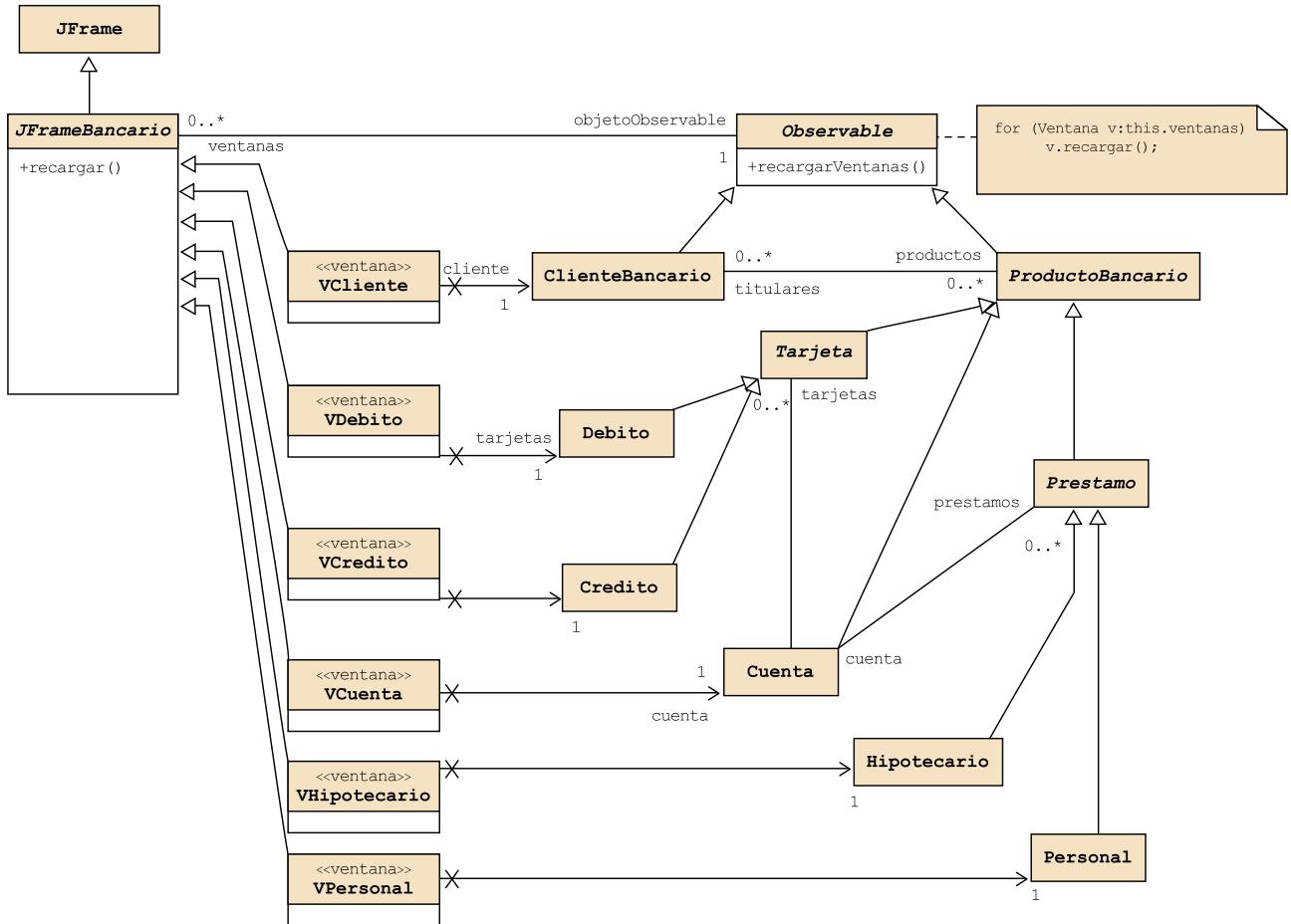


Supongamos, además, que el lenguaje de programación en el que vamos a implementar el sistema soporta herencia múltiple. En este caso, la interfaz *IVentana* que hemos mostrado en la última solución podría sustituirse por una superclase abstracta *Ventana* que incluya la operación concreta *recargar()*. Cada ventana concreta (*VCuenta*, *VDebito*, etcétera) hereda tanto de *Form* como de *Ventana*:



En este último diseño, cada instancia de *Ventana* conoce dos veces a la misma instancia del objeto de dominio: una por el campo *ObjetoObservable* que está heredando de *Ventana*, y otra por la asociación que relaciona a la propia *Ventana* con el objeto de dominio que corresponda (por ejemplo: *VCuenta* conoce a una instancia de tipo *Cuenta* mediante su campo *Cuenta*, pero también conoce a la misma instancia mediante el campo *ObjetoObservable* que hereda de *Ventana*). La referencia heredada se utiliza únicamente para recargar la ventana; la referencia propia, para llamar a los métodos de negocio correspondientes al objeto. El programador deberá garantizar que ambas referencias apunten al mismo objeto.

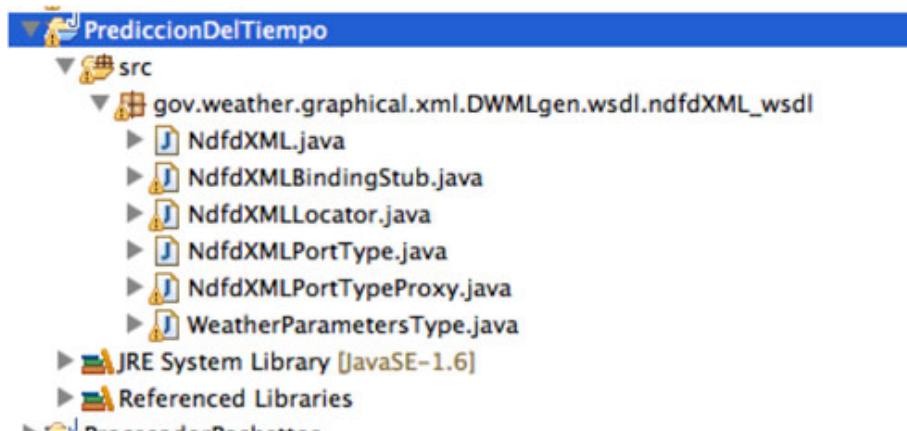
Supongamos ahora que el lenguaje de programación no admite herencia múltiple (como podría ser el caso de Java). En este caso, todas las ventanas serán, probablemente, especializaciones de alguna clase incluida en algún *framework* de diseño de interfaces de usuario (*VCuenta*, *VDebito*, etcétera, pueden ser especializaciones de la clase *JFrame* incluida en Swing). Si no deseamos utilizar interfaces (como en una solución anterior), podemos crear una clase abstracta *JFrameBancario* que herede de *JFrame*, que conecte al objeto observable, de la que hereden las ventanas específicas de gestión y que dé implementación a la operación *recargar()*. Como en la última solución, en la que hemos utilizado herencia múltiple, cada ventana concreta poseerá dos referencias a cada instancia de dominio: una heredada del *JFrameBancario* y otra a la que apunta cada ventana concreta. Igual que antes, el programador debe garantizar que ambos campos apunten a la misma referencia.



Ahora que se han presentado múltiples soluciones para el mismo problema, se pide al lector que piense y razoné sobre los diferentes valores de cohesión, acoplamiento o reutilización que aporta cada una.

8. Para resolver esta actividad, debemos utilizar el asistente de nuestro entorno de desarrollo para conectar al documento WSDL publicado en la dirección especificada en el enunciado del problema.

En el entorno eclipse, por ejemplo, podemos crear un proyecto llamado PrediccionDelTiempo y crear, mediante el asistente, las clases necesarias para acceder al servicio:



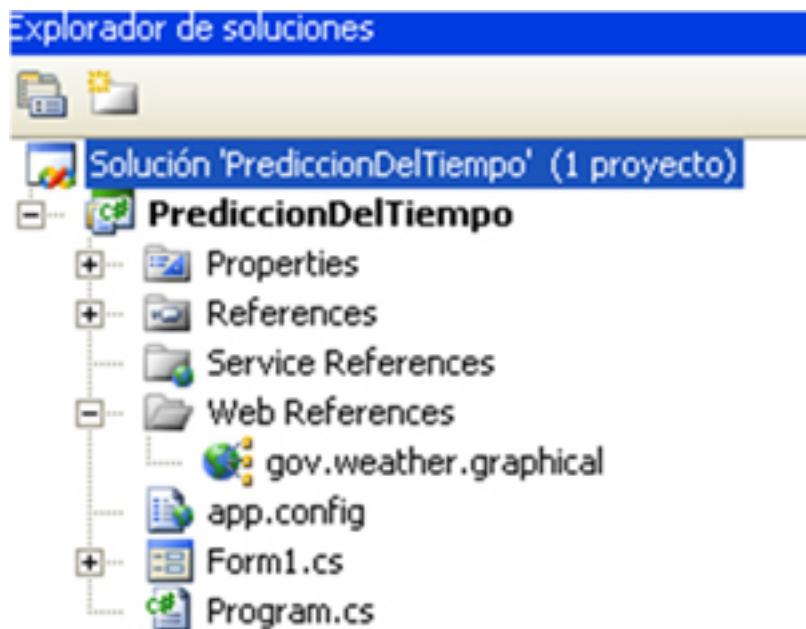
A continuación, escribimos el código para realizar la llamada solicitada:

```

public static void main(String[] args) {
    BigDecimal latitud=new BigDecimal("+35.35");
    BigDecimal longitud=new BigDecimal("-82.33");
    GregorianCalendar gc=new GregorianCalendar(2011, 11, 11);
    Date fechaDeInicio=new Date(gc.getTimeInMillis());
    BigInteger numeroDeDias=new BigInteger("7");
    String unidad="m";
    String formato="24 hourly";
    try {
        NdfdXMLPortTypeProxy proxy = new NdfdXMLPortTypeProxy();
        String result = proxy.NDFDgenByDay(latitud, longitud, fechaDeInicio,
            numeroDeDias, unidad, formato);
        System.out.println(result);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

En Microsoft Visual Studio, la forma de proceder es muy parecida: se agrega la referencia web...



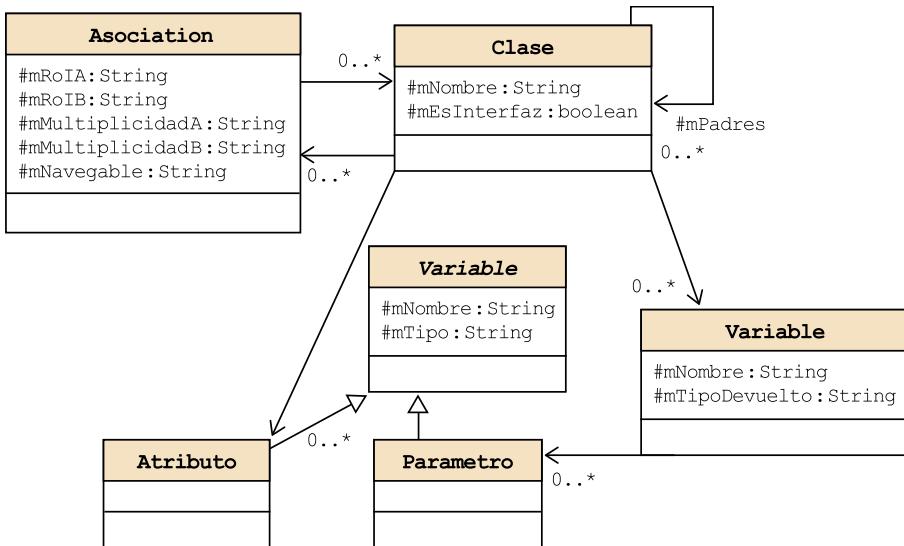
...y se escribe el código:

```

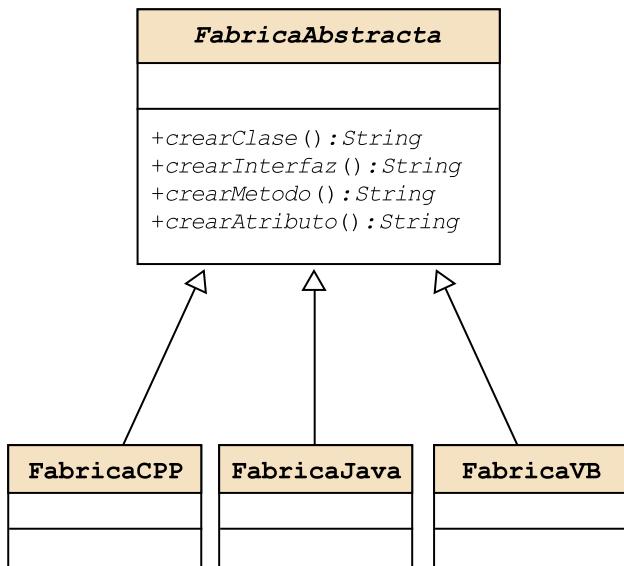
private void button1_Click(object sender, EventArgs e)
{
    gov.weather.graphical.ndfdXML proxy =
    new WeatherForecast.gov.weather.graphical.ndfdXML();
    string result = proxy.NDFDgenByDay(35.35M, -82.33M,
        new DateTime(2011, 12, 11), "7", "m", "24 hourly");
    Console.WriteLine(result);
}

```

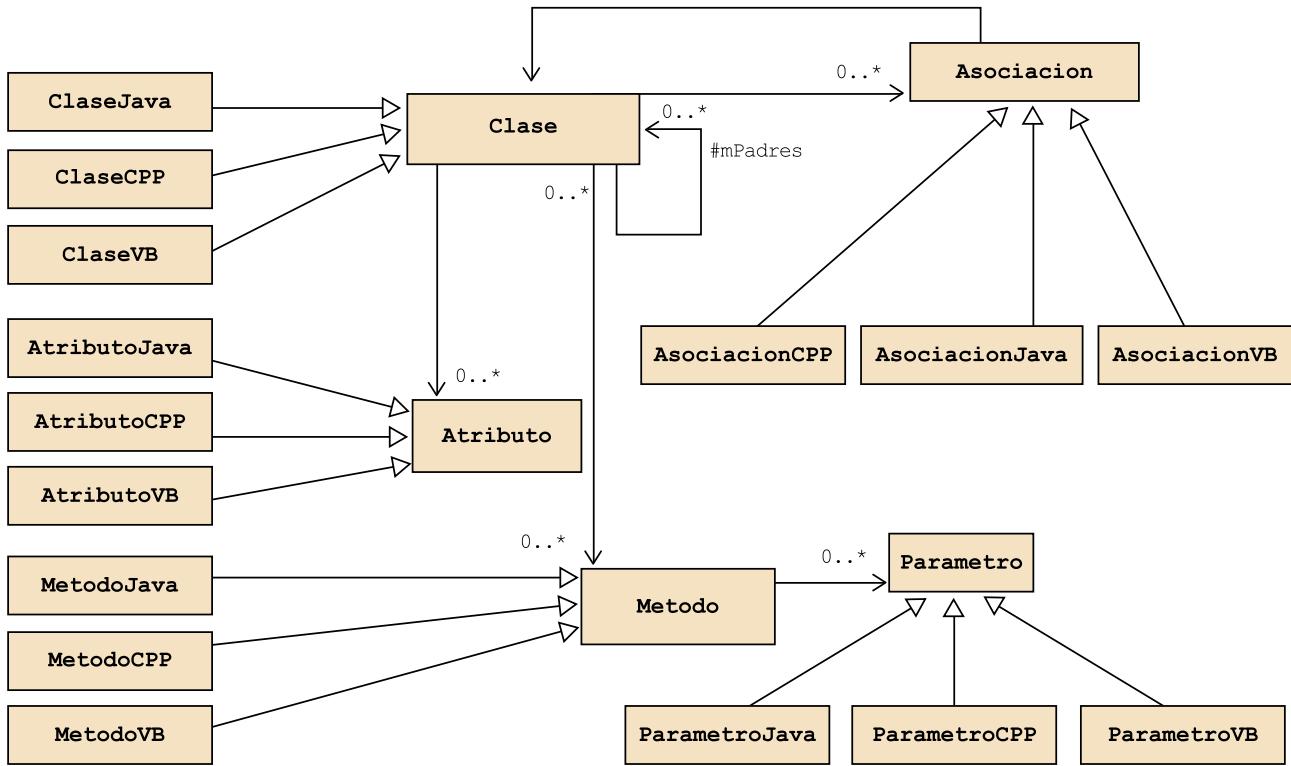
9. Este ejercicio combina conceptos del tema anterior (el patrón Abstract Factory) con la idea de los metamodelos descritos en este. Idealmente, el metamodelo que tal herramienta debería soportar para representar diagramas de clase debería ser del de UML 2.0. No obstante, y por simplificar, supondremos que utilizamos el siguiente metamodelo reducido: una clase hereda, se relaciona con otras clases mediante asociaciones, y tiene atributos y métodos, que pueden tener parámetros:



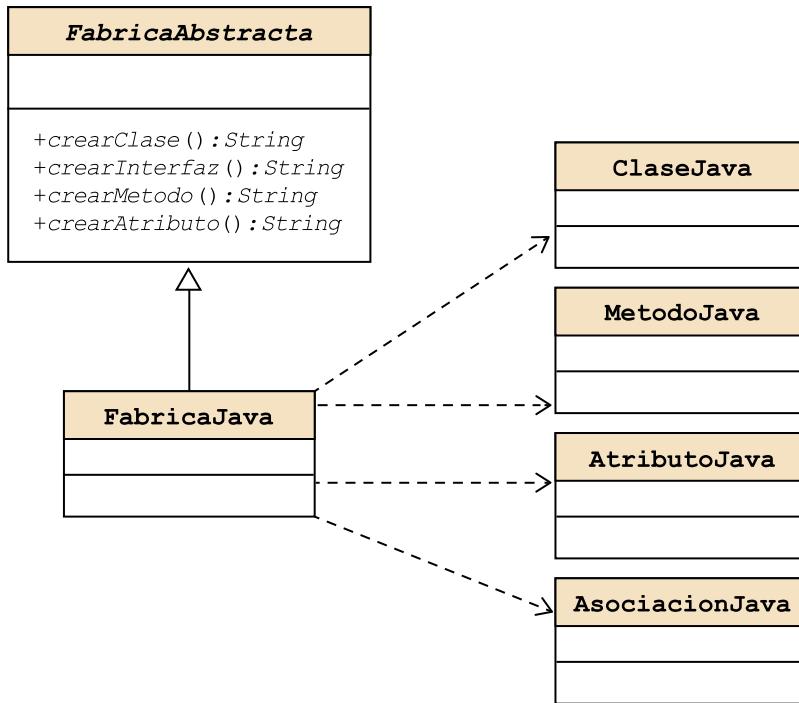
Puesto que se desea generar código para tres lenguajes diferentes, crearemos tres fábricas concretas, que serán especializaciones de una abstracta. En la fábrica abstracta declararemos las operaciones necesarias para crear los tipos de elementos cuyo código se desea generar:



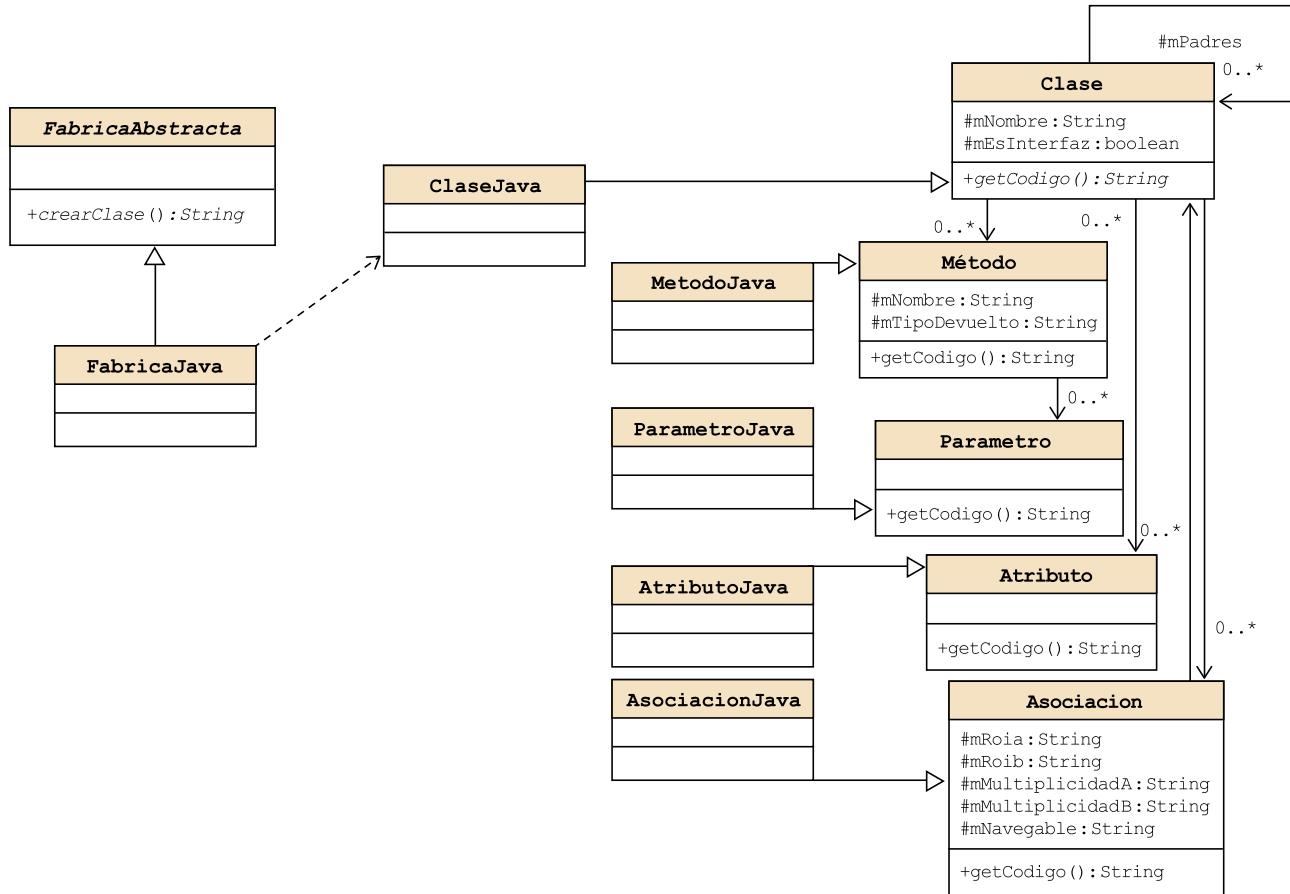
Por otro lado, especializamos las clases definidas en el metamodelo creando, por ejemplo, las clases *ClaseJava*, *ClaseCPP* y *ClaseVB* que, respectivamente, representan clases Java, C++ y de Visual Basic:



Por último, enlazamos cada fábrica concreta con los tipos de elementos concretos. Por ejemplo, haciendo que la fábrica de Java cree clases, métodos, atributos y asociaciones en Java:



En esta otra solución se combina el patrón *Fábrica Abstracta* con el *Builder*: puesto que una clase es un agregado de atributos y métodos, se puede hacer que cada fábrica concreta cree solamente una instancia concreta de su subtipo correspondiente: por ejemplo, que *FabricaJava* cree objetos de tipo *ClaseJava*. Entonces, consideramos que *ClaseJava* es el *Builder* y la dotamos de la capacidad de generar el código completo de la clase mediante un método *getCodigo*, que es abstracto en clase pero concreto en *ClaseJava*, *ClaseCPP* y *ClaseVB*:



10. La respuesta a esta pregunta es un claro y contundente “sí”: QVT es un lenguaje de transformación definido para modelos MOF. Puesto que tanto el metamodelo de ATL como el metamodelo de UML 2.0 son conformes a MOF, cualquier modelo ATL (que no es sino una instancia de su metamodelo) podrá ser transformada a una instancia del metamodelo de UML 2.0 (es decir, a un modelo). Recuérdese que un modelo (nivel 2 de MOF) es una instancia de su correspondiente metamodelo (nivel 1 de MOF).

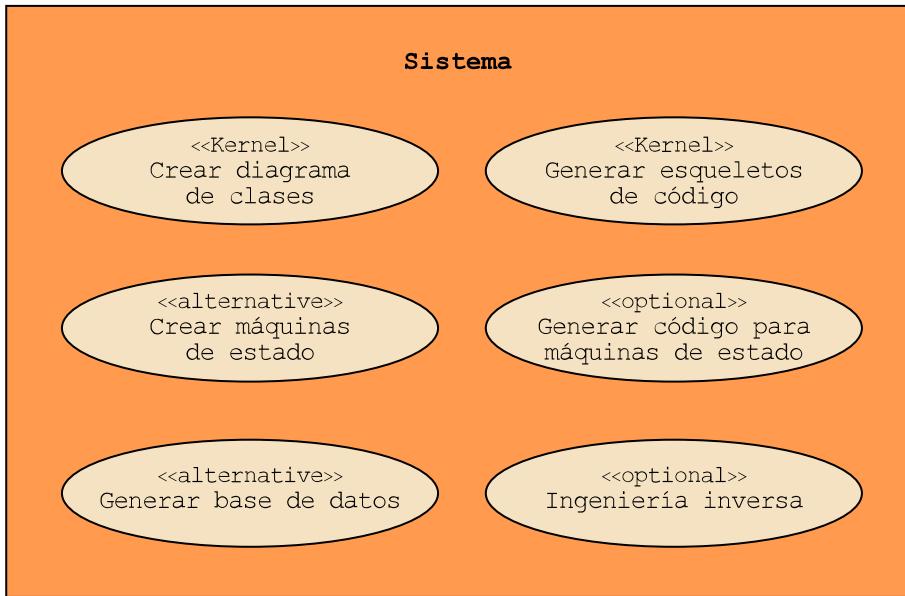
Otra cosa es la posible dificultad inherente a la determinación de las equivalencias entre los elementos de cada metamodelo, que puede hacer muy complejas las transformaciones.

11. El enunciado queda suficientemente abierto (nos dice que modelemos “algunos aspectos de esta herramienta”) para que practiquemos diferentes aspectos en cuanto a líneas de producto.

La siguiente tabla resume las características de cada edición:

Edición	Diagramas de clase	Generación esqueletos	Máquinas de estado	Generación código m. estado	Generación de BB. DD.	Ingeniería inversa
Estándar	Sí	Sí	Parcialmente			
Ejecutiva	Sí	Sí	Sí	Sí	Parcialmente (sólo SQL)	
Empresarial	Sí	Sí	Sí	Sí	Sí	Sí

Un posible conjunto de casos de uso para este sistema podría ser el siguiente:



- Crear diagramas de clases y generar esqueletos de código son casos de uso kernel porque son funcionalidades que se incluirán en todas las ediciones y, además, funcionarán en todas exactamente de la misma manera.
- La creación de máquinas de estado está incluida en todas las ediciones, pero su implementación será diferente en la edición estándar. Por ello, se ha decidido estereotipar este caso de uso como alternativa. La generación de base de datos solo pertenece a las ediciones ejecutiva y empresarial y, además, es algo diferente en una y en otra, por lo que también se ha usado el mismo estereotipo.
- Finalmente, ingeniería inversa y generar código para máquinas de estado son optional porque se incluyen, siempre de la misma manera, solo en dos de los tres productos.

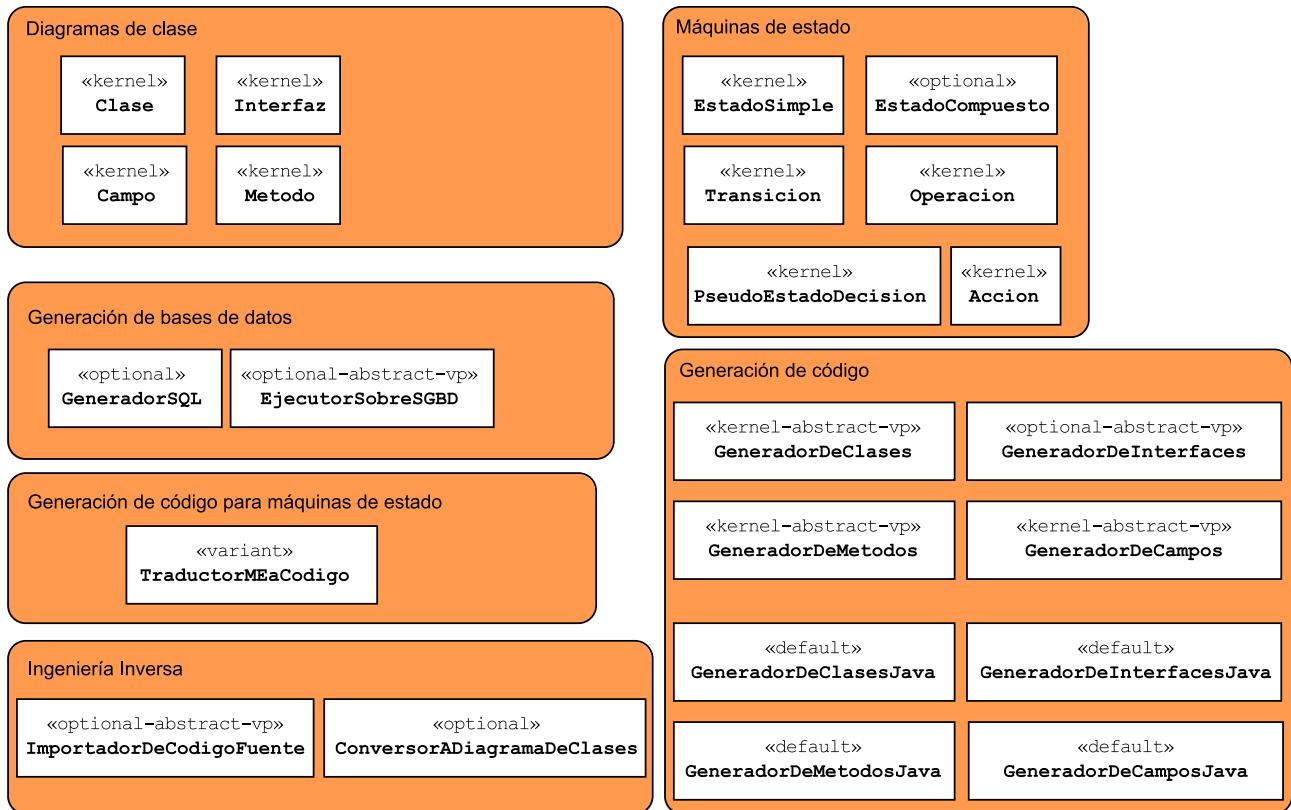
Si saltamos a la etapa de diseño, y teniendo en cuenta los conocimientos que tenemos sobre metamodelado (ejemplos y actividades), podemos utilizar, a modo de ejemplo y simplificando mucho, las siguientes clases:

- Para la creación de diagramas de clases: clase, campo, método, interfaz.
- Para la creación de máquinas de estado: estado simple, estado compuesto, seudoestado de decisión, transición, operación, acción.
- Para la ingeniería inversa tendremos un importador de código y un conversor (que lo transformará al diagrama de clases).
- Para generación de esqueletos de código, un generador para cada tipo de elemento de los diagramas de clase.
- Para la generación de código para máquinas de estado, un traductor.
- Para la generación de bases de datos, utilizaremos un generador de SQL y otra clase ejecutor, que ejecuta las sentencias SQL sobre el gestor.

En esta figura se muestran las clases, agrupadas según la funcionalidad para la que están pensadas.

- Todas las clases del grupo diagramas de clase son kernel porque están incluidas en las tres ediciones de la herramienta. Además, se incluirán en ellas exactamente de la misma forma.
- En Generación de bases de datos, GeneradorSQL es optional porque está incluida solo en dos de las tres ediciones y, además, su implementación será la misma en ellas dos. El EjecutorSobreSGB es optional-abstract-vp porque es optional (no está incluida en todos los productos) pero, además, su implementación dependerá del gestor concreto sobre el que se ejecuten las sentencias de creación de tabla.
- Es discutible (como realmente cualquier solución que se dé a un problema en ingeniería del software) la adecuación del estereotipo *variant* con que se ha anotado Traductor-MEaCódigo del grupo generación de código para máquinas de estado. Se le ha aplicado porque es una clase que no estará en todos los productos pero, además, tiene dos implementaciones diferentes según se genere código para máquinas de estados simples o compuestos. No obstante, el enunciado es suficientemente abierto como para utilizar otros estereotipos.
- En ingeniería inversa, el ImportadorDeCódigoFuente es optional-abstract-vp porque estará incluido solamente en algunos productos y, además, con diferentes implementaciones según el lenguaje de programación que deba leerse. Esta clase obtendrá un diagrama

- de clases (del grupo diagramas de clase, cuyas cuatro clases son kernel) que será procesado por un ConversorADiagramasDeClases, que es optional.
- En máquinas de estado todas las clases son kernel excepto EstadoCompuesto, que no está incluida en la edición estándar.
 - En generación de código hay varias clases kernel-abstract-vp porque todas las ediciones generan el esqueleto de código, pero hay varios lenguajes de programación. Ilustramos el hecho de que la edición estándar genera en lenguaje Java mediante las clases estereotipadas con default.



Ejercicios de autoevaluación

1. El enfoque “proactivo” representa el hecho de que la reutilización de software se concibe desde el principio del desarrollo, y no después. Así, el grado de reutilización de los artefactos software viene a ser, desde el comienzo del proyecto, un requisito no funcional más.
2. Las clases de dominio persistentes delegan a las clases DAO la responsabilidad de mantenerlas actualizadas en el sistema de almacenamiento secundario (ficheros o bases de datos). Así, las clases de dominio, que son las que tienen la mayor parte de la complejidad del sistema, quedan desacopladas del sistema de almacenamiento, teniendo un mayor potencial de reuso.
3. Un caso de uso optional está presente en varios productos de una línea de producto software, y en todos ellos tiene la misma implementación. Un caso alternative también está presente solo en algunos productos pero, sin embargo, en cada uno de ellos puede tener una implementación diferente.
4. Se utilizan fundamentalmente: lenguajes de dominio específico, modelado de características, programación orientada a aspectos y programación genérica.
5. Un buen diseño arquitectónico multicapa mantiene la lógica de negocio bien separada de la capa de presentación, con lo que la lógica que resida en esta se limitará, prácticamente, a la gestión de los eventos del usuario, validación de datos, etcétera. Mantener este desacoplamiento facilita la reutilización de la capa de negocio o dominio, que podrá conectarse sin especiales dificultades (salvo las puramente tecnológicas) a la nueva capa de presentación.

Glosario

acoplamiento *m* Grado en el que están relacionados un elemento de un conjunto con los demás elementos. En el caso de un sistema software, el acoplamiento es una medida del número y calidad de las relaciones existentes entre clases, subsistemas, etc. Un elemento muy acoplado a otros se verá muy afectado por los cambios en estos elementos. El acoplamiento puede medirse cuantitativamente con métricas como *Coupling Between Objects*, *Depth of Inheritance of Tree* o *Number of Childs*.

cohesión *f* En ingeniería del software, la cohesión da una medida del grado en que están relacionados los miembros de una clase o las clases pertenecientes a un subsistema. Una clase con baja cohesión hace varias cosas diferentes que no tienen relación entre sí, lo que lleva a clases más difíciles de entender, de reutilizar y de mantener. Se puede medir cuantitativamente con la métrica *Lack of Cohesion in Methods*.

CBSE (component-based software engineering) *f* Subdisciplina de la ingeniería del software que se ocupa de la construcción de sistemas a partir de componentes reutilizables.

cloud computing *f* Paradigma de distribución de software y compartición de recursos por el que se ofrecen multitud de servicios a los usuarios (ejecución de aplicaciones, servicios de almacenamiento, herramientas colaborativas, etcétera), que los utilizan de manera totalmente independiente de su ubicación y, por lo general, sin demasiados requisitos de hardware.

DAO (data access object) *m* Objeto que se encarga de gestionar la persistencia de otros objetos de dominio.

DLL (dynamic-link library) *f* Librería o biblioteca de clases o funciones que se carga en tiempo de ejecución: esto es, en el momento en que el programa en ejecución la necesita.

DSL (domain-specific language) *m* Lenguaje de modelado que proporciona los conceptos, notaciones y mecanismos propios de un dominio en cuestión, semejantes a los que manejan los expertos de ese dominio, y que permite expresar los modelos del sistema a un nivel de abstracción adecuado.

enlace dinámico *m* Véase **DLL**

enlace estático *m* Librería o biblioteca de clases o funciones que se carga en tiempo de compilación para producir el programa ejecutable. Véase también, para comparar, **DLL**.

fabricación industrial del software *f* Modelo de construcción de software que se emplea en las llamadas fábricas de software, en las que un conjunto numeroso de ingenieros desarrolla aplicaciones grandes y complejas, utilizando de manera más o menos rigurosa metodologías de desarrollo, técnicas de reutilización, etc.

feature model *m* Modelo en el que se representan las *features* o características de un sistema software.

feature *f* Característica importante (desde el punto de vista funcional) de un producto software. El análisis de *features* se utiliza en paradigmas en los que la reutilización es una actividad importante.

genericidad *f* Capacidad que tienen algunos lenguajes de programación para definir y combinar clases parametrizando tipos de datos, que se declaran en tiempo de escritura del código, pero que se resuelven e instancian en tiempo de compilación.

ingeniería de dominio *f* Uno de los dos niveles de desarrollo de software en líneas de producto y en programación generativa. En la ingeniería de dominio, se construyen los elementos comunes a todos los productos.

ingeniería de producto *f* Uno de los dos niveles de desarrollo de software usados en líneas de producto y en programación generativa. En la ingeniería de producto, se construye cada uno de los productos software a partir de las propias características del producto y de lo construido en el nivel de ingeniería de dominio.

LPS (línea de producto software) *f* Método de producción de software en el que se hace un uso muy intensivo de la reutilización y se trabaja a dos niveles de desarrollo: ingeniería de dominio e ingeniería de producto.

MDE (model-driven engineering) *f* Paradigma dentro de la ingeniería del software que aboga por el uso de los modelos y las transformaciones entre ellas como piezas claves para dirigir todas las actividades relacionadas con la ingeniería del software.

metamodelo *m* Modelo que especifica los conceptos de un lenguaje, las relaciones entre ellos y las reglas estructurales que restringen los posibles elementos de los modelos válidos, así como aquellas combinaciones entre elementos que respetan las reglas semánticas del dominio.

MOF2Text *m* Especificación del OMG para la transformación de modelos a texto.

MOFScript *m* Implementación de *MOF2Text* distribuida con licencia GNU, integrable en el entorno Eclipse y desarrollada por la comunidad de desarrolladores SINTEF con el patrocinio de la Unión Europea.

OCL (*object constraint language*) *m* Lenguaje que complementa a UML y que se utiliza principalmente para escribir restricciones sobre sistemas orientados a objeto, de modo que se elimine la ambigüedad inherente al lenguaje natural.

OMG (Object Management Group) *m* Consorcio para la estandarización de lenguajes, modelos y sistemas para el desarrollo de aplicaciones distribuidas y su interoperabilidad.

perfil *m* Extensión de un subconjunto de UML orientado a un dominio, utilizando este-
reotipos, valores etiquetados y restricciones.

programación generativa *f* Método de producción de software que trata del diseño e implementación de software reutilizable para generar familias de sistemas de un dominio, y no sistemas independientes. Se apoya de manera importante en lenguajes específicos de dominio, aspectos y programación genérica.

Bibliografía

Bibliografía básica

Czarnecki, K.; Eisenecker, U. (2000). *Generative Programming: Methods, Tools and Applications*. Addison-Wesley.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (2002). *Patrones de diseño: elementos de software orientado a objetos reutilizables*. Addison-Wesley.

Gomaa, H. (2005). *Designing Software Product Lines with UML. From use cases to pattern-based software architectures*. Addison-Wesley.

Kiselev, I. (2003). *Aspect-Oriented Programming with AspectJ*. Editorial SAMS.

Larman, C. (2002). *UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Prentice-Hall.

Piattini M.; Garzás, J. (2010). *Fábricas de software: experiencias, tecnologías y organización*. Madrid: Ra-Ma.

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming* (2.^a ed.). Boston: MA: Addison-Wesley.

Bibliografía complementaria

Clements, P.; Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley.

Díaz, Ó.; Trujillo, S. (2010). "Líneas de Producto Software". En: Piattini; Garzás (eds.). *Fábricas de Software*. Madrid: Ra-Ma.

Griss, M. L. (1993). "Software reuse: from library to factory". *IBM Software Journal* (vol. 4, núm. 32, págs. 548-566).

Jacobson, I.; Christeson, M.; Jonsson, P.; Övergaard, G. (1992). *Object-oriented software engineering. A use case driven approach*. ACM Press.

Krueger, C. W. (1992). *Software Reuse*. *ACM Computing Surveys* (núm. 24, págs. 131-183).

Kung, H.-J.; Hsu, Ch. (1998). *Software Maintenance Life Cycle Model. International Conference on Software Maintenance* (págs. 113-121). IEEE Computer Society.

May, R. M. (1974). "Biological Populations with Nonoverlapping Generations: Stable Points, Stable Cycles, and Chaos". *Science* (vol. 4164, núm. 186, págs. 645-647).

Meyer, B. (1997). *Construcción de software orientado a objetos* (2.^a ed.). Santa Bárbara: Prentice-Hall.

Piattini, M.; Calvo-Manzano, J. A.; Cervera, J.; Fernández, L. (1996). *Análisis y diseño detallado de Aplicaciones informáticas de gestión*. Madrid: Ra-Ma.

Polo, M.; García-Rodríguez, I.; Piattini, M. (2007). "An MDA-based approach for database reengineering". *Journal of Software Maintenance & Evolution: Research and Practice* (vol. 6, núm. 19, págs. 383-417).

Pressman, R. S. (2009). *Ingeniería del software: un enfoque práctico*. Editorial McGraw-Hill.

Sommerville, I. (1992). *Software engineering* (8.^a ed.). Addison Wesley.